1986 MID-YEAR REPORT

NASA Grant NAG 1-138

SAGA: A Project to Automate the Management of Software Production Systems

Principal Investigator
Roy H. Campbell

C. S. Beckman-Davies
L. Benzinger
G. Beshers
D. Laliberte[1]
H. Render[1]
R. Sum[1]
W. Smith
R. Terwilliger[1]

University of Illinois
Department of Computer Science
1304 W. Springfield Ave.
Urbana, IL 61801-2987.
217-333-0215

This report details work in progress on the SAGA project during the first half of 1986.

[1] These graduates were research assistants on NASA Grant NAG 1-138.

1986 MID-YEAR REPORT

NASA Grant NAG 1-138

SAGA: A Project to Automate the Management of Software Production Systems

Principal Investigator
Roy H. Campbell

C. S. Beckman-Davies
L. Benzinger
G. Beshers
D. Laliberte[1]
H. Render[1]
R. Sum[1]
W. Smith
R. Terwilliger[1]

University of Illinois
Department of Computer Science
1304 W. Springfield Ave.
Urbana, IL 61801-2987.
217-333-0215

This report details work in progress on the SAGA project during the first half of 1986.

[1] These graduates were research assistants on NASA Grant NAG 1-138.

TABLE OF CONTENTS

APPENDICIES

1. Summary

This report describes the current work in progress for the SAGA project. The highlights of the research in the last six months are:

- **Clemma**, an automated configuration librarian, is undergoing development. Clemma will provide configuration management and version control capabilities for the SAGA system. Clemma is being implemented using the Troll database and the UNIX file system. A prototype of Clemma will be completed in the Fall of 1986.
- GNU Emacs as an alternative user interface for the Epos editor.
- A formal foundation for the stepwise development of software components including a formal model for the stepwise development of verified programs and an example of a stepwise development method which falls within the framework of the formal model.
- A survey of software management techniques in AT&T.
- A design for a project management utility for SAGA.
- An implementation of the Cocomo cost model in a software package.
- A prototype implementation of ENCOMPASS written in a combination of C, Csh, Prolog and Ada.
- Simple implementations of the project management and configuration control systems in the ENCOMPASS prototype supporting "programming in the small".
- An initial version of ISLET, the language-oriented editor used to create PLEASE specifications and refine them into Ada implementations.
- An initial version of the software which automatically translates PLEASE specifications into Prolog procedures and generates the support code necessary to call these procedures from Ada.
- The run-time support routines and axiom sets for a number of pre-defined types in ENCOMPASS.
- Interfaces to the ENCOMPASS test harness and TED.
- PLEASE features to support *if*, *while*, and assignment statements, as well as procedure calls with *in*, *out* or *in out* parameters.
- PLEASE features to support a small, fixed set of types including natural numbers, lists, booleans and characters.
- PLEASE and ENCOMPASS use to develop small programs, including specification, prototyping, and mechanical verification.

Appendix A contains a list of twenty theses and papers that document the project. Six of these were produced since the last mid-year report. Appendices B through P contain reports, thesis proposals, papers, and other work produced as part of the NASA project.

2. Overview

Large scale software development is so expensive that new techniques and methods are required to improve productivity. The software development environment is a proposed solution in which software development methods and paradigms are embedded within a computer software system. The goal of an environment is to provide software developers with a computer-aided specification, design, coding, testing and maintenance system that operates at the level of abstraction of the software development process and the application domains of its intended products.

Proposed software development environments range from simple collections of software tools that enhance the development process to complex systems that support sophisticated software production methods. Every environment must include a representation for the eventual software products and a, perhaps informal, notion of the software development process. In the SAGA project, we have been investigating the principles and practices underlying the construction of a software development environment. In this report, we review our studies and results and discuss the issues of providing practical environments in the short and long term.

Research into software development is required to reduce the cost of producing software and to improve software quality. Modern software systems, such as the embedded software required for NASA's space station initiative, stretch current software engineering techniques. The requirements to build large, reliable, and maintainable software systems increases with time. Much theoretical and practical research is in progress to improve software engineering techniques. One such technique is to build a software system or environment which directly supports the software engineering process. In this report, we will describe research in the SAGA project to design and build a software development environment which automates the software engineering process.

The design of a computer-aided software development environment should be guided by the problems that arise in manual software development methods. Many of these problems are reflected in software cost estimation models and measurements. Software costs are very sensitive to mistakes in the early requirements and design phases of development. Programmers and program testers vary greatly in the productivity and quality of their work. However, high-level languages and software tools to support development may increase the productivity of a programmer. Orders of magnitude improvement in the productivity of software engineers might be achieved in many application areas if the products of software engineering can become reusable, that is, if the requirements, design, documentation, validation, and verification of a software system can be reused in maintenance and in building new systems.

The SAGA project is investigating the design and construction of practical software engineering environments for developing and maintaining aerospace systems and applications software. The research includes the practical organization of the software lifecycle, configuration management, software requirements specification, executable specifications, design methodologies, programming, verification, validation and testing, version control, maintenance, the reuse of software, software libraries, documentation and automated management. An overview of the SAGA project components is described in Appendices C and D.

In several of the papers we have produced, we argue for research into formal models of the software development process (Appendices D, F, G, and H.) Such formal models should aid experimental evaluation of the practical techniques that are used in the construction of software development environments.

The SAGA project is developing models of configuration, design, incremental development, and management. The concepts and tools resulting from SAGA are being used to develop a prototype software development system called ENCOMPASS (Appendices I and B[2]). Although the research has developed many general tools and concepts that are independent of the application language and domain, we hope to extend ENCOMPASS to support the development of large, embedded software systems written mainly in ADA.

In the remainder of this report, we describe in more detail the work accomplished this year.

3. Encompass

An initial prototype of the ENCOMPASS environment has been constructed on a Sun workstation running Unix[3]. The system uses the Verdix Ada[4] Development System as well as many tools developed by the SAGA project. The prototype contains simple facilities for configuration control and project management and has a uniform, object-oriented user interface. From ENCOMPASS, the user can invoke IDEAL (Incremental Development Environment for Annotated Languages) which provides facilities for specifying, prototyping, testing and implementing Ada programs.

IDEAL implements a development methodology similar to VDM. Procedures are first specified using pre- and post-conditions written in a subset of first order predicate logic. These specification can be automatically transformed into prototypes written in a combination of Ada and Prolog. ENCOMPASS provides tools that support the creation of acceptance tests using these prototypes. To create and refine specifications, the programmer uses ISLET (Incredibly Simple Language-oriented Editing Tool) an incremental, language-oriented editor specifically for incremental refinement of the PLEASE language.

Using ISLET, the PLEASE specification is incrementally refined into an Ada program. This process is viewed as the construction of a proof in the Hoare Calculus. Each refinement is verified before another is applied; therefore, the final program satisfies the original specification. Verification conditions are generated from each refinement step. ISLET can certify many VCs using algebraic simplifications and simple proof procedures. If these measures fail, ISLET invokes TED as an interface to a general purpose

[2] B contains an early description of our work.

[3] Unix is a trademark of AT&T

[4] Ada is a trademark of the United States government.

theorem prover.

Appendices B and I report more fully on PLEASE and ENCOMPASS. Appendix I contains Bob Terwilliger's Ph.D. Preliminary proposal and two supporting papers. The PLEASE paper in Appendix B has been presented at a conference. Appendix J contains a thesis by Phillip Roberts on the translation of predicates to Prolog.

4. Configuration Control

A prototype configuration librarian, **Clemma**, is currently under development. The goal of the system is to provide a means of organizing, indexing and storing the on–line components of software projects. Users will be able to store both individual files and hierarchies of files as configuration items in the library. An overview of some of the issues involved in configuration management and a description of a small Saga prototype can be found in the ENCOMPASS paper in Appendix I.

Because (as Nestor pointed out in a recent CMU technical report) there are many deficiencies with using just a file system or data base to represent components of a software development, we have adopted a combined approach in which both a data base and a file system are used. The deficiencies of traditional data bases and file systems for representing components of software development has been known for some time and several projects are attempting to implement persistent object storage (a French Esprit project is already implementing such a data base under Unix). It is unclear, as of this moment, whether these attempts will be successful.

Our approach of combining data bases with file systems has the advantage that it does permit the rapid prototyping of many of the facilities which are needed. It also obviates the need to construct a complex piece of software, at least until the performance characteristics of persistent object storage are better understood.

Clemma will provide several capabilities:

- Baselines of software modules can be recorded and updates can be tracked and used to form new baselines.
- Stored modules can be checked out for re–use, with access lists provided to handle problems of permission and change control.
- A browser will be incorporated so that users may more easily find useful modules in the library. This should greatly promote software re–use.
- "Views" of modules will be implemented as hierarchical groupings of stored configuration items. This will greatly aid testing, validation and re–use of software systems.
- By placing constraints on the state of items checked into the library (whether an item is fully documented, tested, etc.) one will be able to implement a development methodology for the software, and control the construction and use of individual components.

The system will be written primarily in the C programming language, and will use the Troll DBMS and Unix$^{\text{TM}}$ file system for support. The current prototype of **Clemma** is expected to be completed in the Fall of 1986.

Appendix M contains an early draft of Clemma's design, a more detailed document is being prepared. As of September, major parts of Clemma have been programmed.

5. The Epos Editor

Peter Kirslis completed the major parts of the Epos editor and finished his Ph.D. which is included as Appendix E. He is continuing development of a SAGA-based editor in his current employment at AT&T in Denver. His new editor will be based on Lex and Yacc and an internal AT&T editor interface. George Beshers regular-right part grammar based Olorin editor generator system is near completion. George is currently revising his Ph.D. thesis having passed the oral examination.

The prototype user interface to the Epos editor became the major obstacle to deploying Epos for practical software development. In order to facilitate the integration of several Saga utilities, we decided to adopt the GNU Emacs extensible editor as the front end user interface. The EPOS incremental parser, the incremental semantics processor, and other Saga utilities may now be added to the GNU Emacs environment as background processes which will communicate with each other through Emacs. Each pair of communicating processes requires an interface which is programmed in the GNU Emacs extension language, ELisp.

The interface between GNU Emacs and the incremental parser has been completed. GNU Emacs itself was changed to pass all text changes to the interface. The interface collects these changes within local regions, and eventually passes them on to the incremental parser. Parsing errors are signalled with an error message and the unparsed text is highlighted. Highlighting required another, more difficult change to GNU Emacs. User commands which need to look at the parse tree, such as token movement or tree selection, ask the parser to return the appropriate information.

A number of modifications were made to the Epos incremental parser to allow it to be used with the Emacs front-end. The primary task was to extract the parser from the Epos editor and to develop an interface of primitive commands to be used by Emacs.

The parse tree representation was upgraded to allow arbitrary text to be stored in the tree (including tabs and trailing blanks). Standard Pascal multi-line comments are now supported, although a change of the termination of a comment is not yet properly reparsed. Also added was a module to allow selection and modification of a range of the parse tree for use by the editor. A number of previously-existing bugs in the parser were revealed and fixed while developing this new interface. Appendix L contains a description of the new GNU EMACS-based Epos.

6. Software Engineering Management

We wish to automate much of the control, communications, and tracking that is associated with the products involved during the lifetime of a software system. To date, we have been looking at various global pictures of the software lifetime to determine what management structures are used and what they require to be used effectively. We would like the management tool to support most management structures of workers (including managers) and documents (including program and management).

Appendix O contains a summary of management techniques used in AT&T Middletown to support the software for System 75, the digital telephone exchange. The summary was collected by Bob Sum on a visit to AT&T. The summary is being correlated with the various NASA proposed lifecycle tasks. We have also being studying other proposed project management systems. As part of these studies, Professor Campbell attended the Lancaster Software Environments conference, Trondheim Software Engineering conference, and RADC KPSA meeting. The most advanced of project management systems appear to be that of the Carnegie Group Inc., the Kestrel Institute, Boeing, and TRW. It is clear from these studies that there still remains much to be done to integrate project management with the other activities in software development and that most systems remain primitive or are prototypes.

In Appendix C, Campbell and Terwilliger discuss the notion of tasks being passed between the *in trays* and *out trays* of software developers. That paper begins to address the problem of interrelating project management with configuration control and other SAGA tools. Project management and configuration control interaction have also been prototyped as part of ENCOMPASS and a description of this work can be found in the ENCOMPASS paper in Appendix I. In particular, the need for a finer granularity of milestone is discussed. Further extension of these ideas that should form part of an eventual management tool may be found in Appendix N.

Work is now progressing on developing an implementation of these ideas. This work will build upon Clemma and earlier designs for the project management system.

7. A Model for Stepwise Development of Programs

The task of specifying and designing a software a software component and verifying that the component satisfies a given specification is quite difficult. An approach which makes this task more manageable is to divide the development of a software component into a series of steps. At each step the following occur:

(1) The software component is specified. At each step after the first, the specification is an augmentation of the specification at the preceding step.

(2) Design decisions which are consistent with design decisions at preceding steps are made.

(3) It is determined that the (possibly incomplete) software component satisfies its specification.

The Vienna Development Method (VDM) [Jones, 80] is an example of such a stepwise development method.

In order to study the properties of a particular stepwise development method or to compare different stepwise development methods, it would be advantageous to have a formal model for the stepwise development process. In addition, any attempt to automate this process would benefit from formalizing the notions involved. A formal model has been constructed and is described in some detail in Appendix H. More concise statements of the model will be found in Appendices F and G. It is conceptually simple and independent of both the specification method used and the method used for determining that a software component satisfies its specification. It contains formal definitions for such basic ideas as a development, a correct development, a development step, and a correct development step.

The model has been used in the study of an example of a stepwise development method. The example is a method for the stepwise development of programs which are verified to be partially correct with respect to specifications. The specifications are expressed in terms of pre– and post–conditions. The model has been most helpful in the construction of the example. In one case, the requirements of the model were met in the example because of the soundness and relative completeness of the Hoare calculus. If the example is viewed apart from the model, it is not obvious that these properties of the Hoare calculus are needed. The model was also useful in modifying the Hoare calculus, which is a method for program verification, into a stepwise verification method for software components.

A description of the formal model and results concerning the properties of the model have been obtained. An example of a stepwise development method based upon the Hoare logic and calculus has been studied in detail. It has been proved that this development method has the properties of the formal model. The details of this model, the results, and examples are given in the Appendices.

8. Comparison Tools and Software Environments

Carol Beckman has continued her studies into the uses of differences in software development. Her Ph.D. preliminary thesis proposal surveys differencing techniques and discusses the various approaches she is investigating to improve the use of these methods in software development environments (see Appendix K.)

9. A COCOMO cost estimating package

As part of a Software Engineering course during the Spring of 1986, Professor Campbell's students implemented a cost estimating package for software development based on Barry Boehm's COCOMO model. Documentation of the package is included in Appendix P.

SAGA Bibliography

Roy H. Campbell

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, Illinois

SAGA Bibliography

October 8, 1986

1. Campbell, Roy H. and Paul G. Richards. *SAGA: A system to automate the management of software production.* **Proceedings of the National Computer Conference** (May 1981) pp. 231-234.

2. Dever, Steve. "A Multi-Language Syntax-Directed Editor", M.S. Thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1981.

3. Essick, Raymond B., IV and Robert B. Kolstad. "Notesfile Reference Manual", Report No. UIUCDCS-R1081, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1982.

4. Richards, Paul G. "A Prototype Symbol Table Manager for the SAGA Environment", M.S. Thesis, Dept. of Computer Science, U. of Illinois at Urbana-Champaign, 1984.

5. Badger, Wayne H. "MAKE: A Separate Compilation Facility for the SAGA Environment", M.S. Thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1984.

6. Essick, Raymond B., IV. "Notesfiles: A Unix Communication Tool", M.S. Thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1984.

7. Campbell, Roy H. and Peter A. Kirslis. *The SAGA Project: A System for Software Development.* **Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments** (April 1984) pp. 73-80.

8. Campbell, Roy H. and P. E. Lauer. *RECIPE: Requirements for an Evolutionary Computer-based Information Processing Environment.* **Proceedings of the IEEE Software Process Workshop** (1984) pp. 67-76.

9. Hammerslag, David H. "TED: A Tree Editor with Applications for Theorem Proving", Report No. UIUCDCS-R-84-1190, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1984.

10. Kirslis, Peter A., Robert B. Terwilliger and Roy H. Campbell. *The SAGA Approach to Large Program Development in an Integrated Modular Environment.* **Proceedings of the GTE Workshop on Software Engineering Environments for Programming-in-the-Large** (June 1985) pp. 44-53.

11. Beshers, George M. and Roy H. Campbell. *Maintained and Constructor Attributes.* **Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments** (June 1985) pp. 34-42.

12. Hammerslag, David H., Samuel N. Kamin and Roy H. Campbell. *Tree-Oriented Interactive Processing with an Application to Theorem-Proving.* **Proceedings of the Second ACM/IEEE Conference on Software Development Tools, Techniques, and Alternatives** (December, 1985).

13. Kimball, John. "PCG: A Prototype Incremental Compilation Facility for the SAGA Environment", M.S. Thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1985.

14. Terwilliger, Robert B. and Roy H. Campbell. *ENCOMPASS: a SAGA Based Environment for the Composition of Programs and Specifications.* **Proceedings of the 19th Hawaii International Conference on System Sciences** (January 1986) pp. 436-447.

15. Kirslis, Peter A. "The SAGA Editor: A Language-Oriented Editor Based on an Incremental LR(1) Parser", Ph. D. Dissertation, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1986.

16. Terwilliger, Robert B. and Roy H. Campbell. *PLEASE: Predicate Logic based ExecutAble SpEcifications.* **Proceedings of the 1986 ACM Computer Science Conference** (February, 1986) pp. 349-358.

17. Kirslis, Peter A., Robert B. Terwilliger and Roy H. Campbell. *An Integrated Modular Environment for SAGA (Abstract).* **Proceedings of the 19th Annual Hawaii International Conference on System Sciences** (January, 1986).

18. Roberts, Philip R. "Prolog Support Libraries for the PLEASE Language", M.S. Thesis, Dept. of Computer Science, U. of Illinois at Urbana-Champaign, 1986.

19. Campbell, Roy H. *SAGA: A Project to Automate the Management of Software Production Systems.* In: **Software Engineering Environments**, Ian Sommerville, ed. Peter Perigrinus Ltd, 1986, pp. 182-201.

20. Campbell, Roy H. and Robert B. Terwilliger,. *The SAGA Approach to Automated Project Management.* In: **International Workshop on Advanced Programming Environments**, Lynn R. Carter, ed. Springer-Verlag Lecture Notes in Computer Science, New York, 1986, pp. 145-159.

21. Terwilliger, Robert B. and Roy H. Campbell. "PLEASE: Executable Specifications for Incremental Software Development", Report No. UIUCDCS-R-86-1295, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1986.

22. ——. "ENCOMPASS: an Environment for the Incremental Development of Software", Report No. UIUCDCS-R-86-1296, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1986.

PLEASE: Predicate Logic based ExecutAble SpEcifications

Robert Terwilliger

Roy H. Campbell

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois

Appeared in the
Proceedings of the 14th Annual ACM
Computer Science Conference
Cincinnati, Ohio, February, 1986

PLEASE:
Predicate Logic based ExecutAble SpEcifications

Robert B. Terwilliger
Roy H. Campbell

Department of Computer Science
1304 W. Springfield Ave.
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
217-333-0215

To appear in
Proceedings of the 14th Annual ACM
Computer Science Conference
Cincinnati, Ohio, February, 1986

Preprint November 30, 1985

This research is supported by NASA Grant NAG 1-138

PLEASE:
Predicate Logic based
ExecutAble SpEcifications[1]

Robert B. Terwilliger
Roy H. Campbell

Department of Computer Science
University of Illinois at Urbana-Champaign
252 Digital Computer Laboratory
1304 West Springfield Avenue
Urbana, IL 61801
(217) 333-4428

Abstract

PLEASE is an executable specification language which supports program development by incremental refinement. Software components are first specified using a combination of conventional programming languages and mathematics. These abstract components are then incrementally refined into components in an implementation language. Each refinement is verified before another is applied; therefore, the final components produced by the development satisfy the original specifications. PLEASE allows a procedure or function to be specified using pre- and post-conditions written in predicate logic and an abstract data type to have a type invariant. PLEASE specifications may be used in proofs of correctness, and may also be transformed into prototypes which use Prolog to "execute" pre- and post-conditions. The early production of executable prototypes for experimentation and evaluation may enhance the development process.

1. Introduction

It is widely acknowledged that producing correct software is both difficult and expensive. To help remedy this situation, methods of specifying[13,19,20,26,29,31] and verifying[14,16,19,27,38] software have been developed. The SAGA (Software Automation, Generation and Administration) project is investigating both the formal and practical aspects of providing automated support for the full range of software engineering activities[2,6,8,15,23,35]. PLEASE is a language being developed by the SAGA group to support the specification, prototyping, and rigorous development of software components. In this paper we describe the development methodology for which PLEASE was created, give an example of development using the language, and describe the methods used to prototype PLEASE specifications.

A *life-cycle model* describes the sequence of distinct stages through which a software product passes during its lifetime[10]. There is no single, universally accepted model of the software life-cycle[3,40]. The

[1] This research is supported by NASA grant NAG 1-138.

stages of the life-cycle generate *software components*, such as code written in programming languages, test data or results, and many types of documentation. In many models, a *specification* of the system to be built is created early in the life-cycle; as components are produced they are *verified*[10] for correctness with respect to this specification. The specification is *validated*[10] when it is shown to satisfy the customers requirements.

Producing a valid specification is a difficult task. The users of the system may not really know what they want, and they may be unable to communicate their desires to the development team. If the specification is in a formal notation it may be an ineffective medium for communication with the customers, but natural language specifications are notoriously ambiguous and incomplete. *Prototyping*[12,24] and the use of executable specification languages[21,22,29,41] have been suggested as partial solutions to these problems. Providing the customers with prototypes for experimentation and evaluation early in the development process may increase customer/developer communication and enhance the validation and design processes.

To help manage the complexity of software design and development, methodologies which combine standard representations, intellectual disciplines, and well defined techniques have been proposed[17,19,37,39]. For example, it has been suggested that *top-down development* can help control the complexity of program construction. By using *stepwise refinement* to create a concrete implementation from an abstract specification we divide the decisions necessary into smaller, more comprehensible groups. Methods to support the top-down development of programs have been devised[19,32] and put into use[34]. It has also been proposed that software development may be viewed as a sequence of transformations between specifications written at different *linguistic levels*[25]; systems to support similar development methodologies have been constructed[30].

The Vienna Development Method[19,34] supports the top-down development of programs specified in a notation suitable for mathematical verification. In this method, programs are first written in a language combining elements from conventional programming languages and mathematics. A procedure or function may be specified using *pre-* and *post-conditions* written in predicate logic; similarly, an *invari-*

ant may be specified for a data type. Then these *abstract programs* are incrementally refined into programs in an implementation language. The refinements are performed one at a time, and each is verified before another is applied; therefore, the final program produced by the development satisfies the original specification.

Path Pascal[7] is an extension to standard Pascal allowing concurrent programming and encapsulated data types. In Path Pascal, a *process* is a program structure which has an independent thread of execution; independently executing processes communicate through shared data structures. Encapsulated data types called *objects* are manipulated only by the predefined routines associated with the type. *Path expressions*[4,5] specify synchronization constraints that apply to the execution of the processes, functions and procedures within objects.

PLEASE is an extension of Path Pascal, which supports a methodology similar to the Vienna Development Method. In PLEASE, a procedure or function may be specified with pre– and post–conditions written in predicate logic, and similarly an object may be specified using an invariant. For ease of expression, several data types have been added to the language. PLEASE specifications may be used in proofs of correctness; they also may be transformed into prototypes which use Prolog[9] to "execute" pre– and post–conditions, and may interact with other modules written in conventional languages. We believe that the early production of executable prototypes for experimentation and evaluation will enhance the software development process.

In section two of this paper, we describe the development methodology PLEASE was designed to support, and in section three, we give an example of program development using PLEASE. First we discuss an example program specification and describe how an executable prototype could be created for it. Then we show a refinement of this specification and discuss the process of verifying that the refined specification satisfies the original. In section four, we give an example of data type specification in PLEASE, and in section five, we discuss the implementation of the system. In section six, we describe the work we have planned for the future and in section seven, we summarize and draw some conclusions from our experience.

2. Incremental Program Development

Figure 1 shows a view of the life-cycle model which PLEASE was designed to support; a different perspective is given in[35]. In our model, a *customer* comes to a software development team to have a system constructed. In the *requirements definition phase*, the functions and properties of the software to be produced by the development are determined[10]. A *systems analyst* produces a *software requirement specification*[10], which precisely describes each requirement of the software to be produced. In our model, software requirements specifications are a combination of natural language and components specified in PLEASE. PLEASE specifications may be transformed into prototypes which can be used for experimentation and evaluation; they are also formal specifications of components to be produced which can be used throughout the rest of the life-cycle. By providing executable components early in the development process, errors in the requirements specification may be discovered and corrected before the internal structure of the system has been defined.

Although a software system may be shown to meet the specification, this does not imply that the system satisfies the customers requirements. The *validation phase* attempts to show that any system which satisfies the specification will also satisfy the customers requirements, that is, that the requirements specification is valid. If not, then the requirements specification should be corrected before the development proceeds any further. In this phase the systems analyst interacts with the users to produce the *system validation summary*[35], which describes the customer's evaluation of the software requirements specification.

To aid in the validation process, the PLEASE components in the specification may be passed to a *prototyping expert* who transforms them into executable prototypes which satisfy the specifications. These prototypes may be used by the systems analyst in his interactions with the customers; they may be subjected to a series of tests, be delivered to the customers for experimentation and evaluation, or be installed for production use on a trial basis. The use of prototypes may increase customer/developer communication and enhance the validation process. If it is found that the specification does not satisfy the customers, then it is revised, new prototypes are produced, and the validation process is reinitiated; this cycle is

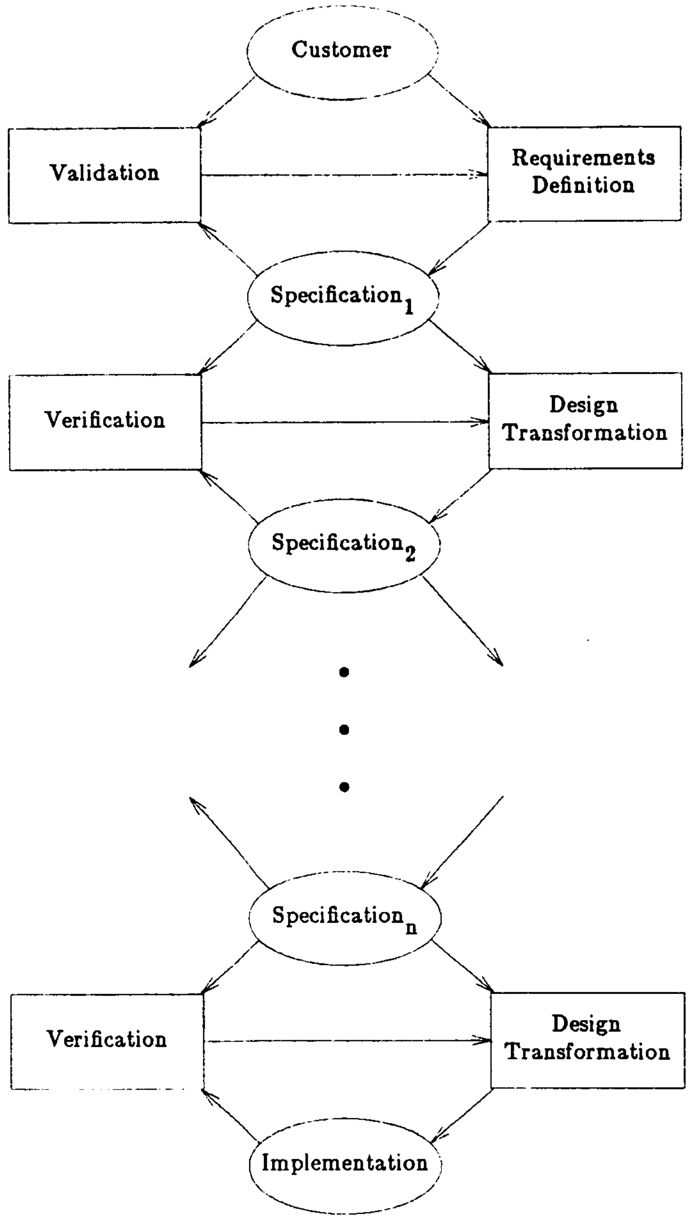


Figure 1. Program Development Model

repeated until a validated specification is produced.

The validated specification then undergoes a refinement, or *design transformation*, in which more of the structure of the system is defined and implemented. This phase produces a *software design specification*[10], which provides a record of the design decisions made during the transformation. During the transformation, prototypes produced from PLEASE specifications may be used in experiments performed to guide the design process. The design transformation may produce components in the implementation language Path Pascal as well as an updated requirements specification. Components which have been implemented need not be refined further, but components which are only specified will undergo further refinements until a complete implementation is produced.

Although a new specification has been created, it's relationship to the original is unknown. Before further refinements are performed, a *verification phase* must show that any implementation which satisfies the lower level specification will also satisfy the upper level one. In our model, this may be accomplished using any combination of mathematical reasoning[14,19,27,38], testing[11,18,28], technical review[36], and inspection. The use of PLEASE specifications enhances the verification of system components using either testing or proof techniques. The specification of a component can be transformed into a prototype. This prototype may be used as a test oracle against which the implementation can be compared. Since the specification is formal, proof techniques may be used which range from a very detailed, completely formal proof using mechanical theorem proving to an argument presented as in a mathematics text. PLEASE provides a framework for the *rigorous*[19] development of programs. Although detailed formal proofs are not required at every step, the framework is present so that they can be constructed if necessary. Parts of a project may use detailed formal verification while other, less critical parts may be handled using less expensive techniques.

To clarify our model further and show how PLEASE specifications enhance the development process, we will consider an example of system development. We will follow the development through requirements definition, validation of the original requirements specification, a single refinement step, and verification of the design transformation.

3. An Example of Program Development

Assume that a customer needs a program which sorts a list of integers. The program should read the list from input, produce a sorted list which is a permutation of the original, and write the sorted list to output. A pre-existing module implementing lists of integers is to be reused. In the requirements definition phase, the customer discusses his needs with the systems analyst and a requirements specification is produced. Along with other documentation, this specification might contain a sort program specified in PLEASE.

3.1. Specifying a Program

Figure 2 shows a PLEASE specification for such a program. The specification uses the component *integer_list.spec* which specifies the module *integer_list*[2]. This module uses the PLEASE type *list* to define the type *integer_list* as *list of integer*. In PLEASE, as in Lisp or Prolog, lists may have varying lengths and there is no explicit allocation or release of storage. However, in PLEASE the strong typing of Pascal is retained and all the elements of a list must have the same type. In PLEASE, a list is denoted by a comma separated list of elements surrounded by $<$ and $>$. The function *hd(L)* returns the first element in a list L and the function *tl(L)* returns L with the first element removed. The function $L_1 \mid\mid L_2$ yields the concatenation of the elements of L_1 and L_2, and the constant *empty_list* denotes a list containing no elements.

The specification for the sort program defines the predicates *permutation* and *sort*, as well as giving pre- and post-conditions for the program. In PLEASE, a *predicate* defines a logical expression which can be used elsewhere. It syntactically resembles a procedure and may contain local type, variable, function or predicate definitions. The predicate *permutation* states that two lists are permutations of each other if both of the lists are empty, or if the first element in the second list is in the first list, and the remainder of the two lists are permutations of each other. The predicate *sorted* states that a list is sorted if it is empty, or if the first element in the list is the smallest and the rest of the list is also sorted. This predicate may be

[2] The statement *#include "integer_list.spec"* instructs a pre-processor to include text from the file integer_list.spec into the specification before further processing.

```
program sort(input, output) ;

#include "integer_list.spec"

var     input_list, output_list : integer_list ;

predicate permutation(list1, list2 : integer_list) ;
        var     front, back : integer_list ;
        begin
                (list1 = empty_list) and (list2 = empty_list)
                               or
                (list1 = front || < hd(list2)> || back) and
                permutation(front || back, tl(list2))
        end ;

predicate sorted(l:integer_list) ;
        var     x : integer ;
        begin
                (l = empty_list)
                        or
                forall( x | member(x,tl(l)), x >= hd(l)) and
                sorted(tl(l))
        end ;

pre_condition ;
        begin
                text_to_integer_list(input) <> integer_list_error
        end ;
post_condition ;
        begin
                (input_list = text_to_integer_list(input)) and
                permutation(input_list, output_list) and
                sorted(output_list) and
                (output_list = text_to_integer_list(output'))
        end ;

begin
end.
```

Figure 2. Specification of Sort Program

read as, a list L is sorted if L is empty, or, if for all X such that X is a member of the tail of L, X is greater than or equal to the head of L, and the tail of L is sorted.

In PLEASE, the pre-condition for a program specifies the conditions that the input data must meet before execution begins. The post-condition specifies the conditions, possibly relative to the input, that the output must meet after execution has been completed. The pre-condition for the program *sort* specifies that the input file must contain the text representation for a valid list of integers. The function *text_to_integer_list* projects from objects of type text onto objects of type *integer_list*, and returns the constant *integer_list_error* for inputs which are not valid. The post-condition for *sort* states that when the input and output files are projected onto integer_lists, the output is a permutation of the input and the output is sorted. The notation *output'* denotes the value of output after the program has executed, while *output* denotes the value before execution begins.

After the requirements specification has been created, it must be validated. The systems analyst can discuss the specification with the customer and obtain test data and expected results for the system. The PLEASE specification then can be given to an expert prototyper, who can produce a prototype which satisfies the specification. If the prototype performs correctly on the test data it can be delivered to the customer for evaluation. If the prototype does not perform correctly, then we know the specification is invalid[3].

3.2. Prototyping the Specification

Figure 3 shows a simplified version of the Prolog code which might be produced from the specification of the sort program by an expert prototyper. There are Prolog procedures for the predicates *permutation* and *sort*, as well as for the program pre- and post-conditions and the program as a whole. The procedure *sort* simply reads the input, executes the pre-condition, executes the post-condition, and then writes the output. The notion of execution is quite different for pre- and post-conditions. Executing a pre-condition involves checking that given data satisfies a logical expression; for example, *sort_pre_condition* simply checks that the function *text_to_integer_list* does not return the error indication when called with the input to the program. Executing a post-condition means finding data that satisfies a logical expression; for example, *sort_post_condition* must find a value for the output such that when the

[3] Note that if the prototype does satisfy the customer, we know only that a particular implementation does so. This does not necessarily mean that all implementations which satisfy the specification would be considered adequate by the customer. While prototypes may enhance the validation process, they do not replace communication with customers and review of the specification.

```
permutation([],[]).
permutation(List1,[Head2|Tail2]) :-
        append(Front,[Head2|Back],List1),
        append(Front,Back,Temp),
        permutation(Temp,Tail2)
.

sorted([]).
sorted(L) :-
        tl(L,Tail),
        hd(L,Head),
        forall(member(X,Tail),(X >= Head)),
        sorted(Tail)
.

sort_pre_condition(Input) :-
        not(text_to_integer_list(Input,integer_list_error))
.

sort_post_condition(Input,Output) :-
        text_to_integer_list(Input,Input_list),
        permutation(Input_list, Output_list),
        sorted(Output_list),
        text_to_integer_list(Output,Output_list)
.

sort :-
        read(Input),
        sort_pre_condition(Input),
        sort_post_condition(Input,Output),
        write(Output)
.
```

Figure 3. Prolog Code Produced from Sort Specification

input and output are projected to lists of integers, the input and output are permutations of each other and the output is sorted.

To accomplish this, *sort_post_condition* converts the input data from text form, performs a naive sort, and converts the output back to text. The procedure *permutation* functions as a *generator* and the procedure *sorted* as a *selector*. When *sort_post_condition* is invoked *text_to_integer_list* is called to convert from text to lists of integers, *permutation* is called to generate a permutation of the input list, and then *sorted* is then called to determine if the permutation is sorted. If *sorted* fails, then execution backtracks and *permutation* generates the next permutation to be evaluated. This continues until a sorted permutation is generated. At this point *sorted* succeeds, *text_to_integer_list* is called to convert the output to text format, and *sort_post_condition* returns.

Although this program produces a sorted list of integers it's performance will be quite poor; in the worst case, all the permutations of the input list will be generated and tested. The performance could be improved by substituting a pre-existing procedure which implements a superior sorting algorithm for the section of *sort_post_condition* which actually performs the sort. A prototyping expert might search libraries of specifications and prototypes to find reusable components which would improve the performance of the prototype under construction. A prototype with better performance characteristics might be subjected to more extensive testing and evaluation before further design transformations are applied. After the specification for *sort* has been validated, it can be transformed into a more concrete form.

3.3. Refining the Specification

Assume that a decision is made to implement the program using the quicksort algorithm. As a first step, the original specification might be refined to produce a PLEASE program which converts the input from text to lists of integers, calls a procedure *sort* to produce a sorted list, converts this list to text, and then writes the text to output. Figure 4 shows the specification of the procedure *sort* which would be used in such a program. This procedure takes a list of integers as input and produces a sorted list as output. First, an element is selected from the input list and the list is partitioned into two sublists, *low* and *high*, so that all the members of *low* are less than the selected element and all the members of *high* are greater. The lists *high* and *low* are then sorted recursively and the results combined to form a sorted permutation of the input.

Although this refinement has narrowed the possible implementations to those using the quicksort algorithm, there are still many design decisions left unmade. The new specification may be refined into a *family* of quicksort programs; these programs might differ in many characteristics, but all would satisfy the specification. For example, the specification for the procedure *select* only requires that *element* be a member of *list*; the algorithm used to select a particular element is not specified at this level of abstraction.

```
procedure sort(input : integer_list ; var output : integer_list) ;

var     element : integer ;
        less, greater, sorted_high, sorted_low : integer_list ;

procedure select(input : integer_list, var element : integer) ;
        pre_condition ;
                begin  true   end ;
        post_condition ;
                begin  member(element, input)      end ;

procedure partition(list : integer_list ; element : integer ;
                   var low, high : integer_list ) ;
        pre_condition ;
                begin  member(element, list)       end ;
        post_condition ;
                var    l, h : integer ;
                begin
                        permutation(list, low || < element > || high) and
                        forall( l | member(l, low), l <= element ) and
                        forall( h | member(h, high), h >= element)
                end ;

procedure combine(sorted_low : integer_list ; element : integer ;
                 sorted_high : integer_list ; var output : integer_list) ;
        pre_condition ;
                begin  true   end ;
        post_condition ;
                begin output' = sorted_low || element || sorted_high  end ;

pre_condition ;
        begin  true   end ;
post_condition ;
        begin  permutation(input, output) and sorted(output)  end ;

begin   (* sort *)
        if (input = empty_list) then output := empty_list
        else begin
                select(input, element) ;
                partition(input, element, low, high) ;
                sort(low, sorted_low) ; sort(high, sorted_high) ;
                combine(sorted_low, element, sorted_high, output) ;
        end ;
end ;   (* sort *)
```

Figure 4. Part of Refinement of Sort Specification

Similarly, the specification for *partition* only states that all the elements in *low* are less than or equal to *element* and all the elements in *high* are greater than or equal to *element*; it says nothing about the algorithm used to produce these lists. As the specification is refined further these algorithms will be defined, thereby narrowing the acceptable implementations. The data types used may undergo refinement as well as the algorithms; for example, the module *integer_list* may be refined to use an array instead of a list representation. However, before the new specification is refined further, it must be shown that any program which satisfies the new specification will also satisfy the original.

3.4. Verifying the Refinement

A number of different methods may be used to show that the refined specification satisfies the original. In the most informal case, inspection of the original and refined specifications by a senior designer, or some type of peer review process might be used. A more rigorous approach might run prototypes produced from the original and refined specifications on the same test data and compare the results; this method gives significant assurance at low cost. However, in the words of E. W. Dijkstra, "Program testing can be used to show the presence of bugs, never to show their absence." In the most rigorous case, mathematical reasoning would be used.

The Vienna Development Method[19] provides rules that can be used to generate *verification conditions* for a refinement. If the verification conditions are always true, then any implementation which satisfies the refined specification will also satisfy the original. Figure 5 shows the verification rules for sequential and conditional statements. $Pre\_OP_i(\sigma)$ is the pre-condition for OP_i; σ represents the parameters, explicit or implicit, to the pre-condition. Each OP_i is verified separately. Rule *di* guarantees that if the pre-condition for OP is true before the sequence begins execution and OP_1 through OP_{i-1} execute correctly, then the pre-condition for OP_i will be true. Rule *r1* guarantees that if OP_1 through OP_n execute correctly, then the post-condition for the entire sequence will be true.

To generate verification conditions, the appropriate pre- and post-conditions are simply substituted into the verification rules. For example, to generate verification conditions for the *sort* procedure, the rule for conditional statements is applied first; the expression

$$input = empty\_list$$

For $OP \equiv OP_1 ; OP_2 ; \ldots ; OP_n$ to be correct, show :

d1. pre_OP(σ) => pre_OP$_1$(σ)

d2. pre_OP(σ_1) and post_OP$_1$(σ_1,σ_2) =>
pre_OP$_2$(σ_2)

d3. pre_OP(σ_1) and post_OP$_1$(σ_1,σ_2) and
post_OP$_2$(σ_2,σ_3) => pre_OP$_3$(σ_3)

•
•
•

dn. pre_OP(σ_1) and post_OP$_1$(σ_1,σ_2) and
post_OP$_2$(σ_2,σ_3) and . . . and
post_OP$_{(n-1)}$(σ_{n-1},σ_n) => pre_OP$_n$(σ_n)

r1. pre_OP(σ_1) and post_OP$_1$(σ_1,σ_2) and
post_OP$_2$(σ_2,σ_3) and . . . and
post_OP$_n$(σ_n,σ_{n+1}) => post_OP(σ_1,σ_{n+1})

For $OP \equiv$ **IF e THEN** OP_1 **ELSE** OP_2 to be correct, show :

da. pre_OP(σ) and eval(e,σ) => pre_OP$_1$(σ)

db. pre_OP(σ) and not eval(e,σ) => pre_OP$_2$(σ)

ra. pre_OP(σ_1) and eval(e,σ_1) and
post_OP$_1$(σ_1,σ_2) => post_OP(σ_1,σ_2)

rb. pre_OP(σ_1) and not eval(e,σ_1) and
post_OP$_2$(σ_1,σ_2) => post_OP(σ_1,σ_2)

Figure 5. Verification Rules for Sequential and Conditional Statements

is substituted for *e*,

output := empty_list

for OP_1, and

begin select(input,element) ; ... end

for OP_2. Pre- and post-conditions for the *begin ... end* block then are generated to facilitate the proof.

The rule for sequential statements then is applied with *begin ... end* substituted for *OP*, *select(...)* for OP_1, *partition(...)* for OP_2, *sort(low,sorted_low)* for OP_3, *sort(high,sorted_high)* for OP_4, and *combine(...)* for OP_5. If the formulae produced by these substitutions are always true, then any implementations of *select*, *partition*, and *combine* which satisfy the appropriate pre– and post–conditions will produce a correct implementation of *sort*.

Automated tools may be used to perform the appropriate substitutions and format the resulting logical formulae. These formulae may then be proved by inspection, rigorous argument, or using an automatic theorem prover; the SAGA project has developed a system which supports the creation and management of proofs using a number of automated theorem provers[15]. Once the refinement has been verified, the new specification may be refined further, and the process repeated until an implementation is produced. Although this example shows only the specification of an entire program, PLEASE may also be used to specify separately compiled components such as abstract data types.

4. Specifying Abstract Data Types

It has been proposed that the use of *abstract data types* can enhance program specification and verification[13,14,20,26,29]. In PLEASE, abstract data types may be specified using an extension of Path Pascal objects. Figure 6 shows the specification of an object implementing a stack of integers in terms of the type *integer_list* or *list of integer*. An object has a scope like a procedure or function; the variables declared local to the object form its *state*[19], in this case a single variable of type *integer_list*. The *invariant* defines the set of legal states, in other words the permitted values of the state variables; the invariant must be true both before and after the execution of any procedure which manipulates the state. The post–condition for a procedure or function associated with an object should specify the value of the state at the end of execution, as well as the values of any output parameters.

The *stack* has four *entry procedures* which may be called from outside the object; any procedures or functions not so declared may not be invoked from an external scope. The first item in the object is the path expression, which can be used to specify synchronization constraints; in this case no constraints are

```
type    stack = object

        path push, pop, top, empty end ;

        var s : integer_list ;

        invariant ;
                begin true end ;

        entry procedure push(elmt : integer) ;
                pre_condition ;
                        begin true end ;
                post_condition ;
                        begin s' = < element > || s end ;

        entry procedure pop ;
                pre_condition ;
                        begin true end ;
                post_condition ;
                        begin s' = tl(s) end ;

        entry function top : integer ;
                pre_condition ;
                        begin not(empty) end ;
                post_condition ;
                        begin s' = s and top' = hd(s) end ;

        entry function empty : boolean ;
                pre_condition ;
                        begin true end ;
                post_condition ;
                        begin
                                (empty' = true and s = empty_list) or
                                (empty' = false and s <> empty_list)
                        end ;

        initially ;
                pre_condition ;
                        begin true end ;
                post_condition ;
                        begin s' = empty_list end ;

end ; (* stack *)
```

Figure 6. Stack of Integers in Terms of *integer_list*

specified, so all execution sequences are allowed. The procedure *push* takes an integer and puts it on the stack, while the procedure *pop* removes the top element from the stack. The function *top* returns the integer at the top of the stack while the function *empty* checks if any items are on the stack. The *initially* block is executed when storage for the object is allocated and may be used to set the initial value of the state.

5. Implementation

A prototype implementation of PLEASE is being constructed on a Vax running BSD 4.2 Unix[4]. In this implementation, PLEASE specifications are transformed into code for the UNSW Prolog Interpreter[33]. In a program which combines modules written in conventional languages with PLEASE prototypes, the Prolog interpreter is run as a co-routine which uses Unix pipes to communicate with the rest of the program. When a call is made to a routine which is implemented using Prolog, the parameters are converted to the appropriate format and sent down the pipe to the interpreter. When the execution is complete, the results are sent back up the pipe, converted to the proper format, and the call returns. A set of standard representations for PLEASE data types has been devised, and routines to manipulate these representations have been added to the Prolog run-time library.

To prototype a module with a procedure call interface, the PLEASE specification is transformed into a *body* and a number of *headers*. The body contains code in a programming language which may be compiled using standard tools to produce an object file. The headers contain interface specifications, which may be included during the separate compilation of other components which use the body. The object code for the body can then be linked in with the object files produced to create an executable system. Using this method we have created systems which integrate modules written in C, Pascal, and Path Pascal with prototypes created from PLEASE specifications.

6. Future Work

Although PLEASE is currently an extension to Path Pascal, the basic specification, verification and prototyping methods are independent of the implementation language used. In the long term, we plan to

[4] Unix is a trademark of AT&T Bell Laboratories

use Ada[5] as our implementation language.

At present, the transformation of PLEASE specifications into Prolog code is largely a manual process. We have designed a system to perform many of these transformations automatically. This system will search libraries of specifications and implementations for components to be reused in the prototype being constructed. We hope this will allow the automatic prototyping of a large class of PLEASE programs. We plan to build a prototype implementation to better judge the feasibility of this approach. We also plan to investigate the possibility of extending these tools into an expert system for prototyping. For example, if the system could not find a component with an logically equivalent specification, then specifications with weaker pre–conditions and stronger post–conditions could be considered. The system also might aid in the reconfiguration of prototypes for different operating environments.

In the current implementation, prototypes produced from PLEASE specifications run quite slowly as the Prolog code is interpreted and the interface between languages is inefficient. We expect that the performance of these prototypes can be dramatically increased by the use of commercially available Prolog compilers, such as[1], which produce high quality machine code and provide interfaces to conventional languages. We plan to adapt our implementation for use with a Prolog compiler and continue our efforts to increase the performance of the prototypes produced from PLEASE specifications.

We are investigating the problems involved with the formal verification of systems specified in PLEASE, and plan to investigate the problems encountered in using our methods on large projects. We plan to gain experience by specifying, prototyping, implementing, and verifying a medium sized system using our methods.

7. Summary and Conclusions

PLEASE is an executable specification language which supports program development by incremental refinement. Software components are first specified using a combination of conventional programming languages and mathematics. These abstract components are then incrementally refined into programs in

[5] ADA is a trademark of the U.S. Government, Ada Joint Program Office.

an implementation language. Each refinement is verified before another is applied; therefore, the final components produced by the development satisfy the original specifications.

Path Pascal is an extension to standard Pascal which supports concurrency and encapsulated data types. PLEASE is an extension to Path Pascal which allows a procedure or function to be specified using pre– and post–conditions written in predicate logic and an abstract data type to have a type invariant. PLEASE specifications may be used in proofs of correctness, and may also be transformed into executable prototypes.

We believe that the early production of executable prototypes for experimentation and evaluation will enhance the development process. Prototypes may increase the communication between customer and developer, thereby enhancing the validation process. Prototypes produced from PLEASE specifications may be used in experiments performed to guide the design process. PLEASE specifications may enhance the verification phase by providing a framework for the rigorous development of programs. Prototypes produced from different level PLEASE specifications can be run on the same test data and the results compared; this method can give significant assurance that a refinement is correct at a low cost. PLEASE specifications may also be used in formal proofs of correctness. PLEASE prototypes are based on existing Prolog technology, and their performance will improve as the speed of Prolog implementations increases. We believe that the use of PLEASE specifications will enhance the design, development, verification and reuse of software.

8. References

1. "Quintus Prolog Users Guide and Reference Manual (Version 3)", Quintus Computer Systems, Palo Alto, California, 1985.
2. Beshers, George M. and Roy H. Campbell. *Maintained and Constructor Attributes.* **Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments** (June 1985) pp. 34–42.
3. Blum, B. I. *The Life-Cycle – A Debate Over Alternative Models.* **Software Engineering Notes** (October 1982) vol. 7, pp. 18–20.
4. Campbell, Roy H. "Path Expressions: A Technique for Specifying Process Synchronization", Technical Report UIUCDCS-R-80-1008, Dept. of Computer Science, University of Illinois at Urbana–Champaign, 1977.
5. Campbell, R. H. and A. N. Habermann. *The Specification of Process Synchronization by Path*

Expressions. In: **Lecture Notes in Computer Science, Vol. 16**, G. Goos J. Hartmanis, ed. Springer–Verlag, 1974, pp. 89–102.

6. Campbell, Roy H. and Peter A. Kirslis. *The SAGA Project: A System for Software Development.* **Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments** (April 1984) pp. 73–80.

7. Campbell, Roy H. and Robert B. Kolstad. *Path Expressions in Pascal.* **Proceedings of the Fourth International Conference on Software Engineering** (September 1979).

8. Campbell, Roy H. and Paul G. Richards. *SAGA: A system to automate the management of software production.* **Proceedings of the National Computer Conference** (May 1981) pp. 231–234.

9. Clocksin, W. F. and C. S. Mellish. **Programming in Prolog.** Springer–Verlag, New York, 1981.

10. Fairley, Richard. **Software Engineering Concepts.** McGraw–Hill, New York, 1985.

11. Gannon, John, Paul McMullin and Richard Hamlet. *Data-Abstraction Implementation, Specification, and Testing.* **ACM Transactions on Programming Languages and Systems** (July 1981) vol. 3, no. 3, pp. 211–223.

12. Goguen, Joseph and Jose Meseguer. *Rapid Prototyping in the OBJ Exececutable Specification Laguage.* **Software Engineering Notes** (December 1982) vol. 7, no. 5, pp. 75–84.

13. Guttag, J. V. and J. J. Horning. *The Algebraic Specification of Abstract Data Types.* **Acta Informatica** (1978) vol. 10, pp. 27–52.

14. Guttag, John V., Ellis Horowitz and David R. Musser. *Abstract Data Types and Software Validation.* **Communications of the ACM** (December 1978) vol. 21, no. 12, pp. 1048–1063.

15. Hammerslag, David H., Samuel N. Kamin and Roy H. Campbell. *Tree-Oriented Interactive Processing with an Application to Theorem-Proving.* **Proceedings of the Second ACM/IEEE Conference on Software Development Tools, Techniques, and Alternatives** (December, 1985).

16. Hoare, C. A. R. *Proof of Correctness of Data Representations.* **Acta Informatica** (1972) vol. 1, pp. 271–281.

17. Jackson, M. **System Development.** Prentice–Hall, Englewood Cliffs, N.J., 1983.

18. Jalote, Pankaj. *Specification and Testing of Abstract Data Types.* **Proceedings of the IEEE Computer Software and Applications Conference** (November 1983) pp. 508–511.

19. Jones, Cliff B. **Software Development: A Rigorous Approach.** Prentice–Hall International, Engelwood Cliffs, N.J., 1980.

20. Kamin, Samuel. *Final Data Types and Their Specification.* **ACM Transactions on Programming Languages and Systems** (January 1983) vol. 5, no. 1, pp. 97–121.

21. Kamin, S. N., S. Jefferson and M. Archer. *The Role of Executable Specifications: The FASE System.* **Proceedings of the IEEE Symposium on Application and Assessment of Automated Tools for Software Development** (November 1983).

22. Kemmerer, Richard A. *Testing Formal Specifications to Detect Design Errors.* **IEEE Transactions on Software Engineering** (January 1985) vol. SE–11, no. 1, pp. 32–43.

23. Kirslis, Peter A., Robert B. Terwilliger and Roy H. Campbell. *The SAGA Approach to Large Program Development in an Integrated Modular Environment.* **Proceedings of the GTE Workshop on Software Engineering Environments for Programming–in–the–Large** (June 1985).

24. Kruchten, Philippe, Edmond Schonberg and Jacob Schwartz. *Software Prototyping Using the SETL Programming Language.* **IEEE Software** (October 1984) vol. 1, no. 4, pp. 66–75.

25. Lehman, M. M., V. Stenning and W. M. Turski. *Another Look at Software Design Methodology.*

Software Engineering Notes (April 1984) vol. 9, no. 2, pp. 38–53.

26. Liskov, Barbara H. and Stephen N. Zilles. *Specification Techniques for Data Abstractions.* IEEE Transactions on Software Engineering (March 1975) vol. SE–1, no. 1, pp. 7–18.

27. Loeckx, Jacques and Kurt Sieber. The Foundations of Program Verification. John Wiley & Sons, New York, 1984.

28. Meyers, G. J. The Art of Software Testing. John Wiley & Sons, New York, 1979.

29. Musser, David R. *Abstract Data Type Specification in the AFFIRM System.* IEEE Transactions on Software Engineering (January 1980) vol. SE–6, no. 1, pp. 24–32.

30. Neighbors, James M. *The Draco Approach to Constructing Software from Reusable Components.* IEEE Transactions on Software Engineering (September 1984) vol. SE–10, no. 5, pp. 564–574.

31. Parnas, D. L. *The Use of Precise Specifications in the Development of Software.* IFIP Congress Proceedings (1977) pp. 861–867.

32. Ross, Douglas T. *Structured Analysis (SA): A Language for Communicating Ideas.* IEEE Transactions on Software Engineering (January 1977) vol. SE–3, no. 1, pp. 16–34.

33. Sammut, C. A. and R. A. Sammut. *The Implementation of UNSW-Prolog.* The Australian Computer Journal (May 1983) vol. 15, no. 2, pp. 58–64.

34. Shaw, R. C., P. N. Hudson and N. W. Davis. *Introduction of A Formal Technique into a Software Development Environment (Early Observations).* Software Engineering Notes (April 1984) vol. 9, no. 2, pp. 54–79.

35. Terwilliger, Robert B. and Roy H. Campbell. *ENCOMPASS: a SAGA Based Environment for the Composition of Programs and Specifications.* Proceedings of the 19th Hawaii International Conference on System Sciences (January 1986).

36. Weinberg, Gerald M. and Daniel P. Freedman. *Reviews, Walkthroughs, and Inspections.* IEEE Transactions on Software Engineering (January 1984) vol. SE–10, no. 1, pp. 68–72.

37. Wirth, Niklaus. *Program Development by Stepwise Refinement.* Communications of the ACM (April 1971) vol. 14, no. 4, pp. 221–227.

38. Wulf, William A., Ralph L London and Mary Shaw. *An Introduction to the Construction and Verification of Alphard Programs.* IEEE Transactions on Software Engineering (December 1976) vol. SE–2, no. 4, pp. 253–265.

39. Yourdon, E. and L. L. Constantine. Structured Design. Prentice-Hall, Englewood Cliffs, N.J., 1979.

40. Zave, Pamela. *The Operational Versus the Conventional Approach to Software Development.* Communications of the ACM (February 1984) vol. 27, no. 2, pp. 104–118.

41. ——. *An Overview of the PAISLey Project – 1984.* Software Engineering Notes (July 1984) vol. 9, no. 4, pp. 12–19.

The SAGA Approach to Automated Project Management

Roy H. Campbell

Robert B. Terwilliger

Department of Computer Science
University of Illinois at Urbana–Champaign
Urbana, Illinois

Appeared in the
*Proceedings of the International Workshop on
Advanced Programming Environments*
Trondheim, Norway, June 16–18, 1986

The SAGA Approach to Automated Project Management

(Supported by NASA grant NAG 1-138 and an AT&T Corporation research grant)

Roy H. Campbell
Robert B. Terwilliger

Department of Computer Science
University of Illinois at Urbana-Champaign
252 Digital Computer Laboratory
1304 West Springfield Avenue
Urbana, IL 61801-2987
(217) 333-4428

Abstract

ENCOMPASS, a prototype software development environment, is being constructed from components built by the SAGA project. Application of SAGA to the major phases of the lifecycle will be demonstrated through ENCOMPASS. The system will include configuration management; a software design paradigm based on the Vienna Development Method; executable specifications; languages which can be used to support modular programming, like Berkeley Pascal or ADA; verification and validation tools and methods; and basic management tools. ENCOMPASS is intended to examine many of the requirements for the design of complex software development environments such as might be used to construct the space station software. It is intended to be used as a prototype for examining many of the more advanced features that will be required in future generations of software development environments which support aerospace applications. In this paper, we describe the framework adopted within ENCOMPASS to provide automated management. We exemplify the approach using an example taken from problem tracking and change control during software maintenance.

1. Introduction.

Research into the software development process is required to reduce the cost of producing software and to improve software quality. Modern software systems, such as the embedded software required for NASA's space station initiative, stretch current software engineering techniques. Embedded software systems often are large, must be reliable, and must be maintainable over a period of decades. The software support environment for building such software systems must ensure a high-level of quality while enabling the embedded software and the hardware on which the software runs to change and the applications for which the embedded system is designed to evolve. Furthermore, such environments must be cost effective.

The SAGA project is investigating the design and construction of software engineering environments for developing and maintaining aerospace systems and applications software (5,7). The research includes the practical organization of the software lifecycle; configuration management; software requirements specification; executable specifications; design methodologies; programming; verification; validation and testing; version control; maintenance; the reuse of software; software libraries; documentation and automated management (5,11,15,17,18,19,23,24,27,28). An overview of the SAGA project components is shown in Figure 1. The tools and concepts resulting from SAGA are being used to develop a prototype software development system called ENCOMPASS (28). The ENCOMPASS software development paradigm is shown in a diagrammatic form in Figure 2. Although the research has developed many general tools and concepts that are independent of the application language and

IFIP WG2.4 International Workshop on Advanced Programming Environments
Trondheim, Norway, June 16-18 1986

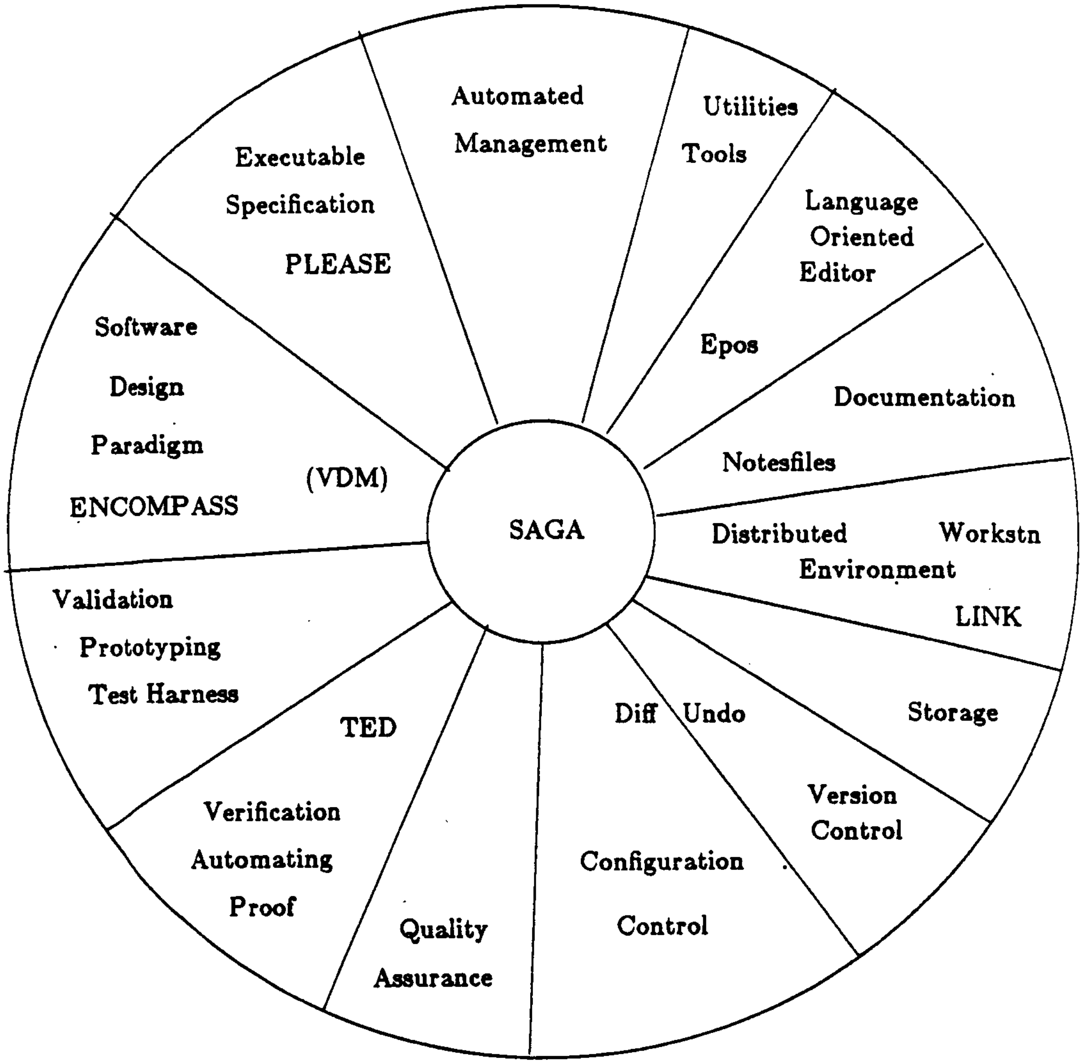


Figure 1: The SAGA workbench components

domain, we hope to extend ENCOMPASS to support the development of large, embedded software systems written mainly in ADA.

In this paper, we study mechanisms to automate the management of ENCOMPASS using a simple example based on the maintenance activities of problem tracking and change control. We describe the prototype configuration management system underlying ENCOMPASS and discuss the interelationships between this system and the automated management mechanisms.

2. The Software Development Environment.

To be effective, a software development environment must actively support the software development process (5). It must be easier to use the software development tools and the environment than to use other tools and a general operating system.

The SAGA project is concerned with software development environments, not with the construction of a general operating system. We assume that SAGA will be used in conjunction with a general operating system such as Berkeley UNIX 4.2BSD that provides a hierarchically structured file system,

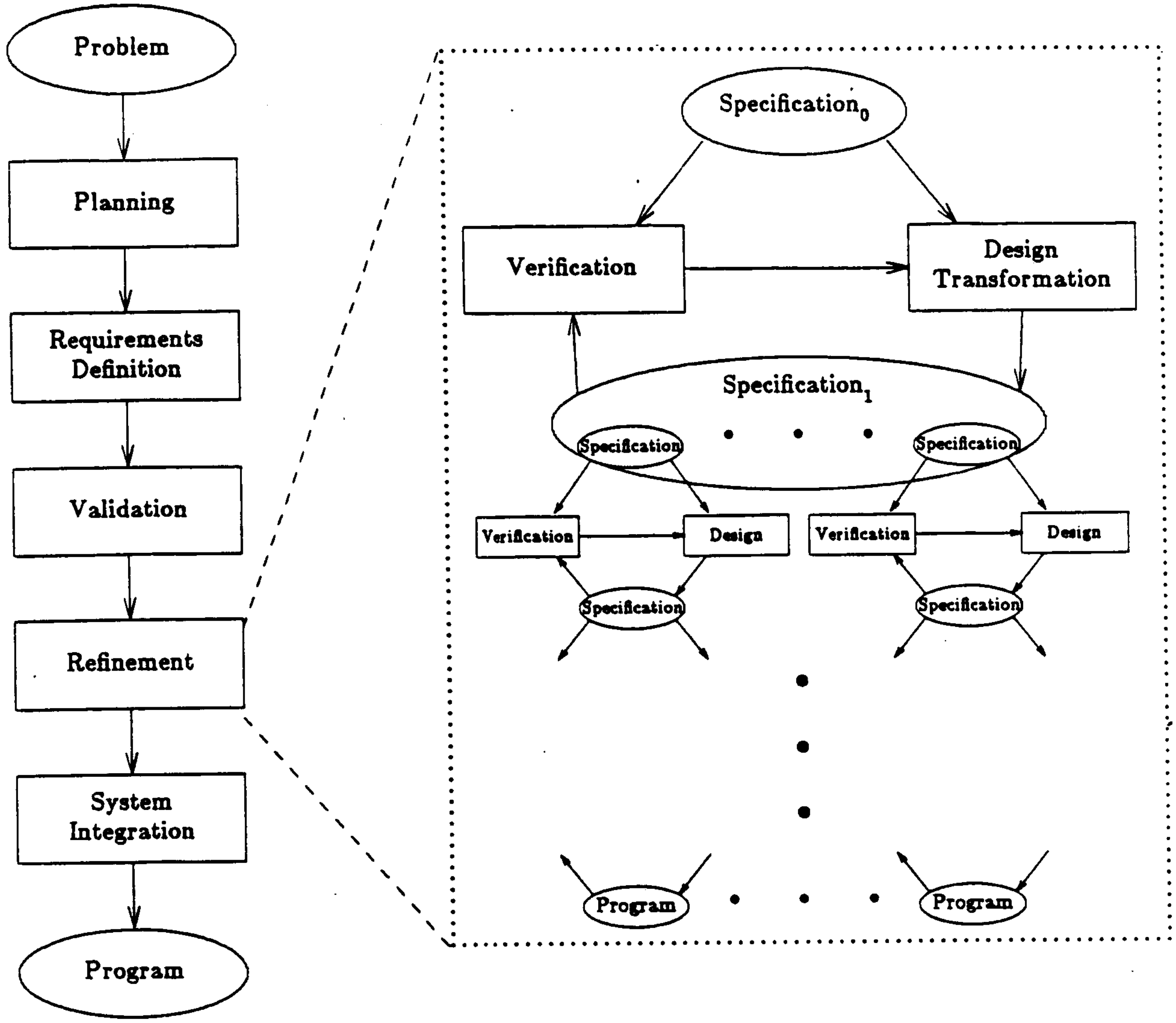


Figure 2: The ENCOMPASS software development paradigm.

virtual memory, processing operations, and mail service. Further, we assume that SAGA will be used in conjunction with an extension of the operating system that supports a networked workstation environment, perhaps using LINK (25), a kernel based version of UNIX United (2), that supports transparent remote network file access, remote spooling and remote processing.

The SAGA environment consists of a configuration management system and a workbench of software development tools which are used in a set of development, management and maintenance activities.

The *configuration management system* stores and structures the software components developed by a project which may include programs, test data, documents, manuals, designs, proofs, specifications, and contracts.

The *development, management and maintenance activities* manipulate the software components being built. They include the actions of the software developers, managers, testers, quality assurance teams, and librarians, such as the editing, compilation, or testing of a program, formatting of a document, or delegation of a task.

The *workbench of software development tools* provides the means by which activities can manipulate the software components. In ENCOMPASS (28), this workbench is the set of SAGA tools. Development, management and maintenance activities interact with the configuration management system through the SAGA user interface, which includes the SAGA language-oriented editor Epos (5,18).

3. The Software Lifecycle.

The SAGA project has adopted a "management by objectives" (14) approach to the definition of the software lifecycle (1,12). Each phase in the lifecycle is oriented towards satisfying an objective by producing a milestone. For example, the requirements specification phase produces a set of properties that the software system to be constructed must satisfy. Validation consists of determining that the specification of the system satisfies the requirements of the system and provides an important milestone in the development process. Using PLEASE (27), an executable specification language, validation can take the form of "testing" or executing the system specification. In a large project such as the space station software development program, validation may take the form of prototyping using a mixture of tools including PLEASE, simulation, standardized library routines and walk-throughs.

The design phase consists of incrementally refining the requirements specification into algorithms and component specifications. It has been shown that neither testing nor formal verifications alone can guarantee correct software (9,10). ENCOMPASS can provide an effective verification process that utilizes both testing and formal methods. The execution of the PLEASE specification for a component provides a test oracle for later use in the verification of refinements. Formal specifications and design methods also aid software reuse (20,21,22).

In ENCOMPASS, we use the specifications not only for testing, but also as the basis for rigorous and formal proofs of correctness. Thus, we intend that the system specification can also be used to prove theorems concerning the requirements of the system and to prove that a design or refinement step correctly implements a specification.

PLEASE is based on specifying programs using pre- and post-conditions. PLEASE specifications are implemented as an extension of a programming language. Both ADA and Path Pascal (6) are being used as vehicles for ENCOMPASS. The predicates are transformed into logic programs which are executed in a Prolog environment (8) that is invoked from the principal programming language. Many of the transformations may be performed automatically. Research into automating these transformations continues.

Verification conditions for the refinement of an abstract program into a more concrete one can be generated during program design. These verification conditions may be inserted into a proof tree and TED (15), a proof tree editor, may be used to manipulate them. In particular, TED permits proofs to be decomposed into sequences of lemmas. Various theorem provers may be invoked to mechanically certify the verification condition.

The development methodology used for refining system specifications into programs is similar to the Vienna Development Method (16,26). A set of rules specifies the verification conditions that are required for a given form of refinement. These rules can be applied automatically, but in general proof of the verification conditions requires some manual labor. Figure 2 summarizes the ENCOMPASS approach.

The use of formal specifications in ENCOMPASS is encouraged not only to assist code and design reuse, to promote clarity, to aid testing, and to support verification, but also to provide acceptance criteria which may be used as management objectives for a design step. The objectives can range from a mechanical proof of the correctness of a design decision to a substantial set of test data for which the design is valid.

Many of the objectives of each software development phase can be made into a milestone by requiring the activities of the phase to generate a list of documented products. These products must be validated before the phase is complete to ensure that the phase has been successful. In SAGA and ENCOMPASS, we can use language–oriented tools such as the Epos editor to further enhance the documentation of milestones. These tools can, we believe, automate repetitive effort in preparing and validating the achievement of objectives (4).

Management for the software development lifecycle must identify, control, and record the development process. A management model can be based on a *trace* of the activities within the project. Such a trace can be used to understand the meaning of management in a similar manner to the use of traces in defining the meaning of a programming language (Campbell and Lauer (3)). In ENCOMPASS, we are implementing a limited set of management functions to record, monitor, initiate activities, and inhibit inappropriate activities. Instead of using a detailed model of management, we have adopted a simpler approach based on the larger granularity provided by milestones.

4. A Framework for Automated Management.

The use of a management by objectives approach (14) in the software lifecycle introduces clearly defined milestones that are agreed upon by the developer and manager. The management objectives for each activity must define the pre–conditions under which the activity may occur, acceptance criteria for the products produced by the activity, and a procedure for evaluating whether the acceptance criteria have been met. These objectives provide a framework around which the management of the software project can be automated.

A simple demonstration of how effective such a management scheme can be is given by the following simplified example of managing software maintenance. Figure 3 shows the organizational structure of a software maintenance group. Analysts and programmers are responsible to a change control board for their contributions to the maintenance activity. Bugs and requests for modifications to maintained software are received by the maintenance group. The change control board manages the manpower and resources of the maintenance group and decides which change requests should be satisfied and which change requests should be ignored.

Figure 4 shows a simplified diagram of the flow of information that occurs within the maintenance group. Users submit change requests to the maintenance group. The change control board assigns program change requests to an analyst for further examination. A program change request may consist of a bug report or a proposal for enhancements to the software. The analyst reviews the requests and pro-

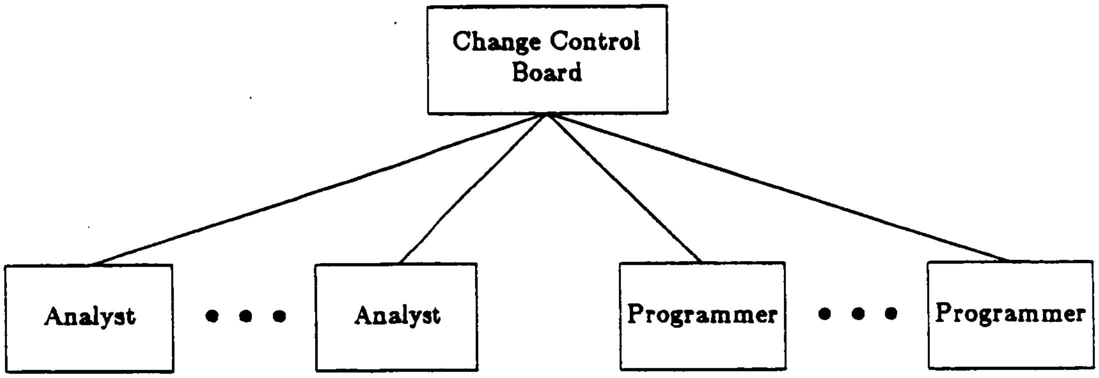


Figure 3: Organization structure

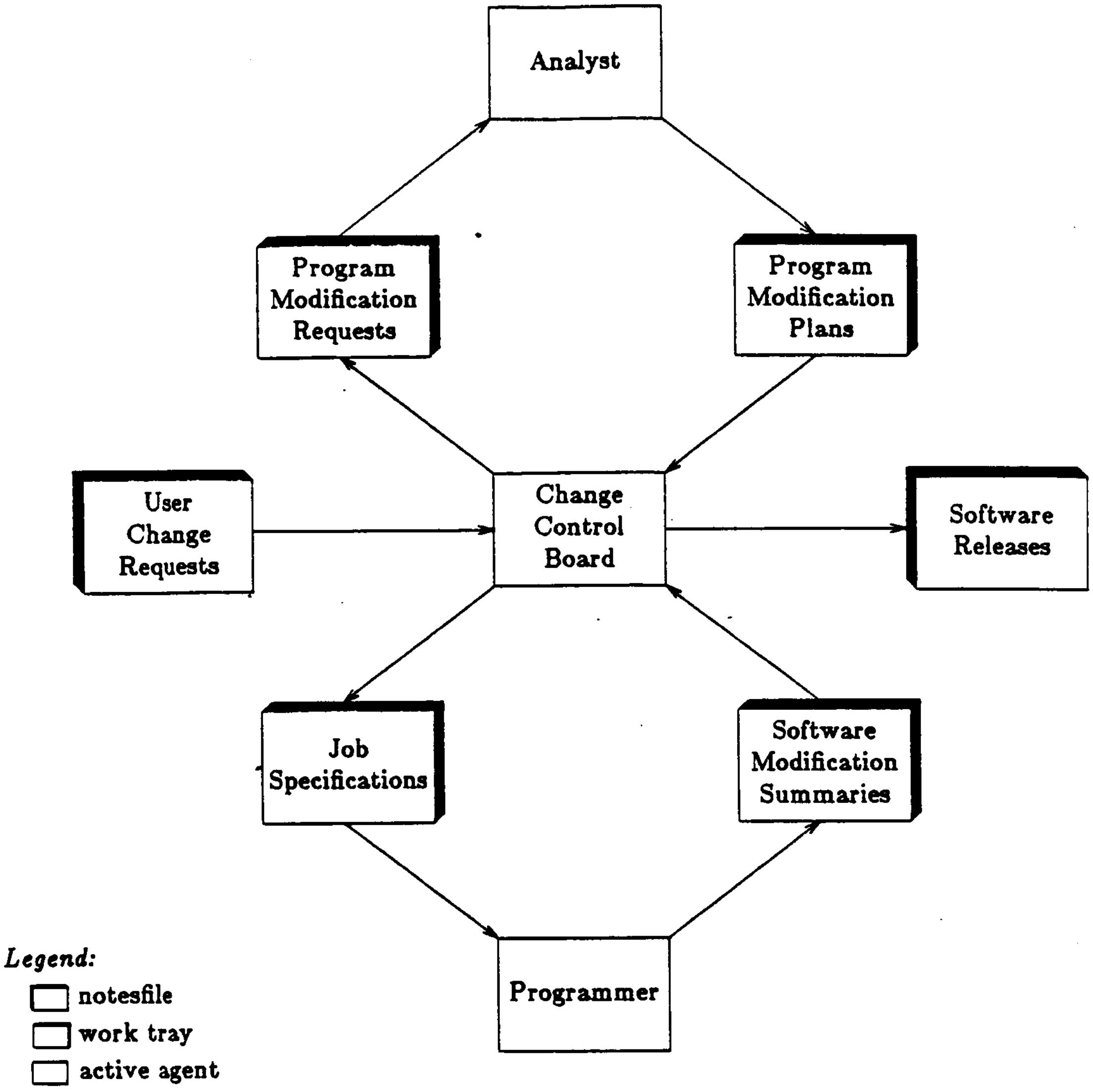


Figure 4: Data flow for change requests

duces program modification plans for those that are valid. These plans are forwarded to the change control board for approval and scheduling. The change control board may either allocate a programmer to work on a job specification based on the plan, or it may reject the plan. A rejected plan will be reconsidered by the analyst.

The programmer produces the appropriate software modifications and submits them to the change control board. The board examines the modifications and may either produce a new software release or generate a new job specification to reconsider the software modifications.

A more detailed flow diagram for the change requests would include additional feedback stages to allow analysts and programmers to negotiate their objectives with the change control board. For example, the programmer may wish to question the time allotted to accomplish the analyst's plan.

In ENCOMPASS, the management system for change control is implemented using SAGA tools. Activities within the change control system are coordinated using a combination of notesfiles, mail, makefiles, and work trays.

4.1. The Notesfile System

The *Notesfiles* system is a distributed project information base constructed for SAGA on the UNIX operating system (11). A file of notes can be maintained across a network of heterogeneous machines. Each file of notes has a topic; each note has a title. A sequence of responses is associated with each note. Notes and responses may be exchanged between separate notesfiles. Notes and responses are documented with their authors and times of creation. Updates to the notes and responses are transmitted among networked systems to maintain consistency. Notesfiles use the standard electronic mail facility to facilitate the updates. A library and standard interface permits any user program to submit a note or response to a notesfile. This library has been used in the construction of automatic logging and error reporting facilities in software and test harnesses. Within the SAGA project, we have used the Notesfile system to organize technical discussions, product reviews, problem tracking, agendas and minutes, grievances, design and specification documentation, lists of work to be done, appointments, news and mail.

4.2. Work Trays

A *work tray* is a new mechanism which has been introduced in order to manage and record the allocation, progress, and completion of work within a software development project. Each user may have a number of work trays, each of which may contain a number of *tasks* that contain software *products*. Products are stored as entities within the ENCOMPASS configuration management system. There are three types of trays: *input trays, in-progress trays,* and *file trays.* Each user receives tasks in one or more input trays. The user may then transfer these tasks to an in-progress tray where he will perform the actions required of him and produce new products. The user may then return the task via a conceptual *output* tray to an input tray for the originator of the task. A user may also create new tasks in in-progress trays that he owns. These tasks may then be transferred to another user's input tray. A task that has been transferred back into the in-progress tray of the user who created the task may be marked as complete and transferred to a file tray for long term storage.

Each task has a *home*, which is the tray where the task was created, a *location,* which is the tray where the task currently resides. and an attribute *time*, which is the time the last action involving that task took place. Status commands allow examination of the tasks in a tray and the products in a task.

4.3. Implementation of the Change Control Scheme

User change requests can be generated because of bug reports or user requests for enhanced functionality. These are sent to the change control system by electronic mail and are stored in a notesfile "User Change Requests".

A user change request is a form that can be filled in manually using an editor tailored for form filling or can be generated by software error reporting tools. It is entered into the notesfile mail system by standard mailing utilities. In this way, user change requests can be generated from a wide range of sources, some local and some remote.

The User Change Requests notesfile is the receiving station for all requests to change the software. The Change Control Board manager creates a particular "Program Modification" task in an in-progress tray. In addition to the details extracted from the notesfile, the manager may also add the amount of time within and the urgency with which a response to the request should be created. The manager transfers the task to the "Program Modification Request" input tray of an analyst, see Figure 4. The analyst will transfer the request to a in-progress tray in order to respond to the request. The analyst may create a product called an "Invalid Request" report as a result of his analysis if he believes that such a report is appropriate. Alternatively, the analyst may create a detailed description of the steps needed to implement the change or bug fix. The analyst transfers the task with the analysis of the request back to the manager's "Program Modification Plan" input tray. Should the analyst not respond to the request within a reasonable time, the periodic invocation of consistency checking programs can automatically detect the delay and enter a complaint in the "Problem Tracking Management" notesfile (which is not shown) and flag the Program Modification task with an item that documents the warning.

The manager may transfer the task back into his in-progress tray. Depending upon the products produced by the analyst, he may register the task as completed, transfer it to a file tray and write a response to the request in the notesfile that further action is unnecessary, convene the change control board, or reject the plan and reassign the task to the analyst with recommendations for a revised plan or to reject the request.

Should the manager wish to review the plan, the Change Control Board will be convened to discuss the Program Modification Plans. Alternatively, the Board may discuss the Plans electronically through the notesfile system. Given acceptance of a plan, the manager of the problem tracking system checks out the products that are needed to make the modification from the project library and enters them into the task. He then transfers the task to the "Job Specification" input tray of a programmer.

The programmer receives the task and transfers it into an in-progress tray. The programmer will add and modify code, documentation, test cases, and proofs of correctness to the products of the task. When complete, the programmer will transfer the task to the "Software Modification Summary" input tray of the manager.

When a Software Modification Summary is received, the manager will again convene the Change Control Board. If the review is satisfactory, he will check the new product into the project library as a new version of the software and announce the release of the software through the "Software Release" notesfile. If the review is unsatisfactory, he may create a new Job Specification.

At any time, the manager or programmers may query any of the tasks they have been assigned or have created. Acceptance criteria may be in the form of executable procedures which produce reports (for example, executable acceptance tests), records of compilations or examinations of the file activity of program files. These acceptance criteria may be automatically stored as products of the task. Status commands will summarize such records, report on who is currently working on the task, who is waiting for completion of the task, and what other tasks are needed to be completed before the current task can be completed.

Thus, very simple mechanisms can be used to automate management, provided that the objectives being managed are well-defined. In the example given, the problem and the resulting corrective maintenance need to be well-defined. In addition, the corrective maintenance must be validated. A feasibility study of the work tray concept has been completed and the concept is being extended. In the following section, we discuss the interaction between maintenance and the configuration management system.

5. Configuration Management System

The configuration management system is responsible for maintaining the consistency of, integrity of and relationships between the products of software development. In the SAGA project, Terwilliger and Campbell (28) model software configurations using a graph in which the nodes represent uniquely named entities or uniquely named collections of entities and the arcs represent relationships between entities. Layers within the graph represent different abstract properties of the software products. The graph also represents the organization of the software products into separate concerns.

In ENCOMPASS, software configurations can be decomposed by organizational relationships into vertical and horizontal structures. The vertical structures form a hierarchy and decompose the system into independent components. For example, within a software development *project*, the configuration may be structured into *subsystems*. These, in turn, are decomposed into *modules* which are decomposed into *compilation units*.

The horizontal structures represent dependencies between entities at the same hierarchical level. Thus, each project, subsystem, module, and unit may have a horizontal structure which includes dependencies between documents, version information, requirements and system specification, shared definitions, architectural design, detailed design, code, binaries, linked binaries, test cases, procedures for generating executable binaries, listings, reports, authors, managers, time and tool certification stamps, development histories, and concurrency control locks. Relationships may specify design, compilation and version dependencies. Depending upon the granularity of the entities, the graph can be represented by the UNIX directory structure, by symbolic links, or by databases. For example, in ENCOMPASS the vertical structure is stored using the UNIX directory structure. Shared definitions are represented by symbolic links. A database at each level in the vertical structure is being built to provide data dictionary capabilities and author manager relations.

Abstractions of the collection of software products are provided by *views*. A view represents a particular abstract property or concern and is implemented as a mapping from names into products. The "base view" is a complete collection of the software products. For example, a "functional test" view might represent the system as a collection of functional specifications, object code, test programs and test data. Other examples of views include a single version abstraction of a system that has many concurrent versions, documentation, and the work of a particular developer.

Continuing our discussion of change control within maintenance, we consider the problems arising in modifying an existing program. Figure 5 shows an example tree traversal program stored in an ENCOMPASS configuration management system (Kirslis et al (19)). Not all the dependencies and details are shown. The program is presented as a subsystem containing four modules, preorder, stack, tree, and item. Each module contains entities including a makefile (Feldman (13)), specification, body or source code, compiled object code, and executable program. Only one type of relationship is shown, the *uses*

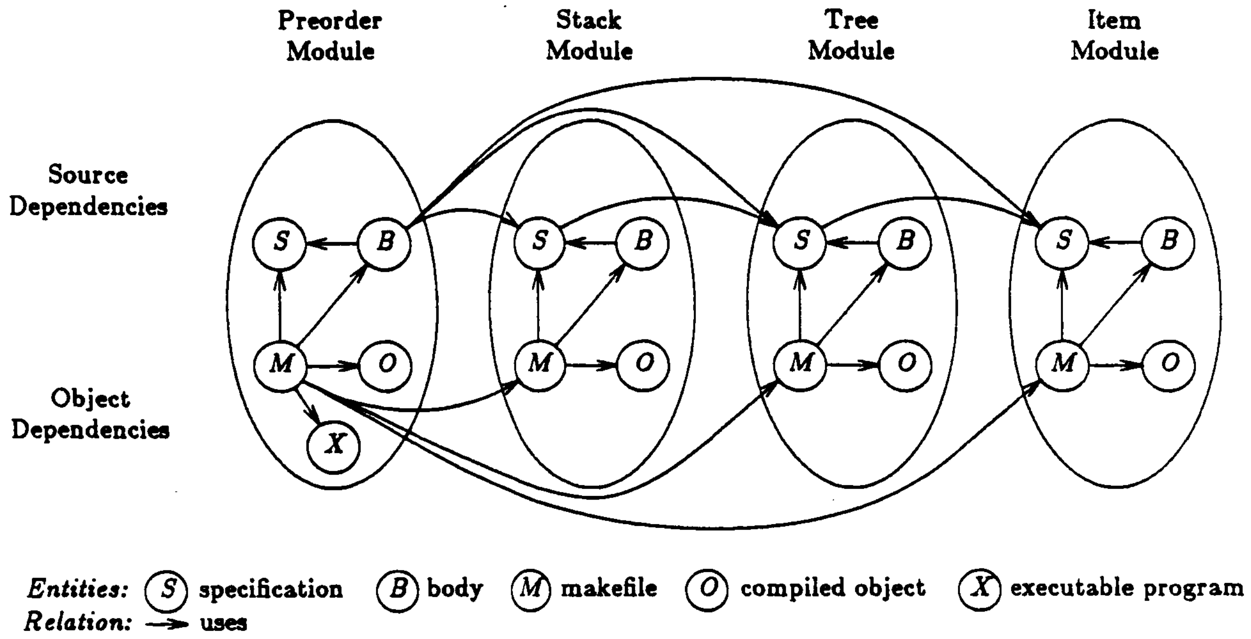


Figure 5: Base view for the *preorder* program

relationship, which associates an entity with another entity if the former entity references the latter one. Each "uses" relationship should be accompanied by a "used by" relationship, not shown in the figure, which is simply the inverse of of the "uses" relationship, and which permits the references to a module/entity to be determined from that module/entity. Each body within a module references its own specification. The body of *preorder* references the specifications in the other modules. The makefile for each module references the specification and body to be compiled, and the compiled object which will be produced. In addition, the makefile in the preorder module also references the makefiles and objects in the other modules, since it needs these in order to produce an executable program.

A number of benefits are realized if this dependency graph is stored in machine accessible form and if the software tools in use are adapted to refer to the graph. A data retrieval tool can provide information about the hierarchical structure of the program. For a given module, the tool can show its dependencies with respect to other modules.

An editor, adapted to use this graph, can permit a programmer to specify a routine, module, or program to edit. If the programmer specifies a module, that module becomes the locus at the beginning of the editing session. The programmer edits within the context of that layer of abstraction. Only the local context of the module is important. The programmer can find and display other modules, routines, or programs which use this module. These references may be checked easily to determine how a change in the current module will affect them. Similarly, other modules that are referenced by the modules which reference the module under consideration may be located easily and displayed.

Compilation tools, which access the dependency graph, can support automatic, incremental recompilation on a module by module basis. For example since the body of *preorder* depends on the specification for *stack*, if the specification for *stack* has been changed since the time *preorder* was last compiled, then *preorder* will be recompiled. A compilation tool can use the dependency graph to resolve the dependencies at compile time and access all files needed to perform a compilation.

Versions of the preorder program are stored in a program library (28). Conceptually, these versions appear in the library as independent entities as depicted in Figure 6. The versions are, however, interdependent because of the history of their construction. The versions are constructed from revision

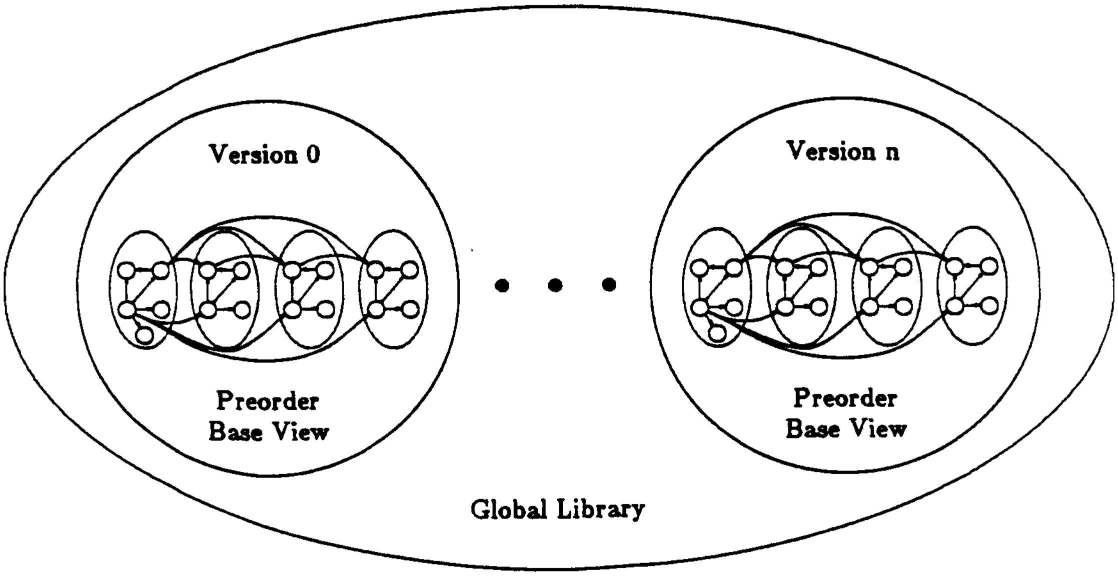


Figure 6: Global library containing versions of *preorder* base view

control histories of preorder. Each preorder version is a collection of versions of the other modules. Each version of a module is stored using a history mechanism based on the Revision Control System (RCS) of Tichy (29). Similarly, the information describing a version of preorder is also stored under RCS. Library makefiles construct a specific version of preorder within the Library on a demand or check out basis. The specific version of preorder specifies the versions of each module that are needed to be extracted. The dependencies between modules and within modules are recorded in a format that can be stored within RCS. (In our prototype ENCOMPASS environment, these dependencies are recorded using the UNIX tape archiving facility *tar* and placed directly under RCS.)

To modify preorder, a read–only copy of the latest version of preorder is checked out. This version is still under configuration management and resides within the protection provided by the global library. Figure 7 shows how a view of preorder is constructed in a workspace. The workspace facilitates chang-

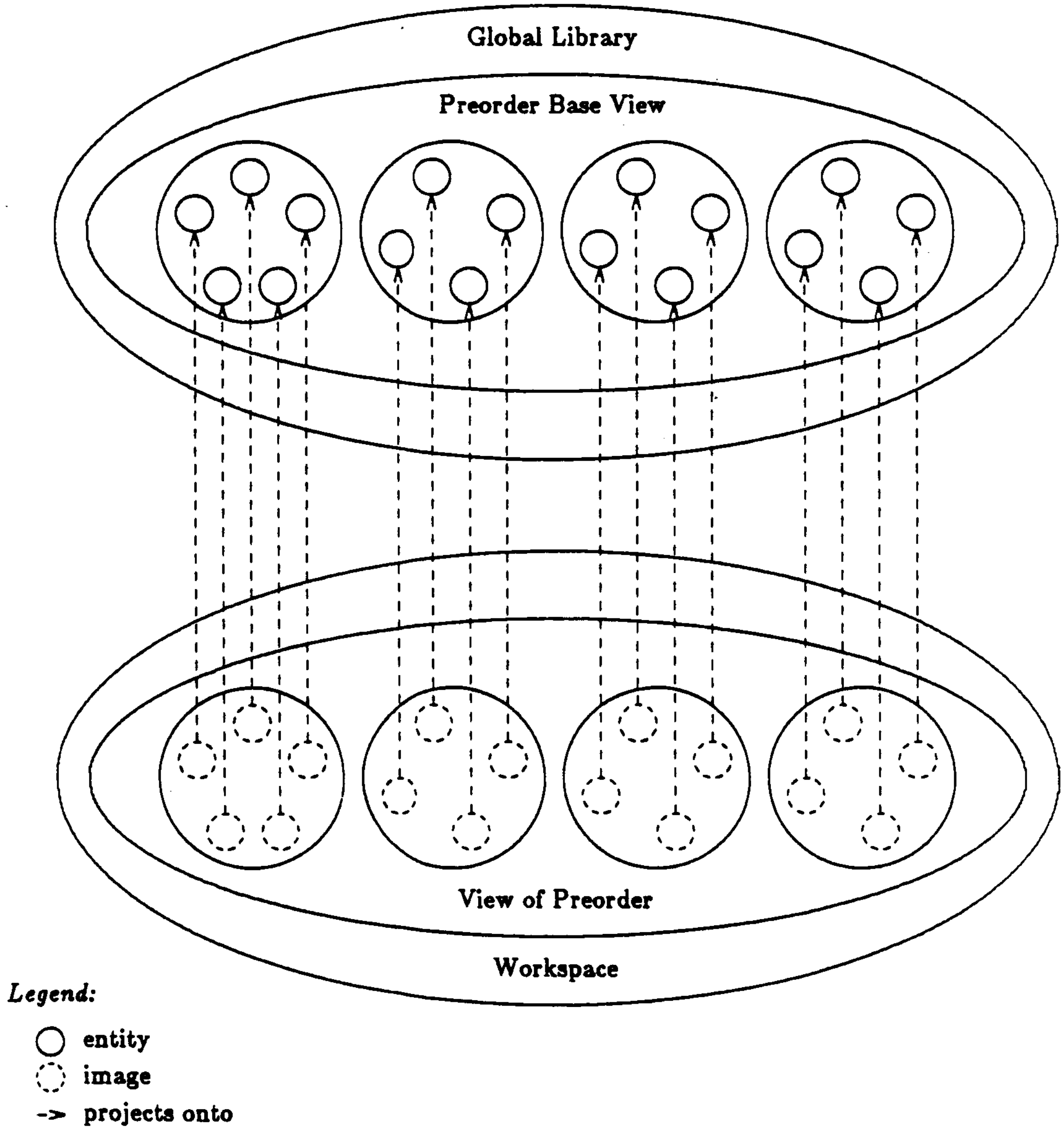


Figure 7: Workspace containing view of *preorder*

ing preorder. Each entity within preorder can be accessed read–only through this view. In the ENCOMPASS prototype, the view is implemented as a hierarchical directory structure which initially only contains symbolic links to the base view stored in the library.

In order to modify components of preorder, the entities concerned are checked into the workspace. In terms of implementation, the symbolic links are replaced by copies of the actual entities to which they correspond. Figure 8 shows a new version of preorder being developed in which two entities within module item are being modified. If the new version being developed is a sequential revision of preorder, locks are placed within the library on those modules checked into the workspace. These locks prevent any parallel development of the same entities. The next version number of preorder and the modules concerned are assigned. If the new version is instead a parallel revision of preorder, locks are not imposed but parallel revision version numbers for preorder and the modules concerned are assigned.

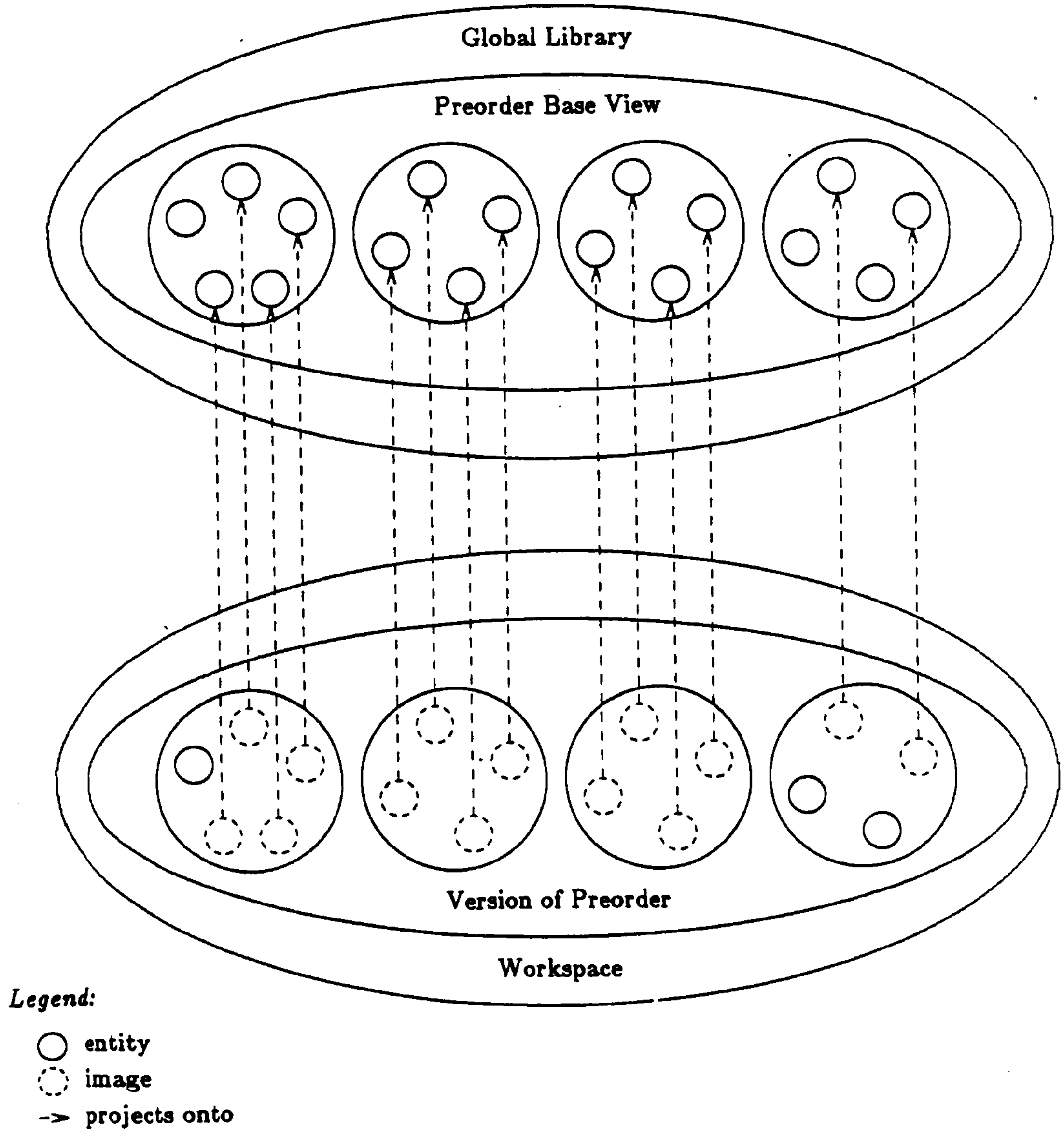


Figure 8: Workspace containing new version of *preorder*

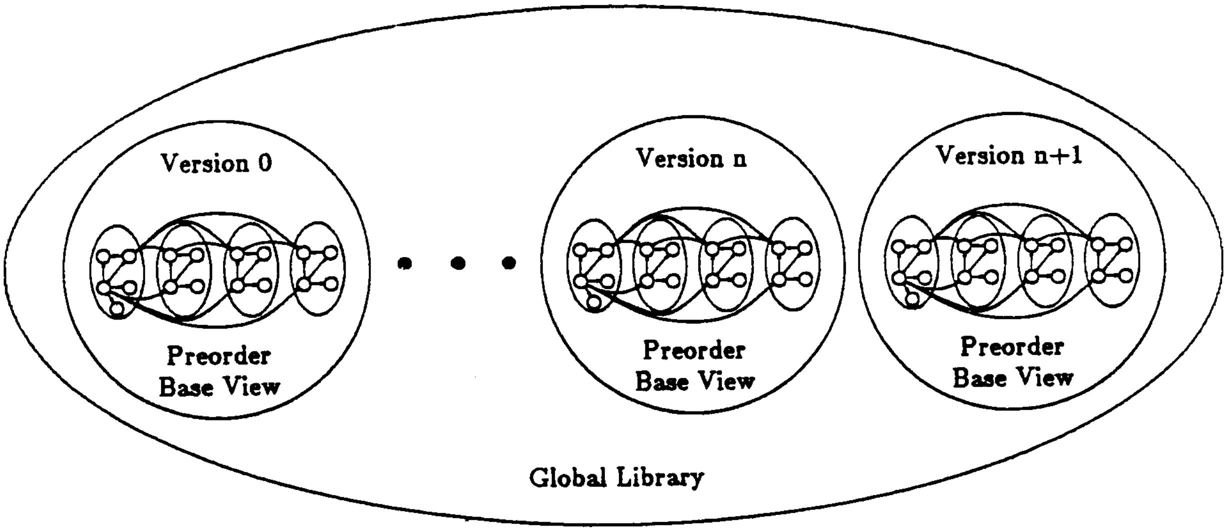


Figure 9: New version of *preorder* installed in global library

Once the development and testing of a new version is complete, the programmer submits a summary of the modifications to the change control board. The change control board evaluates the modifications and makes a recommendation as to whether the work constitutes a valid version. (In a more complex change control system, the evaluation of the new software might be performed by a quality assurance group. Our management model and implementation are easy to extend to permit such a system.) Following a software release, the new version is integrated into the library system, as shown in Figure 9, and the RCS files of the individual modules that are altered are updated.

6. Summary

This paper describes a prototype management system that has been constructed on UNIX as part of the ENCOMPASS environment. The example change control system described has been built using the system. The prototype system demonstrates the feasibility of the approach, but further research and refinement are required to develop a practical management system.

The prototype implementation is not robust and offers no protection from misuse. A complete log of the actions performed on the tasks should be kept in a secure location to support auditing. Further. the implementation has limited goals and is not fully integrated into the SAGA set of tools and the configuration system. The system permits a task to be decomposed into subtasks but should maintain records of those relationships. Finally, the system ought to be coupled to management tools such as report generators, Pert chart analyzers and flow charting displays.

However, the approach is simple and provides a framework for building automated management. We believe our approach can be refined into a production quality system for managing software projects. We shall be exploring refinements of our approach to accomplish this end.

7. References.

1. Blum, B. I., 1982, *The Life-Cycle - A Debate Over Alternative Models*, Software Engineering Notes, vol. 7, pp. 18–20.

2. Brownbridge, D. R., L. F. Marshall and B. Randell, 1982, *The Newcastle Connection or UNIXes of the World Unite!*, Software - Practice and Experience, V. 12, pp. 1147–1162.

3. Campbell, R. H. and P. E. Lauer, 1984, *RECIPE: Requirements for an Evolutionary Computer-based Information Processing Environment*, Proc. of the IEEE Software Process Workshop, pp. 67–76.

4. Campbell, R. H. and Paul G. Richards, 1981, *SAGA: A system to automate the management of software production*, **Proc. of the National Computer Conference**, pp. 231–234.

5. Campbell, R. H. and P. A. Kirslis, 1984, *The SAGA Project: A System for Software Development*, **Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments**, pp. 73–80.

6. Campbell, R. H. and R. B. Kolstad, 1979, *Path Expressions in Pascal*, **Proceedings of the Fourth International Conference on Software Engineering.**

7. Campbell, R. H., C. S. Beckman, L. Bensinger, G. Beshers, D. Hammerslag, J. Kimball, P. A. Kirslis, H. Render, P. Richards, R. Terwilliger, 1985, *SAGA*, Mid-Year Report, Dept. of Comp. Sci., University of Illinois.

8. Clocksin, W. F. and C. S. Mellish, 1981, **Programming in Prolog.** Springer–Verlag, New York.

9. DeMillo, R. A., R. J. Lipton and A. J. Perlis, 1979, *Social Processes and Proofs of Theorems.* **Communications of the ACM,** vol. 22, no. 5, pp. 271–280.

10. Dijkstra, E. W., 1970, *Structured Programming.* In: **Software Engineering Principles**, J. N. Buxton and B. Randall, ed. NATO Science Committee, Brussels, Belgium.

11. Essick, Raymond B., IV., 1984, *Notesfiles: A Unix Communication Tool*, M.S. Thesis, Dept. Comp. Sci., University of Illinois at Urbana–Champaign.

12. Fairley, Richard, 1985, **Software Engineering Concepts.** McGraw–Hill, New York.

13. Feldman, S. I., 1979, *Make - A Program for Maintaining Computer Programs,* **Software – Practice and Experience,** Vol. 9, No. 4, pp. 255–265.

14. Gunther, R., 1978, **Management Methodology for Software Product Engineering,** Wiley Interscience, New York.

15. Hammerslag, D. H., S. N. Kamin and R. H. Campbell, 1985, *Tree-Oriented Interactive Processing with an Application to Theorem-Proving.* **Proc. of the Second ACM/IEEE Conference on Software Development Tools, Techniques, and Alternatives.**

16. Jones, C., 1980, **Software Development: A Rigorous Approach,** Prentice–Hall International, Inc., London.

17. Kimball, J., 1985, *PCG: A Prototype Incremental Compilation Facility for the SAGA Environment,* M.S. **Thesis,** Dept. Comp. Sci., University of Illinois at Urbana–Champaign.

18. Kirslis, P. A., 1986, *The SAGA Editor: A Language-Oriented Editor Based on an Incremental LR(1) Parser.* Ph. D. Dissertation, Dept. Comp. Sci., University of Illinois at Urbana–Champaign.

19. Kirslis, P. A., R. B. Terwilliger and R. H. Campbell, 1985, *The SAGA Approach to Large Program Development in an Integrated Modular Environment,* **Proceedings of the GTE Workshop on Software Engineering Environments for Programming–in–the–Large.**

20. Lanergan, R. G. and C. A. Grasso, 1984, *Software Engineering with Reusable Designs and Code,* **IEEE Trans. on Software Engineering,** Vol. 10, No. 5.

21. Matsumoto, Y., 1984, *Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels,* **IEEE Trans. on Software Engineering,** Vol. 10, No. 5.

22. Neighbors, J. M., 1984, *The Draco Approach to Constructing Software from Reusable Components,* **IEEE Transactions on Software Engineering,** vol. SE–10, no. 5, pp. 564–574.

23. Richards, P., 1984, *A Prototype Symbol Table Manager for the SAGA Environment,* Master's Thesis, Dept. Comp. Sci., University of Illinois at Urbana–Champaign.

24. Roberts, P. R. 1986, *Prolog Support Libraries for the PLEASE Language,* Master's Thesis, Dept. of Comp. Sci., University of Illinois at Urbana–Champaign.

25. Russo, V. F., 1985, *ILINK: Illinois Loadable InterUNIX Networked Kernel,* M.S. thesis, University of Illinois, Urbana, Il 61801.

26. Shaw, R. C., P. N. Hudson and N. W. Davis, 1984, *Introduction of A Formal Technique into a Software Development Environment (Early Observations),* **Software Engineering Notes,** vol. 9, no. 2, pp. 54–79.

27. Terwilliger, R. B. and R. H. Campbell, 1986, *PLEASE: Predicate Logic based ExecutAble SpEcifications,* **Proc. 1986 ACM Computer Science Conference.**

28. Terwilliger, R. B. and R. H. Campbell, 1986, *ENCOMPASS: a SAGA Based Environment for the Composition of Programs and Specifications,* **Proceedings of the 19th Hawaii International Conference on System Sciences.**

29. Tichy, W., 1982, *Design, Implementation, and Evaluation of a Revision Control System*, **Proceedings of the 6th IEEE International Conference on Software Engineering**, pp. 58–67.

30. Wirth, N., 1971, *Program Development by Stepwise Refinement*, **Communications of the ACM**, vol. 14, no. 4, pp. 221–227.

Roy H. Campbell received the B.Sc. degree in mathematics from the University of Sussex, Sussex, England in 1969 and the M.Sc. and Ph.D. degrees in computer science from the University of Newcastle upon Tyne, Newcastle–upon–Tyne, England in 1972 and 1977 respectively. Since 1976, he has been with the Department of Computer Science at the University of Illinois at Urbana–Champaign and was promoted to Full Professor in 1985. His research interests include distributed systems, operating systems, programming language design, and software engineering. He is principle investigator of the EOS project to develop techniques and methodologies for programming embedded real–time operating systems (NASA) and the SAGA project which is building a prototype software development environment (NASA). Dr. Campbell is a member of the IEEE Computer Society and the Association for Computing Machinery.

Robert B. Terwilliger received the B.A. degree in chemistry from Ithaca College, Ithaca, N.Y., in 1980, and the M.S. degree in computer science from the University of Illinois at Urbana–Champaign in 1982. He attended the University of Wisconsin–Madison from 1982 until 1984. Mr. Terwilliger is currently a Ph.D. candidate in the Computer Science Department at the University of Illinois at Urbana–Champaign. Roy Campbell is his thesis advisor. Mr. Terwilliger's research interests include software engineering environments, executable specification languages, and configuration management. Mr. Terwilliger is a member of the IEEE Computer Society and the Association for Computing Machinery.

SAGA: A Project to Automate the Management of Software Production Systems

Roy H. Campbell

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois

Presented at the
Second Annual Conference on Software Engineering
University of Lancaster, England
April, 1986

Presented at the Second Annual Conference on Software Engineering Environments, University of Lancaster, England, April, 1986.

In: Software Engineering Environments, Ian Sommerville, ed. Peter Perigrinus Ltd., 1986.

SAGA: A Project to Automate the Management of Software Production Systems

(Funded by NASA Grant NAG 1-138)

Roy H. Campbell

Abstract. Large scale software development is so expensive that new techniques and methods are required to improve productivity. The software development environment is a proposed solution in which software development methods and paradigms are embedded within a computer software system. The goal of an environment is to provide software developers with a computer-aided specification, design, coding, testing and maintenance system that operates at the level of abstraction of the software development process and the application domains of its intended products.

Proposed software development environments range from simple collections of software tools that enhance the development process to complex systems that support sophisticated software production methods. Every environment must include a representation for the eventual software products and a, perhaps informal, notion of the software development process. In the SAGA project, we have been investigating the principles and practices underlying the construction of a software development environment. In this paper, we review our studies and results and discuss the issues of providing practical environments in the short and long term.

1. Introduction

Research into software development is required to reduce the cost of producing software and to improve software quality. Modern software systems, such as the embedded software required for NASA's space station initiative, stretch current software engineering techniques. The requirements to build large, reliable, and maintainable software systems increases with time. Much theoretical and practical research is in progress to improve software engineering techniques. One such technique is to build a software system or environment which directly supports the software engineering process. In this paper, we will describe research in the SAGA project to design and build a software development environment which automates the software engineering process.

The design of a computer-aided software development environment should be guided by the problems that arise in manual software development methods. Many of these problems are reflected in software cost estimation models and measurements (Boehm (4)). A major proportion of the cost of a software system is in its maintenance (60%), and testing (20%). Fairley (13) comments that software costs are very sensitive to mistakes in the early requirements and design phases of development. Sackman et al (37) and Myers (32) have demonstrated that programmers and program testers vary greatly in the productivity and quality of their work. However, high-level languages and software tools to support development may increase the productivity of a programmer by as much as 222% (4). Orders of magnitude improvement in the productivity of software engineers might be achieved in many application areas if the products of software engineering can become reusable, that is, if the requirements, design, documentation, validation, and verification of a software system can be reused in maintenance and in building new systems.

The SAGA project is investigating the design and construction of practical software engineering environments for developing and maintaining aerospace systems and applications software (Campbell and Kirslis (8)). The research includes the practical organization of the software lifecycle, configuration management, software requirements specification, executable specifications, design methodologies, programming, verification, validation and testing, version control, maintenance, the reuse of software, software libraries, documentation and automated management. The research is documented in the mid-year report (Campbell et al (10)). An overview of the SAGA project components is shown in Fig. 1.

In this paper, we will argue for research into formal models of the software development process. Such formal models should aid experimental evaluation of the practical techniques that are used in the construction of software development environments. The SAGA project is developing models of configuration, design, incremental development, and management. The concepts and tools resulting from SAGA are being used to develop a prototype software development system called ENCOMPASS (Terwilliger and Campbell, (41)). Although the research has developed many general tools and concepts that are independent of the application language and domain, we hope to extend ENCOMPASS to support the development of large, embedded software systems written mainly in ADA.

2. The Requirements of a Software Engineering Environment

Practical software development environments will be used by software developers and software managers with several years experience in software development. Although some components of the system may be used as educational tools, this is not a major goal. The requirements for a practical software development environment can be structured into three components:

1. the organization and representation of software products produced by the development process (the configuration management system,)
2. the software development processes (the lifecycle model, software development, management, and methodologies,)
3. the tools by which software development processes interface to, name, and manipulate software products.

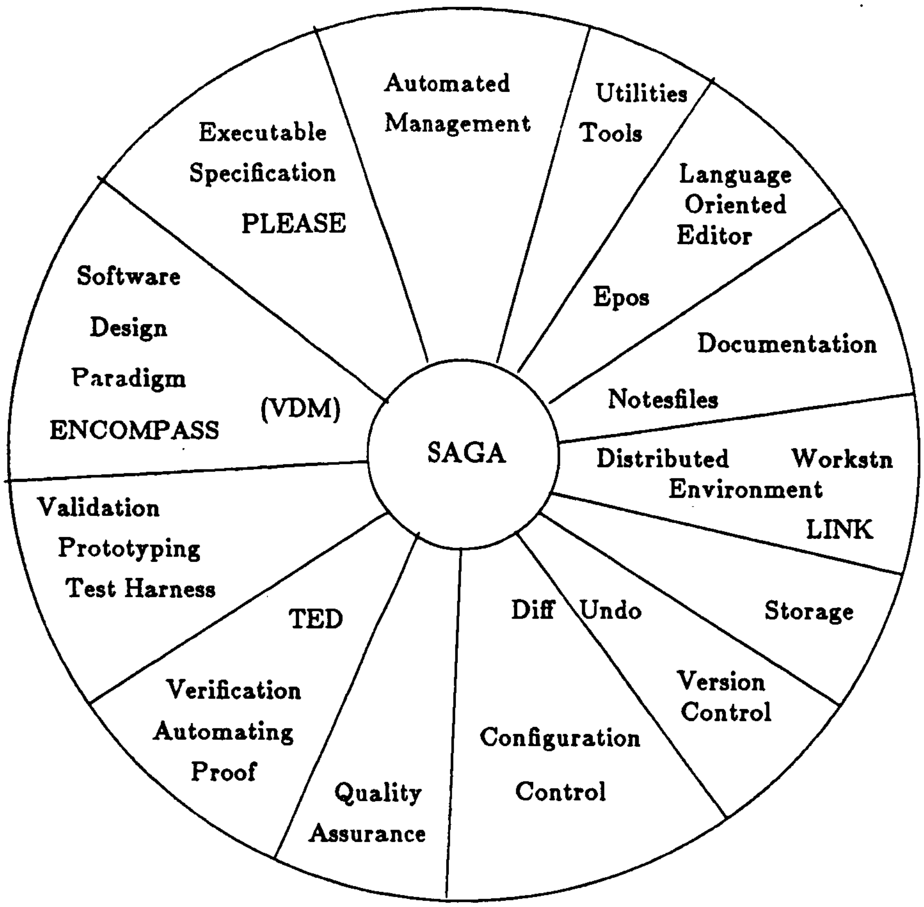


Fig. 1 The SAGA Workbench Components

Guiding the selection of requirements for each of these components, we propose the following principles:

1. A formal basis should be provided for the software environment and its components. This basis should serve to validate the software development paradigms and methodologies used in the environment and also verify the correct operation of the components. The formal basis should allow the specification of such concepts as the model of the software lifecycle in use, the design methodologies, maintenance methods as well as the dependency relationships between products of software development (including requirements specifications, design, tests, documentation, problem tracking, as well as code and versions.)

2. Management by objectives. Each software engineering task should have well–defined goals, participants, and managers. The developers should be able to interact with their managers in refining these goals (Gunther (17)). The task should produce clearly identified software products which may be validated or verified with respect to the goals of the task (Lehman et al (30)) and a method of certifying that the validation or verification has occurred.

3. Automated management aids should provide a project manager with tools which summarize project activity and progress. A project manager should be allowed to review the progress of the project in detail or in summary at any time.

4. Automated development tools should actively support software development and enhance the software developer's abilities. Campbell and Kirslis (8) argue that a software developer must be convinced that a task can be better performed using a tool than without it, irrespective of what other services the tool might provide.

5. Automated quality control tools should permit inspections and audits of the derivation of any software product. This should include examination of any certification process, audits of the software development process, and analyses of the project management. Tools should also support the verification that a software product or development process meets appropriate acceptance criteria and that the configuration management system is kept consistent and up to date.

Many of the principles require further research. In the following sections, we discuss the state of our current research in applying these principles to the construction of software systems.

3. Configuration Management System

The configuration management system is responsible for maintaining the consistency of, integrity of and relationships between the products of software development. In the SAGA project, Terwilliger and Campbell (41) model the configuration management system using a graph in which the nodes represent uniquely named entities or uniquely named collections of entities and the arcs represent relationships between entities. Layers within the graph represent different abstract properties of the software products. The graph also represents the organization of the software products into separate concerns.

The configuration system for ENCOMPASS can be decomposed by organizational relationships into vertical and horizontal structures. The vertical structures form a hierarchy. For example, within a software development *project*, the configuration may be structured into *subsystems*. These, in turn, are decomposed into *modules* which are decomposed into *compilation units*.

The horizontal structures represent attributes of the hierarchy. Thus, each project, subsystem, module, and unit may have an attribute for documentation, version information, requirements specification, shared definitions, architectural design, detailed design, code, binaries, linked binaries, test cases, procedures for generating executable binaries, listings, reports, authors, managers, time and tool certification stamps, development histories, and concurrency control locks. Interattribute relationships specify design, compilation and version dependencies. Depending upon the granularity of the entities, the graph can be represented by the UNIX directory structure, by symbolic links, or by databases. For example, in ENCOMPASS the vertical structure is stored using the UNIX directory structure. Shared definitions are represented by symbolic links. A database at each level in the vertical structure is being built to provide data dictionary capabilities and author manager relations.

Abstractions of the collection of software products are provided by *views*. The "base view" is a complete collection of the software products and other views. A "view" is a layer in the graph which represents a particular abstract property or concern. For example, a "functional test" view might represent the system as a collection of functional specifications, object code, test programs and test data. Other examples of views include a single version abstraction of a system that has many concurrent versions, documentation, and the work of a particular developer.

Fig. 3 shows an example tree traversal program stored in an ENCOMPASS configuration management system (Kirslis et al (26)). It shows a base view, which includes all the details of the software, and a test view, which is a projection onto the base view that abstracts some of the details of the base view and supports the testing of the software. Not all the dependencies and details are shown. The program is presented as a subsystem containing four modules, preorder, stack, tree, and item. Each module contains entities including a makefile (Feldman (14)), specification, body or source code, compiled object code, executable program, test specifications, test body, test makefile, compiled test object, executable test

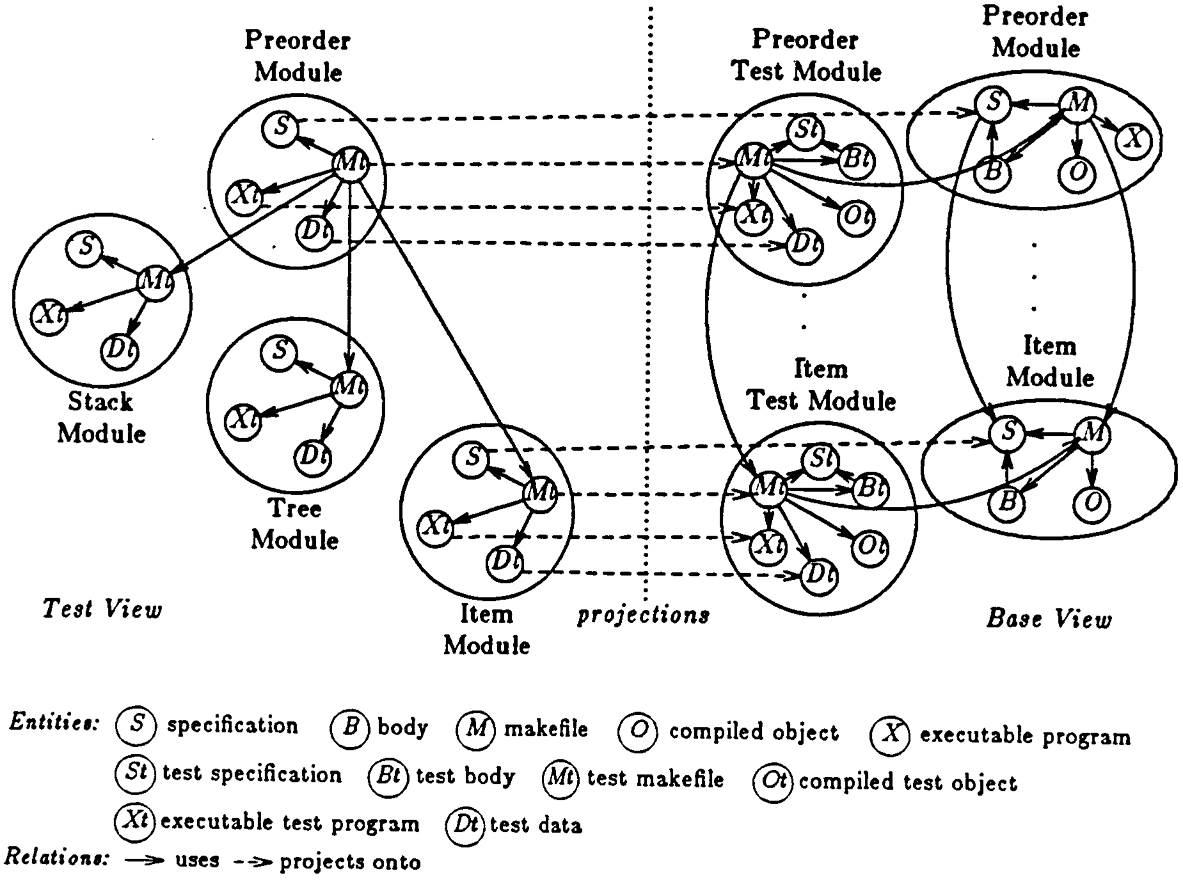

Preorder Module
S
Mt
Xt
Dt
Stack Module
S
Mt
Xt
Dt
Tree Module
S
Mt
Xt
Dt
Item Module
S
Mt
Xt
Dt
Test View
projections
Preorder Test Module
St
Mt
Bt
Xt
Ot
Dt
Item Test Module
St
Mt
Bt
Xt
Ot
Dt
Preorder Module
S
M
X
B
O
Item Module
S
M
B
O
Base View
Entities: S specification B body M makefile O compiled object X executable program
St test specification Bt test body Mt test makefile Ot compiled test object
Xt executable test program Dt test data
Relations: —> uses --> projects onto

Fig. 3 A view of the preorder program

program, and test data. Only one type of relationship is shown, the *uses* relationship, which associates an entity with another entity if the former entity references the latter one. Each "uses" relationship should be accompanied by a "used by" relationship, not shown in the figure, which is simply the inverse of of the "uses" relationship, and which permits the references to a module/entity to be determined from that module/entity. Each body within a module references its own specification. The body of *preorder* references the specifications in the other modules. The makefile for each module references the specification and body to be compiled, and the compiled object which will be produced. In addition, the makefile in the preorder module also references the makefiles and objects in the other modules, since it needs these in order to produce an executable program.

A number of benefits are realized if this dependency graph is stored in machine accessible form and if the software tools in use are adapted to refer to the graph. A data retrieval tool can provide information about the hierarchical structure of the program. For a given module, the tool can show its dependencies with respect to other modules.

An editor, adapted to use this graph, can permit a programmer to specify a routine, module, or program to edit. If the programmer specifies a module, that module becomes the locus at the beginning of the editing session. The programmer edits within the context of that layer of abstraction. Only the implementation details of the module are important. The programmer can find and display other modules, routines, or programs which use this module. These references may be checked easily to determine how a change in the current module will affect them. Similarly, other modules which this module references may be located easily and displayed.

Compilation tools, which access the dependency graph, can support automatic, incremental recompilation on a module by module basis. For example since the body of *preorder* depends on the specification for *stack*, if the specification for *stack* has been changed since the time *preorder* was last compiled, then *preorder* will be recompiled. A compilation tool can use the dependency graph to resolve the dependencies at compile time and access all files needed to perform a compilation[1].

Test tools can use the dependency graph to provide incremental, hierarchical testing for modular programs. A test suite and driver may be associated with each module. A program can then be incrementally tested in a bottom up manner, that is, all modules referenced by module A will be tested before module A is tested. If any of the referenced modules fail their tests then the system can print an appropriate message and terminate the testing session. If the test driver, test suite, or module has not been changed since the tests were last run, the system can report the previous results without rerunning the tests.

[1] In practice, by using UNIX we can do better than this. By an appropriate implementation of the source dependency information, we can make it appear as though all files needed for a compilation are resident in one place, permitting us to use an existing makefile interpreter program and compiler without modification (26).

Fig. 3 includes a test view which might be used by a quality assurance team to test *preorder* after it has been completed. The test view contains a module corresponding to each code module in the base view. The dashed arrows represent the *projection* relationship which shows the correspondence between entities in the test and base view. Each projection relationship is accompanied by an *abstraction* relationship, not shown in the figure, which is its inverse. Each module in the test view contains the specification of the code module to be tested as well as the makefile, load module, and test data from the corresponding test module in the base view.

4. Software Development Processes

Fairley (13) describes a life-cycle model as the sequence of distinct stages through which a software product passes during its lifetime. There is no single, universally accepted model of the software life-cycle according to Blum (3) and Zave (44). In SAGA, we have investigated several aspects of the software life-cycle.

4.1. Software Design Model

In many models of the life-cycle, a requirements specification of the system to be built is created early in the lifecycle. As the project proceeds, components of the software system are built and *verified* for correctness with respect to this specification. The specification is *validated* when it is shown to satisfy the customers requirements. To help manage the complexity of software design and development, methodologies which combine standard representations, intellectual disciplines, and well-defined techniques have been proposed (Jackson (20), Wirth (42), and Yourdon (43)). In the SAGA project, we are developing a formal model for the development process and using it to study a methodology similar to the Vienna Development method described by Jones (21).

A document describing the function of a software system is called a functional specification (13). Design introduces the algorithms and data structures to implement a functional specification. In this paper, we will argue that there are three separate fundamental issues involved in developing computer-based software design aids. We will assume that the development process consists of a number of refinement steps. The first concern is the design decision to select one refinement step instead of another. Design decisions are difficult to formalize without a better understanding of the development process and the application domain.

The second concern is the documentation and verification of a refinement step or implementation decision. Several researchers have argued the need for rigorous argument or formal verification of a refinement step using proof methods (21). The refinement step can be regarded as a correctness preserving transformation from an abstract program to a more concrete program. Using such an approach, the verification becomes a record of the refinement steps.

The third concern is the development process. We argue that a model for the development process is required in order to reason about different development methodologies and the different methods of verifying refinement steps.

In our model of a development process, a functional specification defines a potentially infinite number of implementations. The development process selects a single implementation from this large set. Each refinement step produces a derived functional specification or "abstract program" which constrains the number of possible implementations. The purpose of the model is to allow a study of incremental program development. Within the framework provided by the model we can compare different development methodologies and investigate subtle problems in a rigorous manner. By separating the development process from the issues involved in performing a refinement step, our approach provides a framework to build tools that support a general notion of a development process and that are independent from particular design methodologies. We hope that the model can also help justify design rules which permit rigorous, but not formal, arguments of correctness by construction.

4.2. Executable Specifications

A major problem arising in the design of software is the accurate determination of the function that the software is to perform. The users of the system being constructed may not really know what they want and they may be unable to communicate their desires to the development team. If a functional specification is in a formal notation, it may be an ineffective medium for communication with the customers, but natural language specifications are notoriously ambiguous and incomplete.

Functional specifications may be introduced as part of the design process (perhaps describing the elements of an abstract program) and should help document the design process as well as enhance the designer's understanding of the design. If a formal notation is used for such specifications, a designer may not be sufficiently well-motivated to document his design with a specification because it does not directly contribute towards the act of creating a program. However, a natural language specification may be too imprecise.

Prototyping (Kruchten et al (28)) and the use of executable specification languages (Goguen and Meseguer (16), Kamin et al (22), Zave (44), (Kemmerer(23)) have been suggested as partial solutions to these problems. Providing the customers with prototypes for experimentation and evaluation may increase communication between customers and developers and enhance the validation process. Executable specifications used in the design process provide stubs that allow experimental evaluation of the algorithms and data structures of a program being developed without requiring the program's completion.

Terwilliger and Campbell (41) describe the design of an executable specification language called PLEASE for use in the SAGA Project. By providing executable programs early in the development process, errors in the specification may be discovered before the internal structure of the system has been defined. We believe that this approach will enhance the software development process. A methodology for using executable specification languages in the software lifecycle is being examined as part of ENCOMPASS (41).

4.3. An Executable Specification Design Method

ENCOMPASS supports program development by successive refinement using a similar approach to that of the Vienna Development Method (Jones (21), Shaw et al (39)). In this method, programs are first specified in a language combining elements from conventional programming languages and mathematics. These *abstract programs* are then incrementally refined into programs in an implementation language. The refinements are performed one at a time and each is verified before another is applied. Therefore, the final program produced by the development correctly implements the original abstract program. The ENCOMPASS software development paradigm is shown in Fig. 4.1.

Terwilliger and Campbell (40) describe how abstract programs may be written in PLEASE and refined into the implementation language Path Pascal (Campbell and Kolstad (9)). In PLEASE, a procedure or function may be specified with pre– and post–conditions written in predicate logic. Similarly, an abstract data type may be specified using an invariant. PLEASE specifications

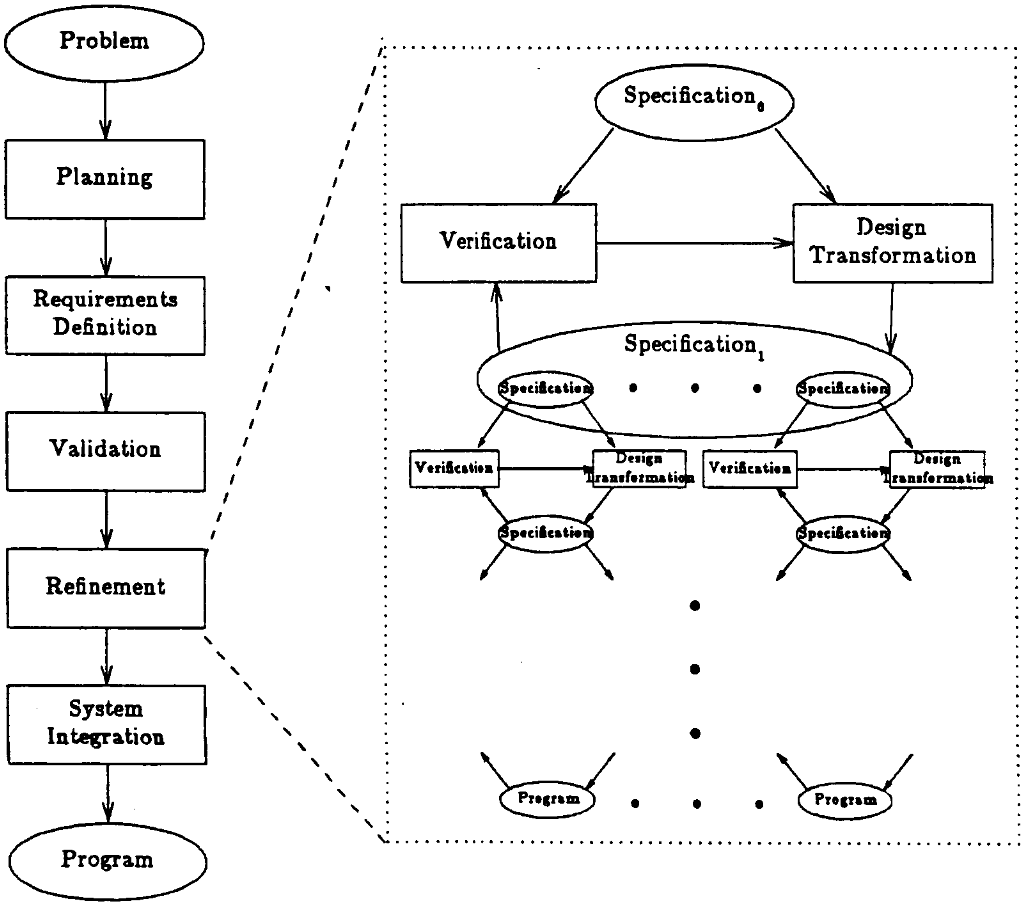


Fig. 4.1 The ENCOMPASS Software Development Paradigm

may be used to argue correctness. They also may be transformed into prototypes which use Prolog (Clocksin and Mellish (11)) to "execute" pre– and post–conditions. These prototypes may interact with other modules written in conventional languages.

Lehman et al (30) propose that software development may be viewed as a sequence of transformations between specifications written at different *linguistic levels*. Neighbors (33) describes the construction of a system that supports a similar development methodology. ENCOMPASS supports this view of software development by allowing abstract, predicate logic based definitions of data types or routines to be transformed into successively more concrete realizations. The use of executable specifications allows prototypes for two or more linguistic levels to be executed using the same input data and the results compared for the purposes of verification or debugging. An executable specification provides a framework for the rigorous development of programs in a manner similar to (21). Although detailed formal proofs are not required at every step, the framework is present so that they may be constructed if necessary. (However, it is our experience that many problems arise in changing a rigorous argument into a mathematical proof.)

Fig. 4.2 shows an example of a PLEASE specification for a SORT program. The specification is given in terms of a pre–condition and post–condition for sort. Two predicates, "permutation" and "sorted", are used by the post–conditions. Terwilliger and Campbell (40) describe the translation of the specification into Prolog. In general, the translation of arbitrary specifications into executable programs is difficult. Theoretically, the guaranteed automatic production of a terminating program from an arbitrary specification written in first order logic is not possible. One aspect of our future research will be to study what. is possible in practice.

The specification may be used to validate the user requirements for *sort* or they may be used as a test oracle for the subsequent refinements of *sort* (40). In addition, using rules similar to those provided in the Vienna Definition Method (21), an argument for correctness can be constructed for the sort program based on the refinement steps used to build the program. Examples of some of the rules are given in (40).

4.4. Software Management Model

A management model for software development must identify, control, and record the development process. A management model can be based on a *trace* of the activities within the project. Such a trace can be used to understand the meaning of management in a similar manner to the use of traces in defining the meaning of a programming language (Campbell and Lauer (6)). The trace represents a complete history of all significant events that have occurred in the project. Projections from the trace permit identification of particular sequences of activities. Control can be expressed in terms of the valid continuations of a partially completed trace.

```
program sort (input, output);

#include "integer_list.spec"

var  input_list, output_list: integer_list;

predicate permutation (list1, list2: integer_list);
     var  front, back: integer_list;
     begin
          (list1 = empty_list) and (list2 = empty_list)
               or
          (list1 = front || <hd (list2) > || back) and
          permutation (front || back, tl(list2))
     end;

predicate sorted (l: integer_list);
     var  x: integer;
     begin
          (l = empty_list)
               or
          forall (x | member (x, tl(l)), x >= hd(l)) and
          sorted(tl(l))
     end;

pre_condition;
     begin
          text_to_integer_list(input) <> integer_list_error
     end;
post_condition;
     begin
          (input_list = text_to_integer_list(input)) and
          permutation(input_list, output_list) and
          sorted (output_list) and
          (output_list = text_to_integer_list (output'))
     end;

     begin
     end;
```

Fig. 4.2 A specification of a sort program

In ENCOMPASS, we are implementing a limited set of management functions to record, monitor, initiate activities, and inhibit inappropriate activities. Instead of using a detailed trace model of management, we have adopted a practical approach based on the larger granularity provided by milestones. We structure the management model of a software project into units of work which create well–defined products (Gunther (17)). The management objectives for each activity must define the pre–conditions under which the activity may occur, acceptance criteria for the products produced by the activity, and a procedure for evaluating whether the acceptance criteria have been met. The acceptance criteria evaluation procedure may be invoked at any time during the activity and produces status reports of the software product. Satisfaction of the pre–condition and the acceptance criteria provide "milestone" events. A record of the occurrence of these milestones is stored in a management log. Accounting information may be associated with each unit of work. The log and accounting information can be used to generate reports and, when used with other information such as PERT schedules, to control the project.

Work units form a hierarchical structure. The reports generated by one work unit may satisfy a pre-condition or acceptance criteria for another activity.

In ENCOMPASS, management monitoring, assessment, and control is implemented using makefiles, predicate evaluation, and Notesfiles. Periodic execution of makefiles are used to implement automated management and assessment of the project. The makefiles incorporate automatic evaluation of work unit pre-conditions, the creation of work units, the invocation of acceptance criteria evaluation procedures, and the creation of milestones when a pre-condition or acceptance criteria is met. The Notesfiles (Essick (12)) record milestones and reports and propagate traceable management information to developers and managers.

For example, consider the implementation of a problem tracking system. Bug reports are mailed to the "problem definition" notesfile. They can be created by a user, a developer, or by the execution of a program at a remote or local site. Debugging facilities within a software product can automatically report an internal error by invoking the Notesfiles mailer. Similarly, development tools may report errors, for example the test harness may automatically report the detection of an error.

The problem definition notesfile records the site, author, time, address, and complaint. The "problem tracking manager" may set a timeout on the notesfile sequencer which specifies the acceptable interval within which a "problem definition analyst" should respond to the note. After expiration of the timeout, the notesfile automatically notifies the manager using a "management" notesfile.

The problem definition analyst may respond to the note in several ways. A response may be created that identifies the problem as a user error. Alternatively, the analyst may create a request in a maintenance programmer's "activity" Notesfile to consider possible solutions to the problem.

The acceptance criteria for the programmers task is to assess the practical design issues involved in correcting the problem, provide a cost estimate of the work involved, and produce an implementation plan. While the programmer is considering possible solutions, the problem definition analyst or problem tracking manager may request progress reports. These reports may consist of any milestones accomplished and preliminary documentation generated.

When the problem definition analyst is satisfied that the acceptance criteria for the task have been satisfied, he may then submit a change request note to the project change request board (Fairley (13)). This milestone and the timetable of the change request board determine the conditions under which a meeting of the board is scheduled.

4.5. Project Libraries.

Horowitz and Munson (19) suggest that the *reuse* of software can significantly reduce the cost of program development, and systems which contain libraries of previously coded modules and/or a number of standard designs for programs have been proposed by Lanergan and Grasso (29) and Matsumoto (31). In ENCOMPASS, any software component or group of components can be saved for later reuse. In addition to source and object code, documentation, formal specifications, proofs of correctness, test data and test results can all be stored in the central library and later retrieved. The library can support a number of projects, both accepting and supplying components for reuse in all phases of development. The structure and organization of the library is shown in Fig. 4.3.

A programmer, developing code, will use a view of the project library to access shared code and data, test cases, specifications, design, and other products of the project. The workspace extends the view with local copies of code that are being modified and with new code. Eventually, the programmer will submit his workspace to be placed under the configuration management of the library. The configuration management of the workspace must be consistent with that of the library and acceptance criteria may be applied to the software products before the library is updated. An integration test may be required as a pre-condition to a library update performed on a working version of the software system. A

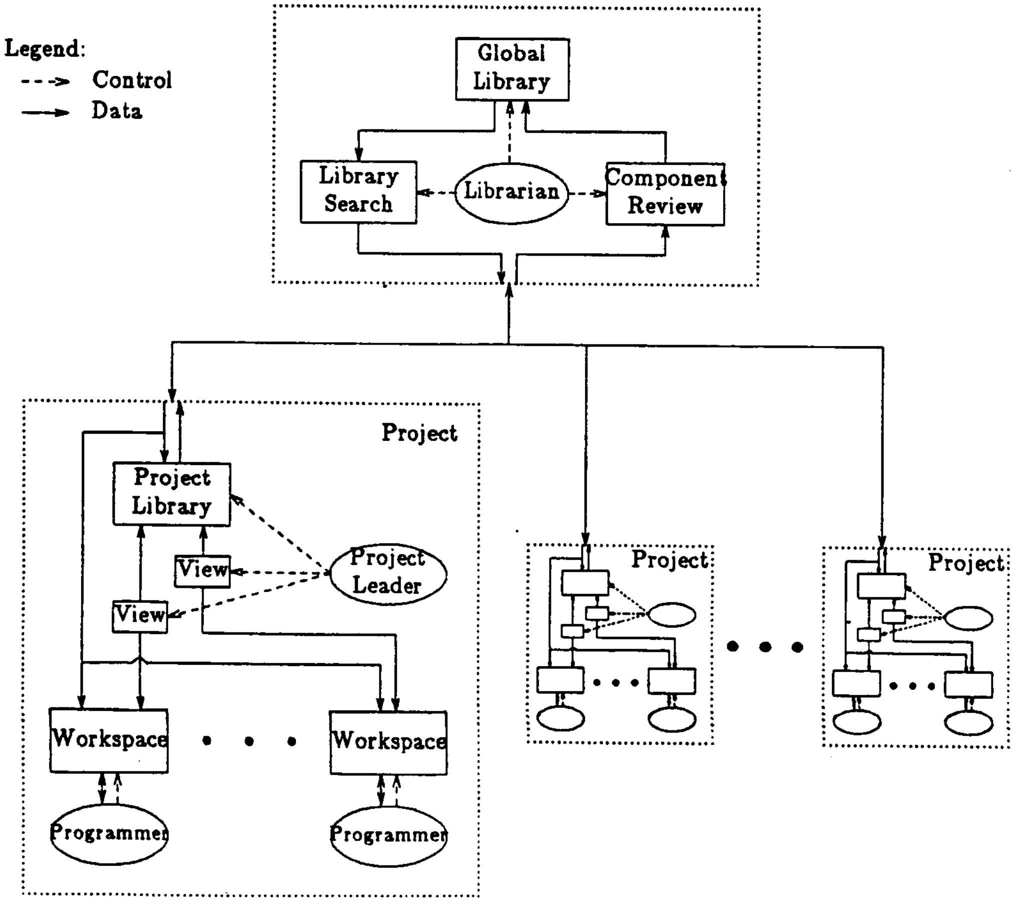


Fig. 4.3 The ENCOMPASS Library Structure

system acceptance test may be required as a pre-condition to a library update performed on a stable version of the software system. The project leader has responsibility for correct library operation including the view and workspace creation and workspace integration.

5. Software Tools

A large number of software tools are required to implement computer-aided software development environment. Rather than build a large number of specialized tools, in SAGA we have chosen to build a small number of tools that can be specialized for specific purposes. Examples of such "generic" facilities are the Notesfiles system (Essick (12), the SAGA language-oriented editor (Campbell and Kirslis (8), the symbol table manager (Richards (35)), tree editor (Hammerslag et al (18)) and the attribute evaluation schemes used for semantic evaluation (Beshers and Campbell (2)). The Notesfile system is used for documentation and management. The editor can be specialized to edit many different languages and specialized editors have been built for Pascal, ADA, PLEASE, and C. In addition to their number, the software tools in a software environment must also have other properties.

Software development environments need to be maintainable for the duration that they are used to support a software development project (Campbell and Lauer (6)). The software tools in the environment must accommodate change and modification of the environment over the lifetime of the software project. In many applications, the software support environment and its tools must be maintained for the duration of the maintenance of the software product; in the case of an embedded system like the space station software system this might be for twenty or more years. Changes in hardware technology may require the environment to be ported to new computer systems. New tools may be integrated into the environment. A solution to the problem of maintaining the environment and tools for a long period of time is to design them as part of an "open architecture".

In such an open architecture, modular tools are built which use standard interface to access other tools. The approach we have adopted in SAGA is to use the UNIX operating system to define a standard interface. UNIX processes become the mechanism to modularize the tools. New software tools built for UNIX can be integrated into the environment and UNIX provides a method of migrating the environment from one computer technology to the next. UNIX UNITED (Brownbridge et al (5)), LINK (Russo (36)) and other distributed UNIX systems permit the support of software development environments on networks of workstations.

Software engineering studies reported by Bauer et al (1) suggest providing the user with a high-level interface which reflects the levels of abstractions in programming. By allowing the user to phrase commands in terms of high-level concepts, the quality of the user's interaction with the computer can be improved. Less time is needed to accomplish a given task, and fewer operations mean fewer errors made during the software development process. Since users spend a large amount of their time using editors, Scofield (38) proposed using an editor as an appropriate program in which to implement a high-level interface.

In the remainder of this section we discuss some of the SAGA tools that have been developed based on these ideas.

5.1. Language-Oriented Editor

Language-oriented editors supply a high-level interface for software development tools (Campbell and Richards (7), Campbell and Kirslis (8)). Since the editor is the primary tool for constructing software products, enhancing the editor with features that aid the editing of specific specification languages and programming languages should be beneficial to the development process. The editors can have semantic and syntactic oriented editing commands and may help the program development process by preventing or providing immediate diagnosis of syntactic and semantic errors in the program text.

Two different approaches may be used to construct a language-oriented editor: the generator, or "template", approach and the recognizer approach. The SAGA project has developed a recognizer-based editor. The editor incorporates an LALR(1) parser augmented for the interactive environment with incremental parsing techniques (Kirslis (25), Ghezzi and Mandrioli (15)). An editor generator (25) allows editors to be generated for a particular language.

The SAGA project has demonstrated (25) that the recognizer approach is a practical basis for constructing language-oriented editors and has several advantages:

1. The recognition approach can be applied consistently to the editing of the lexical, syntactic, and semantic components of the language. This simplifies providing uniform editing commands that manipulate lexical, syntactic, and semantic entities. Template editors are tedious to use if they do not use a recognizer to enter expressions, variable names, and constants. An editing command will differ in operation depending upon whether an entity is recognized or generated.

2. The recognition approach permits arbitrary editing operations on the program. Rectangular blocks of characters may be copied from one part of a screen of program text to another as when initial assignments are being made to array elements. Global string substitutions may be made. Program code may be commented out and comments may be changed into program code. The generator approach cannot handle arbitrary editing commands unless the resulting edit generates text which is reparsed into a form suitable for the editor. Problems occur when such an edit creates a lexical or syntactic error.

3. Program editing during the debugging and maintenance phases of a project will invariably require transforming the program through a number of illegal lexical, syntactic, and semantic constructs. Many editors using the generator approach expressly forbid the creation of incorrect programs. However, the recognition approach permits illegal programs which may have many incorrect semantic, syntactic and lexical errors. The errors may be introduced in any order and may be removed in any order. When a lexical or

syntactic error is introduced, the editor can mark the discontinuity in the corresponding token or parse tree. When an error is removed, the incremental parsing technique will examine the surrounding context of the change only as far as it is necessary to determine that the change results in a lexically and syntactically correct program fragment. The parse tree will be repaired in the local context of the change.

4. The recognition approach allows a lexical or syntactic entity such as a Pascal **while** loop to be incrementally changed into a **repeat** loop whereas the generator approach must include a transformation rule to support such a modification. Although it is simple to generate a set of useful transformation rules, it is not clear whether it is possible to generate all useful transformations of this form.

5. The recognition approach uses existing compiler generation and parsing techniques without major alteration. If standard compiler generation and parsing tools are used, then many existing specifications of the lexical, syntactic, and semantic components of a programming language can be used directly by an editor generator facility to produce corresponding language-oriented editors.

6. Semantic analysis is performed in most language-oriented editors using recognition techniques that extend those developed for compilers. For example, the attribute evaluation schemes proposed by Knuth (27) have been used directly or encoded in a procedural manner to provide semantic evaluation of edited programming languages (Reps et al (34)).

The SAGA editor has been used with various semantic evaluation methods. Beshers and Campbell (2) describe an approach combining the editor with right regular expression grammars, attributed grammars, and maintained and constructor attributes. This method was proposed to overcome some of the overhead that occurs in direct attribute evaluation schemes. A SAGA editor for a subset of Pascal has been built that incrementally compiles Pascal programs using more conventional techniques (Kimball (24)).

One of the major problems in building language-oriented editors is that they provide an unfamiliar interface to the user. To overcome this problem, a new version of the SAGA editor is being constructed using an EMACS editor front end.

5.2. Notesfiles

An important software development tool for any project is a means to record, document and retrieve information. Such a tool can be used to support technical discussions, product reviews, problem tracking, agendas and minutes, grievances, design and specification documentation, lists of work to be done, appointments, news and mail. The SAGA Notesfiles system (Essick, (12)) has been in use for some time to support all these functions within the SAGA project.

The Notesfiles system is a distributed project information base constructed for SAGA on the UNIX operating system. A file of notes can be maintained across a network of heterogeneous machines. Each file of notes has a topic; each notesfile has a title. A sequence of notes is associated with each notesfile. Notes and responses may be exchanged between separate notesfiles. Notes and responses are documented with their authors and times of creation. Updates to the notes and responses are transmitted among networked systems to maintain consistency. Notesfiles use the standard electronic mail facility to facilitate the updates. A library and standard interface permits any user program to submit a note or response to a notesfile. This library has been particularly useful in the construction of automatic logging and error reporting facilities in test harnesses and "beta test" uses of SAGA code.

6. Conclusion

One approach to improving the productivity of large scale software development is to construct software systems that support the software development process. The design of such systems requires an understanding of the principles underlying the software development and maintenance process as well as methods and technologies for building complex design aids. We argue that the experimental research required to build such environments should be based on formal models of the software development process. Much research is required to produce both the appropriate formal models and the methods and techniques of implementation and environment.

In the SAGA Project, we have been studying the construction of an environment to support the software development and maintenance. In this paper, we have outlined some of the models being developed in association with the construction of an experimental environment called ENCOMPASS.

Acknowledgements. The ideas presented in this paper have been summarized from the work of the SAGA Project team. In particular, I would like to thank Bob Terwilliger for his work on PLEASE and ENCOMPASS, Peter Kirslis for building the first SAGA editor, and Carol Beckman-Davies, Leonora Benzinger, George Beshers, David Hammerslag, Dan Laliberte, Craig Neth, Hal Render and Paul Richards for their various contributions to the research described in this paper.

7. References.

1. Bauer, F., J. Dennis, G. Goos, C. Gotlieb, R. Graham, M. Griffiths, H. Helms, B. Morton, P. Poole, D. Tsichritzis, and W. Waite, 1977, **Software Engineering — An Advanced Course,** F. Bauer, Ed., Springer Verlag, New York.

2. Beshers, G. M. and R. H. Campbell, 1985, *Maintained and Constructor Attributes*, **Proc. of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments**, pp. 34-42.

3. Blum, B. I., 1982, *The Life-Cycle - A Debate Over Alternative Models*, **Software Engineering Notes**, vol. 7, pp. 18-20.

4. Boehm, B., 1981, *Software Engineering Economics,* **Prentice Hall**, Englewood Cliffs, NJ..

5. Brownbridge, D. R., L. F. Marshall and B. Randell, 1982, *The Newcastle Connection or UNIXes of the World Unite!*, **Software - Practice and Experience**, V. 12, pp.

1147–1162.

6. Campbell, R. H. and P. E. Lauer, 1984, *RECIPE: Requirements for an Evolutionary Computer-based Information Processing Environment*, **Proc. of the IEEE Software Process Workshop**, pp. 67–76.

7. Campbell, R. H. and Paul G. Richards, 1981, *SAGA: A system to automate the management of software production*, **Proc. of the National Computer Conference**, pp. 231–234.

8. Campbell, R. H. and P. A. Kirslis, 1984, *The SAGA Project: A System for Software Development*, **Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments**, pp. 73–80.

9. Campbell, R. H. and R. B. Kolstad, 1979, *Path Expressions in Pascal*, **Proceedings of the Fourth International Conference on Software Engineering.**

10. Campbell, R. H., C. S. Beckman, L. Benzinger, G. Beshers, D. Hammerslag, J. Kimball, P. A. Kirslis, H. Render, P. Richards, R. Terwilliger, 1985, *SAGA*, Mid–Year Report, Dept. of Comp. Sci., University of Illinois.

11. Clocksin, W. F. and C. S. Mellish, 1981, **Programming in Prolog**. Springer–Verlag, New York.

12. Essick, Raymond B., IV., 1984, *Notesfiles: A Unix Communication Tool*, M.S. Thesis, Dept. Comp. Sci., University of Illinois at Urbana–Champaign.

13. Fairley, Richard, 1985, **Software Engineering Concepts**. McGraw–Hill, New York.

14. Feldman, S. I., 1979, *Make – A Program for Maintaining Computer Programs,* **Software – Practice and Experience,** Vol. 9, No. 4, pp. 255–265.

15. Ghezzi, C. and D. Mandrioli, 1980, *Augmenting Parsers to Support Incrementality,* **Journal of the ACM,** Vol. 27, No. 3.

16. Goguen, J. and J. Meseguer, 1982, *Rapid Prototyping in the OBJ Executable Specification Language*, **Software Engineering Notes**, vol. 7, no. 5, pp. 75–84.

17. Gunther, R., 1978, **Management Methodology for Software Product Engineering,** Wiley Interscience, New York.

18. Hammerslag, D. H., S. N. Kamin and R. H. Campbell, 1985, *Tree-Oriented Interactive Processing with an Application to Theorem-Proving.* **Proc. of the Second ACM/IEEE Conference on Software Development Tools, Techniques, and Alternatives.**

19. Horowitz, E. and J. B. Munson, 1984, *An Expansive view of Reusable Software,* **IEEE Trans. on Software Engineering,** Vol. 10, No. 5.

20. Jackson, M., 1983, **System Development**, Prentice–Hall, Englewood Cliffs, N.J..

21. Jones, C., 1980, **Software Development: A Rigorous Approach**, Prentice–Hall International, Inc., London.

22. Kamin, S. N., S. Jefferson and M. Archer, 1983, *The Role of Executable Specifications: The FASE System*, **Proc. of the IEEE Symposium on Application and Assessment of Automated Tools for Software Development.**

23. Kemmerer, R. A., 1985, *Testing Formal Specifications to Detect Design Errors*, **IEEE Transactions on Software Engineering**, vol. SE–11, no. 1, pp. 32–43.

24. Kimball, J., 1985, *PCG: A Prototype Incremental Compilation Facility for the SAGA Environment*, **M.S. Thesis**, Dept. Comp. Sci., University of Illinois at Urbana-Champaign.

25. Kirslis, P. A., 1986, *The SAGA Editor: A Language-Oriented Editor Based on an Incremental LR(1) Parser*, Ph. D. Dissertation, Dept. Comp. Sci., University of Illinois at Urbana-Champaign.

26. Kirslis, P. A., R. B. Terwilliger and R. H. Campbell, 1985, *The SAGA Approach to Large Program Development in an Integrated Modular Environment*, **Proceedings of the GTE Workshop on Software Engineering Environments for Programming-in-the-Large.**

27. Knuth, D. E., 1968, *Semantics of context-free languages,* **Math. Syst. Theory**, Vol. 2, No. 2.

28. Kruchten, P., E. Schonberg and J. Schwartz, 1984, *Software Prototyping Using the SETL Programming Language.* **IEEE Software**, vol. 1, no. 4, pp. 66–75.

29. Lanergan, R. G. and C. A. Grasso, 1984, *Software Engineering with Reusable Designs and Code,* **IEEE Trans. on Software Engineering**, Vol. 10, No. 5.

30. Lehman, M. M., V. Stenning and W. M. Turski, 1984, *Another Look at Software Design Methodology,* **Software Engineering Notes**, vol. 9, no. 2, pp. 38–53.

31. Matsumoto, Y., 1984, *Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels,* **IEEE Trans. on Software Engineering,** Vol. 10, No. 5.

32. Myers, G. J., 1978, *A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections,* **Comm. ACM**, pp. 760–767.

33. Neighbors, J. M., 1984, *The Draco Approach to Constructing Software from Reusable Components,* **IEEE Transactions on Software Engineering,** vol. SE–10, no. 5, pp. 564–574.

34. Reps, T., Teitelbaum, T., and Demers, A., 1983, *Incremental Context-Dependent Analysis for Language-Based Editors,* **ACM Transactions on Programming Languages and Systems,** Vol. 5, No. 3.

35. Richards, P., 1984, *A Prototype Symbol Table Manager for the SAGA Environment,* Master's Thesis, Dept. Comp. Sci., University of Illinois at Urbana-Champaign.

36. Russo, V. F., 1985, *ILINK: Illinois Loadable InterUNIX Networked Kernel,* M.S. thesis, University of Illinois, Urbana, Il 61801.

37. Sackman, H., et al., 1968, *Exploratory Experimental Studies Comparing Online and Offline Programming Performance,* **Comm. ACM,** vol. 11, no. 1.

38. Scofield, Alan, 1985, *Editing as a Paradigm for User Interaction*, Technical Report 85–08–10, Dept. Comp. Sci., Univ. of Washington, Seattle.

39. Shaw, R. C., P. N. Hudson and N. W. Davis, 1984, *Introduction of A Formal Technique into a Software Development Environment (Early Observations)*, **Software Engineering Notes**, vol. 9, no. 2, pp. 54–79.

40. Terwilliger, R. B. and R. H. Campbell, 1986, *PLEASE: Predicate Logic based ExecutAble SpEcifications,* **Proc. 1986 ACM Computer Science Conference.**

41. Terwilliger, R. B. and R. H. Campbell, 1986, *ENCOMPASS: a SAGA Based Environment for the Composition of Programs and Specifications*, **Proceedings of the 19th Hawaii International Conference on System Sciences.**

42. Wirth, N., 1971, *Program Development by Stepwise Refinement*, **Communications of the ACM**, vol. 14, no. 4, pp. 221–227.

43. Yourdon, E. and L. L. Constantine, 1979, **Structured Design**, Prentice-Hall, Englewood Cliffs, N.J..

44. Zave, P., 1984, *The Operational Versus the Conventional Approach to Software Development*, **Communications of the ACM**, vol. 27, no. 2, pp. 104–118.

SAGA Project Mid-Year Report 1986 Appendix E

The SAGA Editor: A Language-Oriented Editor Based on Incremental LR(1) Parser

Peter Andre Christopher Kirslis

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois

December, 1985

THE SAGA EDITOR: A LANGUAGE-ORIENTED EDITOR BASED ON AN INCREMENTAL LR(1) PARSER

BY

PETER ANDRÉ CHRISTOPHER KIRSLIS

A.B., Harvard University, 1975
M.S., University of Illinois, 1977

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1986

Urbana, Illinois

THE SAGA EDITOR: A LANGUAGE-ORIENTED EDITOR BASED ON AN INCREMENTAL LR(1) PARSER

Peter Andre Christopher Kirslis, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1986
Dr. Roy H. Campbell, Adviser

The research described in this dissertation supports the thesis that a language-oriented editor for *full* programming languages, and other languages specifiable with context-free LR(1) grammars, can be based upon an *incremental LR(1) parser* employing incremental analysis techniques. The resulting editor is flexible, supporting a higher-level command interface which includes structure-oriented commands involving tokens and sub-trees, while retaining common text editing commands which operate on arbitrary groups of characters and lines. This editor can be used to develop practical programs which incorporate software engineering principles concerning the design and construction of software systems. In this dissertation, an incremental parsing algorithm suitable for use with an interactive editor is developed. A new solution to the handling of comments in syntax trees is proposed, and an error-recovery algorithm which permits editing of the parse tree in the midst of syntax errors is presented. The resulting editor, its commands, and environment are described. The editor can be retargeted to other languages, and can use any parser-generating system which can meet its interface. A prototype editor which employs these algorithms has been implemented as a part of the SAGA project as a demonstration of the practicality and flexibility of this approach; this editor has been in experimental use during the past couple of years at the University of Illinois at Urbana-Champaign.

ACKNOWLEDGEMENTS

I would like to thank my adviser, Prof. Roy Campbell, for his advice and support throughout this research. Our many conversations greatly contributed to the final form of the SAGA editor. I would also like to thank the members of my committee, Prof. M. Dennis Mickunas, Prof. Bill Gear, Prof. Mehdi Harandi and Prof. Sam Kamin for their interest in my work. The many comments from the members of the SAGA group helped me develop the editor into its present form.

I received many helpful comments during the proofreading of the dissertation manuscript from Prof. M. Dennis Mickunas, Leonora Benzinger, Judy Grass, Simon Kaplan, Dan LaLiberte, and Hal Render, for which I am grateful. This research was supported under NASA grant NAG 1–138; I appreciate the feedback that I have received from Ed Senn of NASA Langley, C. Michael Holloway, and others at NASA who took an interest in this work.

PREFACE

This dissertation details an alternate approach to the syntax–directed, template driven, language–sensitive editors which are receiving much attention at present. Rather than displaying the internal tree structure of the program being edited, our editor displays the program in text form on the terminal screen, no non–terminals appear. Instead of restricting the editing commands to structure–only commands at certain points in the program, and text–only at other points, our approach permits the use of both structure–oriented commands on tokens and trees, and common text–oriented commands on arbitrary groups of characters and lines, permitting each type of command *anywhere* in the program. The syntax checking provided by the parser provides feedback to the programmer about the correctness of his program as he edits it, without requiring him to always keep the program text syntactically correct or to immediately repair syntax errors which arise. This combination of feedback and flexibility should appeal to experienced programmers, and I believe that this approach to editing is practical and will be favorably received.

An understanding of the SAGA editor and the ideas behind it can be obtained through a reading of Chapters 1, 3, 6, and 8. More in–depth information about the internal structure of the parse tree and the incremental LR(1) parsing algorithm in use can be found in Chapters 4 and 5, although a reading of these chapters is only necessary to gain insight into *how* the editor works. Chapter 5 presents the incremental parsing algorithm in enough detail to guide another implementation of the incremental parser, should one wish to extend the ideas presented here in future work. Chapter 7 describes the generation of new editors, and the parser–generators in use by the SAGA project. Finally, Chapter 2 contains a detailed look at some of the previous work in the area.

A prototype editor has been produced as a demonstration of the feasibility of the ideas presented in this dissertation, and has been in experimental use since 1982 at the University of Illinois. I have enjoyed the time I have spent on this research, and my contacts with other students who have based Master's Theses and class projects upon this editor. I wish the best to the others who will be continuing this work at the University of Illinois, and to any others who may extend the ideas presented here.

Peter A. C. Kirslis

November, 1985

TABLE OF CONTENTS

CHAPTER PAGE

CHAPTER 1

INTRODUCTION

Software complexity and cost are severe problems in software development. To keep costs down, programmer productivity needs to be improved. The more powerful computers available today permit more complex programs to be written which were previously not feasible, and programmers can now better utilize tools to receive more analysis at an earlier point in the software development cycle. However, the existing software tools are not always adequate to manage the large amount of software required in many new projects; new tools are needed. Properly designed tools can improve programmer productivity, if these new hardware resources are put to best use.

Software engineering research addresses many of the problems in software development and offers formal results and insights to the solution of current problems. Results from software engineering [Bauer et. al., 77] suggest providing the user with a high level interface which reflects the levels of abstraction in programming. Since the user can phrase commands in terms of high level concepts, the quality of the user's interaction with the computer can be improved. Less time will be needed to accomplish a given task, and fewer operations mean fewer errors made during the software development process.

Since users spend a large amount of their computer time using editors, an editor is an appropriate program in which to implement a high level interface. The interface can support the concept of structured programming. Exploiting the syntactic and semantic properties of a programming language supports levels of abstraction and allows a programmer to build his program using either bottom-up integration or top-down refinement. Since more computing resources are available, one approach toward providing an interface is to perform additional analysis that up until now was done at a later time or in another program; the user benefits by receiving more directive or diagnostic information much sooner than before.

Parsing theory is well-developed, but until recently has been applied in a static environment in which parsers are run non-interactively and take their input from a file. Parsing techniques must be modified in an interactive language-oriented editor in which input is received incrementally from a user and a large portion of text which has already been parsed may be modified. Initial results concerning the organization and complexity of incremental parsers have appeared [Celentano, 78], [Ghezzi and Mandrioli, 79], [Ghezzi and Mandrioli, 80] that suggest that such methods can be applied in practical interactive environments such as a language-oriented editor, and that reasonable response times can be maintained while performing this increased computation.

1.1. Syntax-Directed vs. Language-Oriented Editing

Language-oriented editors have been proposed to provide a high-level interface for software development tools. Two different approaches may be used to construct language-oriented editors. The *generator* approach (often called the *template approach*) constrains the editing commands so that only valid programs can be developed. The *recognizer* approach supports both normal text editing commands and additional language-oriented commands, employing an incremental LR(1) parser to detect lexical, syntactic and semantic errors in program fragments. The recognizer approach provides a more flexible editing environment for program development and maintenance. This second approach will be presented in this dissertation.

1.1.1. The Generator Approach

To date, much work with language-specific editors has followed the *generator* approach, producing *syntax-directed* editors [Hansen, 70], [Donzeau-Gouge et. al, 75], [Medina-Mora and Feiler, 81], [Teitelbaum and Reps, 81], [Reiss, 84]. Such editors have a particular language structure imposed upon them, resulting in an editor driven by commands which are constrained to follow the specific language structure. The user of such an editor is presented with a program skeleton. He selects language constructs from a menu and places them into pre-determined points in the program display. The method constrains the user's interaction with the editor to operations which produce error-free syntax, although semantic errors are still possible. Some typing is also saved, since the user never types keywords or punctuation.

This approach has proven popular with implementors for several reasons. First, the user-interface is simple; a small set of menu-driven commands permits construction of the tree in a well-defined (error-free) manner. Second, the implementation is straightforward; since the user is not permitted to make syntax errors, no error detection, recovery, or correction code is needed. Third, a set of templates representing the constructs of a language can be constructed without much difficulty. Fourth, the resulting editor is a very nice teaching tool for novice programmers, since at each editing step, the user is channeled to a narrow path with few choices. Users can build programs free of syntax errors more easily than with traditional text editors, which permit syntax errors that may be obscure to the new programmer.

However, this approach is inflexible, and modifications to existing programs can be difficult. In order to replace one construct with another, the sub-trees first must be removed from the template and saved somewhere, then the template deleted, another selected to replace it, and finally, the trees re-inserted. Two examples of modifications which illustrate this difficulty are: the addition of an **else** clause to an **if-then** statement, and the alteration of a statement to a block of two or more statements. It is also not practical to build the program entirely by selection of templates down to the lowest expression level, since the many selections needed become tedious. Therefore, at the lowest levels of the parse tree, expressions are input from the keyboard and parsed, and certain kinds of errors are possible.

Unfortunately, syntax-directed template editors have not been accepted by experienced programmers [Waters, 82]. In fact, one indication of the lack of utility of such editors is that the developers of these editors do not use them themselves in their own program development. In addition, since experienced programmers are not troubled by syntax errors, error repair is a simple and straightforward task; the restrictive editing environment provided by these editors is of no benefit to these programmers. Commands which operate on arbitrary groups of characters or lines are not provided. Comments also cause great difficulty to template editors. They are usually handled by permitting (or requiring) comments at certain places, and prohibiting them anywhere else. When permitted, their placement is often restricted to a certain format, and block copying of combined syntactic structures and comments is difficult.

1.1.2. The Recognizer Approach

The *recognizer* approach employs some type of parser to analyze character strings entered by the user. A recursive descent parser was used by [Wilcox et. al, 76] in an educational system, and a bottom-up parser by [Horton, 81] in his editor; both provided text interfaces to the user, and supported editing operations which manipulated strings of characters. When editing programs using the recognizer approach, the user typically inputs his text in free format; this input is analyzed using an incremental parser and immediate feedback is provided about the correctness of the program. With this approach, it becomes possible to specify editing operations on syntactic and semantic entities such as tokens, sub-trees or items with particular semantic attributes, in addition to operations on arbitrary groups of characters or lines. With a parser in the editor performing incremental compilations on portions of the parse tree, less use of the compiler is required for detecting syntactic and semantic errors; potentially many compilation runs can be saved since a successful compilation will result the first time that the user runs the compiler. Still further improvement results if the editor provides the compiler the parse tree directly, and the compiler selectively recompiles those program fragments which changed during the editing session.

Since a recognizer is used, editing commands can be supported which take the program through intermediate, incorrect states, which facilitates some editing operations such as the insertion of a widely spaced **begin** ... **end** pair. It also permits the editor to provide the user with program specific information in the form of valid continuations of a parse, which can be calculated by the recognizer given the current parse state and parse stack context, so error repair is simplified in cases when the error is not immediately obvious.

The productions of the grammar used to specify the language are user–transparent; that is, none of the editing commands, error diagnostics, or development aids are based upon information that is not directly representable as elements of the *concrete* syntax. The user sees a text–oriented display of his program similar to the screen–oriented text editors available today; no non–terminal symbols appear, and it is not necessary to become acquainted with the internal grammatical structure of the production rules used to describe the language in order to effectively use the editor.

The resulting editor is flexible, incorporating an incremental LR(1) parser with incremental analysis techniques to analyze the user's input and provide immediate feedback about its correctness. The editor supports a higher–level command interface, which includes structure–oriented commands involving tokens and sub–trees, and retains common text editing commands, which operate on arbitrary groups of characters and lines. In this dissertation, an incremental parsing algorithm suitable for use with an interactive editor is developed. A new solution to the handling of comments in syntax trees is proposed, and an error-recovery algorithm which permits editing of the parse tree in the midst of syntax errors is presented. A prototype editor which employs these algorithms was implemented beginning in 1981 as a demonstration of the practicality and flexibility of this approach; this editor has been in experimental use over the past couple of years.

1.2. The SAGA Project

The SAGA (Software Automation, Generation, and Administration) project is investigating formal and practical aspects of computer-aided support for program development in the software life cycle [Campbell and Kirslis, 84], [Campbell and Richards, 81]. The goal of the project is to design a practical software development environment that supports all major phases of the life cycle. The design of the system requires facilities to allow the construction of a language-oriented editor for a large class of formal languages including many programming languages, specification languages and design languages. The language-oriented editor presented in this dissertation is the editor of the SAGA project, and will at times be referred to as the *SAGA editor*.

The SAGA editor provides a means by which the syntactic and semantic properties of a programming language (or other formal language) can be exploited to provide a more useful interactive environment for the user. Character, line, and screen editing commands are augmented by commands based on the syntax and semantics of the particular language being edited. Fragments of the edited text may be selected by their syntactic (and eventually semantic) structure and moved, copied, deleted, or even transformed into other well-defined syntactic constructs. The same properties may be exploited to constrain a programmer to structure the development of a program using a particular methodology if desired. The editor is being applied in a software development environment of coordinated tools [Kirslis et al., 85]. The environment provides additional support for the management of software development.

Structured editing can also be applied to abstract specification languages. The editor can be used to enter and verify sentences in the language; other software tools can use the structures generated by the editor to verify both the specifications and subsequent programs written to implement them. In each case, providing the higher level interface lets the user deal productively with relevant concepts instead of lower level components; fewer operations are needed, fewer errors will be made, and less time will be needed to accomplish the task.

1.3. Chapter Summary

The remainder of this dissertation discusses the design and structure of an editor based upon an incremental LR(1) parser. Chapter 2 relates some recent previous work. In Chapter 3, *Shift/reduce* parsing is reviewed, with an emphasis on attributes of a parse tree node that can be added to provide support for incremental parsing. Some possible parse tree structures are investigated in Chapter 4, and one chosen which will best support the incremental parser, permit the editor to operate directly from the parse tree, and support related software development tools to be used in the SAGA environment. The incremental LR parser proposed by Ghezzi and Mandrioli is taken as a starting point in Chapter 5, and the extensions necessary to support an editor with incremental parsing are presented and discussed. The integration of the incremental parser with the editor, the basic text and structure editing capabilities, flexibility of the user interface, and the design of the SAGA editor as a hierarchy of modules are presented in Chapter 6. Chapter 7 describes the SAGA editor generating facility, which permits the use of different *parser-generator* and *compiler-generator* systems to automatically construct a SAGA editor for formally specified languages. The lexical, syntactic, and semantic analyses are performed by separate modules within the editor, each containing logically independent data structures. This independence is required in order to effectively implement separate incremental lexical, syntactic, and semantic analysis. It permits reuse of the remaining editor modules whenever an editor is constructed for a new language, since none of these modules contain any language–specific information. Finally, Chapter 8 presents the conclusions from this research, describes some applications in which the editor has already been tested in the SAGA development environment, and suggests some future work.

CHAPTER 2

PREVIOUS WORK

The idea of an editor for formally specified structured data is not a new one. In the late 1960s, attention was directed to the editing of general hierarchies of text [Englebart and English, 68] and sections of annotated, linked text [Carmody et al, 68]. In 1971, Hansen used an extended BNF formalism to describe hierarchic text, and produced EMILY, a template editor for structured programs, but which was retargetable to other formally specified structures [Hansen, 71].

A user of EMILY generated programs by the application of syntactic rules. The user selected syntactic constructs from a menu, building a tree from the root out to the leaves in successive refinement steps. EMILY was based on two principles: *selection not entry*, applied to text construction and operation invocation, and *predictable behavior*, which used a small set of concepts that a user can perceive with a little practice.

The extended BNF formalism provided three features: indentation and carriage–returns could be specified for the formatting of the display; conditional display operations could test contents of sub–nodes and the identifier of the parent of the node to provide flexible display operations; and identifier and block structure could be described in the formalism so that the system could keep track of all references to identifiers. EMILY also supported the elision of selected lines of the display, which Hansen termed *holophrasting*; he also defined a visible marker, the *holophrast*, to represent the elided subtrees on the display.

The languages implemented for EMILY were PL/I, GEDANKEN [Reynolds, 70], a hierarchy language for thesis outlines, and the Emily syntax language. Emily was written in PL/I, and implemented on an IBM 2250 Graphics Display Unit attached to an IBM 360 Model 75. Hansen found that program construction took longer than with a text editor, but that the user made fewer mistakes. EMILY consumed too many CPU cycles and memory to be practical at the time, but Hansen postulated that with decreasing computer and increasing human costs, his approach would eventually become more feasible, and history has shown him to be correct.

In 1975, Donzeau-Gouge, Huet, Kahn, Lang, and Levy produced a text editor specialized for editing program texts, and applied it to the Pascal programming language. It was a first step in building what became the MENTOR programming environment at INRIA-LABORIA, in Rocquencourt, France [Donzeau-Gouge et al., 75, 79, 80]. In their system, programs were manipulated as abstract objects; no parse tree existed. The abstract objects were *labeled trees*, also called operator-operand trees, in which internal nodes are operators. The programs were written in a *concrete syntax*, but stored in an *abstract syntax* tree. An *unparser* was used to regenerate the program. The user of the editor used structural addresses to specify sub-trees. A *constructor* performed syntax analysis, to permit pre-existing code to be edited. A separate process later performed semantic analysis.

The user looked at sub-trees through a window, and an integer *n*, the *holophrasting depth*, was attached to the sub-tree to specify the level of detail to display. Long lists could be *rolled*; the beginning and end hidden, with a portion of the middle displayed. Comments could be attached to a node, either as a prefix or postfix. Comments were not normally displayed, but could be called up. *Evaluators* computed on the abstract tree. Their editor was written in Pascal.

A table-driven, interactive, diagnostic programming system, CAPS, was produced by Wilcox, Davis, and Tindall in 1976 [Wilcox et al., 76]. CAPS was a highly interactive, menu-driven editor, diagnostic compiler and interpreter which was used to prepare, debug, and execute simple programs. Errors were diagnosed both at compile and run time. The analysis was performed character by character; when an error occurred, a box was flashed around the invalid character, any additional input data was ignored, and CAPS began an interaction with the student to find the cause of the error. The user could back up the cursor to erase the box and resume editing, or press a HELP key for auto-generated diagnostic assistance. The first press of the HELP key displayed an error message; subsequent presses suggested possible repairs. CAPS employed a recursive-descent parser with complete syntax checking. The internal representation was a list of tokens, including spacing information and comments. Static semantic analysis was also performed. Execution interpretation included a trace facility and run-time error analysis. CAPS was table-driven and could be retargeted to other languages. New interpreters had to be designed and implemented for a new language, but many modules could be reused, since the internal structure had the same form, regardless of the language. CAPS was available for Fortran, PL/I, and COBOL. It was implemented on the PLATO IV Computer-Based Education System at the University of Illinois at Urbana-Champaign.

In 1977, Teitelman produced INTERLISP, a display-oriented programmer's assistant [Teitelman, 77]. It was a programming system for LISP, based on interpretation, with emphasis on the debugging and execution of programs. An interpreter linked program pieces for execution. A system debugger worked by interpreting code. Code pieces could be compiled, but the debugger only accessed the interpreted code.

In the late 1970s, Teitelbaum designed and built the Cornell Program Synthesizer [Teitelbaum, 79], [Teitelbaum and Reps, 81]. The Synthesizer provides a syntax-directed programming environment. It incorporates a grammar via templates which are predefined in the editor. Programs are created *top-down*. New templates are inserted within the skeleton of previously entered templates. However, *phrases* (assignment statements, expressions, and lists of variables) are entered directly as text. Programs are translated into interpretable form during program entry. Program development and testing can be interleaved; interpretation is suspended when an unexpanded placeholder is encountered; it can be resumed after the placeholder is expanded. A batch LR parser is used at the expression level. Errors are detected as soon as the user moves the editing cursor out of a field; the cursor is positioned at the point of the error. Modifications are performed by the *clip, delete,* and *insert* commands. For example, to change a statement into a block of two statements: the statement is clipped and replaced by the original placeholder, a block template is selected, the clipped statement is inserted, and the new statement added. Semantic analysis is performed through an incremental attribute re–evaluation scheme [Reps, 82]. The Synthesizer has been shown to be a good educational tool: it has been used in introductory programming courses at several universities since June 1979. It has been used with the language PL/CS [Conway and Constable, 76], a subset of PL/I.

The Synthesizer is limited in utility in that editing operations (modifications, moving, copying) are cumbersome and not likely to be favored by experienced programmers. Simple text editing commands are limited to phrases only. It cannot be used with pre–existing software because there is no way provided to convert fragments into template form. Comments can only be inserted at selected locations, and are required in certain locations. The Synthesizer employs a hybrid approach: recursive descent at high levels (templates), and parsing of character strings at low levels (phrases).

Unlike the Cornell Program Synthesizer, the BABEL editor by Horton [Horton, 81] presents a text–like interface to the user, and provides commands which operate on sequences of characters. It performs lexical analysis on the input text, producing a list of tokens, can perform optional checking of the syntax based on an earlier Ghezzi & Mandrioli parsing algorithm published in 1979 [Ghezzi and Mandrioli, 79], and can also perform optional semantic checking using Reps' algorithm. This algorithm operates on grammars of the class LR(1) $\cap$ RL(1). That is, the grammar must be LR(1) and the *reversed* grammar, obtained by reversing the right hand sides of all of the productions, must also be LR(1). The algorithm uses both left–thread and right–thread pointers to store parser state information; Horton replaced some of the links with access routines to generate them in order to save space. Comments are handled by attaching them to the following token. Programs are not permitted to be incomplete, and it is not possible to place unexpanded non–terminals in the tree (that is, there are no placeholders.) Horton defined a *Language Description Language* (LDL) to specify the language for which an editor is to be built; a modified *yacc* parser is used to produce the parse tables.

Horton reported that in BABEL, the running time required to perform syntax checking is 5 times as much as that taken by the *vi* text editor [Joy and Horton, 80] to perform the same editing operation (with no analysis). Semantic checking is 15 times slower when an executable statement is changed; when a declaration is changed, it is slower by a "much larger factor" (unspecified). (Horton reports on one example with semantic analysis which took 62 times as much processing). BABEL trees without semantic information average 30 times the size of the equivalent text file; with semantic information, the size increases to 300 times.

At Carnegie–Mellon, Medina–Mora and Feiler have produced an editor as part of the Incremental Programming Environment (IPE) [Medina–Mora and Feiler, 81]. The environment consists of several tools: the editor, translator, linker and loader, and debugger. The user interacts with the entire system through the editor; other tools are invoked by the editor as needed. The editor is syntax–directed; the programmer constructs his program by inserting templates, and syntactic correctness is enforced. The editor represents the program internally as an *abstract syntax tree*. An *unparser* translates the tree back into readable text to present the programmer with his program. Semantic correctness is not enforced; semantic checking routines perform further analysis, and are automatically invoked.

Their system supports incremental program translation. The program is debugged with a language-oriented debugger. A syntax-directed editor generator is used to prepare additional editors. IPE provides an environment for a single programmer working on a single program. IPE is a component of the Gandalf project, which coordinates programmers and versions of programs [Habermann, 79]. The language supported is GC, a type-checked variation of C [Kernighan and Ritchie, 78] with modular structure. Language descriptions also have been prepared for a subset of Ada; *Alfa*, an non-Algol-like applicative language designed by Habermann; the system language of Gandalf; and the grammatical description itself. An IPE prototype is running under UNIX on a VAX. Medina-Mora and Feiler have found that new users need to get used to the structured editing approach; expression entering and editing is more difficult than text editing; preexisting code cannot be used unless a parser is built to perform a preprocessing pass to convert it into tree form.

At the University of Illinois, Orailoglu has reviewed the design issues involved in the development of hierarchical editors, and has produced an editor which employs a modified LL(1) predictive parser [Orailoglu, 83]. He incorporates language-specific information through a ***user-specified grammar*** with ***incomplete productions***. The user interface of the editor permits movement by characters, words, lines, and from one node to another within the surrounding tree structure. Text characters entered by the user are inserted at the position of the cursor; a delete key deletes the character, word, line, or tree at which the cursor is positioned. User-supplied pretty printing information can be specified in the language description; comments, however, are not pretty printed. Lexical analysis, syntax analysis, and pretty printing are performed character by character; detection of an error causes the remaining input to be displayed in reverse video as the user continues typing. No semantic analysis is performed.

Shilling has extended Orailoglu's editor with a combination of *follow-the-cursor* parsing, which parses only the characters up to the editing cursor, and *soft templates*, which appear following the cursor to indicate the non-terminals the parser is expecting but has not yet had completed by the parser [Shilling, 85]. The templates are *soft* in that the user is not obligated to follow them. Shilling is also adding semantic analysis based upon attribute grammars and the attribute update algorithm of Reps. The editor has language description grammars for Cobol, Fortran, Pascal, C, and some other languages; it is implemented on several systems which run the 4.2BSD UNIX operating system.

There have been other efforts toward improving the editing process or the software development environment in similar ways [Fraser, 81], [Morris and Schwartz, 81], [Osterweil, 82, 83] and [Osterweil and Cowell, 83]. A number of reviews survey the field of editors, summarizing many efforts [Meyrowitz and van Dam, 82], [Reid and Hanson, 81] and [van Dam and Rice, 71], and will be of interest to the reader desiring more detailed background information.

The development of structure-oriented editors has been monitored by several individuals, who have made the following observations. Waters, at MIT, notes that early implementations of syntax-directed editors have been overly restrictive, and that the criticisms about them are generally valid. He believes that the editors need time to mature, and could become quite attractive to use at some point in the future. He is firm that text oriented commands *should not* be replaced, but augmented with structure-oriented commands [Waters, 82].

Meyrowitz and van Dam, at Brown University, note that a well-defined, consistent, conceptual model is needed, instead of the ad hoc methods used today. Documentation is needed which explains the conceptual model and the user interface. A clear, concise, orthogonal user interface that is easy to learn is needed. Today, interfaces are haphazard and contradictory. The sharing of project information and files among a group in a controlled way is also needed [Meyrowitz and van Dam, 82].

The SAGA editor addresses many of these points. Because it is based upon an incremental parser, its user interface is much more flexible than that which has been provided by structure editors. Text–oriented commands have not been replaced, but retained and augmented with structure–oriented commands. The user interface is concise and orthogonal, permitting the specification of groups of characters, tokens, lines, and sub–trees, and applying all built–in operations to all argument types which make sense. Comments are handled as any other token, and not treated as a special case as in all other systems to date. We believe that we also have a solution to the sharing of project information and files among a group through an *Integrated Modular Environment* [Kirslis et al., 85]; we have defined a model, representation, and implementation for an environment which can be used with many standard software development tools available today, and with which additional benefits are possible when combined with a language–oriented editor such as the SAGA editor. The approach we have taken has already yielded a prototype language–oriented editor and environment. We believe that the editor and support environment has practical application in software development, and we believe that it will be possible to refine these prototype tools into a usable system with significant benefit to software engineers.

CHAPTER 3

SHIFT-REDUCE PARSING

We begin our discussion of parsing with a quick review of *shift-reduce* parsing, also called *LR(k)* parsing; we then describe a *left threading* of a parse tree which will be of great use when we turn our attention to incremental parsing.[1] In *LR(k)* parsing, the *k* refers to the number of symbols at the head of the input string that are passed to the parser to enable it to determine the parsing action; these symbols are termed the *lookahead* of the parser. The languages in which we are interested can be defined by *LR* grammars with $k = 1$, so we will restrict our discussion to this class.[2] A subset of *LR(1)* grammars, termed *LALR(1)* grammars, is of particular interest to us, since the parse tables produced for this class are much smaller and better suited for practical use.

3.1. Preliminary Definitions

LR parsers are driven from tables which can be algorithmically generated from a formal specification of the language. Programs which apply these algorithms and produce these tables are called *parser-generators.* Since the specific techniques and algorithms used by parser-generators are well documented elsewhere and are not necessary for the understanding of

[1] Readers unfamiliar with *LR(k)* parsing can find an introduction to the subject in [Aho and Ullman, 77], especially Chapter 5. In this discussion, we assume the reader is familiar with *shift/reduce* parsing notation as defined in [Aho and Ullman, 72].

[2] If there is an interest in a language described by an *LR* grammar with $k > 1$, and if a parser-generator is available which will process the grammar, it would be a simple matter to modify the editor to pass to the parser a list of lookaheads. The incremental parsing algorithm is not affected.

language–oriented editing being presented in this dissertation, a discussion of these techniques is omitted. The reader may consult [Aho and Ullman, 77] for more information about this topic.

The formal specification of a language can take a number of forms. Two of the parser–generators used by the SAGA project [Noonan and Collins, 84], [Mickunas, 86] use a context–free grammar in *Backus–Naur Form* (BNF) as a specification. A third parser–generator, presently under construction, will take an extended BNF specification [Beshers, 84]. For simplicity, we will only discuss BNF syntax, since the extended BNF can always be rewritten in this form.

BNF notation provides a means to write a formal description of a language for which we wish to construct a parser. The description is given as a *context–free grammar* G, which is defined to be the four–tuple (N, Σ, P, S), where N is the finite non–empty set of *non–terminal* symbols, Σ is the finite set of *terminal* symbols, P is the finite set of *productions* $A \longrightarrow \alpha$, where $A \in N$ and $\alpha \in (N \cup \Sigma)^*$, and $S \in N$ is a distinguished non–terminal termed the *start symbol* [Hopcroft and Ullman, 79]. The sets N and Σ are disjoint; that is, $N \cap \Sigma = \emptyset$ (the empty set). Additionally, $N \cup \Sigma$ is conventionally denoted as V. See Figure 3–1 for an example of a grammar.

```
<S> ::= <E>                  (1)
<E> ::= <E> + <E>            (2)
<E> ::= <E> * <E>            (3)
<E> ::= ident                (4)
<E> ::= integer              (5)
<E> ::= ( <E> )              (6)
```

Figure 3-1: An (Ambiguous) Grammar for Simple Expressions. G = (N, Σ, P, <S>), where N = (<S>, <E>), Σ = (*ident, integer*, +, *, '(', ')'), P is shown above, and <S> is the start symbol.

A *string* over V is a finite sequence of symbols from V. V* denotes the set of all strings from V, including the empty string ϵ, with V+ = V* - { ϵ }. If α, γ, $\beta \in$ V*, and $A \in$ N, then $\alpha A \beta$ ***directly derives*** $\alpha\gamma\beta$, denoted $\alpha A \beta \Rightarrow \alpha\gamma\beta$, where $\Rightarrow$ is a relation between strings in V*, and $A \rightarrow \gamma$ is a production in the grammar. (The production $A \rightarrow \gamma$ is applied to the string $\alpha A \beta$ to yield $\alpha\gamma\beta$.)

If α_1, α_2, ..., α_n are strings in V*, and

$$\alpha_1 \Rightarrow \alpha_2, \quad \alpha_2 \Rightarrow \alpha_3, \quad \alpha_{n-1} \Rightarrow \alpha_n,$$

then α_1 ***derives*** α_n, denoted $\alpha_1 \overset{*}{\Rightarrow} \alpha_n$. By convention, $\alpha \overset{*}{\Rightarrow} \alpha$.

A ***derivation tree*** D for a context-free grammar G = (N, Σ, P, S) is a labeled ordered tree in which each node is labeled by a symbol from V; if A labels an interior node and B_1, ..., B_n label the immediate descendants, then $A \rightarrow B_1 B_2 \ldots B_n$ is a production in P.

The ***frontier*** of a derivation tree is the string $w = w_1 w_2 \ldots w_n$, where the $w_i \in \Sigma$ are the labels of the terminal nodes read left to right.

Given a context free grammar G with start symbol S, the language generated by G, denoted L(G), consists of all strings of terminals w such that S $\overset{*}{\Rightarrow} w$. A ***sentential form*** α of G is a string of terminals and/or nonterminals such that S $\overset{*}{\Rightarrow} \alpha$.

A ***rightmost derivation*** is a derivation in which the rightmost nonterminal in a sentential form is replaced at each step in the derivation. Such a derivation is denoted by $\alpha \underset{rm}{\Rightarrow} \beta$ if a single step is taken, $\alpha \underset{rm}{\overset{*}{\Rightarrow}} \beta$ if zero or more steps are taken, and $\alpha \underset{rm}{\overset{+}{\Rightarrow}} \beta$ if at least one and possibly more steps are taken. A ***right sentential form*** α is a sentential form generated from the start symbol S by a rightmost derivation.

It can be determined whether a string of terminal symbols from G is a sentence in L(G) by performing the reverse of a rightmost derivation on the terminal string. A program to perform such an analysis in limited cases of context free languages is termed an *LR parser*, which consists of a *driver routine* to perform the parsing, a set of named or numbered *parse tables* to direct the parse, a *parse stack* on which to store the intermediate results, and an *input* from which to read the string to be parsed.[3] The parser is assigned a *parse state* which identifies one of the parse tables as the table to reference at the current step in the parse. One of these states is distinguished as the *initial state* in which the parser is to begin execution.

A *configuration* of an LR parser for a grammar G is an ordered pair (K, I), where K represents the contents of the parse stack and I is the input not yet read by the parser. Each entry on the parse stack K consists of a symbol Y followed by a parse state P, where

$$Y \in V \cup \textit{bof} \cup \textit{eof}.$$

The special symbols *bof* and *eof* serve as left and right pads, respectively, to delimit the contents of the stack and the input.[4]

3.2. Shift–Reduce Parsing

Given a grammar that describes a language, and a parser–generator to produce parse tables from that grammar, we can build a parser which uses those tables to analyze strings in the language. We illustrate the operation of this parser with an example of the parsing of a string in the language generated by the grammar given in Figure 3–1. In this grammar, the *ident* symbol represents an alphanumeric identifier, and the *integer* symbol represents an integer.

[3] The *LR* in *LR parser* stands for *Left–to–right* scan of the input and construction of a *Rightmost* derivation in reverse.

[4] The use of a left and right pad theoretically restricts us to strict deterministic context–free languages with end markers, and also less than full LR parsers, since the parser should only check that the input is exhausted once it has decided to accept the input already scanned. But this restriction is of no practical significance. During initialisation of the editor, the parser is called with the empty string to produce an initial tree which provides a uniform context for subsequent editing. An *eof* symbol is always present at the right end of the parse tree frontier, and so is always passed to the parser whenever the end of the parse tree is reached. While the parser always will be passed both a parse state and this *eof* symbol when the input has been exhausted, it is not required to look at this symbol, and can behave in the above manner. In addition, grammars need not include the end markers; the choice is left to the parser–generating system.

Assume that the grammar given in Figure 3–1 has been processed by a parser–generating system to produce the set of parse tables given in Table 3–1. We are not concerned here with how the tables are produced from the grammar; such techniques are well–understood, and the reader can find these details in [Aho and Ullman, 77] if interested. In Table 3–1, each row is a single parse table. The parser begins in state 1, and enters other states as directed by entries in the table for its current state. There is a single column for each terminal and non–terminal symbol in the grammar. The parser reads a symbol from the input; this symbol becomes the lookahead symbol which is used to select an entry from the parse table to direct the parser's next step.

Given this set of parse tables and an input string to be parsed, the parser begins in an initial configuration (K, I) which is given by the 2–tuple with only the *bof* (beginning of file) token and initial parse state stored in the stack component K, and the input string in the second component I. At each step in the parse, the parse state on the top of the stack is the current state of the parser, and the first token in the input string is the lookahead symbol.

State		Action							
		id	+	*	(	)	eof	S	E
1		s10			s4		r2	s2	s3
2	S						acc		
3	E		s6	s5			r3		
4	(	s10			s4				s7
5	*	s10			s4				s9
6	+	s10			s4				s8
7	E		s6	s5		s11			
8	E		r4	s5			r4		
9	E		r5	r5		r5	r5		
10	id		r6	r6		r6	r6		
11	)		r8	r8		r8	r8		

Table 3–1: *LR(0)* parsing tables for the simple expression grammar given in Figure 3–1.

Four types of parsing actions are shown in the parse tables: *shift* (s), *reduce* (r), *accept* (acc), and *error* (blank). When a *shift* action is indicated, the parser removes the input token from the second component of the tuple and pushes it onto the stack (the first component) together with the state given in the parse table with the shift action. This state becomes the new state of the parser.

When a *reduce* action is indicated, the parser uses the associated number to determine which production rule of the grammar is to be used to perform the reduction. The parser pops the entries corresponding to the right hand side of the rule off the stack, and then prepends the token code corresponding to the left hand side of the production rule to the head of the input in the second component.

Standard parsers at this point typically replace the tokens and states corresponding to the right hand side of the production rule with the non–terminal on the left hand side of the production rule and the new parse state. The new state is determined by applying a *goto function* to the *state uncovered* on the parse stack after the right hand side of the production rule has been removed, and the non–terminal on the left hand side of the production. The *goto function* will always return a *shift* action, and there can never be an error when it is computed. But by prepending the non–terminal to the head of the input stream instead of continuing with the reduction in this manner, the next action that the parser will perform will be exactly this *goto function*. By having the parser treat the *goto function* as the standard parsing action, instead of separating it out as a special case, we gain the ability to have non–terminal nodes treated identically to terminal nodes. This uniformity is important, since it permits the SAGA editor to pass previously parsed sub–trees to the parser intact to be inserted into the parse tree at a new location. The contents of this (potentially large) sub–tree need not be reparsed, saving computation time, and we can provide an editing command to move sub–trees around easily in the editor.

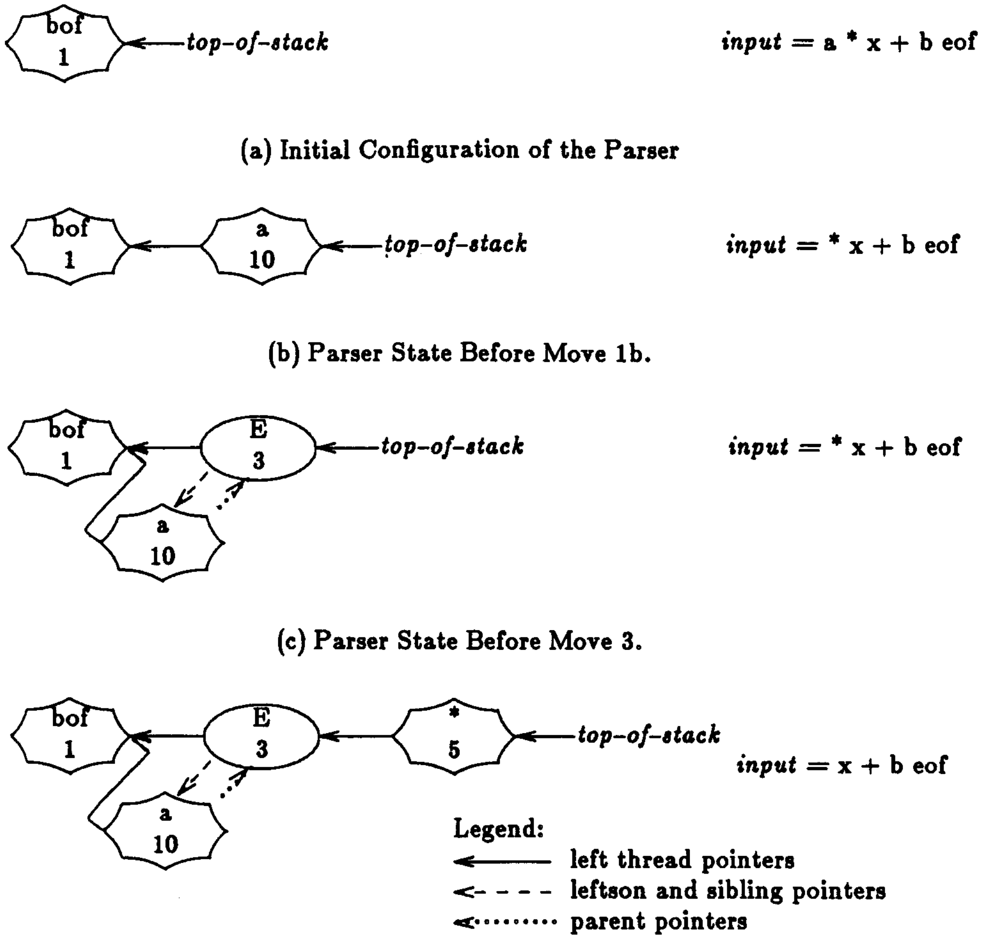
Move	(Stack,	Input)	Action
0	(bof 1,	a * x + b eof)	
1a	(bof 1,	a * x + b eof)	s10
1b	(bof 1 a 10,	* x + b eof)	r6
2	(bof 1,	<E> * x + b eof)	s3
3	(bof 1 <E> 3,	* x + b eof)	s5
4a	(bof 1 <E> 3 * 5,	x + b eof)	s10
4b	(bof 1 <E> 3 * 5 x 10,	+ b eof)	r6
5a	(bof 1 <E> 3 * 5,	<E> + b eof)	s9
5b	(bof 1 <E> 3 * 5 <E> 9,	+ b eof)	r5
6	(bof 1,	<E> + b eof)	s3
7	(bof 1 <E> 3,	+ b eof)	s6
8a	(bof 1 <E> 3 + 6,	b eof)	s10
8b	(bof 1 <E> 3 + 6 b 10,	eof)	r6
9	(bof 1 <E> 3 + 6,	<E> eof)	s8
10	(bof 1 <E> 3 + 6 <E> 8,	eof)	r4
11	(bof 1,	<E> eof)	s3
12	(bof 1 <E> 3,	eof)	r3
13	(bof 1,	<S> eof)	s2
14	(bof 1 <S> 2,	eof)	acc

Figure 3-2: Configurations through which the parser passes during an *LR* parse of the input string: $a * x + b$. The current parse state is stored on the top of the stack, and the lookahead symbol is at the head of the input. The simple expression grammar is given in Figure 3-1, and the set of parse tables produced from this grammar in Table 3-1.

An *accept* action tells the parser to terminate and accept the input string as a legal sentence in the grammar. When the parser terminates in this way, the start symbol of the grammar will be the only token on the parse stack (not counting the *bof* token which is always present), and the *eof* token will be the only token remaining in the input string.

A *blank entry* in the table indicates that an error has occurred; in this case the parser terminates and rejects the string as a non-sentence in the language.[5]

[5] In the case of the SAGA editor, the parser invokes an error handler to save the information necessary to enable the parser to resume the parse at a later time, and to enable the editor to display this portion of the tree in the meantime.



(a) Initial Configuration of the Parser

(b) Parser State Before Move 1b.

(c) Parser State Before Move 3.

(d) Parser State Before Move 4a.

Figure 3–3: Construction of the parse tree for the first few moves shown in Figure 3–2. The numbers in the nodes refer to the state of the parser just after a *shift* action was performed with that node.

Let us now consider the input string $a * x + b$. Figure 3–2 shows the moves that our parser makes at each step in the parse. The first row in the table shows the initial configuration of the parser. At each step in the parse, the rightmost number in the first component is the parser's current state, and the leftmost symbol in the second component is the lookahead symbol at the head of the unread input string. At the conclusion of this successful parse, the stack contains only the *bof* symbol and the start symbol, and the input string only the *eof* symbol.

3.3. Constructing the Parse Tree

The approach used by the SAGA editor's parser is similar to the one described above, except that a parse tree actually is constructed during the parsing operation. In addition, *no explicit parse stack is used* by the editor's parser. Instead, the parse stack is directly incorporated into the parse tree as it is constructed. Each parse tree node is augmented with a *left thread* attribute, which contains a pointer to the node that would be directly beneath this one on a parse stack if one were used. A *top-of-stack* variable points to the node which would be on the top of this stack at each point in the parse. Figure 3–3 illustrates the parse tree and input string manipulated by the parser for the first few moves of the parse given in Figure 3–2. Figure 3–4 presents the completed parse tree constructed by the parser for this input string.

In Figure 3–4, the reader should take note of the left thread pointers, shown as solid arrows, which connect the nodes of the tree. The number in each node is the new parse state of the

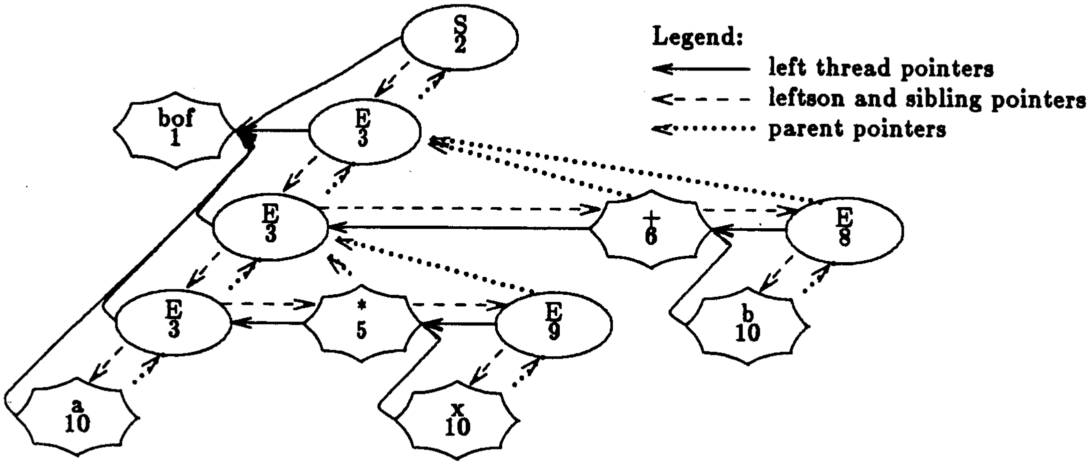


Figure 3–4: The parse tree constructed for $a * x + b$.

parser just after that node is shifted. By storing these two pieces of information in the parse tree, *each and every configuration* through which the parser has passed during the entire parse is captured. By setting the top–of–stack variable to any of these nodes, and the parser state to the state found in that node, we recreate the exact configuration the parser was in at this point in the parse, just as though we had begun the parse from scratch and proceeded up to this point, pausing just after this node had been shifted. This ability to recreate any intermediate configuration quickly in the parse is central to the editor's ability to efficiently and incrementally reparse a user's modifications as they are made. The ability to terminate the reparse after the modification is complete, and not completely reparse the remainder of the program is also required if this approach is to prove feasible to use. The incremental parser also has this second property, but we will defer discussion of it until Chapter 5, when the incremental parsing algorithm is discussed in detail.

CHAPTER 4

PARSE TREE STRUCTURE

In chapter 3, construction of a parse tree was discussed. In this chapter, we will look at some possible parse tree structures and then decide upon the one to be used with the incremental parsing algorithm to be presented in chapter 5. To avoid implementation complexity, it is desirable to have the editor use the parse tree structure directly instead of maintaining both a parse tree and equivalent text representation and then maintaining the consistency between them. Therefore, the parse tree must be able to support both the incremental parsing algorithm and the editor's command interpreter and display module.

The parse tree proposed by [Ghezzi and Mandrioli, 80] is sufficient to support their parsing algorithm, but is not suitable for use with an editor since a number of operations are required which cannot be performed efficiently using their structure. In particular, the editor's command interpreter requires the ability to move from node to node throughout the tree in response to user commands which select token sequences and sub-trees for editing. In addition, the editor's display module needs a convenient and efficient way to sequentially access the terminal nodes in the tree to generate the display; it is much too inefficient to force a walk through the internal structure of the tree in order to retrieve these terminal nodes.

We will begin with a summary of common tree traversal methods for both *binary trees* and *trees*. By *tree*, we mean an *ordered tree* in which each non-terminal node has one or more children in a particular order from left to right, as opposed to an *oriented tree* in which no ordering is imposed upon the children. By *binary tree*, we mean an *ordered tree* in which each internal node has at most two children, distinguished as the left child and the right child. Then we will review the parse tree structure proposed by Ghezzi and Mandrioli and show what access routines their structures require to support the traversals, and what difficulties arise. Lastly, we propose improvements to their linking structure, and show how the modified linking structure better supports the tree traversals and editor tree access.

4.1. Traversing the Parse Tree

Tree traversal algorithms visit each node in the tree in some order. Recursive or iterative programs can easily be written which visit each node and its sub–trees. Three common traversal methods for *binary* trees are listed in Table 4–1, headed by some of the names commonly used to refer to them. These traversals assume that each internal node in the binary tree contains pointers to its left and right children. The editor's parse tree is actually implemented as a binary tree by using a standard correspondence between trees and binary trees [Knuth, 73]. Therefore, programs which need to visit each node of the tree *and* can use a binary tree traversal may do so if a simpler program results. However, the parse tree should conceptually be thought of as a (non–binary) tree since each internal node has one, two or more children, and a given traversal algorithm will visit the nodes in a tree in a different order, depending upon the way in which the

preorder *depth–first order*	*inorder* *symmetric order* *lexicographic order*	*postorder* *endorder* *bottom up order*
visit the node traverse the left subtree traverse the right subtree	traverse the left subtree visit the node traverse the right subtree	traverse the left subtree traverse the right subtree visit the node

Table 4–1: Some *binary tree* traversal methods.

tree is viewed. Therefore, we will not speak of the left and right sons of a node, but of the *children* of a node, and a node with *n* children will have *n* subtrees to be visited. Using this tree structure, the preorder and postorder traversals may be described as shown in Table 4–2. There is no simple equivalent for *inorder*, since the root node needs to be visited somewhere in between the visits to the first and last children.

4.2. The Ghezzi and Mandrioli Parse Tree

In the Ghezzi and Mandrioli parse tree, nodes are linked together by four types of links: *lthread* (left thread), *parent, rmost* (rightmost sibling) and *rdescend* (rightmost descendant). These links are all that are required to support an incremental parser; thus the leftmost son and sibling links, shown in the parse trees presented in Chapter 3, do not exist in this tree. The *lthread* link is identical to the left thread link previously described; it points to the node which was shifted by the parser immediately preceding this one. Each node in the tree points to its parent through its *parent* link, and to the rightmost sibling in its production through its *rmost* link. Lastly, each node points to the terminal node at the right end of its sub–tree through its *rdescend* link.

Unfortunately, we have no pointers to any of the subtrees to use for the tree traversals, except for the single pointer to the rightmost descendant. So either a new method for traversal must be devised which uses only those links which are available, or the left and right son links of

preorder	*postorder*
visit the node traverse the subtrees	traverse the subtrees visit the node

Table 4–2: Equivalent *tree* traversal methods for *preorder* and *postorder* traversals.

a binary tree must be simulated in some fashion. By providing functions to return the *leftson* and *right sibling* of a node, we can implement the above traversals without requiring any other links in the tree.

4.2.1. Retrieving the Right Sibling of a Node

With this tree structure, the right sibling of a node may be determined by accessing only the *rmost* and *lthread* attributes in the tree. For the nodes which are children of a single parent, the *lthread* attribute links each child except the leftmost son to its left brother (the leftmost son is linked to the left sibling of its closest ancestor with a left sibling). Using these two links, we can construct a function which returns the *right sibling* (hereafter referred to simply as the *sibling*) of a node, or *nil* if the node is itself a rightmost sibling. This function is presented in Figure 4–1.

Algorithm 4–1: Retrieve the Right Sibling of a Node

```
sibling($\mathcal{N}$):
    Input: A parse tree node pointer.
    Output: A pointer to the sibling of the node, or nil if none exists.

    Let $X$ be a pointer to a parse tree node.

    $X \leftarrow$ rmost($\mathcal{N}$);                      1
    if $X = \mathcal{N}$ then                                 1
        return(nil);                                          0
    while $\mathcal{N} \neq$ lthread($X$)                     $R$
        $X$ = lthread($X$);                                   $R - 1$
    return($X$).                                              1
```

Figure 4–1: Given a parse tree whose nodes are linked together only by *parent, lthread, rmost,* and *rdescend* links, retrieve the right sibling of a node. The column to the right counts the number of times each statement is executed, under the assumption that $\mathcal{N}$ is not itself a rightmost sibling. (If $\mathcal{N}$ is a rightmost sibling, then the running time is 3, since only the first three lines are executed.)

4.2.1.1. Discussion of Algorithm 4–1

Given any node in a completed parse tree, this algorithm will always return the right sibling of a node if one exists, or *nil* otherwise. This can be seen as follows: Let $\mathcal{N}$ be a node in the parse tree. If $\mathcal{N}$ is the rightmost child in some production, then its *rmost* attribute will have been set to itself when this portion of the tree was built, X will test equal to $\mathcal{N}$, and *nil* will be returned. If $\mathcal{N}$ is not the rightmost child in some production, then its *rmost* attribute will have been set to the node which is the rightmost in the production. The *while* loop above will succeed in locating the next sibling of $\mathcal{N}$ if and only if $\mathcal{N}$ appears on the *lthread* list beginning at *rmost($\mathcal{N}$)*, and is located immediately after the node which is its next sibling.

$\mathcal{N}$ does appear on this *lthread* list because an LR parser constructs the parse tree by a sequence of operations which correspond to the reverse of a rightmost derivation of the terminal string represented by the parse tree. In a rightmost derivation, the rightmost non–terminal is replaced at each step. The right–sentential form produced in this way can be written as $S \underset{rm}{\overset{+}{\Rightarrow}} \alpha w$, where α consists of a mixture of both non–terminal and terminal symbols, while w consists only of terminal symbols. As an LR parse progresses, α will correspond to the contents of the parse stack, and w to the unexpended input string. Each node on the parse stack corresponding to a symbol in α will have its *lthread* attribute set to the node which represents the symbol to its left in α, since the nodes in this list are by definition those on the parse stack.

In addition, if $\alpha = \gamma\beta$, and $B \rightarrow \beta$ is a production in the grammar, where β is on the top of the parse stack, then β is called a ***handle***. Whenever the parser recognizes a handle β on the parse stack, it performs the reduction $B \rightarrow \beta$, replacing β with the non–terminal B. Because the reverse of a rightmost derivation is being performed, the symbols that comprise β are exactly the immediate children of the production B $\rightarrow \beta$. Therefore, the first n nodes at the top of the parse stack, where $n = |\beta|$, will be the nodes corresponding to β, with the rightmost child on top.

Since the *lthread* attribute of the first $n - 1$ nodes at the top of the parse stack is set to their left siblings, and the parse stack is finite in length, by following the *lthread* links, eventually a node X must be found whose *lthread* attribute is $\mathcal{N}$, and the algorithm will successfully terminate.

4.2.1.2. Running time of Algorithm 4–1

When $\mathcal{N}$ is the rightmost sibling, only the first three lines of the algorithm are executed, so the running time of the above algorithm is 3, or *O(constant)*. Otherwise, the running time is given by the sum of the counts shown to the right in Figure 4–1, which is $2R + 2$, where $R = |\beta| - 1$, $B \rightarrow \beta$ being the production for which $\mathcal{N}$ represents a symbol contained in β. This is also *O(constant)*, being on average the mean of the lengths of the production rules represented in the tree, which is independent of the number of nodes in the tree.

4.2.2. Retrieving the Leftmost Son of a Node

Construction of an algorithm to retrieve the leftmost son of a node using some combination of these four links is slightly more complicated, and unfortunately will not run on average in *O(constant)* time, as the ***sibling*** function does. This algorithm will make the above tree traversals easier to code, although its overall running time may not be very desirable, as we shall see. This algorithm is presented in Figure 4–2.

4.2.2.1. Discussion of Algorithm 4–2

This algorithm works by following the chain of parent pointers back up from the rightmost descendant of a node until the node's rightmost (immediate) child is reached, and then by following the left thread links through the list of children until the leftmost child is reached. Each time that the parser performs a reduction of nodes $X_1...X_n$ to a parent node $\mathcal{N}$, it makes the following assignments (among others):

Algorithm 4–2: Retrieve the Left Son of a Node

leftson($\mathcal{N}$):

Input: A pointer to a parse tree node.

Output: A pointer to the left son of the node, or *nil* if none exists ($\mathcal{N}$ is a terminal node).

Let X be a local variable which is a pointer to a parse tree node.

```
$X \leftarrow$ rdescend($\mathcal{N}$);               1
if $X = \mathcal{N}$ then                          1
    return(nil);                                   0
while parent($X$) $\neq \mathcal{N}$ do            $H$
    $X \leftarrow$ parent($X$);                    $H - 1$
while parent(lthread($X$)) = $\mathcal{N}$ do      $R$
    $X \leftarrow$ lthread($X$);                   $R - 1$
return($X$).                                       1
```

Figure 4–2: Given a parse tree whose nodes are linked together only by *parent, lthread, rmost,* and *rdescend* links, retrieve the leftmost son of a node. The column to the right counts the number of times each statement is executed, under the assumption that $\mathcal{N}$ is not a terminal node. (If $\mathcal{N}$ is a terminal, then the running time is 3, since only the first three lines are executed.)

$$\begin{aligned} &parent(X_k) \leftarrow addr(\mathcal{N}),\ 1 \le k \le n; \qquad (1)\\ &rdescend(\mathcal{N}) \leftarrow rdescend(X_n); \end{aligned}$$

which extends the chain of parent links along the right edge of the tree by one level. The first *while* loop can be shown correct by induction on the height h of the right side of the sub-tree whose root is $\mathcal{N}$. For $h = 0$, $\mathcal{N}$ is a terminal node, and its *rdescend* attribute points to itself. For $h = 1$, the *rdescend* attribute of a non-terminal node points to its rightmost child, which must be a terminal node. Likewise, the *parent* attribute of the rightmost child terminal node points back to this non-terminal node, since it is its immediate parent. For $h = 2, ..., n$, assuming the *rdescend* attribute of $\mathcal{N}$ is the rightmost terminal node in the parse tree, and the *parent* attribute of the rightmost child of $\mathcal{N}$ points to $\mathcal{N}$, then for $h = n + 1$, the *rdescend* attribute of $\mathcal{N}$ is identical to that of its rightmost child since it is copied from the *rdescend* attribute of its rightmost child, and the *parent* of the rightmost child of $\mathcal{N}$ is $\mathcal{N}$ itself, as given in code fragment (1) above. Therefore, the rightmost descendant attribute of all rightmost non–terminals in the sub–tree with root node $\mathcal{N}$ point to the rightmost terminal in the sub–tree, and $\mathcal{N}$ can always be reached by following *parent* pointers beginning at *rdescend*($\mathcal{N}$). Since the first *while* loop in algorithm 4–2 follows *parent* pointers beginning at the rightmost descendant of $\mathcal{N}$, it will always arrive at a node X whose *parent* attribute is set to $\mathcal{N}$, and this node will be the rightmost child of $\mathcal{N}$. Therefore, at the conclusion of the first while loop, X will be set to the rightmost child of $\mathcal{N}$.

From the earlier discussion of algorithm 4–1, we know that if we start at the rightmost child X of $\mathcal{N}$ and follow the pointers stored in the *lthread* attribute, that we reach the children of the production whose left hand side symbol is represented by the parent of this node, that we reach these children in right–to–left order, and that the *parent* attribute of each of these children is set to $\mathcal{N}$. Therefore, by looking one node deeper into the stack than X, that is, to node *lthread*(X), if we find that the *parent* attribute of this node is not set to $\mathcal{N}$ then we know that X is the first and leftmost son of $\mathcal{N}$. Thus the second *while* loop will always terminate with X set to the left(most) son of $\mathcal{N}$, and X is the value which is returned.

4.2.2.2. Running Time of Algorithm 4–2

If $\mathcal{N}$ is a non–terminal node, the running time of *leftson* is $2H + 2R + 1$, where H is the height of the right side of subtree $\mathcal{N}$, and R is the length of the right hand side of the production rule whose left hand side non–terminal is $\mathcal{N}$. (If $\mathcal{N}$ is a terminal node, then the running time is 3.) Since R depends on the length of the production rule and not the number of nodes in the parse tree, it is of *O(constant)*. However, H depends on the height of the right side of the sub–tree of which it is the root, which does depend upon the size of the tree.

The best case for H occurs when the production with parent $\mathcal{N}$ has a rightmost child which is a terminal node; in this case $H = 1$, and the overall algorithm becomes *O(constant)*. It is necessary to look at the grammar to determine the likelihood of this case, by noting the number of productions which end in a terminal node, and their relative frequency in the language.[1]

The worst case for H occurs when all productions in the sub-tree with root $\mathcal{N}$ have length 1; in this case $H = n - 1$, where n is the number of nodes in the tree, and H is $O(n)$. For languages specified with non-ambiguous BNF grammars, the grammar will contain a number of renaming rules, many in high frequency use, such as:.

$$<\text{expression}> \Rightarrow <\text{term}> \Rightarrow <\text{factor}> \Rightarrow <\text{variable}> \Rightarrow <\text{identifier}>.$$

We would like to discover what the average value is for a given grammar. Unfortunately, in general this is a difficult question to answer. Empirical estimates can be made by analyzing collections of programs written in the language, and computing the mean and standard deviation of H and R. However, the choice of programs to include in the study must be carefully made, to arrive at a representative sample.

If the grammar follows a regular pattern, it might lend itself to a more mathematical analysis. Consider for example a grammar in which every production is of length 2. This grammar produces binary parse trees ($R = 2$). The variable H measures the external path length along the right edge of the parse tree. If we assume that the external path length along the right edge of the parse tree is no different than the external path length from the root to any of the terminal nodes, then we can take H to be proportional to the average external path length of these trees. In this case, given a parse tree with n internal nodes, if we further assume sentences will produce complete binary trees (trees with minimum height), we get an average path length of $log_2 n$. If instead we assume that all possible tree constructions with n nodes are equally likely,

[1] For example, the C language grammar in use on the SAGA system contains 288 production rules, of these, 100 have a terminal node as the rightmost child.

we get an average path length of *sqrt(n)*.[2] Unfortunately, grammars for real programming languages are unlikely to produce trees which closely follow either form.

Since LR(1) parsers operate by recognizing production rule handles on the parse stack and reducing them to single node parents, it seems reasonable to expect parse trees to take on an overall shape more similar to a complete tree than to either a degenerate (linear) tree or many of the structures possible given n nodes (although it is certainly possible to construct a grammar with either of these properties). Since a complete tree seems overly optimistic, a value of $log_2 n$ for H is likely to be a good lower bound.

As the value of R increases, where R is based on a weighted (frequency) count of the lengths of the production rules in a grammar, we can expect the average parse tree height H to decrease for a given value of n, the number of nodes in the tree. We can hypothesize that H and R are inversely related to one another, and that a small value of R implies a tree with generally longer external path lengths (an "overhead" factor of non–terminals, in a sense). Table 4–3 presents some measurements of R, both as a simple average of the (unweighted) productions in several grammars, as well as an average based on the frequency of productions found in a set of programs written in the language.

4.3. Providing *leftson* and *right sibling* Attributes.

The four attributes for parse tree linking are the minimum necessary to support the incremental parser. The editor's incremental parser never needs to determine either the leftson or the sibling of a node since the parse tree is built bottom up. However, the editor and other routines which need to perform traversals of a completed parse tree will need to determine this information. An implementor of the parsing algorithm must determine how often *leftson* and *sibling*

[2] [Knuth, 73], section 2.3.4.5 (p. 400).

Language	Number of Production Rules	Average Production Length R Grammar	Average Production Length R in Programs	Ratio of Non–terminals to Terminals
Pascal	217	2.12	1.49	1.76
FP	105	1.35	1.46	2.16
Ada	432	2.48	1.41	2.24
C	288	. 1.94	1.14	7.04

Table 4–3: Average production rule lengths for several SAGA grammars and sets of programs produced by those grammars.

functions are likely to be used. The cost of adding each attribute is the space required to store another link in a parse tree node, and the code to maintain it. The benefit for the *leftson* access is that an operation which previously took somewhere between $O(log_2 n)$ and $O(n)$ running time will take unit time, or $O(1)$, since it is a simple lookup. The benefit for the *sibling* access is that an operation which formerly took *O(constant)* time (a value of 6 time units for $R = 2$) will also take unit time (1 time unit).

Since much longer running times for the traversal programs would be required if the *leftson* and *sibling* functions are used, both a *leftson* and a (right) *sibling* attribute have been added to the parse tree. Due to the similarity between the *sibling* attribute and the *rmost* attribute, the *rmost* attribute has been removed, since it can be calculated by following a sequence of *sibling* pointers. As we will see shortly, the incremental parsing algorithm only accesses the *rmost* attribute during a late phase of the reparse, when it is reparsing a section of the tree previously entered, and then only to avoid shifting nodes in a production which already have their *lthread* fields properly set. By eliminating the *rmost* attribute, we now need to follow a pointer chain to reach the rightmost sibling, but no other work needs to be done (the nodes do not need to have the *shift* operation performed again by the parser, since their links are already correctly set). Because the average production lengths are small, this pointer chain will be short, and little additional work is actually required.

4.4. Linking the Terminal Nodes Together

The editor maintains a text image display of the tree (hiding the internal structure), so it is necessary to be able to efficiently access successive terminal nodes to retrieve the text representation of the token represented by each of these nodes. The initial version of the editor used inherited attributes for the text formatting information and a preorder tree traversal to produce the display, but this scheme had two difficulties. First, the input had to be successfully parsed in order to meaningfully generate the display, and second, the computation time needed to traverse the tree was excessive. By chaining the terminal nodes together into a doubly linked list, and only processing this list in order to produce the display, better response was achieved; in addition, by storing *newline* and spacing information in the terminal nodes, the user's format could be preserved and the display produced whether or not the input could be successfully parsed.[3] (A pretty printer could still be invoked at the conclusion of the parse to reformat the display, if desired.)

Therefore, two additional attributes *prev* and *next* (discussed further in Chapter 5) were added to the parse tree node. With these additional links, the parse tree structure will serve very well to support the functionality required by the editor.

[3] A further improvement in response time was achieved by physically grouping terminal and non-terminal nodes together in memory. This was accomplished by rewriting the node allocation routine to allocate both a block of non-terminal nodes and a block of terminal nodes (it still passed them back one at a time, as new nodes were needed). The editor runs on an operating system with demand paged memory, and its parse tree can also be demand paged within the editor's own data buffers (discussed in Chapter 6). This grouping resulted in fewer page faults when a display was generated, and improved response on a multi-user system since a smaller working set of memory pages was sufficient to support the editor.

4.5. Summary

In this chapter, we have taken a look at the parse tree access required by an editor, investigated some possible parse tree structures, and chosen a structure which is adequate to efficiently support the editor. The net result is that the *leftson, prev* and *next* attributes have been added, and the *rmost* attribute replaced by the *sibling* attribute. In the next chapter, the incremental reparsing algorithm is introduced and extended. More will be said about these modifications at that point, when their impact upon the algorithm is discussed. All of the extensions which were made to the algorithm can still be made independently of the changes made to the attributes of a node, so another implementation could be based on the original parse tree structure together with the *leftson* and *sibling* routines defined in this chapter. We now turn our attention to the incremental LR parsing algorithm, discussed in Chapter 5.

CHAPTER 5

THE INCREMENTAL PARSER

In this chapter, the editor's incremental parser is presented. This parser is based on, and extended from, an incremental LR(0) parsing algorithm by Ghezzi and Mandrioli [Ghezzi and Mandrioli, 80]. *As published*, the algorithm assumes the use of parser tables produced by an LR(0) parser-generator. The input grammar is assumed to be an LR(0) context-free grammar excluding productions with empty right hand sides.

To adapt the algorithm for use with the SAGA editor, a number of extensions were made. Since many programming languages are based on LALR(1) grammars, the algorithm has been extended from LR(0) to LR(1) (also handling LALR(1) and SLR(1) grammars). It also has been extended to support grammars containing productions with empty right hand sides.

We have a new way to handle comments, which eliminates several problems: (1) Restriction of use of comments to limited placement in the language, necessary for syntax-directed template editors. (2) Storage and maintenance of comments in the parse tree. It is a difficult problem to store them in the tree and display them for the user while hiding them from the routines which analyze the syntax of the tree. (3) Uniformity of access by editor commands to both comments and syntactically meaningful tokens in the tree. Comments have been attached to other tokens, not always displayed automatically, and have required additional commands specifically designed to enable them to be edited. We have defined a comment as a lexical class, and modified the parsing algorithm to recognize comments and handle them in an appropriate manner.

We have redefined the *reduce* operation, proposing an alternative which permits the parser to treat non–terminal tokens in a uniform manner with terminal tokens. We also have combined the parsing action and goto function into a single action. Both of these modifications eliminate duplicate code in the incremental parser, improve its efficiency, and permit the editor to pass sub–trees to the parser, as well as lists of terminal tokens.

Explicit error handling actions have been introduced, since a working editor must be able to recover from a user's syntax errors. The original algorithm identifies syntax errors, but states only "Jump to the appropriate error recovery action." While there are several approaches to be taken and a decision to be made about whether to provide automatic error correction, the choice of the best approach for use in an interactive, incrementally parsing editor is not obvious. We have in fact tried a couple different approaches toward the handling of errors before settling upon the current scheme, described below in section 5.9, as the most suitable.

We have altered the attributes associated with the parse tree node proposed in the original statement of the incremental parser, since that structure is not suitable for use with an editor. The alteration of one attribute and the introduction of some additional ones enables the editor's command and display modules to work directly from the parse tree. This eliminates the need to keep an additional text representation and the associated additional complexity that would be required to maintain the consistency between the textual and structural forms of the data.

A parse tree was chosen as the data structure, instead of an abstract syntax tree, since it enables both the editor and the display manager to work directly with the tree. While abstract trees require less space, systems which use them require an *unparser* to reconstruct the original text image for display, and a second data structure to retain this text image for the editor. (In the SAGA editor, a text image of the data actually displayed on the screen is kept by the window management package, but is used only to optimize updates to the display). With abstract trees, the formatting of the displayed text typically is limited, and in some cases no choice is permitted the user. But with a parse tree, the user can format his program any way he pleases, and this information is retained. Pretty–printing programs also can be used to reformat all or any part of the tree in some standardized or customized format.

In this chapter, we describe and present the original Ghezzi and Mandrioli incremental parser, and introduce some additional variables which will help in the subsequent description of our extensions. Then we review the extensions that we made in Chapter 4 to the parse tree structure, discuss their impact upon the parsing algorithm, and present the modifications required to support them. Remaining sections introduce the extensions to the algorithm required to support the editor capabilities mentioned above. Finally, we restate the algorithm at the end of the chapter, with all of the extensions incorporated into it. In Chapter 6, we turn our attention to the editor itself, show how the incremental parser is interfaced to the editor, and discuss the fundamental command capabilities which provide support for both text and structure commands.

5.1. Description of the Algorithm

The original Ghezzi and Mandrioli parsing algorithm is described in this section to give the reader a feel for the operations that occur during incremental reparsing. The algorithm itself is presented in the next section. Following sections then introduce the extensions that have been made. The resulting algorithm, used by the SAGA editor, is presented at the end of this chapter, summarizing the extensions made.

Given a grammar G $=$ (N, Σ, P, S), a terminal string $w = xzy \in$ L(G), with $x, z, y \in \Sigma^*$, and a parse tree $\mathcal{T}$ for w, we wish to substitute the string of tokens $z' \in \Sigma^*$ for z in $\mathcal{T}$, incrementally reconstructing the parse tree $\mathcal{T}'$ for the new string $w' = xz'y$. (Note that any of x, y, z, and z' may be empty, in which case we may have only an insertion or deletion, or an initial parse.) To aid in the description of the algorithm, several variables not present in the original algorithm are introduced. We introduce variables ***activenode, deletecount, nextusernode*** and ***nextnode***. ***Activenode*** points to the terminal node at which an ***editing cursor*** is positioned; all deletions will begin with this node, and all insertions will occur just before it. Thus ***activenode*** is positioned on the node representing the first token in zy. ***Activenode*** is passed to the parser together with ***deletecount***, which is assigned the number of nodes to be deleted (the length of z). ***Nextusernode*** is set to the first symbol in the input string z' to be read by the parser, or ***nil*** if z' is empty. ***Nextnode*** is set to the node corresponding to the first token in y. This variable is initially assigned a pointer to the node ***deletecount*** nodes past ***activenode***. As the parser reads the nodes corresponding to the tokens in y, this variable will be advanced, so it always marks the next node to be read.

Variable ***stacktop***, corresponding to ***top*** in the original algorithm, is set to the node on the top of the parse stack, and ***irmark***, corresponding to ***mark***, to the node on the parse stack just to the left of the first node included in the new sub–tree being constructed by the parser. The ***irmark*** variable will be used later to terminate the reparse; it will be discussed at that time.

5.1.1. Initialization

Let $\mathcal{N}$ be the node at which the editing cursor is positioned (and at which ***activenode*** is set). To perform the initialization, we must restore the state of the parser to that which existed just before the first token of zy was shifted during the previous parse. Since each and every state through which the parser has passed during the previous parse of $\mathcal{T}$ is stored in the ***lthread*** and ***pstate*** attributes associated with the nodes in $\mathcal{T}$, the parse stack can be restored simply by setting ***stacktop*** to the value of ***lthread($\mathcal{N}$)***. The state of the parser is given by the value of *pstate(stacktop).* The reader should be convinced that the view of all of the sub–trees available from the *stacktop* variable is identical to the view that the parser would have obtained had it actually restarted the parse from the beginning of w and proceeded up to this point.

The first time that the parser runs, there is no initial tree; in this case, *stacktop* is set to a *bottom–of–stack* node $\mathcal{B}$, a special parse node whose *pstate* attribute is set to the initial state of the parser. This node serves as a pad token at the bottom of the parse stack, and can only be reached through the *lthread* links in the tree.

In addition, the variable *irmark* is set to the value of *stacktop*, since this node is part of the old tree, and the next node to be shifted will be a part of the new tree being constructed. Finally, the input characters are lexically analyzed by a tokenizing routine, which constructs a linked list of terminal nodes corresponding to the tokens in z', and assigns the first node on this list to *nextusernode.*

5.1.2. Deletion of z

If z is non–empty, it indicates a group of tokens to be deleted. The deletion of z is accomplished by advancing the editing cursor by the number of terminal tokens to be deleted, and setting the *nextnode* variable to the new node to which the editing cursor now points; this node corresponds to the first token of y. Since the terminal nodes in the parse tree corresponding to z lie between the points marked by *stacktop* and *nextnode*, they will be excluded from the new parse tree that will be constructed during this reparse; no further action is required.[1]

[1] It is desirable to place these nodes onto a list of deleted nodes, or onto a "last nodes deleted" list, to support a capability for an *undo* operation. Of course, a *garbage collector* also could be provided to periodically sweep through memory and reclaim those nodes no longer reachable from the root of the parse tree.

5.1.3. Insertion of *z'*

If *z'* is empty, there is no new input, and we immediately skip to the reparsing of *y*, discussed in the following section. Otherwise, ***nextusernode*** points to the list of terminal nodes corresponding to the tokens in *z'* to be inserted, and we proceed as follows. The parsing action *f* is determined using the current parse state ***pstate(stacktop)***. If it is ***shift***, then the node pointed to by ***nextusernode*** is pushed onto the parse stack, and ***nextusernode*** is advanced to the next node in the list.

If *f* is ***reduce***, then a new node is allocated to be the parent of the nodes on the top of the stack which correspond to the right hand side of the production rule; these nodes are popped from the stack. The parse state stored in the node which becomes the top of stack is used to determine the goto function *g*. The parent node is pushed onto the stack, and assigned this state, which becomes the new state of the parser.[2]

If the string *y* in the old tree is empty, or the parser reaches its end, then it is also possible for the parsing action *f* to be ***accept***, in which case the parent node corresponds to the start symbol of the grammar, the indicated reduction is performed, the parser is placed into its final state, and terminates.

If the next input token is invalid in the current parser context, then the parse action is ***error***, and the parse is suspended at this point. Discussion of error handling is deferred until later in the chapter.

Assuming that $w' = xz'y$ is a legal sentence in the language, eventually the parser will shift the last token in *z'*, perform zero or more reductions, and then be ready to shift the first token in

[2] The alternate reduction mentioned earlier in Chapter 3, in which the new parent is prepended to the input stream, will be added as an extension to this algorithm later in this chapter.

y. At this point in the parse, the parsing of *z'* is complete, and we begin the reparse of *y*.

5.1.4. Reparse of *y*

While re–scanning *y*, the parser handles each of the possible parse actions as above, except that it makes some additional checks in an attempt to optimize the reparse, which will save a considerable amount of work. Each time that the parser shifts a node $\mathcal{N}$ corresponding to either a token in *y* or the parent of a token in *y*, it compares the new parse state $g(\mathcal{N})$ against the parse state *pstate*($\mathcal{N}$) the previous time that $\mathcal{N}$ was shifted. Since we are re–scanning *y*, which by definition did not change, if the comparison between these parse states shows the states to be equal, then we are guaranteed that if the parser continues reparsing the subtrees of this node's siblings, that the exact same result will be achieved as before. Therefore, the parser can skip these steps of the analysis, and directly reset its ***stacktop*** variable to be the rightmost sibling of $\mathcal{N}$ (note that the ***lthread*** attribute of the right siblings of $\mathcal{N}$ are all already correctly set, so that this reassignment causes them to appear on the parse stack just as though each had been individually shifted again).

The parser's next action must be to perform a reduction. Let $\mathcal{N}$ now be set to the parent of the node on the top of the stack. If the ***parent*** attribute of each of the children to be included in the reduction is set to point to $\mathcal{N}$, then this node can also be reused. The parser only needs to reset the ***stacktop*** variable to be this node (the ***lthread, rmost, and parent*** links of all of its children are already correctly set), and set the ***lthread*** attribute of $\mathcal{N}$ to the node which is on the top of the stack once the nodes corresponding to the right hand side of this production rule have been popped from the stack.

The parser then repeats this shift/reduce process, comparing the new parse state to the one stored, and continuing to skip steps in the analysis, until it reaches a reduction in which all of the children do not have their *parent* attribute set to $\mathcal{N}$.

At this point, the match condition can be tested, since some of children correspond to elements of, or parents of elements of x and/or z'. If a match is indicated, then the incremental reparse will terminate with this reduction, and our new tree $\mathcal{T}'$ will be complete. If not, a new non-terminal node will be allocated, the reduction performed, and *nextusernode* set to the token following the rightmost descendant of this new sub-tree. When the node to which *nextusernode* now points is shifted, the above parse state comparison can be performed, and the above procedure repeated, until either the match condition does test true, or a reduction is made to the start symbol of the grammar, and the parser accepts the string.

5.1.5. Testing the Match Condition

Whenever the parser is ready to make a reduction while reparsing y, it checks a set of matching conditions to determine whether the parent of the node on the top of the stack can be reused in the reduction that is about to be performed, and whether the left and right edges of the resulting sub-tree mesh cleanly into the structure of the old tree. If these requirements are satisfied, then we are guaranteed that if we do continue the parse beyond this point, all subsequent actions of the parser would be identical to those that were taken when the remainder of the parse tree was last constructed. Therefore, we can terminate the parse at this point, having incrementally produced the tree$\mathcal{T}'$ corresponding to the sentence $w = xz'y$.

Let $A \rightarrow \alpha$ be the reduction that is about to be performed, let *stacktop* be set to the node corresponding to the rightmost token in α, and let $\mathcal{N}$ be the parent of this node, or *nil* if none yet exists (the sub-tree is new). Also, let $\alpha = a_1a_2...a_n$, where $n = |\alpha|$ and $\mathcal{N}_k$ is the node corresponding to a_k, for $1 \leq k \leq n$. Whether the matching condition holds at a given point in the parse tree can be determined by performing the following five tests prior to carrying out the reduction:

(m1) $\mathcal{N} \neq \mathit{nil}$ **and** $\mathit{token}(\mathcal{N}) = A$;
(m2) $\mathit{irmark} = \mathcal{N}_k$, for some k;
(m3) $\mathit{parent}(\mathcal{N}_i) = \mathcal{N}$ for $1 \leq i \leq k$;
(m4) $\mathit{parent}(\mathit{lthread}(\mathcal{N}_1)) \neq \mathcal{N}$;
(m5) $\mathit{rdescend}(\mathcal{N}_n) = \mathit{rdescend}(\mathcal{N})$.

Condition (m1) checks whether the token that labels the parent node $\mathcal{N}$ in the original tree $\mathcal{T}$ is the same as the token A on the left hand side of the production rule about to be used. Clearly, if $\mathcal{N} = \mathit{nil}$, or if these non-terminal tokens do not match, then the new sub-tree about to be produced cannot reuse the node $\mathcal{N}$, and the match condition cannot be satisfied.

Condition (m2) checks that *irmark* points to a node $\mathcal{N}_k$, $1 \leq k \leq n$, which is to be included in the reduction. Recall that *irmark* is always set to the node closest to the top of the stack which existed in the original tree $\mathcal{T}$, and has not yet been included in a new reduction. If *irmark* $= \mathcal{N}_k$, then we know that $\mathcal{N}_k$ existed in the original tree $\mathcal{T}$, and that the newly created nodes which are descendants of the N_j, $1 \leq j \leq n$ mesh correctly into the preexisting nodes in $\mathcal{T}$ to their left. It only remains to be shown that the parent node $\mathcal{N}$ can be cleanly grafted into $\mathcal{T}$, and that the right edge of this new sub-tree fits into that portion of $\mathcal{T}$ to its right.

Nodes $\mathcal{N}_1 \ldots \mathcal{N}_k$ existed originally in $\mathcal{T}$. Condition (m3) checks that the parent of each of these nodes is $\mathcal{N}$. The sub-trees with parent $\mathcal{N}_j$ for $j > k$ are either newly rebuilt or just reparsed by the parser so they either have no parent yet, or their parent reference is not relevant since it is not necessarily based on the original tree $\mathcal{T}$.

Condition (m4) checks that node $\mathcal{N}_1$, which is about to become the leftmost son of $\mathcal{N}$ was previously the leftmost son of $\mathcal{N}$. (The node on the parse stack beneath $\mathcal{N}_1$ must have a *different* parent than $\mathcal{N}$.) If $\mathcal{N}_1$ previously was a son of $\mathcal{N}$, but not its leftmost son, then we will not be

SAGA Notation	Ghezzi & Mandrioli	Description
		Variables
$\mathcal{B}$	$\mathcal{B}$	Node at bottom of parse stack of $\mathcal{T}$
$\mathcal{N}$	$\mathcal{N}$	Node of $\mathcal{T}$
$\mathcal{T}$	$\mathcal{T}$	Threaded parse tree
activenode	*first-y*	First node in *y*
irmark	*mark*	Incremental reparse marker
oldtable	*old-table*	Temporary table value
stacktop	*top*	Top node of the stack
deletecount		Number of tokens to be deleted
nextusernode		Next node in *z'* to be read
nextnode		Next node in *y* to be read
		Attributes of nodes
addr($\mathcal{N}$)	r($\mathcal{N}$)	Address of $\mathcal{N}$
lthread($\mathcal{N}$)	p($\mathcal{N}$)	Next in pushdown list after $\mathcal{N}$
parent($\mathcal{N}$)	F($\mathcal{N}$)	Father of $\mathcal{N}$
pstate($\mathcal{N}$)	t($\mathcal{N}$)	LR table of $\mathcal{N}$ (parse state)
rdescend($\mathcal{N}$)	rd($\mathcal{N}$)	Rightmost descendant of $\mathcal{N}$
rmost($\mathcal{N}$)	rb($\mathcal{N}$)	Rightmost brother of $\mathcal{N}$
token($\mathcal{N}$)	v($\mathcal{N}$)	Element of V in $\mathcal{N}$
leftson($\mathcal{N}$)		Leftmost son of non-terminal $\mathcal{N}$
next($\mathcal{N}$)		Next terminal in tree after $\mathcal{N}$
prev($\mathcal{N}$)		Previous terminal before $\mathcal{N}$
sibling($\mathcal{N}$)		Next sibling to right of $\mathcal{N}$
		Functions
alloc()	*take()*	Allocate a new node
apply_match()	*apply-matching()*	Graft new tree into old
f()	*f*	Current parsing action
g()	*g*	Current goto function
matchcond()	*matching-condition()*	Can new tree be grafted into old tree at current spot?
reduce(*i*, $\mathcal{N}$)	*apply-reduction*(*i*, $\mathcal{N}$)	Reduce by production *i*, making $\mathcal{N}$ the parent node.
shift($\mathcal{N}$)	*apply-shift*($\mathcal{N}$)	Shift $\mathcal{N}$ onto stack
shift($\mathcal{N}$, *i*)	*apply-shift*($\mathcal{N}$)	Shift $\mathcal{N}$ onto stack, go to parse state *i* (replaces *shift* above)
stack($\mathcal{N}$, *j*)	$p^j(\mathcal{N})$	$p^0(\mathcal{N}) = \mathcal{N}$ $p^1(\mathcal{N}) = lthread(\mathcal{N})$ $p^j(\mathcal{N}) = p^1(p^{j-1}(\mathcal{N}))$
action(...)		Combined *f* and *g* functions
chain($\mathcal{N}$, ...)		Link $\mathcal{N}$ into *next, prev* list
nextsym()		Next node to be read by parser.
unchain($\mathcal{N}$)		Unlink $\mathcal{N}$ from *next, prev* list

Table 5–1: Notation used in the description of the LR(0) incremental reparsing algorithm.

able to terminate the parse at $\mathcal{N}$ since it would leave the original leftson of $\mathcal{N}$ as a dangling node within the tree, which would no longer be well-structured.

Lastly, condition (m5) checks that the rightmost descendant of $\mathcal{N}_n$ matches the rightmost descendant of the parent node $\mathcal{N}$, to guarantee that the newly created sub-tree which is re-using pre-existing parent node $\mathcal{N}$ has the same right edge as the sub-tree rooted in $\mathcal{N}$. If these nodes do not match, then we cannot terminate the parse at $\mathcal{N}$, since some nodes will be left dangling where the right edge of the new-subtree meets the old tree, and the tree will not be well-structured.

If conditions (m1) through (m5) are all true, then $\mathcal{N}$ is re-used with new children $\mathcal{N}_1 \dots \mathcal{N}_n$. This newly created sub-tree is unified with that part of the original tree $\mathcal{T}$ which remains to produce $\mathcal{T}$', and the parser terminates, accepting the new input.

5.2. Algorithm 5.1: The Ghezzi and Mandrioli Incremental LR(0) Parser

Having described the algorithm, we now include the actual algorithm in this section. This is Ghezzi and Mandrioli's LR(0) algorithm as published, but described using different terms. Table 5–1 gives the correspondence between the terms used here and those in the originally published paper. The different notation is used in part to provide longer, more mnemonic names for the attributes of the parse tree nodes, and to permit the algorithm to be described in terms that match the code used in the actual implementation. Curly braces are introduced for grouping, to make the algorithm more readable.

5.2.1. Routines used in the Parser

```
alloc():
        N ← alloc();
        addr(N) ← N;
        rdescend(N) ← N;
        return(N).
```

```
apply_match:
        Let $A \rightarrow \alpha$ be the reduction for which the matching condition holds.

        $parent(stack(stacktop, j)) \leftarrow parent(irmark), \forall 0 \leq j < |\alpha|$;
        $rmost(stack(stacktop, j) \leftarrow stacktop, \forall 0 \leq j < |\alpha|$.

matchcond():
        Let $A \rightarrow \alpha$ be the reduction to be applied.

        if $irmark = stack(stacktop, j)$, for some $0 \leq j < |\alpha|$
        and $parent(irmark) = parent(stack(irmark, h)) \forall 0 \leq h < |\alpha| - j$
        and $parent(irmark) \neq parent(stack(irmark, |\alpha| - j))$
        and $token(parent(irmark)) = A$
        and $rdescend(stacktop) = rdescend(parent(irmark))$
        then
                $matchcond \leftarrow true$;
        else
                $matchcond \leftarrow false$.

reduce(i, $\mathcal{N}$):
        Let $i$ be production $A \rightarrow \alpha$.

        $parent(stack(stacktop, j)) \leftarrow addr(\mathcal{N}), \forall 0 \leq j < |\alpha|$;
        $rmost(stack(stacktop, j)) \leftarrow stacktop, \forall 0 \leq j < |\alpha|$;
        $token(\mathcal{N}) \leftarrow A$;
        let $g$ be the goto function of $pstate(stack(stacktop, |\alpha|))$;
        $pstate(\mathcal{N}) \leftarrow g(A)$;
        $rdescend(\mathcal{N}) \leftarrow rdescend(stacktop)$;
        $stacktop \leftarrow addr(\mathcal{N})$.

shift($\mathcal{N}$):
        Let $g$ be the goto function of $pstate(stacktop)$.

        $lthread(\mathcal{N}) \leftarrow stacktop$;
        $pstate(\mathcal{N}) \leftarrow g(token(\mathcal{N}))$;
        $stacktop \leftarrow addr(\mathcal{N})$.
```

5.2.2. The Parser

Let $\mathcal{T}$ be the parse tree for the string $w = xzy$.
Let z' be a replacement string for z, and $w' = xz'y$ the result.

1. Initialization

```
(1.1)   if $w \neq e$ (the empty string), then {
                let $\mathcal{N}$ be the node in $\mathcal{T}$ which stores the first symbol of $zy$;

                $irmark \leftarrow lthread(\mathcal{N})$;
                $stacktop \leftarrow lthread(\mathcal{N})$;
        }.
```

```
(1.2)  if $w = e$ (i.e., $w'$ is being parsed from scratch) then {
            irmark ← $\mathcal{B}$;
            stacktop ← $\mathcal{B}$;
       }.
```

2. Analysis of *z'*

```
(2.1)  Let $f$ be the parsing action of pstate(stacktop).
       Execute (a), (b), (c), or (d) according to the value of $f$.

(a)    $f = SHIFT$.
       if the symbol to be shifted is first(y) then
            jump to (3);
       else {
            $\mathcal{N}$ ← alloc();
            name($\mathcal{N}$) ← next-symbol-from-the-input;
            shift($\mathcal{N}$);
       }.
(b)    $f = REDUCE\ i$. Let $i$ be the production $A \rightarrow \alpha$.
       if irmark = stack($\mathcal{N}$, j) for some $0 <= j < |\alpha|$ (i.e., irmark must be updated)
       then
            irmark ← stack($\mathcal{N}$, $|\alpha|$).
       $\mathcal{N}$ ← alloc();
       reduce(i, $\mathcal{N}$).

(c)    $f = ERROR$.
       Jump to the appropriate error recovery action.

(d)    $f = ACCEPT$.
       The algorithm terminates, accepting the string so far scanned.
```

3. Analysis of *y*

```
(3.1)  Let $\mathcal{N}$ be the node which stores the first symbol of $y$.

       oldtable ← pstate($\mathcal{N}$);
       shift($\mathcal{N}$);

(3.2)  if oldtable $\neq$ pstate(stacktop) then
            jump to 3.3;

       Otherwise, skip steps of the analysis of $y$ as follows:
       stacktop ← rmost(stacktop) (we enter directly in a reduction state).
       Let $f$ be the parsing action of pstate(stacktop), where $f = REDUCE\ i$, $i$ being
            production $A \rightarrow \alpha$.

       if matchcond holds then {
            apply_match;
            accept $w'$, terminating the algorithm.
       }
       if irmark = stack(stacktop, j) for some $0 <= j < |\alpha|$ then
            irmark ← stack(stacktop, $|\alpha|$);
```

```
          oldtable ← pstate(parent(stacktop));
          if parent(stack(stacktop, j)) = parent(stack(stacktop, k)) $\forall$ 0 <= j, k < $|\alpha|$ then
      {
                 the entire subtree of $\mathcal{T}$ rooted in parent(stacktop) is reused:
                 $\mathcal{N}$ ← parent(stacktop);
                 lthread($\mathcal{N}$) ← stack(stacktop, $|\alpha|$).

                 Let g be the goto function of pstate(stack(stacktop, $|\alpha|$)).
                 pstate($\mathcal{N}$) ← g(A).
          } else {
                 a new node is allocated:
                 $\mathcal{N}$ ← alloc();
                 reduce(i, $\mathcal{N}$);
          }
          Jump to 3.2.

(3.3)     Let f be the parsing action of pstate(stacktop).
          Execute (a), (b), (c), or (d) according to the value of f.

(a)       f = SHIFT. Let $\mathcal{N}$ be the node corresponding to the next symbol of y.
          oldtable ← pstate($\mathcal{N}$);
          shift($\mathcal{N}$);
          jump to 3.2.

(b)       f = REDUCE i. Let i be production $A \rightarrow \alpha$;
          if matchcond holds then {
                 apply_match;
                 accept w', terminating the algorithm.
          }
          if irmark = stack(stacktop, j) for some 0 <= j < $|\alpha|$ then
                 irmark ← stack(stacktop, $|\alpha|$);
          $\mathcal{N}$ ← alloc();
          reduce(i, $\mathcal{N}$);
          jump to 3.3.

(c)       f = ERROR.
          Jump to the appropriate error recovery action.

(d)       f = ACCEPT.
          The algorithm terminates.
```

5.3. Modifications to the Parse Tree Node

In the previous chapter, we decided that parse tree access for other software tools would be improved by modifying some of the attributes of a parse tree node. In addition, parse tree access for the editor is improved if some additional attributes are also added to each parse tree node. In this section we discuss these alterations.

5.3.1. Addition of a *leftson* Attribute

A *leftson* attribute has been added to each non–terminal parse tree node, so that each non–terminal node now contains a pointer to its leftmost child. The *leftson* attribute is not required by the incremental parser, so the addition of one has no effect on the operation of the algorithm since this attribute is never referenced by it. The existence of this attribute requires only one additional assignment in the *reduce()* routine of the algorithm, to set the *lthread* attribute of the parent node to its leftmost child at the time that the reduction is being performed. This is accomplished by including the following statement in this routine just after the *token* attribute is set:

$$lthread(\mathcal{N}) \leftarrow stack(stacktop, |\alpha| - 1);$$

This assignment is the only one necessary since the only time the relationship between a parent and its children changes is during a reduction. All reductions occur in the *reduce* routine, with one exception, in *apply_match*, when the parent node is re–used in performing the final reduction which terminates the reparse. In this situation, for the match condition to test *true*, the *irmark* variable must point to one of the nodes within this production. This can only occur if the node existed in the original tree, so the value of the *leftson* attribute will already be correctly set, and no further action is necessary.

5.3.2. Replacement of *rmost* by the *sibling* Attribute

The *rmost* attribute has been replaced by the *sibling* attribute. The *rmost* attribute contained a pointer to the rightmost brother of a node in a production; the new *sibling* attribute contains a pointer to the sibling to the immediate right of each node, or *nil* if the node is itself a rightmost sibling. The replacement of the *rmost* attribute by the *sibling* attribute does slightly affect the parsing algorithm in that wherever the *rmost* attribute had been referenced, it is now necessary to traverse the *sibling* links to reach the rightmost brother of that node. This occurs in only one location, in section (3.2) of the algorithm, in the reparse optimization section. The reference to *rmost* is made in order to obtain the rightmost sibling of a production which the parser has decided need not be reparsed since it would have an identical outcome. The production is entered onto the parse stack simply by setting *stacktop* to the value of this attribute. The same result is obtained using the *sibling* attribute, if the original statement in section (3.2):

```
stacktop ← rmost(stacktop)
```

is replaced by:

```
while sibling(stacktop) ≠ nil do
    stacktop ← sibling(stacktop);
```

While it may appear that the replacement of a simple assignment statement by a loop may slow the algorithm, our analysis at the end of chapter 4 showed that the average production rule length *R* in sample parse trees tends to be approximately *2* or less, depending on the grammar, so that the additional work required is indeed small.

To maintain the *rmost* attribute, assignment statements were required in both *apply_match()* and *reduce()*, where the *rmost* attribute previously was set. The assignment statement:

```
rmost(stack(stacktop, j)) ← stacktop, $\forall 0 \leq j < |\alpha|$
```

is replaced in each instance by:

```
sibling(stack(stacktop, j)) ← stack(stacktop, j - 1), $\forall 0 < j < |\alpha|$;
sibling(stacktop) ← nil.
```

The same number of assignments to the *sibling* attribute of these nodes is required as before; only the value of the assignment is different, since each node now receives the address of the sibling to its immediate right, instead of the rightmost sibling. It should be clear that by following this list of pointers, *stacktop* will always be set to the rightmost sibling at the conclusion of this loop.

5.3.3. Chaining the Terminal Nodes Together

Two other attributes have been added to the parse tree node: the *next* and *prev* attributes. These are used to chain the terminal leaves of the parse tree together into a doubly-linked list. While these attributes are not necessary for the parsing algorithm, they are of great utility to the editor in producing the display and executing user commands. Since the parse tree is constructed and maintained by the incremental parsing algorithm, the maintenance of these additional attributes is best done by the algorithm. We add two new routines, *chain* and *unchain*, which add and remove nodes from this doubly-linked list:

```
chain($\mathcal{N}$, at, $\mathcal{M}$):
    Let $\mathcal{M}$ be a node in the frontier of $\mathcal{T}$, $\mathcal{N}$ a node to be added,
        and at be one of BEFORE or AFTER.
    if at = BEFORE then {
        next($\mathcal{N}$) = addr($\mathcal{M}$);
        prev($\mathcal{N}$) = prev($\mathcal{M}$);
        if prev($\mathcal{M}$) $\neq$ nil then
            next(prev($\mathcal{M}$)) = addr($\mathcal{N}$);
        prev($\mathcal{M}$) = addr($\mathcal{N}$);
    }
    if at = AFTER then {
        next($\mathcal{N}$) = next($\mathcal{M}$);
        prev($\mathcal{N}$) = addr($\mathcal{M}$);
        if next($\mathcal{M}$) $\neq$ nil then
            prev(next($\mathcal{M}$)) = addr($\mathcal{N}$);
        next($\mathcal{M}$) = addr($\mathcal{N}$);
    }.

unchain($\mathcal{N}$):
    prev(next($\mathcal{N}$)) = prev($\mathcal{N}$);
    next(prev($\mathcal{N}$)) = next($\mathcal{N}$).
```

The *chain* routine only needs to be called from one location in the parser, at the point that a node corresponding to a token in z' is shifted by the parser. Since the nodes corresponding to the tokens in both x and y previously existed in $\mathcal{T}$, their *next* and *prev* attributes are already correctly set. The *unchain* command only needs to be called during parser re–initialization to remove each of the terminal nodes associated with the tokens of *z* which are being deleted. Other calls will be needed for error handling, but these calls will be discussed later, when this topic is presented.

5.4. Extension of the Algorithm to LR(1)

Since programming languages of interest are easily expressible using LR(1) grammars, the algorithm is extended to include this class of context–free grammars. Almost all of the added complexity caused by this extension affects the parser–generator program itself, since different algorithms need to be used to produce the parse tables. But once produced, the difference for the incremental parsing algorithm is that the parsing action becomes a product of both the state of the parser and the next symbol from the input. Since these parse table generation algorithms are well understood, and parser–generators for this class of grammars have been written and are commonly available, they will not be covered here.

To implement this extension in the parsing algorithm, We extend both *f* and *g* so that they depend upon both of these parameters. We introduce a function, *nextsym*, which returns the node corresponding to the next input token and advances *nextusernode.* If there are no new input nodes left, then the node in *y* to which *nextnode* points is returned, and *nextnode* is advanced. If *nextnode* is *nil*, then a node corresponding to the end–of–file token is generated and returned. The only time this occurs is during the very first parse of a new file; in all other invocations of the parser, the last node in the list headed by *nextnode* will be the end–of–file token, which the parser will never go past. If the end–of–file token is legal, then the parser will receive an *accept* action before a new node is needed; if not, then an unexpected end–of–file error will occur, and the parser will suspend at this point. Thus, *nextsym* is defined as:

```
nextsym():
    Let $\mathcal{N}$ be a pointer to a parse tree node.
    if nextusernode $\neq$ nil then {
        $\mathcal{N} \leftarrow$ nextusernode;
        nextusernode $\leftarrow$ next(nextusernode);
    } else if nextnode $\neq$ nil then {
        $\mathcal{N} \leftarrow$ nextnode;
        nextnode $\leftarrow$ next(nextnode);
    } else {
        $\mathcal{N} \leftarrow$ alloc();
        token($\mathcal{N}$) $\leftarrow$ eof;  (the end-of-file token code)
    }
    return($\mathcal{N}$).
```

This extension requires a change to the algorithm, as follows; wherever

Let *f* be the parsing action of *pstate(stacktop)*

appears, it is replaced by:

Let *f* be the parsing action of *pstate(stacktop)* and *nextsym()*.

Wherever

Let *g* be the *goto function* of *pstate(...)*

appears, it is replaced by:

Let *g* be the *goto function* of *pstate(...)* and *nextsym()*.

Modifications to the incremental parsing algorithm occur in part (2.1), the end of part (3.2), part (3.3) and in the routines *shift* and *reduce*.

The major change to the parser occurs in the initialization section, since it is no longer sufficient to initialize *stacktop* the stack pointer contained in *activenode*. Because of the lookahead requirement, it may now happen that the parse action previously taken on the node preceding *activenode* will differ from the action that will be taken during this parse, since *activenode* will not necessarily be the lookahead token this time; the first token in *z'* will be instead. Therefore, the parser must back up one token (since the grammar is LR(1)) and then reset the parser variables using this node instead. Section (1.1) must be altered to reset $\mathcal{N}$ to *prev($\mathcal{N}$)* before any of the assignments are made. Thus, section (1.1) becomes:

```
(1.1) if $w \neq e$ (the empty string), then {
          let $\mathcal{N}$ be the node in $\mathcal{T}$ which stores the first symbol of $zy$;
          $\mathcal{N} \leftarrow prev(\mathcal{N})$;
          $irmark \leftarrow lthread(\mathcal{N})$;
          $stacktop \leftarrow lthread(\mathcal{N})$;
      }.
```

5.5. Redefinition of the *reduce* Operation

To permit the editor to pass non-terminal nodes to the parser in the ***nextusernode*** list (corresponding to tokens in z'), and to permit the f and g functions to be combined (covered in the next section), the ***reduce*** action is redefined to prepend the parent node onto the input instead of placing it on the top of the parse stack. Routines ***nextsym, reduce*** and ***shift*** need to be modified, and a new routine, ***prepend***, added to place the new parent node at the head of the input list.

Since the parent node which is to be prepended to the input list will immediately be shifted during the next step of the parser, it is sufficient to define a new variable ***savenode*** to retain this node, and add an initial test to the ***nextsym*** routine to return the node assigned to ***savenode*** if it is non-***nil***, setting ***savenode*** to ***nil*** when this is done. The new routine ***prepend*** is defined as follows:

```
prepend($\mathcal{N}$):
    $savenode \leftarrow \mathcal{N}$.
```

The new test, placed at the beginning of ***nextsym***, is

```
if $savenode \neq nil$ then {
    $\mathcal{N} \leftarrow savenode$;
    $savenode \leftarrow nil$;
} else ...
```

Several lines of code at the end of the *reduce* routine, which place $\mathcal{N}$ onto the parse stack, are deleted. These lines are:

```
**let** $g$ be the goto function of $pstate(stack(stacktop, |\alpha|))$;
$pstate(\mathcal{N}) \leftarrow g(A)$;
$stacktop \leftarrow addr(\mathcal{N})$.
```

This modification simplifies the parsing algorithm, since redundant code for shifting nodes is removed from *reduce*.

5.6. Combining the f and g Parsing Functions

At each step in an incremental reparse, the parser first determines the parsing action f and then a short time later the goto function g. We have redefined the reduction action as a reduction by a given production rule number, with the resulting parent node prepended to the input rather than immediately placed on top of the parse stack. With this redefinition, g now depends upon the current parse state and the head of the input just as f has always done. The goto function in a reduction, which previously depended upon the parse state uncovered in the parse stack has become the goto function of a shift action of a non–terminal symbol.

This uniformity permits us to combine the f and g functions into a single *action* routine. This is a desirable alteration, since parse tables usually code both the action and new state or rule number together in a single entry. By retrieving both with a single call, we eliminate a duplicate lookup that would otherwise have to occur at each step in the parse.

The *action* routine takes as arguments the current parse state and input symbol, as f and g did before, and returns two values. In the case of a *shift* or *accept* action, these new values correspond to the old values returned by f and g, namely the action and the new parse state. In the case of a *reduce* action, the second value is assigned the production rule number by which the reduction is to be made. In other cases, this second value is unused.

This change is incorporated into the incremental parsing algorithm by modifying sections (2.1), (3.2) and (3.3). The introductory line:

Let *f* be the parsing action of *pstate(stacktop)* and *nextsym()*

is replaced by an explicit call to the *action* routine:

action(pstate(stacktop), nextsym(), f, newvalue)

where *f* now becomes a variable assigned by *action*, and the variable in the position of *newvalue* above is assigned either the new parse state or a production rule number, according to the value of *f*.

The *shift* routine must also be modified to pass in the new parse state, now contained in *newvalue*, since it is no longer necessary to call *g* from within it. The call to *g* is replaced by the value of this variable. No further modifications are necessary, and it should be evident that the algorithm still runs as before since its flow is still the same; only the form in which this information is passed has changed. The significance of this change is that the efficiency of the algorithm is improved, and we gain the ability to pass an intact sub-tree to the parser.

5.7. Extension to Support Grammars with *epsilon* Productions

The parsing theory for LR(1) context-free grammars is well developed, and *epsilon* productions (productions with empty right hand sides) are well understood. While any grammar containing epsilon productions can be represented by an equivalent grammar with none [Hopcroft and Ullman, 69], it is much more convenient for the language implementor to be able to use epsilon rules in his specification.

The addition of epsilon rules adds some complexity to the algorithm. First, their representation in the tree must be decided. Two approaches are common: the first places *nil* pointers into the parents of epsilon productions; the second represents the empty right hand side with an epsilon terminal node, and adds a token code to the terminal vocabulary of the grammar. The second method has been chosen for the SAGA editor, since it results in a more uniform parse tree. All non-terminal nodes always have at least one child, and when descending through the tree toward the frontier, one is guaranteed to eventually reach a terminal node.

The initialization of the algorithm is affected, since it is no longer sufficient to use *prev(activenode)* to initialize the parser. Any section of the frontier can contain an unlimited sequence of epsilon nodes, depending on the form of the grammar in use. Therefore, it is necessary to check the token type of the preceding node and if it is an epsilon token, to continue traversal back along the *prev* links. Since the stack is finite, either a non–epsilon terminal token, or the *bottom-of-stack* node $\mathcal{B}$ will eventually be reached. This node then can be used to initialize the parser. In the initialization of the parser, part (1.1) is replaced by the following code:

```
(1.1)  if w $\neq \epsilon$, then {
            let $\mathcal{N}$ be the node in $\mathcal{T}$ which stores the first symbol of $zy$;

            $\mathcal{N} \leftarrow prev(\mathcal{N})$;
            while $token(\mathcal{N}) = \epsilon$ do
                  $\mathcal{N} \leftarrow prev(\mathcal{N})$;
            $irmark \leftarrow lthread(\mathcal{N})$;
            $stacktop \leftarrow lthread(\mathcal{N})$;
       }.
```

Part (2.1)(b) is affected, since in the production $A \rightarrow \alpha$, α can now be of length zero. In this case, we shift an epsilon terminal node onto the stack, and then perform a reduction, using a length of *1* for the rule instead of *0*. Replace "$\mathcal{N} \leftarrow$ *alloc()*; *reduce*(i, $\mathcal{N}$)" in (2.1)(b) with:

```
(2.1)(b)
       ...
       if $|\alpha| > 0$ then {
             $\mathcal{N} \leftarrow alloc()$;
             $reduce(i, \mathcal{N})$;
       } else {
             $\mathcal{N} \leftarrow alloc()$;
             $token(\mathcal{N}) \leftarrow \epsilon$;
             $shift(\mathcal{N}, -1)$;
```

```
        $\mathcal{N} \leftarrow$ alloc();
        reduce(i, $\mathcal{N}$);
    }.
```

The *shift* routine is passed an unused parse state code, in this case *-1*, for assignment to the pstate field, since the epsilon token has no goto function. The *reduce* routine must be modified to check whether $|\alpha| = 0$, and if so to assume that $|\alpha| = 1$ instead, since an epsilon node now resides on the stack. The *matchcond* routine needs an identical check, although the test will always fail when called with the parent of an epsilon node, since a production must have length > 1 in order to pass all of the tests.

A modification to *nextsym* is also required, to test for an epsilon token and delete it from the list. A **while** loop suffices, which will continue testing tokens until one is found which is not an epsilon token. Because the editor produces an initial parse tree during initialization on a new file, the last token in the *nextnode* list will always be the *end-of-file* token, so this loop will always succeed in locating a non-epsilon token.

In *nextsym*, the following code fragment:

```
    ...
    } else if nextnode $\neq$ nil then {
        $\mathcal{N} \leftarrow$ nextnode;
        nextnode $\leftarrow$ next(nextnode);
    } else {
    ...
```

is replaced by:

```
    ...
    } else if nextnode $\neq$ nil then {
        while nodetype(nextnode) $= \epsilon$ do
            nextnode $\leftarrow$ next(nextnode);
        $\mathcal{N} \leftarrow$ nextnode;
        nextnode $\leftarrow$ next(nextnode);
    } else {
    ...
```

It should be evident that any epsilon tokens existing in the original tree, after the point of the modification, which are reached by the parser will be removed, and that the parser will receive the correct lookahead token. If the epsilon tokens should be retained in the new tree, the parser will produce new nodes as necessary, when directed by the *action* routine to perform reductions in which the length of the right hand side of the production rule is zero.

No other modifications are needed to support epsilon rules. We now turn our attention to comment tokens and their handling.

5.8. Extension to Support Comments

Providing support for comments is one of the more challenging tasks for language-based editors. Programming languages which include comments typically permit them to appear between any two tokens in the input; some, such as the C programming language [Kernighan and Ritchie, 78], also permit them even within tokens, between any two characters. This flexibility is easy for a batch compiler to support, since all comments are stripped out of the input and discarded as soon as they are read, and do not affect further processing of the input data. But language-based editors do not have this option, since they are expected to retain and display a user's comments along with his program text. Unfortunately, while a lexical class for comments is easily definable, incorporating a comment token into the production rules of a grammar is usually not possible; if all of the permissible locations for comments in the language are specified in the grammar, it becomes ambiguous and cannot be successfully processed by a parser-generating system. An alternate method of handling comments needs to be used.

Some syntax-directed editors solve this problem by restricting the locations at which comments can appear so that comment tokens can be specified in the formal description of the language [Teitelbaum and Reps, 81]. Comments are required in certain locations, such as preceding procedure declarations, permitted in others, and prohibited in most remaining ones. Restrictions such as these are often justified by the implementors by stating that they are producing a structured environment, that the use of comments is to be encouraged, and that by limiting them to a few key locations, they are encouraging a more standardized development style.

Other editors [Horton, 81] construct comment tokens and attach them to a nearby terminal token in the parse tree. This has the advantage of hiding the comment from the parser, but the disadvantage of forcing the comment to be treated as an attribute of a neighboring node, when no such relationship necessarily exists. There is also the added problem of deciding whether to attach the comment to the preceding or following parse tree node. This is often solved either by picking one by default and letting the user override the choice, or prompting the user for the node to use each time he enters a comment. This choice is not simple: a comment documenting a routine is usually placed before the routine in the file, immediately preceding the first token in the routine, while a comment documenting a variable declaration is usually placed after the declaration (and any trailing punctuation that may be present). Trying to determine the node to which the comment should be attached based on the surrounding context can be attempted if language–dependent information is used, but suffers the difficulty of not knowing where to place the comment when a syntax error occurs.

The SAGA editor uses a third, and new, approach. Comments are tokenized by the lexical analyzer and allocated their own terminal node, one per comment. These nodes are attached to the parse tree along the *prev/next* doubly–linked list of terminal nodes in the parser. Each time a comment token is detected in the input, it is linked into this list, and the following token is retrieved from the input to be passed to the *action* routine, so that it never encounters a comment token, and the parse tables do not need entries for comments. Since the *prev/next* list is not used by the algorithm, once the comment tokens are in the tree, they are never seen again by the parser. Even walking the tree in the traditional way from the root will not discover any comment tokens in the tree, so that the routines which only are concerned only with syntactic structure need not be modified to process comments, even though comments can in fact occur between any two tokens in the parse tree. In subsequent editing, the comment will be included in the operation being performed if it is selected by the user, and not otherwise. Routines which need to process comments while walking the tree can do so by checking the *next* attribute of each terminal node they encounter, and testing the *token* attribute of this node.

Many programming languages permit single comments to span more than one line. While the comment text could be stored in one long string and a single node allocated for the entire comment, this representation is not convenient for the routines which must track the position of the editing cursor as it moves past such comments. Therefore, multi–line comments are represented by a *comment tree* of unit height, in which the text of each separate line of the multi–line comment is stored separately, and allocated its own terminal node. A single non–terminal node is allocated to be the parent of all of these terminal nodes. This parent token is linked into the *prev/next* terminal list in the parse tree, so that the comment is represented by a single token. At the same time, by accessing the children of this node, information about the formatting of the comment across lines can be obtained without needing to actually read the text string itself, making the calculations for editing cursor positioning more efficient.

The major change to the parsing algorithm is to the lexical analyzer, which must recognize a multi–line comment and construct the tree described above. Section (1.1) of the initialization must also be modified to back along the frontier past comment tokens as well as epsilon tokens. The while loop becomes:

```
while $token(\mathcal{N}) = \epsilon$ or $token(\mathcal{N}) = commentcode$ do
    $\mathcal{N} \leftarrow prev(\mathcal{N})$;
```

An additional parse action, *noaction*, is added to the possible parse actions which can be taken by the parser. When a comment token is detected, the parser takes this parsing action instead of passing the parse state and comment to the *action* routine for lookup in the parse tables. If the parser is parsing z', this action causes the node representing the token to be chained into the terminal list of the parse tree; if y is being reparsed, then no action is required, and the parser simply moves on to the next token.

Specifically, both sections (2.1) and (3.3) of the parsing algorithm need to be modified to add a fifth parse action (e), in which f is *noaction*. In section (2.1) only, a call is made to the *chain* routine to insert the comment node into the terminal list.

5.9. Extension to Support Exception Handling for Errors

Error handling is a difficult issue, and one which significantly complicates the parsing algorithm. Many syntax-directed editors avoid the issue by limiting the user to operations which permit only a correct program to be produced. But these limitations are overly restrictive, making many simple modifications tedious. By permitting a user to make changes which take a program through *intermediate incorrect* states, much more flexible editing becomes possible. The SAGA editor has followed this approach.

The first question which arises in error handling is whether to provide error *correction*, or error *recovery*. Error *correction* can simplify the implementation, since a trap-door error recovery mechanism can be used to restore a correct environment and permit the parser to continue to completion. However, the correction method used often restructures the input into a different form than the user intended, and can create more work for the user to restore his correct input from the system-corrected result than if he simply fixed the original error. Error *recovery* does not repair the error automatically, but permits the editor to continue in operation

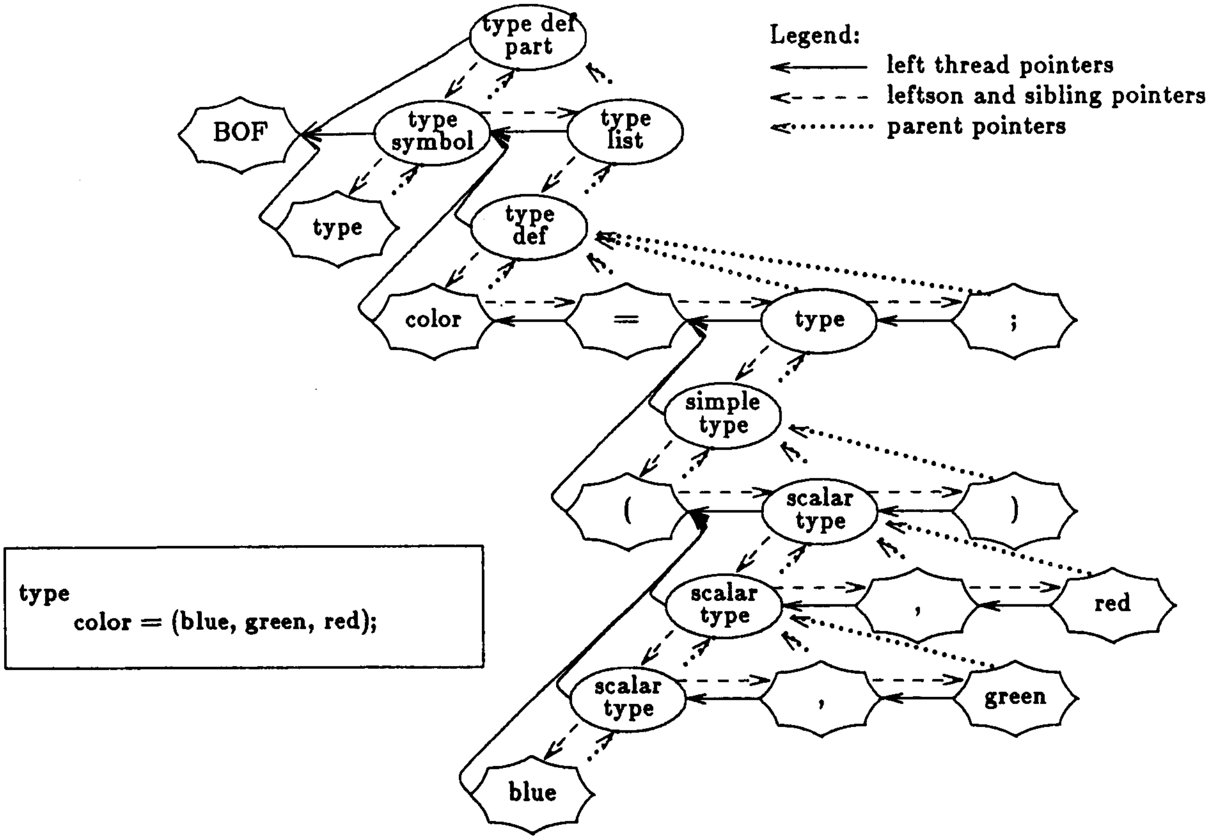
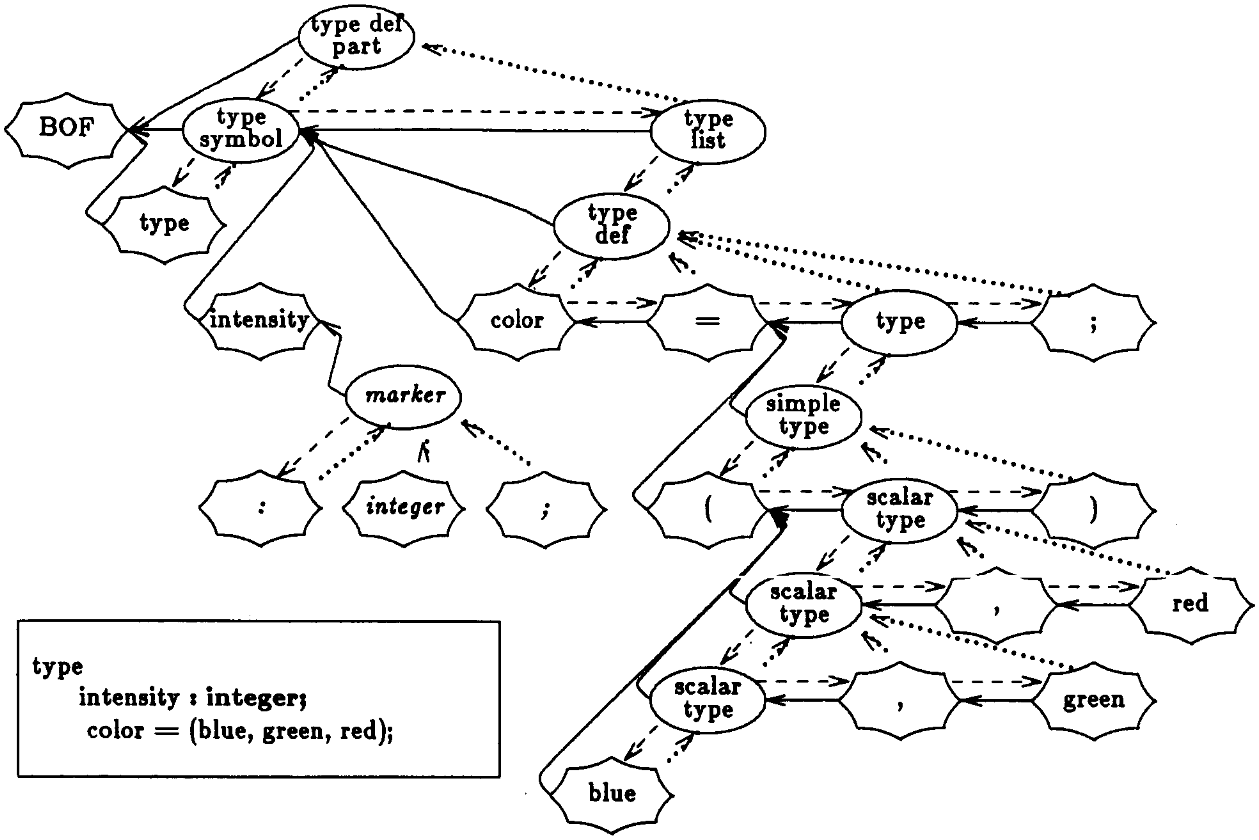


Figure 5-1: Parse tree produced by the insertion of "type color = (blue, green, red);". The rectangle gives the display on the user's terminal. In addition to the links shown, each node is also linked to its rightmost descendant, and each terminal node is contained in a doubly-linked list connecting it to the immediately preceding and following terminal nodes along the frontier of the parse tree. To avoid clutter, these links have been omitted since they can be determined by inspection of the parse tree.

until some later time when the user can repair the error himself. To recover from an error, the implementation must be able to save the state of the parse and local tree structure for later continuation, suspend the parse, and return to the editor.

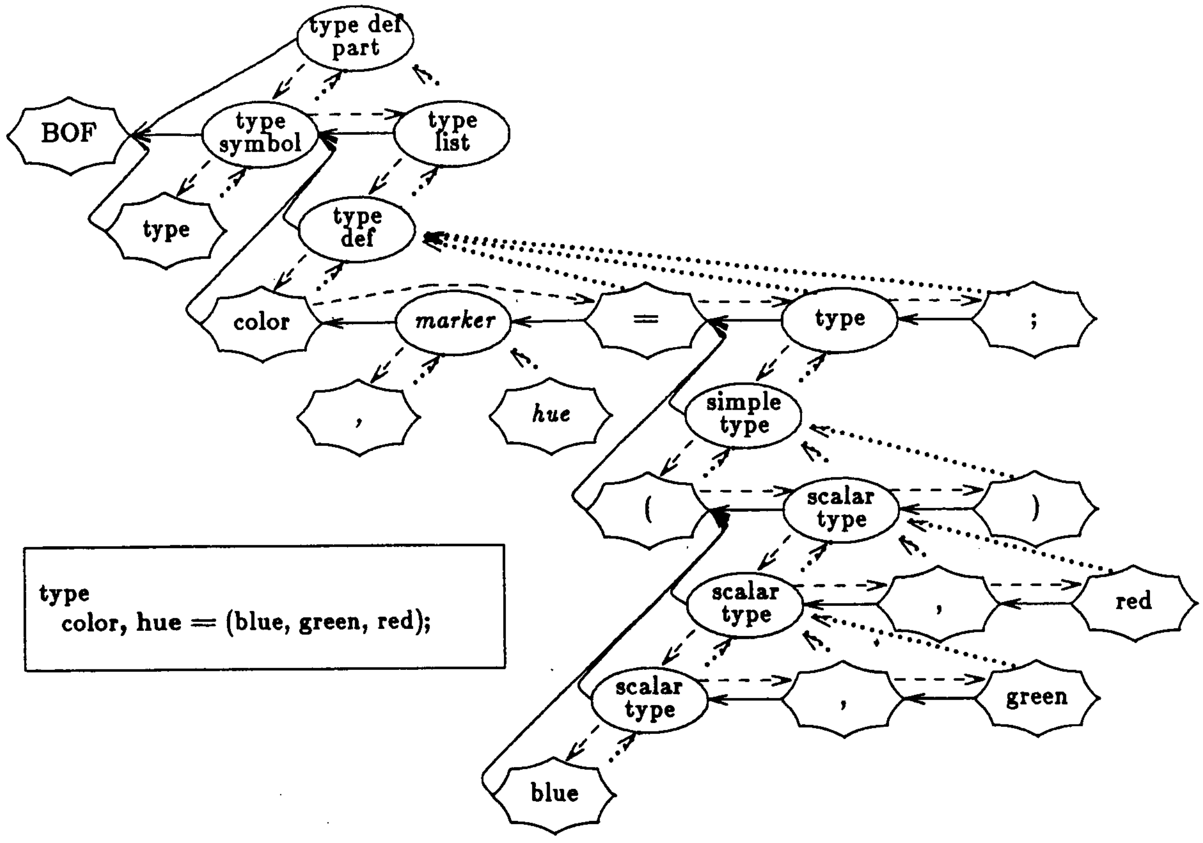


```
type
    intensity : integer;
     color = (blue, green, red);
```

Figure 5-2: Parse tree from Figure 5-?a after the incorrect insertion of "intensity : integer;". The text which was not successfully parsed is highlighted on the screen. A *marker* token has been inserted into the tree to note the point of the error.

5.9.1. Single Exception Handling

The SAGA parser divides exceptions into two types: *errors* and *suspensions*. An error occurs when a syntax error action is returned to the parse tree constructor by the *action* routine. A suspension occurs either when a user requests a partial parse and the parser finishes parsing *z'*, or when the parser attempts to perform a reduction and detects an insufficient context on the parse stack, caused by a previous error or suspension.



```
type
  color, hue = (blue, green, red);
```

Figure 5-3: Parse tree from Figure 5-1 after the incorrect insertion of ", hue" before the equal sign. In this case, since the following terminal node had its *lthread* attribute set to a terminal node, this attribute has been reset to point to the *marker* node.

To process errors, a third type of parse tree node is introduced: the *marker* node. In addition, a new attribute *nodetype* is added to all nodes; this attribute will be set to TERM, NT, or MARKER, according to whether the node is a terminal, non-terminal, or marker node.

When an error occurs, the parser allocates a *marker* node, takes the offending terminal node and all of the remaining nodes in the *nextusernode* list (which correspond to the tokens in z' not yet parsed), and makes them children of the marker. The terminal nodes are linked into the *prev/next* list. If the *lthread* attribute of the node following the rightmost child of the marker node points to a terminal node, it is reset to point to the marker node, as are the *lthread* attributes of any of its parents which also point to the same node. This relinking guarantees that the parser will detect the marker if it later is reparsing a section following this one, and a reduction brings it into this area of the tree. The current value of *stacktop* is saved in the marker node, for later restoration of the parse stack if a parse is resumed at this point in the tree. An example of the handling of a syntax error is illustrated in Figures 5–1, 5–2 and 5–3.

When a suspension is indicated, the parser allocates a marker node and links it directly into the *prev/next* terminal list just before the node that would be returned by the next call to *nextsym*. The node following the marker has its *lthread* attribute reset to point to the marker node, as do any of its parents whose *lthread* attribute is identical. The current value of *stacktop* is stored in the marker node, and the parser returns to the editor.

By tokenizing all new input before invoking the parser and linking the nodes into the terminal list when an error/suspension occurs, the editor can display the unparsed nodes even though the parser has not yet completely incorporated them into the internal structure of the parse tree. This ability is important, since it permits the user to view his input at the points of discontinuity, and even perform further modifications before, at, or after these points. Since the marker node is an integral part of the parse tree, trees containing errors can be saved between editor sessions and repaired at a later time.

A number of modifications to the parsing algorithm are necessary to support exception handling. First, a new routine *exception* is introduced, to mark the point of discontinuity in the parse tree:

```
exception(kind):
    Let kind be either ERROR or NOACTION, according to whether a syntax error
        or a suspension has occurred.
    Let $\mathcal{M}$ be a marker node, to note the point of error.
    Let $\mathcal{N}$ be the incorrect parse node, or nil if a suspension has occurred.
```

```
$\mathcal{M} = alloc()$;
if $kind = ERROR$ then {
    $leftson(\mathcal{M}) = \mathcal{N}$;
    $parent(\mathcal{N}) = addr(\mathcal{M})$;
    if the parser is in section (2) then {
        $chain(\mathcal{N}, BEFORE, activenode)$;
        $chain(\mathcal{N}_j$, BEFORE, $activenode)$, $\forall 1 < j < n$, where the $\mathcal{N}_j$ are on the
            $nextusernode$ list;
        $parent(\mathcal{N}_j) = addr(\mathcal{M})$; $\forall 1 < j < n$.
    } (otherwise we are reparsing $y$, and $\mathcal{N}$ is already in the terminal list)
} else {
    $leftson(\mathcal{M}) = nil$;
    $chain(\mathcal{M}$, BEFORE, $activenode)$;
}
$lthread(\mathcal{M}) = stacktop$;
if $nodetype(lthread(activenode)) =$ TERM then
    $lthread(\mathcal{N}_a) = addr(\mathcal{M})$, $\forall \mathcal{N}_a$, where $lthread(\mathcal{N}_a) = lthread(activenode)$.
```

Since the parser can now terminate in one of two ways, either by a completion or a suspension, a fourth section is added to the algorithm to handle termination. If the parser completes, accepting the modification just made, then the algorithm jumps to (4.1). Section (3.3d) is changed from "the algorithm terminates" to "jump to (4.1)". If the parser suspends, then it will jump to (4.2) to terminate. To handle suspensions, sections (2.1c) and (3.3c) of the incremental parsing algorithm must be altered from "jump to the appropriate error recovery action" to "*exception(ERROR)*; jump to (4.2)". In addition, a test is inserted at the beginning of section (3) to determine whether a partial parse has been requested by the user, and if so, the code "*exception(NOACTION)*; jump to (4.2)" is executed.

These modifications are sufficient to recover from an initial exception which the parser might encounter. If subsequent parsing is now restricted to requiring the repair of this error before permitting any other editing, then no further alterations are necessary, and the extensions to the parser are finished. But a practical editor should be more flexible than this, and so we will investigate parsing in the midst of multiple errors.

5.9.2. Multiple Exception Handling

By making two other extensions, to permit the parser to encounter marker nodes in its input, and to detect marker tokens in the parse stack when a reduction is about to be performed, we can relax the single error restriction, and permit editing anywhere within the tree no matter how many errors or suspensions are outstanding.

A parse tree containing a single error or suspension point will have either a marker node in the terminal list, or a continuous sequence of one or more unparsed nodes, all with their *parent* attribute set to the marker node which manages the discontinuity. If a parse can occur elsewhere in a tree containing one of these discontinuities, then the marker or an unparsed node can be encountered in one of three ways: (1) during reinitialization, (2) if the *nextnode* variable becomes set to one of these nodes, and *nextsym* is called to return the next node as the parser moves forward, or (3) if a marker node is found on the parse stack during a reduction operation. If each of these cases is addressed, then parsing can be permitted anywhere along the frontier of the tree no matter how many points of discontinuity exist in the parse tree.

During reinitialization, the parser backs along the frontier immediately before *activenode*, to find the most recent token (excluding epsilon tokens and comments) that previously had been shifted by the parser. If during this operation an unparsed or marker node is encountered, then the initialization cannot be completed, since there is no previous parse context to retrieve. The user's modification can still be permitted, however, by deleting the number of nodes specified, and then calling *exception(NOACTION)* to link the new input from the *nextusernode* list into the frontier together with the marker node. The new input will be retained in the frontier of the tree, but its parsing will need to be deferred until the earlier exception is repaired.

If the parser succeeds in its reinitialization, successfully processes all of the nodes in the *nextusernode* list, and then encounters an unparsed or marker node during a call to *nextsym*, an attempt can be made to continue the parse. If *nextnode* points directly to a marker node, the marker can simply be deleted from the tree and *nextnode* advanced. If *nextnode* points to an unparsed node, then the node can be returned to the parser, and *nextnode* advanced. The parse should be continued because a node which previously caused an error might parse correctly now, since the parse context immediately before it may be different than before. Once the last unparsed node is passed to the parser, the marker node effectively drops out of the tree. Only the *lthread* attribute of the terminal node and zero or more of its parents following the last unparsed node still point to the marker node. We must add a test to *exception* to reset the *lthread* attribute if the node type is a *marker* as well as a terminal node, then if a new suspension were to occur at this point, these fields would all be reset to a new marker node, leaving no further reference to the original marker. If the parse does continue beyond this point, the *lthread* attribute of this terminal node will be altered as soon as it is pushed onto the parse stack, along with those of its parents as soon as they are processed. If the reparse progresses in such a way that these non–terminal nodes are not reprocessed, then they will be excluded from the final tree when the match condition holds, and their reference to the deleted marker node will be irrelevant. Therefore, the only modification required to the algorithm is made to the block of code in *nextsym* headed by "**if** *nextnode* $\neq$ *nil* **then** { ... }", which is changed to:

```
if $nextnode \neq nil$ then {
    while $nodetype(nextnode) = \epsilon$ or $nodetype(nextnode) = MARKER$ do
        $nextnode \leftarrow next(nextnode)$;
    $\mathcal{N} \leftarrow nextnode$;
    $nextnode \leftarrow next(nextnode)$;
}
```

Only one other case remains. Recall that whenever a marker node is inserted into the tree, the *lthread* attribute of the following terminal node and any of its parents with an identical *lthread* attribute all have it reset to point to the marker node. Therefore, any stack traces which pass through these nodes will pass through the marker node and continue to the left, always terminating in the *bottom-of-stack* node *B*. Any other stack traces which pass through a parent node whose *lthread* attribute was not reset will not encounter the marker and the parse can proceed normally. So the only additional check by the parser occurs in the *reduce* routine, to determine if any node in the handle about to be reduced is a marker node. If one is detected, the reduction cannot be made, and the parser must suspend, since there is inadequate context to be used. A call to *exception* is made instead, and the parser inserts a suspension point just before *nextnode*. The parse must now be suspended, so *reduce* must be further modified to return a value: *0* if the reduction proceeded normally, *1* otherwise. The parser checks the return code from *reduce*, and if it is non-zero, immediately jumps to (4.2) for termination.

With these three cases accounted for, the parser can now support general editing throughout the tree, regardless of the number of outstanding errors. Although in two of these cases the parser must suspend when it encounters a marker or unparsed node, the user's input will still be entered into the tree and displayed, so that flexible editing is supported.

5.10. Extension to Support a *shiftreduce* Parse Action Optimization

Some parser-generators optimize their parse tables by providing another parse action in addition to the basic four actions: *shift, reduce, error,* and *accept.* This fifth action, *shiftreduce*, is returned whenever it has been determined that a *shift* action will produce a stack containing a complete handle to be reduced, and a reduction can immediately be performed. By providing this fifth action, both the number of states required for the parse tables, and the number of steps the parser must take, can be reduced.

Back in chapter 3, a sample parse was presented in Figure 3–2. Although the example did not show any *shiftreduce* actions, moves 1a, 1b, 4a, 4b, 5a, 5b, 8a and 8b made by that parser could have been combined into single moves 1, 4, 5, and 8, the actions replaced in the parse tables by actions "sr6", "sr6", "sr5", and "sr6" respectively, and parse states 9, 10, and 11 deleted from the parse tables. The reader should note that the parse table states eligible for this treatment are the ones in which the only non–error actions are reductions by a single production. In this case, all "s9" actions would be replaced by "sr5" actions, all "s10" actions replaced by "sr6" actions, and all "s11" actions replaced by "sr8" actions.

Adapting the parsing algorithm for this extension requires simple extensions to sections (2.1) and (3.3). A sixth case, labeled (f) in the final presentation of the algorithm, for $f =$ *shiftreduce* needs to be added. The code for this new case is simply the code for the *shift* action followed immediately by the code for the *reduce* action which appears in accompanying sections (a) and (b).

5.11. Algorithm 5.2: The SAGA Incremental LR(1) Parser

The extensions to the incremental parser are now complete. Some other attributes are added to the parse tree nodes in the next chapter, and used to support editor operations. But these attributes are not required for incremental parsing, nor are they maintained or referenced by the incremental parser, so their presentation has been left for later chapters, so that only the essential parser extensions could be discussed in this chapter, to simplify the presentation.

The extended incremental LR(1) parser is now restated. This algorithm now handles LR(1) grammars, epsilon productions, comments, multiple syntax errors, and partial parses (suspensions). The next chapter discusses the editor/parser interface and the command interpreter of the SAGA editor.

5.11.1. Routines used in the Parser

```
alloc():
    Allocate a new node $\mathcal{N}$.

    addr($\mathcal{N}$) ← $\mathcal{N}$;
    rdescend($\mathcal{N}$) ← $\mathcal{N}$;
    return($\mathcal{N}$).

apply_match:
    Let $A \rightarrow \alpha$ be the reduction for which the matching condition holds.

    parent(stack(stacktop, j)) ← parent(irmark), $\forall 0 \leq j < |\alpha|$;
    sibling(stack(stacktop, j)) ← stack(stacktop, j-1), $\forall 0 < j < |\alpha|$.
    sibling(stacktop) ← nil.

chain($\mathcal{N}$, at, $\mathcal{M}$):
    Let $\mathcal{M}$ be a node in the frontier of $\mathcal{T}$, $\mathcal{N}$ a node to be added,
        and at be one of BEFORE or AFTER.

    if at = BEFORE then {
        next($\mathcal{N}$) = addr($\mathcal{M}$);
        prev($\mathcal{N}$) = prev($\mathcal{M}$);
        if prev($\mathcal{M}$) $\neq$ nil then
            next(prev($\mathcal{M}$)) = addr($\mathcal{N}$);
        prev($\mathcal{M}$) = addr($\mathcal{N}$);
    }
    if at = AFTER then {
        next($\mathcal{N}$) = next($\mathcal{M}$);
        prev($\mathcal{N}$) = addr($\mathcal{M}$);
        if next($\mathcal{M}$) $\neq$ nil then
            prev(next($\mathcal{M}$)) = addr($\mathcal{N}$);
        next($\mathcal{M}$) = addr($\mathcal{N}$);
    }.

exception(kind):
    Let kind be either ERROR or NOACTION, according to whether a syntax error
        or a suspension has occurred.
    Let $\mathcal{M}$ be a marker node, to note the point of error.
    Let $\mathcal{N}$ be the incorrect parse node, or nil if a suspension has occurred.

    $\mathcal{M}$ = alloc();
    if kind = ERROR then {
        leftson($\mathcal{M}$) = $\mathcal{N}$;
        parent($\mathcal{N}$) = addr($\mathcal{M}$);
        if the parser is in section (2) then {
            chain($\mathcal{N}$, BEFORE, activenode);
            chain($\mathcal{N}_j$, BEFORE, activenode), $\forall 1 \leq j < n$, where the $\mathcal{N}_j$ are on the
                nextusernode list;
            parent($\mathcal{N}_j$) = addr($\mathcal{M}$); $\forall 1 \leq j < n$.
        } (otherwise we are reparsing y, and $\mathcal{N}$ is already in the terminal list)
    } else {
        leftson($\mathcal{M}$) = nil;
        chain($\mathcal{M}$, BEFORE, activenode);
    }
    lthread($\mathcal{M}$) = stacktop;
    if nodetype(lthread(activenode)) = TERM        or nodetype(lthread(activenode))
= MARKER then
        lthread($\mathcal{N}_a$) = addr($\mathcal{M}$), $\forall \mathcal{N}_a$, where lthread($\mathcal{N}_a$) = lthread(activenode).

matchcond():
    Let $A \rightarrow \alpha$ be the reduction to be applied.

    if irmark = stack(stacktop, j), for some $0 \leq j < |\alpha|$
    and parent(irmark) = parent(stack(irmark, h)) $\forall 0 \leq h < |\alpha| - j$
    and parent(irmark) $\neq$ parent(stack(irmark, $|\alpha| - j$))
    and token(parent(irmark)) = A
    and rdescend(stacktop) = rdescend(parent(irmark))
    then
        return(true);
    else
        return(false).

nextsym():
    Let $\mathcal{N}$ be a pointer to a parse tree node.
    Variable savenode is set in routine prepend below.

    if savenode $\neq$ nil then {
        $\mathcal{N} \leftarrow$ savenode;
        savenode $\leftarrow$ nil;
    } else if nextusernode $\neq$ nil then {
        $\mathcal{N} \leftarrow$ nextusernode;
        nextusernode $\leftarrow$ next(nextusernode);
    } else if nextnode $\neq$ nil then {
        while nodetype(nextnode) = $\epsilon$ or nodetype(nextnode) = MARKER do
            nextnode $\leftarrow$ next(nextnode);
        $\mathcal{N} \leftarrow$ nextnode;
        nextnode $\leftarrow$ next(nextnode);
    } else {
        $\mathcal{N} \leftarrow$ alloc();
        token($\mathcal{N}$) $\leftarrow$ eof;  (the end-of-file token code)
    }
    return($\mathcal{N}$).

prepend($\mathcal{N}$):
    savenode $\leftarrow$ addr($\mathcal{N}$).

reduce(i, $\mathcal{N}$):
    Let i be production $A \rightarrow \alpha$, and $\mathcal{N}_j$ be the nodes in the handle to be reduced,
        $1 \leq j \leq n$, $n = |\alpha|$.

    If $n = 0$ then
```

```
        $n \leftarrow 1$;
        if nodetype($\mathcal{N}_j$) = MARKER, $\forall 1 \leq j \leq n$ then {
            exception(NOACTION);
            return(1);
        }
        parent(stack(stacktop, j)) $\leftarrow$ addr($\mathcal{N}$), $\forall 0 \leq j < n$;
        sibling(stack(stacktop, j)) $\leftarrow$ stack(stacktop, j - 1), $\forall 0 < j < n$;
        sibling(stacktop) $\leftarrow$ nil;
        rdescend($\mathcal{N}$) $\leftarrow$ rdescend(stacktop);
        token($\mathcal{N}$) $\leftarrow$ A;
        prepend($\mathcal{N}$);
        return(0).

    shift($\mathcal{N}$, newstate):
        lthread($\mathcal{N}$) $\leftarrow$ stacktop;
        pstate($\mathcal{N}$) $\leftarrow$ new-parse-state;
        stacktop $\leftarrow$ addr($\mathcal{N}$).

    unchain($\mathcal{N}$):
        prev(next($\mathcal{N}$) = prev($\mathcal{N}$);
        next(prev($\mathcal{N}$) = next($\mathcal{N}$).
```

5.11.2. The SAGA Incremental LR(1) Parser

parse(activenode, deletecount, nextusernode, parseoption):

Let $\mathcal{T}$ be the parse tree for the string $w = xzy$.
Let z' be a replacement string for z, and $w' = xz'y$ the result.

1. Initialization

```
(1.1) if w $\neq$ e (the empty string) then {
          $\mathcal{N}$ $\leftarrow$ activenode; (the first symbol in zy)
          while $\mathcal{N} \in z$ do { (delete z)
              $\mathcal{N}$ $\leftarrow$ next($\mathcal{N}$);
              unchain(prev($\mathcal{N}$));
          }
          activenode $\leftarrow$ $\mathcal{N}$;
          nextnode $\leftarrow$ activenode;

          $\mathcal{N}$ $\leftarrow$ prev($\mathcal{N}$); (reset the parser...)
          while token($\mathcal{N}$) = $\epsilon$ or token($\mathcal{N}$) = commentcode do
              $\mathcal{N}$ $\leftarrow$ prev($\mathcal{N}$);
          irmark $\leftarrow$ lthread($\mathcal{N}$);
          stacktop $\leftarrow$ lthread($\mathcal{N}$);

          if nodetype($\mathcal{N}$) = MARKER then {
              exception(NOACTION);
              jump to (4.2)
```

```
            }
      }.
(1.2) if $w = e$ (i.e., $w'$ is being parsed from scratch) then {
            irmark ← $\mathcal{B}$;
            stacktop ← $\mathcal{B}$;
            nextnode ← nil;
      }.
```

2. Analysis of *z'*

```
(2.1) $\mathcal{N}$ ← nextsym();
      action(pstate(stacktop), token($\mathcal{N}$), f, newvalue);
      Execute (a), (b), (c), (d), (e) or (f) according to the value of f.

(a)   $f = SHIFT$.
      if the symbol to be shifted is activenode then
            jump to (3);
      else {
            shift($\mathcal{N}$, i);
            chain($\mathcal{N}$, before, activenode);
      }.

(b)   $f = REDUCE\ i$. Let $i$ be the production $A \rightarrow \alpha$.
      if irmark = stack($\mathcal{N}$, j) for some $0 <= j < |\alpha|$ (i.e., irmark must be updated)
then
            irmark ← stack($\mathcal{N}$, $|\alpha|$);
      if $|\alpha| > 0$ then {
            $\mathcal{N}$ ← alloc();
            if reduce(i, $\mathcal{N}$) = 1 then
                  jump to (4.2);
      } else {
            $\mathcal{N}$ ← alloc();
            token($\mathcal{N}$) ← $\epsilon$;
            shift($\mathcal{N}$, i);
            $\mathcal{N}$ ← alloc();
            if reduce(i, $\mathcal{N}$) = 1 then
                  jump to (4.2);
      }.

(c)   $f = ERROR$.
            exception(ERROR);
            jump to (4.2).

(d)   $f = ACCEPT$.
      Jump to (4.1).

(e)   $f = NOACTION$.
      chain($\mathcal{N}$, before, activenode).
```

```
(f)     f = SHIFTREDUCE.
        Execute sections (2.1a) and (2.1b) above.
```

3. Analysis of y

```
(3.1)   If parseoption = SUSPEND then
                jump to (4.2).

        N ← nextsym();          (Let N be the node which stores the first symbol of y.)
        oldtable ← pstate(N);
        shift(N, i);

(3.2)   if oldtable $\neq$ pstate(stacktop) then
                jump to (3.3);

        Otherwise, skip steps of the analysis of y as follows:
        while sibling(stacktop) $\neq$ nil do {
                stacktop ← sibling(stacktop); (we enter directly in a reduction state).
                N ← stacktop;
        }
        action(pstate(stacktop), token(N), f, i); (we know f = REDUCE i, i being
                production $A \rightarrow \alpha$).
        if matchcond holds then {
                apply_match;
                accept w', terminating the algorithm.
        }
        if irmark = stack(stacktop, j) for some $0 <= j < |\alpha|$ then
                irmark ← stack(stacktop, $|\alpha|$);
        oldtable ← pstate(parent(stacktop));
        if parent(stack(stacktop, j)) = parent(stack(stacktop, k)) $\forall\, 0 <= j, k < |\alpha|$ then {
                the entire subtree of T rooted in parent(stacktop) is reused:
                N ← parent(stacktop);
                lthread(N) ← stack(stacktop, $|\alpha|$ ).

                action(pstate(stacktop), token(N), f, newvalue);
                pstate(N) ← newvalue.
        } else {
                a new node is allocated:
                N ← alloc();
                if reduce(i, N) = 1 then
                        jump to (4.2);
        }
        Jump to (3.2).

(3.3)   N ← nextsym(input);
        action(pstate(stacktop), token(N), f, i);
        Execute (a), (b), (c), (d), or (e) according to the value of f.

(a)     f = SHIFT.
        oldtable ← pstate(N);
        shift(N, i);
        jump to (3.2).
```

```
(b)  $f = REDUCE\ i$. Let $i$ be production $A \rightarrow \alpha$;
     if matchcond holds then {
          apply_match;
          jump to (4.1);
     }
     if $irmark = stack(stacktop, j)$ for some $0 <= j < |\alpha|$ then
          $irmark \leftarrow stack(stacktop, |\alpha|)$;
     $\mathcal{N} \leftarrow alloc()$;
     if $reduce(i, \mathcal{N}) = 1$ then
          jump to (4.2);
     jump to (3.3).
(c)  $f = ERROR$.
          exception(ERROR);
          jump to (4.2).
(d)  $f = ACCEPT$.
     Jump to (4.1).
(e)  $f = NOACTION$.
(f)  $f = SHIFTREDUCE$.
     Execute sections (3.3a) and (3.3b) above.
```

4. Termination

```
(4.1) $status = COMPLETE$;
      return(status).
(4.2) $status = SUSPEND$;
      return(status).
```

5.12. Summary

In this chapter, we have presented the editor's incremental parser. We altered the attributes associated with the parse tree node to make the parse tree suitable for use with an editor. The parsing algorithm was extended from LR(0) to LR(1) grammars. It also has been extended to support grammars containing productions with empty right hand sides.

We proposed a new way to handle comments, which permits their general use in language–oriented editors, as in text editors, resolves their storage problem in parse trees, and permits uniformity of access by editor commands to both comments and syntactically meaningful tokens in the tree.

We redefined the *reduce* operation, proposing an alternative which permits the parser to treat non–terminal and terminal tokens uniformly. We also combined the parsing action and goto function into a single action. Duplicate code was eliminated, improving efficiency, and provide support for the editor to pass sub–trees to the parser.

Our error handler was described, which permits editing throughout the parse tree in the midst of multiple errors, and the editing of erroneous text which has not yet been parsed. We have elected to provide error recovery, not error correction, since it supports the above abilities while letting the programmer correct his own errors, which we found to be simpler for both the programmer and the editor. We added the ability to perform a partial parse, only analyzing the new input, and then suspending the parse to await further instructions. This feature permits controlled editing which takes the program through incorrect intermediate states, improving the flexibility of the editor.

CHAPTER 6

THE SAGA EDITOR

The SAGA editor has been designed to be modular and retargetable to more than one language. The modularity concentrates the language-dependent code in a few modules, allowing most of the source code to be re-used intact when editors are built for new languages. It also permits experimentation with different parser-generating systems for a given language, so that the strengths and weaknesses of different systems can be compared. A pictorial breakdown of the editor's modular structure is presented in Figure 6–1. This chapter will discuss these modules, and their interactions with one another. The editor/parser interface is discussed first, editor commands next, then editor interaction with other development tools, and finally the editor interface to the file system. The next chapter discusses the generation of editors for different languages.

6.1. The Editor/Parser Interface

The editor/parser interface consists of four modules: the *parse tree constructor* on the editor side of the interface, and the *lexical analysis, syntax analysis* and *semantic analysis* modules on the language-dependent parser side of the interface. The parse tree constructor implements the incremental LR(1) parsing algorithm presented in the previous chapter. Header files are provided for each of the analysis modules; any parser-generating system which produces tables for which code can be written to meet the requirements of this interface can be used with the SAGA editor.

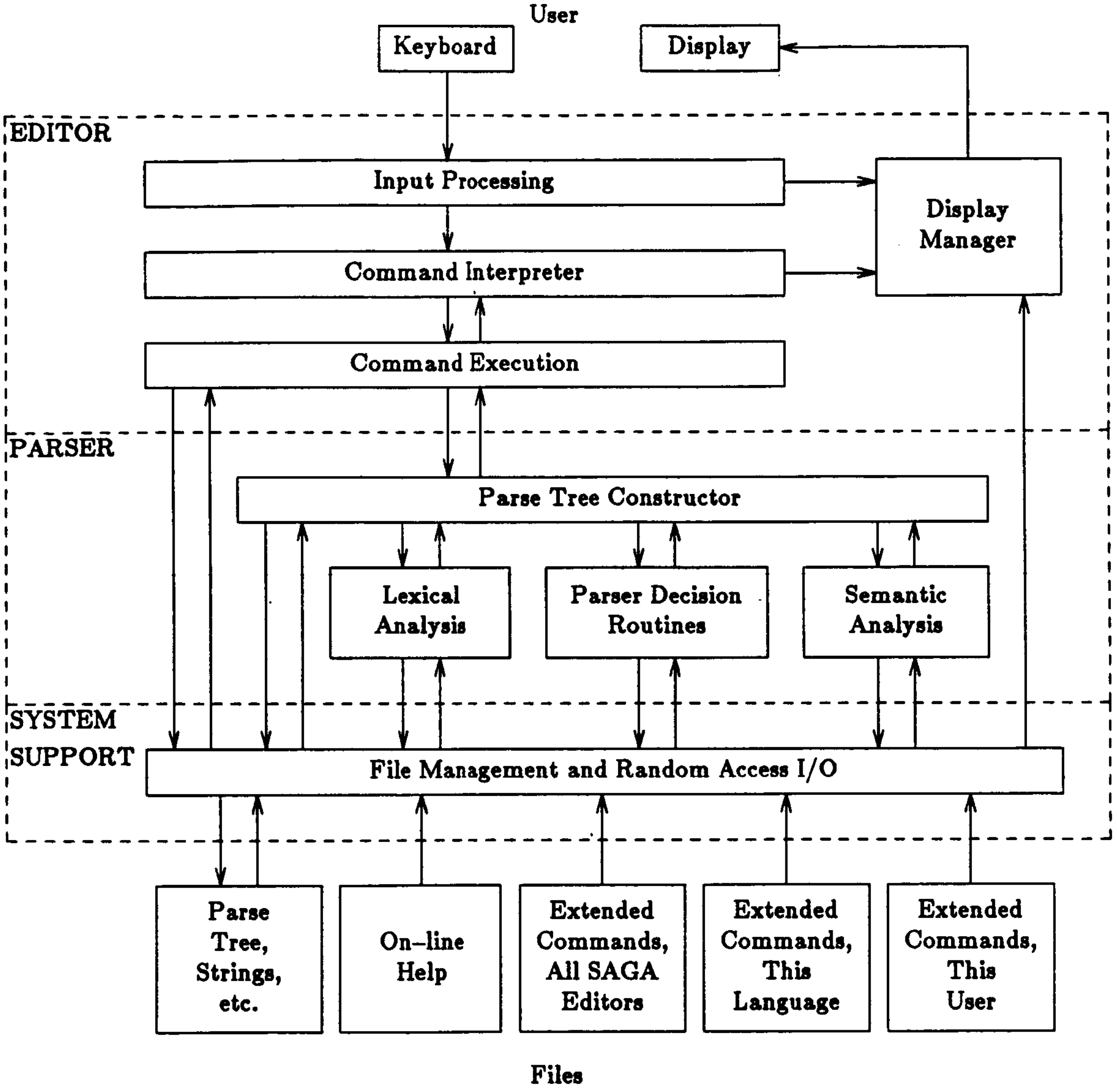


Figure 6–1: SAGA Editor Modular Structure

During editing and language analysis, all interaction between the editor and the parser occurs through two routines: *tokenize* and *parse*. The *tokenize* routine converts a buffer of characters into a linked list of terminal nodes; the *parse* routine inserts these nodes into the parse tree, also removing any nodes which are being deleted. Calls to the semantic analyzer are embedded within the incremental parser. A parser initialization routine also exists, as do a few other routines to generate the follow set for the language and to support the parse tree constructor.

6.1.1. Lexical Analysis

When the user makes a change to his program, the input handler of the editor constructs a text image of the input and the token being modified. If the change is between two tokens with no intervening space, the text from both tokens is included in the image. The lexical analysis routine *tokenize* then tokenizes this image. If the input spans several lines, *tokenize* is called on each line as it is completed, and the returned nodes are appended to the *nextusernode* list being constructed. The change may cause the text to the right of the modification to be re–examined, in which case the analyzer may need to request further input from the input handler in order to properly complete its task. A lookahead character (the character on the screen immediately after the text image) is always passed to the analyzer, which it may use to decide whether it requires further input. If the lookahead character is not part of the current token, then the tokenizer is finished, and returns a list of parse tree terminal nodes which represent the tokens. Otherwise, the tokenizer returns the list of terminal nodes and the remaining text image, with a request to be called again with further input. In the case of a *matchfix* token (such as a comment or string) which has not yet been completely recognized, the tokenizer returns and the input handler enters a "token collection" mode in which the user can skip the cursor forward to include existing text in the new matchfix token. The user can then insert the terminating delimiter at an appropriate point.

The interface to the language–dependent lexical analysis routines is defined in the header file *lexfns.h*, summarized in Figure 6–2.

error ← *lexinit(treeexists);*

nodelist ← *tokenize (buffer, nextc, lastc, lookahead, addinput);*

where:

addinput: A Boolean, set by *tokenize* to request more input.
buffer: A buffer of characters.
lastc: The last character position used in a buffer.
lookahead: The character after the last character passed to *tokenize*.
nextc: The first character position used in a buffer.
treeexists: A Boolean, set to *true* if an existing tree is being edited.

Figure 6–2: Lexical Analysis Interface

When a lexical error is encountered, the remainder of the input string is stored in a separate parse tree terminal node, marked as an unknown token, and returned along with any other terminal nodes that were constructed. The calling routine can still make further calls to the lexical analyzer, until all input has been lexically analyzed. These nodes will still be passed to the parser later.

When the tokenizer requests further input, it sets *addinput* to *true* and returns with *nextc* set to the first character not included in any token. Its caller copies the characters between *nextc* and *lastc* back to the beginning of the buffer, retrieves the text representation of the following token, appends it to this buffer, and marks the token for deletion. It calls the tokenizer again, and this process repeats until the buffer can be entirely tokenized.

By completely tokenizing the input before performing any parsing, it can be guaranteed that text read from an input file will become a part of the frontier of the parse tree whether or not it is syntactically correct, without requiring additional code to handle this situation as a special case. The implementation is simplified, since it is not necessary to treat a syntax error in a text file differently from a syntax error in text typed by the user.

6.1.2. Syntax Analysis

After the lexical analysis is complete, the parser is called to insert the new nodes into the parse tree. The parser can be run in one of two ways: to *suspend* or *complete*, according to the command given by the user. Normally the user asks the parser to run to completion, with it reparsing y after it has finished parsing z', where z' represents the new input, and y the remainder of the terminal string past the new input. However, the user can request a *partial parse*, which causes the parser to suspend parsing after analyzing z', and before any reparsing of y. The parser also will suspend whenever it encounters a syntax error, regardless of the parse requested. A suspension will leave the parse tree with a discontinuity, but with the state and local structure saved so that the parse can be resumed later. The parser also can process deletions using either a full or partial parse.

Parser suspension permits modifications which take the parse tree through intermediate illegal configurations, as, for example, when a **begin** and distant matching **end** symbols are being inserted or deleted. The presence of this option greatly increases the flexibility of editing operations, since the user can make a change in several operations, without concern for maintaining syntactic correctness at each step.

When a syntax error is encountered, the offending token (which could be the unknown token constructed above) is highlighted and diagnostics are displayed. All new terminal nodes are still inserted into the parse tree on the *prev/next* list discussed earlier; thus they may be accessed by the display manager even when they cannot be successfully parsed. The user has the option of repairing the error immediately, or of scanning through other portions of the program and possibly making modifications there (needed, for example, if a **begin** keyword was mistakenly omitted and its matching **end** just encountered).

Syntax analysis is performed in the parse tree constructor and *syntax analysis* modules of the editor. The interface to the language-dependent syntactic analysis routines is defined in the header file *parsefns.h*, summarized in Figure 6-3. The parse tree constructor, *parse*, is the incremental LR(1) parser presented at the end of the previous chapter. It communicates with the

```
var  Vpgenname:  alfa;           Name of the parser-generator used.
     Vpgenrev:   alfa;           Version of the parser-generator.
     Vlangname:  alfa;           Name of the language recognized.
     Vlangrev:   alfa;           Version of the language (grammar spec.).

error ← initparser(treeexists);

action (tokencode, state, f, newvalue);

legalnonterm (state, stackptr, tokenlist, length);

legalterm (state, stackptr, tokenlist, length);

nametokencode (tokencode, buffer, lastc);

tokencode ← ruleleftside (rulenumber);

length ← rulelength (rulenumber);
```

where:
buffer: A buffer of characters.
f: The parsing action returned.
lastc: The last character position used in a buffer.
length: The number of items in a returned list.
rulenumber: A production rule number.
state: The current parse state.
stackptr: A pointer to the node on the top of the parse stack.
tokencode: The code (integer) assigned to a token of the grammar.
tokenlist: An array of token codes.
treeexists: A Boolean, set to *true* if an existing tree is being edited.

Figure 6-3:Interface to Syntax Analysis Routines

parser decision routines through an interface which supports the basic shift–reduce parsing algorithm. This interface permits the use of different parsers from a variety of parser–generating systems in the construction of SAGA editors. Any parser generating system may be used if the resulting parser and its tables can support the functions required by the interface. Since different parser–generators have different capabilities, this permits us to choose a generator best suited for a particular language.

The *parse* routine takes an editing location, a count of the number of nodes to be deleted, the list of nodes to be inserted, and the parsing option *suspend/complete*. It in turn calls a routine *action*, on the language–dependent side of the interface, in the syntactic analysis module. The *action* routine is passed the current parse state and a token code; it uses these values to index into a set of parse tables to produce a parsing action and either a new parse state or a production rule number, according to the parsing action.

Two routines, *ruleleftside* and *rulelength*, are defined which take a production rule number and return the token code of the token on the left hand side of the rule, and the length of the rule, respectively. These routines are used by the *reduce* routine described earlier to obtain the necessary information about the production $A \rightarrow \alpha$ to be used in the reduction.

Routines *legalterm* and *legalnonterm* pass in the current parse state and value of the *stacktop* variable, and expect to receive a list of terminal or non–terminal token codes. These codes are used to construct the follow set to be displayed and to determine whether a non–terminal node can be inserted at a given spot in the terminal list.

Routine *nametokencode* passes in a token code and expects to receive a text string which is the printable form of the token. If the token code is a reserved word or a special symbol (operators and punctuation), then that reserved word or symbol(s) should be returned. If the token code is a generic class, such as an identifier or constant, then that identifier or constant should be retrieved from the string table and returned. If the token code is a non-terminal token, then the text string used to name this token should be retrieved if available, or the production rule number enclosed in angle brackets otherwise. This value is most often used in debugging traces, but could also be used to display a non-terminal follow set (for a language designer, for example) or as a printable name for a place holder for an unexpanded or elided sub-tree.

Lastly, a call to a routine *initparser* is provided which passes in a flag indicating whether an editing session is beginning with a new or preexisting file; this routine should contain code to load the parse tables and initialize any internal data structures to be used by the other routines in this module. It also should initialize several character strings which are used to display the version of the language and parser-generating system in use. Tables from any parser-generator can be used as long as access code to produce the return values from these initial values can be written.

6.1.3. Semantic Analysis

Support is provided for semantic analysis to be performed through a syntax-directed analysis scheme. The interface to the semantic analysis routines is defined in the module *semanfns.h*, summarized in Figure 6–4. As the parser shifts and reduces parse tree nodes, it calls semantic evaluation routines in the *semantic analysis* module. The semantic routines can either evaluate the changes as they are made, or can record the changes as the parser runs and perform the actual evaluation after the reparse has completed. When a semantic error is detected, a *semantic error* flag is set in the node, and that token is displayed in highlighted form. Since semantic errors do not affect the integrity of the parse tree, there is no impact upon the incremental parser. The user can repair the error when convenient.

Calls to these routines are placed into the incremental parsing algorithm, to automatically invoke these routines whenever a parse occurs. Any style of semantic analysis which can be per-

evalinit (treeexists);	Beginning to edit a new file.
evalstart (stacktop, state);	An incremental reparse is beginning.
evalcontinue (tag); *Tag* was returned by ***evalsuspend*** earlier.	Parser has reached a suspension point.
evaldelete (activenode, count);	Delete ***count*** nodes, beginning at ***activenode.***
evalshift (stacktop);	A node has just been shifted.
evalreduce (stacktop, rulenumber, parent);	About to reduce ***stacktop*** by ***rulenumber*** to ***parent.***
evalphase2 (yfirst: nodeindex);	Beginning 2nd phase; the reparse of ***y.***
tag ← *evalsuspend (stacktop, state);*	Suspending a parse; ***tag*** will be passed to ***evalcontinue*** later.
error ← *evalfinish (subtreeroot);*	Completed a parse; ***error*** set if any semantic errors.
error ← *evalclose;*	Ending an editing session; ***error*** set if cannot save semantic data.
error ← *eval (tree, activenode);*	Called by editor ***eval*** command.
evalerror (activenode, buffer, lastchar);	Return semantic error message for ***termnode*** in ***buffer:lastchar.***
evaldebug (tree, activenode);	Called by editor ***evaldebug*** command.

where:

activenode: parse tree node on which the editing cursor is positioned.
buffer: contains message describing semantic error.
lastchar: length of message in ***buffer.***
parent: node to be parent after reduction is made.
rulenumber: production rule number used in this reduction.
stacktop: a pointer to the node on the top of the parse stack.
state: the current parse state.
subtreeroot: non–terminal node at which incremental reparse terminated.
tag: an integer returned by ***evalsuspend***; passed to ***evalcontinue*** later.
tree: a pointer to the header record for the parse tree.
treeexists: set to ***true*** if the parse tree already exists.
yfirst: the first old terminal node after the new input nodes.

Figure 6–4: Interface to Semantic Analysis Routines

formed using the values made available through these calls can be supported. In particular, the semantic analysis may be performed either in step with the syntactic analysis, in a separate pass over parts of the tree after the syntactic analysis has completed, or as a separate process running in parallel with the editor. Since semantic analysis, even when performed incrementally, takes a significant amount of time to complete, analysis can be deferred and performed only when specifically directed by the user.

Routine *evalinit* is called during editor initialization, to permit the semantic evaluator to initialize its data structures, and start up its own process if one is desired. *Evalstart* is called whenever a new parse begins, except if one is beginning at a suspension point, when *evalcontinue* is called instead. *Evalcontinue* is also called each time the parser reaches a discontinuity in the *nextnode* list. *Evaldelete* is called before the parser actually modifies the tree; it is passed both *activenode* and *deletecount* which indicate the nodes to be deleted, to permit the semantic analysis routines to nullify any synthesized attributes, if required. Each time the parser shifts a node, *evalshift* is called with the *stacktop* variable after the operation is complete. Each time the parser performs a reduction, *evalreduce* is called with *stacktop* and the new parent node after the parent and its children are linked together, but before the reduction is performed, to make access to the children easier. When the parser completes parsing the new input string z' and begins reparsing y, already present in the old tree, *evalphase2* is called to indicate the parser has entered the next phase of the parse.

If the parse completes normally, *evalfinish* is called; otherwise *evalsuspend* is called, and the integer value returned by it is saved in the *marker* node to be passed to *evalcontinue* when this discontinuity is later reached. At the end of an editing session, *evalclose* is called to permit any data kept in memory to be written to disk, and to terminate the separate semantic analysis process, if one was begun.

Three other semantic analysis routines exist which are tied to editor commands. *Eval* is called when the "eval" command is executed; it provides support for editing commands which access the symbol table produced during the semantic analysis. *Evalerror* corresponds to the "evalerror" command, is called with the address of a node containing a semantic error, and returns an error message to be displayed for the user. Lastly, ***evaldebug*** is linked to the editor command of the same name, and provides an entry point to the semantic analysis routines to support interactive debugging, such as display of the data structures used by the semantic analysis routines.

These routine skeletons are provided by the editor for the language implementor to enable him to interface the editor to a parser–generator of the implementor's choice, so that different semantic analysis techniques can be tried.

The SAGA group is presently studying incremental semantic analysis and building an attribute evaluator for languages specified by ***regular right part LR(1) grammars*** [Beshers, 84] using ***maintained and constructor attributes*** [Beshers and Campbell, 85]. An independent investigation into semantic analysis based upon the CFF/AML system designed by Kaplan [Kaplan, 85] is also beginning. A non–incremental semantic evaluator was recently produced [Kimball, 85] to provide support for a code generator to produce object code directly from the parse trees constructed by the editor. Further reports about semantic analysis schemes should appear in future Ph.D. dissertations and Master's theses by other members of the SAGA research group as this related research matures.

6.2. The Command Interpreter

The user of a SAGA editor inputs his program in free format from the keyboard; templates are not required, and no non–terminals appear on the screen. The editor is screen–oriented; the user positions the cursor at any point within the file of text on the screen and inserts, replaces or deletes text directly at that position; the input is tokenized, parsed, and inserted into the program display window. At any time during the editing process, the user can request that the editor print the set of legal tokens (the follow set) that can be inserted at the cursor position. The user also can select more complex editor commands by using the command mode of the editor, which temporarily displays commands at the bottom of the screen. As such commands are executed, the screen is updated immediately to display the changed text. Editing commands enable the user to insert, delete, move, copy, or replace arbitrary fragments of text. These fragments can be selected by cursor positions, characters, strings, lines, syntactic constructions and eventually by semantic constructions within the text. For example, in a Pascal program, a user may select an **if** ... **then** ... **else** ... statement, discard the **else** ... part, and copy the remaining fragment to another location.

6.2.1. Basic Commands Capabilities

Since the user's text is parsed and stored in parse tree form, it is possible to take advantage of this structure through structure–oriented commands which specify operations in terms of tokens or sub–trees. But more significantly, unlike template driven syntax–directed editors, which constrain editing to limited sub–tree replacements, by basing the SAGA editor upon an incremental parser and permitting free–form input, it is possible to retain the text–oriented commands that manipulate characters and lines as well.

During the execution of an editing modification, the editor communicates with the parser through two routines: *tokenize*, which converts characters to terminal nodes, and *parse*, which integrates these nodes into the parse tree. All commands which modify the text are executed through this interface. The basic modification operation provided by the parser is to delete and/or insert a sequence of *tokens* at an arbitrary token position along the frontier of the parse tree.

It is possible to extend the editor/parser interface to permit the deletion and/or insertion of an arbitrary sequence of *characters*, at an arbitrary character position along the frontier of the tree, by defining a data structure consisting of a buffer of characters *modbuf*, a pointer *deletenode* to a node to be deleted, and a deletion count *deletecount*,

When a sequence of characters beginning in the middle of a token is to be deleted, the address of the node containing the token in which the sequence starts is assigned to *deletenode*, and *deletecount* is set to *1*. The characters from the beginning of the token, up to but not including the first one to be deleted, are copied into the beginning of *modbuf*. Then *deletecount* is incremented for each additional token that corresponds to the characters to be deleted. If new characters will also be inserted, these characters are appended to *modbuf*, which is tokenized each time it contains a complete line of input. At the end of the insertion, any characters in the last token to be deleted which are not in the character string to be deleted are copied to the end of *modbuf* before it is tokenized.

Now the parser is called, with *activenode* set to *deletenode*, *deletecount* supplying the number of nodes to be deleted, the *nextusernode* list supplying the nodes to be inserted, and *parseoption* set to the user's choice as to whether the parse should *suspend* or *complete* once the immediate modification is complete. The parser integrates the new input into the parse tree, treating any errors as discussed earlier.

Since modifications can be permitted from any character position to any other character position, it is straightforward to provide modifications on any integral number of tokens, lines, or sub–trees, as long as a mechanism is provided to the user to specify these other types of units. All other editor commands are constructed upon this basic mechanism, by decomposing more complex editing operations into sequences of this basic modification.

6.2.2. Screen Mode

When invoked in screen mode, the SAGA editor displays a screen full of text (parse tree terminal nodes), and positions the terminal's cursor on the first node displayed. The user positions this cursor and selects sections of the tree to be acted upon by an editor command. Editor commands are single control characters; a line–mode escape jumps the cursor to the bottom row of the screen to permit a line–mode editor command to be typed. The control characters are tied to the basic line–mode commands, or sequences of these commands. A map table is planned as a future extension which will enable the user to customize the editing interface.

To insert text in screen mode, it is only necessary to position the cursor at the point of the insertion and then directly type the characters to be inserted. All non–control characters are treated as data, and are placed into the text buffer to be tokenized and parsed. Once a partial line of input text has been typed, single characters may be erased by typing a backspace (control–H), and the entire line of new input erased by typing control–U. Once a newline (or carriage return) is typed, the input line is immediately tokenized, and queued for parsing. Each new line of input is treated in the same way. No (syntactic) parsing is actually done until the input is terminated via an *escape* character. Alternatively, The user may request a partial parse by terminating the input with a control–P instead. At this point, the parse is performed, and any lexical, syntactic, or semantic error highlighted on the screen. The user may repair the error right away, scroll through other parts of the file, make another editing change before the point of the error, or exit the session (to repair the error in a future session).

6.2.3. Line Mode

The line mode command syntax has the following form:

[arguments] command [parameters]

Argument	Type	Syntax
Count	character	*n*{ c ¦ C }
	integer	*n*
	line	*n* { l ¦ L }
	subtree	*n* { s ¦ S }
	token	*n* { t ¦ T }
Position	character	$@l_1$
	range	$@l_1:l_2$
	sub-tree	*n*^
	tree node	*a*#
String		"*string-of-characters*"

where: *n* is an *integer*, *, or -*.
l_1, l_2 are single letters that name editor pointers, or if absent, the terminal's cursor.
* stands for *maxint*, the maximum integer value permitted on the system.
-* stands for -*maxint*.
a is the address of a parse tree node (an unsigned integer).

Table 6-1: Editor argument types and their syntax.

Only the command name is required, and only as many characters as necessary to disambiguate it from other commands. The preceding *arguments* generally specify a section of the parse tree to be acted upon by an editing command. These arguments are evaluated by the editor's command interpreter, and placed onto an argument stack before the command is invoked. Arguments only apply to the command they directly precede, unless parentheses are used to distribute them across several commands. Not all commands take all argument types; legal ones are listed with each command, while illegal ones simply cause an *unexpected argument* error message to be displayed. In general, commands take all argument types which "make sense" for that command.

The trailing *parameters* specify additional arguments specifically for that command, and that command only. Unlike preceding arguments, trailing parameters are not evaluated before the command is invoked, but are placed on the argument stack as a string of characters. This is especially useful for the *filter* command, which executes a separate process specified by the user, passing to it these parameters as command line arguments.

In addition to the predefined commands, the user may also define new commands as sequences of already existing editor commands. This mechanism provides a convenient way to experiment with composite commands. Commands which are found to be particularly useful may be added to the basic command set for improved execution.

Nine types of arguments are presently recognized: integers; counts of characters, tokens, lines, trees; a character position; a range (a pair of character positions); a sub–tree root position; and a character string. Counts are all relative to the location of the editing cursor, positions are at a specific tree location, integers are interpreted as appropriate to the command, and character strings represent search strings, file names, and so on. The argument types and their syntax are given in Table 6–1.

6.2.4. Predefined Commands

The editor's predefined commands may be grouped according to function: positioning commands, modification commands, formatting commands, informational commands, control commands, and environmental commands. Table 6–2 presents the argument types permitted with each command. Each of these command groups is described below.

6.2.4.1. Positioning Commands

The positioning commands move the editing cursor through the text displayed on the screen (and through the frontier of the parse tree), and also place auxiliary editing pointers into the parse tree for later reference. There are four commands: *back* and *forward* for cursor positioning, and *set* and *clear* for auxiliary pointer placement. Each of these commands corresponds to a line mode command; in screen mode, characters can be mapped to either specific commands or specific argument/command pairs, so that *move–by–char, move–by–token, move–by–line* and *move–by–tree* commands can be made single key strokes, appearing as individual commands

Command	Argument Types										
	None	Count					Position			String	Parameter
		c	i	l	s	t	ch	ra	st		
Positioning											
back	x	x	x	x	x	x	x	x	x	x	
clear											x
forward	x	x	x	x	x	x	x	x	x	x	
set											x
Modification											
delete		x	x	x	x	x	x	x	x		
fread	x									x	x
insert	x									x	
parse	x										
pdelete		x	x	x	x	x	x	x	x		
pinsert	x									x	x
fwrite	x									x	x
Formatting											
cchar	x		x								
close	x		x								
ochar	x		x								
open	x		x								
Informational											
error	x						x				
follow	x						x				
help	x									x	x
print	x	x	x	x	x	x	x	x	x		
Control											
define	x									x	x
exec										x	x
loop		x	x	x	x	x	x	x	x	x	x
(...)		x	x	x	x	x	x	x	x	x	x
off										x	x
on										x	x
quit	x										
Environmental											
csh										x	x
filter		x	x	x	x	x	x	x	x	x	x
sh										x	x

Table 6–2: Basic editor commands grouped by function showing the argument types permitted with each one.

although they are actually a single command invoked with different arguments.

6.2.4.2. Modification Commands

Modification commands change the text of terminal tokens and/or the structure of the parse tree. These commands are: *delete, partial-delete, insert, partial-insert, fread, fwrite* and *parse.* The *delete* and *insert* commands request a *complete* parse, *fread* corresponds to *insert* taken from a text file, and *parse* invokes the parser at a particular location to remove a suspension point left by some earlier suspended parse.

6.2.4.3. Formatting Commands

Formatting commands rearrange the display of the text on the screen by altering the number of newlines and spaces between terminal tokens. They differ from modification commands in that neither the terminal token text nor the parse tree structure is altered, so that the parser is not invoked. Reformatting which would cause two tokens to be reevaluated into one is prohibited; a deletion command instead is required to remove the intervening spaces.

6.2.4.4. Informational Commands

These commands provide assistance to the user. Four are pre–defined: *help, error, follow–set* and *print.* The *help* command lists the editor commands and available help topics; *help keyword* provides more specific help about the command or topic supplied in *keyword.* The *error* command displays an error message for the highlighted token under the editing cursor. Only *lexical error, syntax error* and *semantic error* are provided by the editor by default; customized code must be written for one of the language–dependent modules or a filter process in order to provide more language–specific diagnostics.

The *follow–set* command asks the editor to display the set of legal tokens which could be inserted just before the token on which the editing cursor is positioned. It can be a valuable diagnostic for a user with a non–obvious syntax error to repair, for a language implementer to verify his parse tables interactively, and to acquaint a programmer with a new language. The *print* command is only necessary when the editor is run in line mode, to list portions of text.

6.2.4.5. Control Commands

The control commands affect the editor's internal environment, setting options and controlling command definition and execution. These commands consist of *parentheses, loop, define, exec, off, on* and *quit.* The *parentheses* command, specified with a pair of parentheses, groups several commands together to distribute arguments or perform an iteration. The *loop* command executes the following command until failure; applied to parentheses, it iterates over a sequence of commands until one produces an error return (e.g. *forward* when positioned at the end of the parse tree).

The *define* command associates a name with a sequence of editor commands; use of this name as a command invokes this command sequence. The *exec* command takes a file name string argument and reads and executes the editor commands specified in the file. It is used to define and execute commands in a file during editor initialization, and to pass command sequences from a filter process back to the editor for execution.

The *off* and *on* commands take several keyword arguments and set or clear corresponding Boolean variables in the editor which control its behavior. These commands are mostly used to interactively toggle debug and trace variables to monitor ad measure editor execution. Lastly, *quit* terminates an editor session.

6.2.4.6. Environmental Commands

The environmental commands affect the editor's external environment, such as its interface to the file system and other processes that are running on the system. Four are presently defined: *sh, csh, filter* and *make.* Both *sh* and *csh* pass their arguments out to UNIX *cshell* and *shell* programs for execution. When run in screen mode, results of these programs can be placed into a pop-up window on the user's terminal.

The *filter* command executes the named process, passing it the name of the file currently being edited, an optional sub-tree root or token range, and any other command-specific parameters given on the command line. Filter processes greatly increase the power of the editor by providing a modular way perform analyses on existing parse trees. More will be said about filter processes later in this chapter.

6.2.5. User-Defined Commands

Given the basic command set described above, a number of additional commands can be defined and included in all editors to provide increased functionality. A *copy* operation can be defined by *pick* and *put* commands:

```
define      pick      fwrite tempfile
define      put       fread tempfile
```

By splitting *copy* into two components, only one location need be specified at an instant, simplifying the operation since the user does not need to keep both the source and destination locations in mind at the same time. An additional variant can be created which picks up and deletes, to perform a move operation.

Commands such as *move to the end of the line* can be constructed from the predefined *move 1 line forward; move 1 character back.* Any of these can be used in screen mode by assigning a control character to the user-defined command.

Command extensibility permits us to try out different command combinations easily, and permits language-dependent operations to be defined for each different type of editor. Advanced language-dependent operations, such as tree transformations, can be performed through a separate process tied to the editor, and invoked through a user-defined command. The extensibility supports customization of the editor toward a specific development environment, and makes many more resources available to the user.

6.2.6. Screen Management

The SAGA editor employs the *Maryland Window Package* [Torek, 83] as its screen manager. This package references the */etc/termcap* terminal capability file available on UNIX systems to determine the characteristics of the terminal in use. The package supports the declaration of a text buffer and associated window into that buffer, with the window placed on some portion of the terminal screen. Multiple, overlayed windows are supported, and the package runs an algorithm to detect moved blocks of text as well as isolated modifications, and attempts to send a minimal number of characters to the terminal to update the display to correspond with its internal text image.

The package provides a flexible environment for the editor, which uses the overlaid window capability for pop-up windows that contain information such as a terminal follow set or output from a filter process run from within the editor.

6.2.7. Invoking the Editor

The editor[1] is invoked with a command of the form:

epos [options] <name>

Options are single letters preceded by a minus sign, and the names are SAGA directories containing structured files. The editor can be run in either screen-mode or line-mode, depending on the

[1] The SAGA editor has been tentatively named *epos*, until a more suitable name is found. Webster's dictionary defines *epos* as "epic poetry", appropriate since the SAGA project is investigating software development and the software life-cycle for full programming languages and grammars, not just simple subsets; and also as "an epic poem, handed down by word of mouth", appropriate for the earlier days of the editor development, since new features became available some time before they became documented!

terminal in use and the user's preference. *Epos <name>* attempts to run the editor in screen mode, and failing that, uses line mode. *Epos -l <name>* forces the editor to use line mode, regardless of the terminal's screen capabilities.

The *<name>* argument is used to create a directory which will contain the files of structured data (parse tree, symbol table, object code library, etc.) that will be produced by the editor. If already existent, *<name>* must be a directory containing the structured data files from an earlier editing session. Files in directory *<name>* should only be modified by SAGA programs, and only files created by SAGA programs should reside there. (During program execution, a number of temporary files of varying names are created, and name collisions are possible.)

Although actually a directory, <name> can be thought of conceptually as a file containing structured information, and since the user need not be concerned with the actual organization of the information in this directory, it will be referred to as a file throughout this section. <Name> must have been produced by a SAGA program recognizing the same language as the editor being invoked.

6.3. Filter Processes

The SAGA editor provides a mechanism by which separate processes can be invoked during an editing session to traverse portions of the parse tree being edited. These processes, termed *filter processes*, read, analyze and possibly transform the parse tree, returning the result to the editor. By defining new commands with the editor's user-defined command facility, which invoke filter processes, authors of filters can provide complex operations as simple commands. A tree plotter, diagrammer, compactor, rule frequency counter, pretty printer, and a Pascal tree transformation program have already been written using this facility.

Since the editor constructs a parse tree, it is a simple matter to make this tree available for additional analysis by other programs. These programs, using pre-defined library routines, walk the parse tree collecting data. They can modify some fields in the tree directly, and can transform the structure of the tree by writing a text file to be passed back to the editor to be parsed and inserted in place of some portion of the existing tree. They can also produce editor command files, to be executed once the filter process terminates. The last command in this file can invoke the filter process again, resulting in effect in a co–routine. The editor provides both user–defined command sequences and command files to facilitate the use of these programs.

The SAGA editor contains a *filter* command which takes the name of the filter process as an argument, and arranges to execute the program as a sub–process to the editor. This command automatically supplies the name of the parse tree directory as the first argument to the program, and optionally supplies a parse tree node number as a second argument if a sub–tree is selected by the user to be passed to the filter command. Any other arguments given to the filter command are passed along to the filter process after these initial arguments. Thus the filter process is executed with the following arguments:

<filtername> <parse-tree-directory> [<tree-node>] [<args to filter cmd>]

At each node in the tree, the appropriate library routine can be used to retrieve the fields of interest in the node. Should it be desired to make modifications to the tree, two approaches may be used. To transform the tree, a text file should be created into which the new text to be inserted into the tree is placed. If the *filter* command in the editor is placed into a user–defined command sequence, then additional commands in this sequence can cause the deletion of the sub–tree which was passed to the filter followed by the insertion of the new text from this file.

For more complex modifications, the filter process can created a command file which contains a combination of editor commands and input data. The user–defined command sequence which executes the filter command can then invoke the editor's *exec* command on the file produced by the filter process; commands in this file will then guide the modifications to be made.

6.4. Demand–Paged Data Structures

Pascal provides no mechanism to support random access to files. Since parse trees can grow large and the editor would like to be able to run with only a small portion of the tree memory–resident, a module was written which permits a program written in Berkeley Pascal to randomly access records in a file. The paging routine module provides an interface by which the records in this file can be accessed and modified. Only a small portion of the file needs to be memory resident at any time; the package implements a demand–pager to move the data in and out of memory as required. The programmer specifies a record to be paged and provides a buffer (an array of records) to contain a portion of the file in memory. The routines in the package can also be used to define an interface to treat the records as an encapsulated data type, and implement additional access routines to provide access to the fields in the record in an implementation independent manner.

The paging system provides access to a potentially large file of records through a possibly small area of memory available to a program. Conceptually, the file may be thought of as an array of records, the first one labeled with index 1, and with no upper bound. As higher and higher indices are referenced, additional pages are added to the file. The file is limited in size only by UNIX system imposed restrictions (typically the amount of free space on the file system containing the file).

Each record in this file can be read or written independently from all others in the file, in any order whatsoever. The programmer using the paging system simply specifies the index of the record in the file he wishes to access, and the record will be swapped into memory if not already present, and made available to him. Figure 6–5 illustrates both the concept and the implementation scheme used by the routines.

The records to be paged can be any size up to but not greater than the size of the disk page which is swapped by the operating system. On older systems, this size is typically 512 bytes, although page sizes of 1024, 4096, and 8192 bytes are also common.

Since all disk i/o is performed a page at a time, no record is stored across two pages, since this doubles the overhead to retrieve the record. So as many records as will fit onto a single page are stored on that page, and the remaining space is left as a "hole", which is not used by the paging system.

The data is stored in memory as an array of records. The user's program must contain a declaration of the record, and a pointer to an array of records to be used as a buffer to contain the pages of records which will be swapped into and out of memory by the paging system. The routines use a page table and buffer table to store the information needed to manage the data. This information is hidden from the user, and it is not necessary to understand these structures in order to use the paging routines; these structures are shown in Figure 6–5 only for completeness and the interest of the reader.

The cost of these functions is the increased overhead of a procedure call per record reference. These routines are used by the SAGA language–oriented editor to manage the parse trees which are constructed during the editing process. This results in faster response time for large programs since the entire tree does not need to be read into memory.

6.5. Summary

This chapter has covered the modules of the SAGA editor. By parsing the user's text as it is input, the editor provides additional analysis sooner than previously available, eliminating the

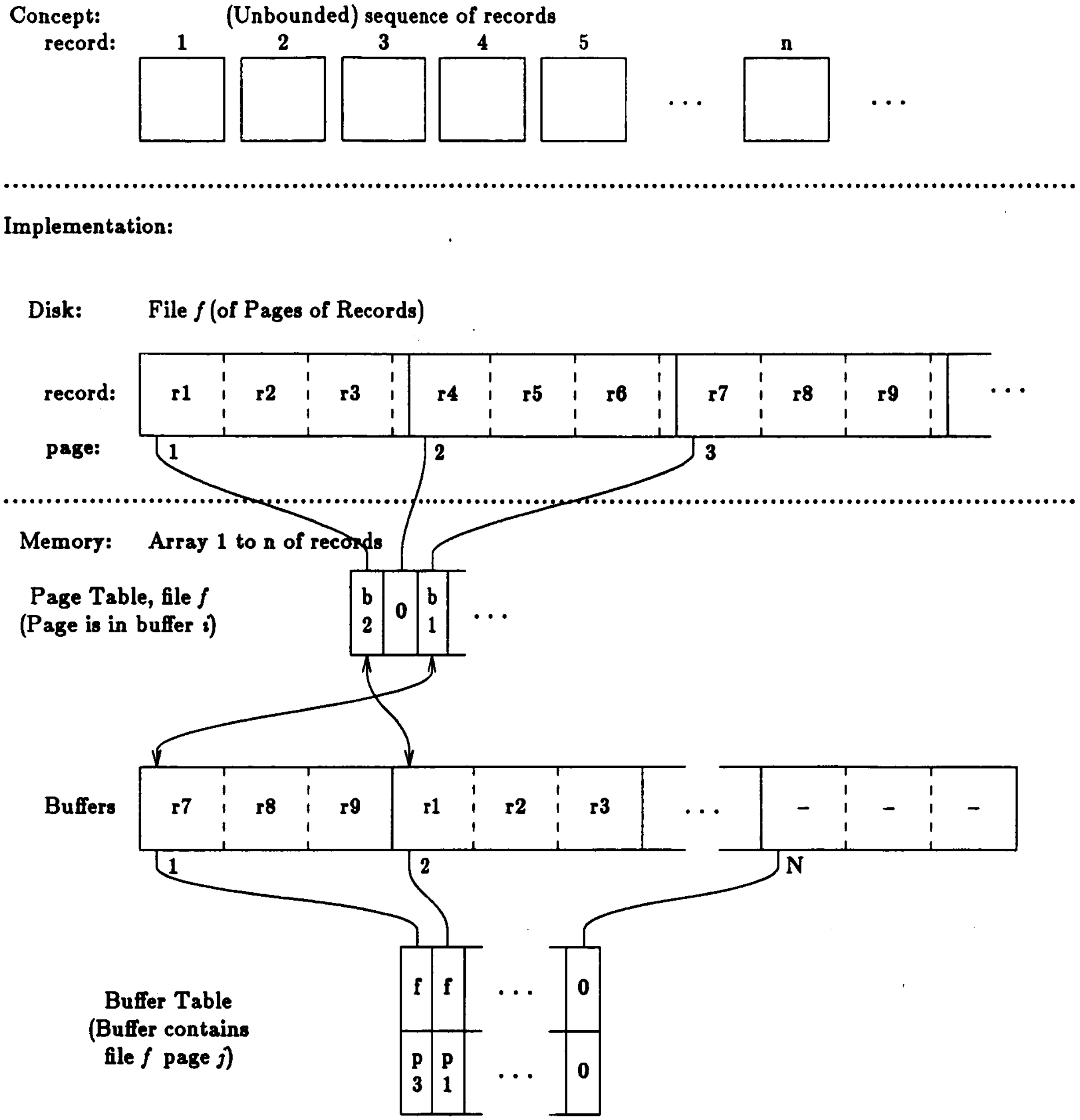


Figure 6-5: A Demand-Paged File, used for the editor's parse tree and string table. These structures are paged into memory on demand, permitting the editor to run with only a small portion of the parse tree memory resident during a editing session. The paging module is available for use with other Pascal programs, and can be used to support any data structures which can be stored as an array of records.

need to run a compiler merely to locate and repair syntax and semantic errors, and reducing the time from coding to test. By implementing the editor's command interpreter over an incremental, table-driven, LR(1) parser, it has been possible to retain common text editing commands while augmenting the user interface with structure-oriented commands which increase the level of abstraction of the user interface. This permits editing operations to be specified in terms closer to the problem at hand. Interfaces are provided to language-dependent lexical, syntactic, and semantic analysis modules, permitting the use of any parser generating system which can meet the requirements of the interface, and permitting language implementors to use a formal specification grammar or other notation with which they are familiar.

An extensible command set permits customization of the editor, and allows it to draw on other tools in the development environment to perform additional analyses and operations for the user from within the editor. Through the use of filter processes, the editor provides the capability of performing semantic analysis in a separate process running in parallel with the editor, which should lessen delays in response time when semantic processing is being performed. The use of a demand-paged data structure to store the parse tree permits use of the editor with large programs on systems with limited available memory. In the next chapter, editor generation is discussed, completing the presentation of the SAGA language-oriented editor.

CHAPTER 7

EDITOR GENERATION

The SAGA Editor has been designed to be easily retargetable to additional languages. Most of the editor's modules are language-independent, and can be used intact when an editor is produced for another language. Only the lexical, syntactic, and semantic analysis modules need to be altered, and the extent of the alterations is dependent upon the parser-generating system being used to process the language specification.

The lexical, syntactic, and semantic analysis modules are generated by or written for use with a specific parser-generator facility. The generator program reads one or more input files which contain formal descriptions of the language-specific information. This information consists of a formal description of the syntax of the language in the form of a grammar, information about the lexical representations of the tokens and semantic evaluation information in the form of executable code fragments or attributed grammars, depending on the parser-generator used. The parser-generator produces parse tables and associated information which is combined with the parser-generator dependent library routines and the common editor object code to produce an editor for a particular language. Figure 7–1 illustrates the generation of a SAGA editor.

7.1. The Mystro Parser–Generator System

The Mystro parser-generating system [Noonan and Collins, 84] uses a customized subroutine to perform the lexical analysis, a formal BNF-grammar description of the syntax of the

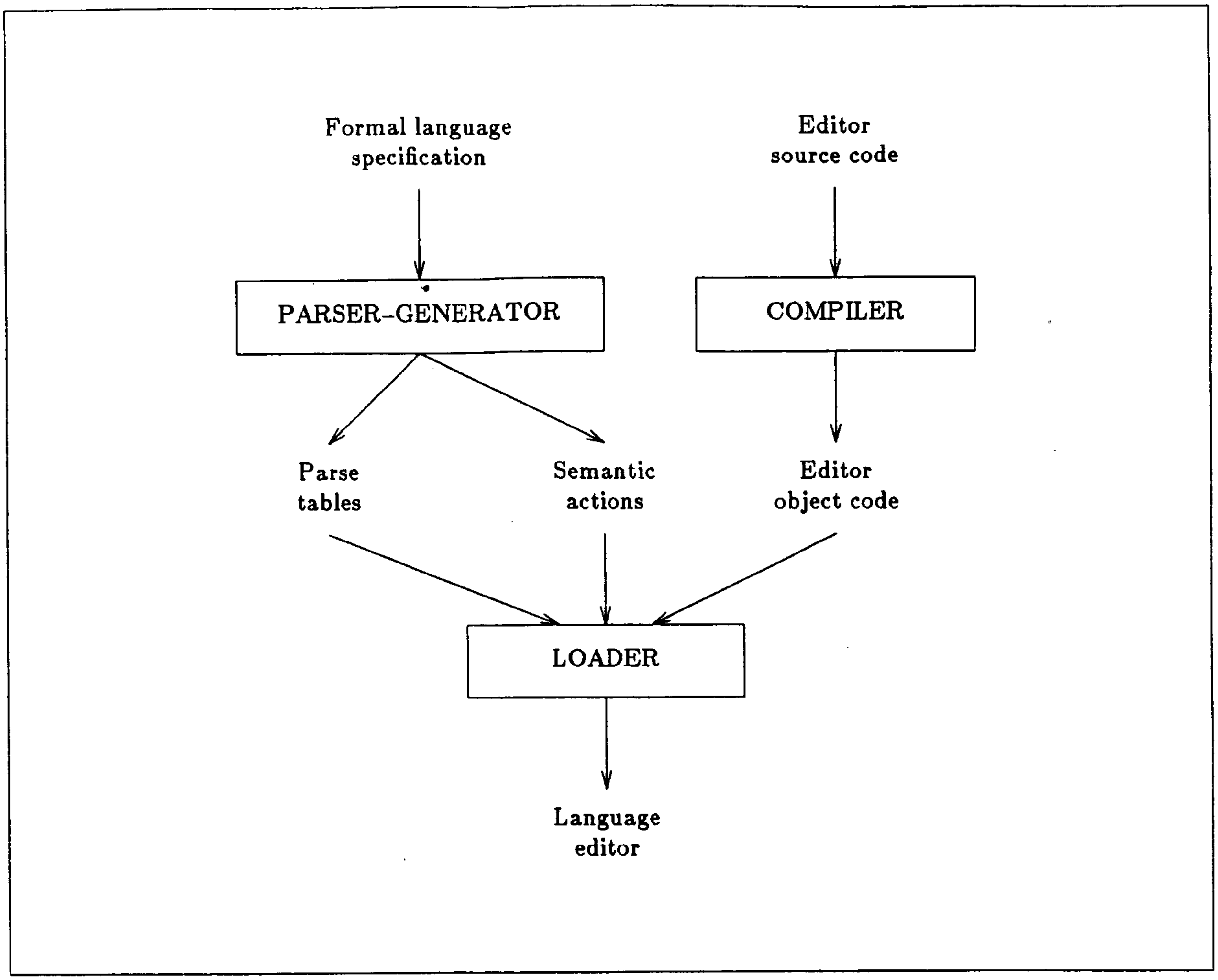


Figure 7-1: SAGA Editor Generation

language, and code fragments attached to the production rules of the grammar to perform semantic evaluations whenever a reduction by this rule is performed by the parser. The lexical analysis is accomplished in the *tokenize* routine which consists of a case statement and associated subroutines to scan the input buffer and recognize specific tokens, followed by code to construct a terminal node for this token and append it to a list to be returned to the caller of the routine. To adapt lexical analysis for another language, this routine can be copied from a file already in existence for another language and edited to insert or delete cases to/from the case statement to reflect the different lexical classes that need to be recognized for the new language. All of the code to scan the input buffer and construct the terminal nodes can be re-used unchanged. If the lexical structure for the new language is similar to that of a language which has already been specified for a SAGA editor, then the modifications are straightforward and take little time.

At the syntax analysis level, a formal BNF grammar must be specified which is LR(1); the challenge to the language implementor is to get the specification into this form, eliminating all *shift/reduce* and *reduce/reduce* conflicts. Unfortunately, at this time the Mystro system does not permit operator precedence specification and ambiguous grammars, so it is necessary to completely specify the precedence of operators in the structure of the production rules and hence the parse tree. For a language like Pascal, this is not too difficult, since there are a limited number of precedence levels. However, for a language such as C, there are so many precedence levels that the parse trees become heavy with renaming rules in the sections involving operators and operands.

The Mystro system will still produce a file of parse tables for the syntax of ambiguous grammars, though the parser will always default to using the first applicable action which it encounters. But because the tables are produced, it is possible to post-process them manually, and in many cases edit the tables and resolve the conflicts in favor of one or another. In the case of the Pascal grammar, it is possible to replace the production rules given in Figure 7-2(a) with those in 7-2(b), run the resulting ambiguous grammar through the parser-generator, and then edit the resulting tables. The effect is that all renaming rules of the form:

$$<\textit{simple-expression}> \rightarrow <\textit{term}> \rightarrow <\textit{factor}> \rightarrow <\textit{id}>$$

disappear from the parse tree and are replaced by:

$$<\textit{simple-expression}> \rightarrow <\textit{id}>$$

C-3

directly. Parse trees produced by editors for both of these grammars were analyzed for production rule frequency and tree size and it was found that the trees resulting from the ambiguous grammar contained 27% fewer nodes, a significant saving in both space and parser processing time.

The parser-generator takes the code fragments associated with the production rules in the grammar specification and combines them into a case statement indexed by rule number; this statement is placed into one of the language-dependent analysis files automatically during parser generation. The SAGA editor provides support for semantic analysis to be performed either in-sequence with the parse, or after the syntax analysis has completed. In this latter case, a separate process can be employed to perform the semantic analysis. The use of a separate process is encouraged since the semantic analysis must presently be done with code fragments, but if the lexical analysis could be made table-driven, then it would be possible to produce a single editor which loads the lexical and syntax tables at run time, instead of customized editors, instan-

```
<simple_expres> → <simple_expres> <add_op> <term> ¦ <term>
<add_op>        → + ¦ - ¦ or
<term>          → <term> <mult_op> <factor> ¦ <factor>
<mult_op>       → * ¦ / ¦ div ¦ mod ¦ and
<factor>        → <variable>
```

(a) Section of original unambiguous grammar

```
<simple_expres> → <simple_expres> <op> <variable> ¦ <variable>
<op>            → + ¦ - ¦ or ¦ * ¦ / ¦ div ¦ mod ¦ and
```

(b) Equivalent ambiguous grammar, assuming +, - and *or* are assigned lower precedence than the remaining operators.

Figure 7–2: A grammar simplification resulting in more efficient parse trees.

tiated one per language. To produce an editor for a new language, only new tables would need to be produced; these could be used with an existing editor binary, saving storage space used for the editor programs and permitting all SAGA editors to run from a single text image in memory.

The SAGA group has found the Mystro parser–generator to be a stable and reliable system, and of great use in the development of new SAGA editors. If a future version could contain a formal specification of lexical classes then the manual code modification of the tokenizing routine could be eliminated; if ambiguous grammars with precedence specification of tokens which arise in ambiguous constructs could be provided, grammars could be specified which produce potentially much more efficient parse trees. These extensions could enhance a very useful system.

7.2. The ILLIPSE Parser–Generating System

Over the past few years, work has been underway at the University of Illinois on an *interactive* parser–generator system [Mickunas, 81], [Mickunas, 86]. The *ILLInois Parsing System Editor* (ILLIPSE) permits a user to build, examine, modify and test context–free grammars interactively. A BNF–style format is used to specify the grammar to be processed. The user selects the type of parser to be generated; LR(1), LALR(1), SLR(1), and NSLR(1)[1] parsers are supported. The user then controls the generation of the sets of items for the parse states of the parser. States can be generated singly, or all at once. The user then can traverse the state tables by state number or transition, adding and deleting items, lookaheads, and transitions. Test strings can be input and parsed to check the behavior of the parser.

ILLIPSE is a very useful tool for the specification of context–free grammars. Ambiguous grammars can be input, and the ambiguities resolved interactively. Many renaming rules[2] in the grammar can be eliminated which results in smaller grammars, parse tables, and resulting parse

[1] Non–deterministic SLR.

[2] Renaming rules are production rules containing a single non–terminal on the right hand side.

trees than otherwise would be possible if an unambiguous grammar without token precedence specification needed to be used. This ability can greatly simplify the task of grammar preparation, and result in production rules which more closely match the constructs in the language.

7.3. The Olorin Parser–Generator

Work has begun in the SAGA group to produce a parser-generator which takes a formal language specification in an extended BNF syntax, with support for formally specified, incrementally evaluatable semantics [Beshers, 84], [Beshers and Campbell, 85]. A prototype generator is still in the design and implementation phase, but should be available for testing some time within the next year.

7.4. Other Parser–Generators

As already mentioned, the SAGA editor can be used with any parser-generating system which can produce tables for which code can be written to meet the requirements of the lexical, syntactic, and semantic interfaces discussed in the previous chapter. Other logical generators to use are the *lex* and *yacc* programs available on UNIX systems [Lesk, 75], [Johnson, 75]. These programs need some modification since they were designed as an encapsulated black-box lexer and parser, which perform more work than is appropriate when applied to the SAGA editor. *Yacc* both performs the syntax analysis and provides parsed output, but the SAGA editor needs structures which can be incrementally reparsed at a later time; only the syntax tables provided are usable since the editor produces its own parser output (the parse tree).

7.5. Summary

By designing the editor to be retargetable, the results of the effort that went into producing a language-oriented editor can be applied more widely. This greatly reduces the time and effort required to produce an editor for a new language. It produces software modules which may be re–used both together and separately, in related and unrelated programs as well.

The editor's modular structure permits, for example, the reuse of only the parser and subordinate modules in programs which need to manipulate parse trees automatically under program control; while such a program could feed a SAGA editor input through a pseudo–teletype interface, it will be more efficient to produce a single program which can communicate directly with the *tokenize* and *parse* routines. Other modules, such as the demand–pager for arrays of records, can find uses in unrelated applications in which a large amount of data can be accessed non–sequentially and processed in small pieces.

By permitting the interfacing of other parser–generating systems, the SAGA editor can take advantage of new systems which come along, and which may provide better support for a particular language than a generator which is currently in use. The use of modular, re–usable software enhances the software development environment, adding power and flexibility to the tasks of efficient software development.

CHAPTER 8

CONCLUSION

This dissertation has shown that a language–oriented editor for context–free languages can be based upon an incremental LR(1) parser with incremental analysis techniques. The editor has been constructed using the *recognition* approach, which permits it to retains common text editing commands while augmenting them with structure–oriented ones. It can handle full programming languages. It is superior to editors based on the *generator* approach, which implement subsets of full programming languages and provide restricted editing environments, and are unable to provide many of the operations currently available in text editors.

The editor incorporates a table–driven, incremental parser. The parser provides an environment in which syntactic errors are permitted; editing is *simplified* since structural modifications which can be tedious can be performed in several pieces. The program being edited can be taken through several intermediate, incorrect states. Since the parser permits the editor to support text–oriented commands, pre–existing code fragments in text form can be directly incorporated anywhere in the parse tree; no preprocessing is required.

We have presented our parse tree node structure, which adds attributes which are of direct benefit to an editor. These attributes permit the parse tree to be used directly by the editor's command interpreter and display module. This eliminates the need to keep an additional text representation, and the additional complexity that would be required to maintain consistency between the textual and structural forms of the data. Since a single data structure suffices for the parser and editor, no *unparser* is needed to retrieve the original syntax and formatting information.

A new solution to the handling of comments in syntax trees has been presented, which eliminates the restrictions placed upon comments by syntax-directed template editors, simplifies storage and maintenance of comments in the parse tree, and supports uniformity of access by editor commands which can reference both comments and syntactically meaningful tokens in the parse tree at the same time.

In the parsing algorithm, we have redefined the *reduce* operation, proposing an alternative which permits the parser to treat non-terminal and terminal tokens uniformly, permitting the specification of non-terminals in the input string. We have combined the parsing action and goto function into a single action. Both of these modifications eliminate duplicate code in the incremental parser, and improve its efficiency.

Explicit error handling actions have been introduced, since a working editor must be able to recover from a user's syntax errors. The error-recovery algorithm handles multiple syntax errors, and permits editing of the parse tree in the midst of errors.

The editor is screen-oriented: It displays the parse tree terminal nodes in text form, no non-terminal nodes appear on the screen, so that the programmer need not know the specific construction of the production rules in the grammar defining the language in order to be able to use the editor. A command is provided to display the set of legal tokens which can appear at a given location in the tree. This feature can aid programmers who are learning a new language, as well as provide diagnostic support to aid in the repair of difficult syntax errors.

The editor is flexible and supports a higher-level command interface which includes both structure-oriented commands and common text editing commands. This editor can be used to develop practical programs which incorporate software engineering principles concerning the design and construction of software systems. A prototype editor which employs these algorithms was implemented beginning in 1981 as a demonstration of the practicality and flexibility of this approach; this editor has been in experimental use over the past couple of years.

The editor is a part of the SAGA system, which is directed towards experienced programmers, who if anything need additional editing flexibility and analysis, and not a tightly constrained environment with restrictive editing options.

The editor's modular structure supports the reuse of code when constructing editors for other languages, making the majority of its code language-independent. By basing it upon standard table-driven LR parser technology, the editor can be used with many of the already existing parser-generator programs which have been independently developed, improving its applicability.

In summary, the construction of a language-oriented editor based upon the recognition approach is very flexible and has several advantages:

1) The technique can be applied *consistently* to the lexical, syntactic, and semantic components of the language. (Many language-oriented editors based on a generation approach nevertheless depend upon the recognition of valid primitive expressions of the language.) We believe this consistency simplifies the implementation of a uniform set of basic editing commands such as insert, delete, move and copy for the lexical, syntactic and semantic components of the language.

2) The approach permits *arbitrary editing operations* on the program. Editors that use the generation approach cannot permit arbitrary changes and often require particular syntactic transformations to be implemented as special cases.

3) The approach *facilitates program maintenance and modification.* It is often simpler to transform an existing program into a desired program if the editing commands can take a program through a sequence of intermediate invalid forms. In addition, these invalid programs may be saved between editing sessions. Such program transformations are difficult to implement using an editor based on the generator approach.

4) Arbitrary lines of *existing program text can be inserted anywhere* into the text of a new program. This allows the editor to be used to combine two different versions of a program in an arbitrary manner to produce a new version.

5) The design of a facility to produce language–oriented editors is simplified if *existing compiler generation and parsing techniques and tools can be employed* without major alteration. If standard compiler generation and parsing techniques are used, then many existing specifications of the lexical, syntactic, and semantic components of a programming language can be used directly by the facility to produce corresponding language–oriented editors.

The editor runs on a DEC VAX 11/780 under the 4.2BSD UNIX operating system. Editors have been created for Pascal, C, Ada, and FP. Most experimentation has involved the Pascal editor, and we have found that enough additional processing is performed that a fast or dedicated processor is necessary to provide reasonable response times, but that with such a processor, the apparent response time perceived by the user is as good as with a text editor. The parse trees for Pascal, using a non–ambiguous grammar, take about ten times as much space as the equivalent text representation. Using an ambiguous grammar, and eliminating renaming rules in expressions, we have found that we can reduce the size of the tree to seven times that required for the text. Additional semantic information will increase this size somewhat.

Since a dedicated processor is desirable, the editor has been ported to a workstation environment. It runs on a Sun workstation under the 4.2BSD UNIX operating system. We have found that a workstation provides an ideal environment for such an editor, since the processing is adequate for its needs and the large amount of available memory permits efficient editing of large programs. Response time is adequate, and the multi–process window environment provided by the system software promotes good interaction between the editor and other tools used in a software development environment.

Looking beyond the editor into the development environment in which it beginning to be run, the parse trees produced by the editor can serve as a uniform data structure for many other tools. Additional programs can easily be written to perform additional analyses or operations on these parse trees. Editors can be produced for *specification* and *design* languages, and tools written to transform parse trees produced by one level into a form suitable for the next. Applied in an integrated modular environment, the editor can take advantage of dependency information to generate displays, noting the interrelationships among components in a system under development; research into such an environment has been performed [Kirslis et al., 85], and is continuing [Terwilliger and Campbell, 86].

The editor has also been used to support the research in several Master's Theses, one covering analysis of changes to semantic scopes [Badger, 84], another implementing a symbol table for use with the editor [Richards, 84], and a third interfacing the editor's parse tree to a code generator [Kimball, 85]. It has been used to support the development of software tools, written as class projects for software engineering classes offered by the Department of Computer Science at the University of Illinois. Among the projects were a tree transformation tool for Pascal and a program slicer for data flow analysis. The editor is being extended to include semantic analysis [Beshers, 84], a table driven lexical analysis based on *lex*, and a table-driven command interpreter for the editor that will permit formal specification of the editor's command language.

The research into an editor based on the recognition approach has shown this approach to be feasible, and through our initial experiments with it, we believe that the prototype editor implemented with this approach is refinable into a practical tool which will better support programmers and enhance the software development process.

REFERENCES

[Aho and Ullman, 77] Aho, Alfred V. and Jeffrey D. Ullman, *Principles of Compiler Design*, Addison-Wesley, 1977, chapter 6.

[Aho and Ullman, 72] Aho, Alfred V. and Jeffrey D. Ullman, *Theory of Parsing, Translation and Compiling, Vol. I*, Prentice Hall, Englewood Cliffs, N.J., 1972.

[ARM, 83] Ada Joint Program Office, *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A-1983, U.S. Dept. of Defense, February, 1983.

[Backus, 78] Backus, J., "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM*, Vol. 21, No. 8, August, 1978, pp. 613-641.

[Baden, 83] Baden, S., *Berkeley FP User's Manual*, Revision 4.1 (4.1BSD UNIX), University of California at Berkeley, July, 1983.

[Badger, 84] Badger, W. H., *MAKE: A Separate Compilation Facility for the SAGA Environment*, M. S. Thesis, Dept. Computer Science, Univ. of Illinois at Urbana-Champaign, 1984.

[Bauer et al., 77] Bauer, F., J. Dennis, G. Goos, C. Gotlieb, R. Graham, M. Griffiths, H. Helms, B. Morton, P. Poole, D. Tsichritzis, and W. Waite, *Software Engineering -- An Advanced Course*, F. Bauer, Ed., Springer-Verlag, New York, 1977.

[Beshers, 84] Beshers, G., *Regular Right Part Grammars and Incrementally Updatable Attributes for Language Oriented Editors*, Preliminary Examination Statement, Dept. Computer Science, Univ. Illinois at Urbana-Champaign, July 1984.

[Beshers and Campbell, 85] Beshers, G. and Roy H. Campbell, "Maintained and Constructor Attributes", *Proc. ACM SIGPLAN Symposium on Language Issues in Programming Environments*, SIGPLAN Notices, Vol. 20, No. 7, July 1985.

[Campbell and Kirslis, 84] Campbell, R. H., and Peter A. Kirslis, "The SAGA Project: A System for Software Development," *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Software Engg. Notes Vol 9., No. 3; SIGPLAN Notices Vol. 19, No. 5, May 1984.

[Campbell and Richards, 81] Campbell, R. and P. Richards, "SAGA: A System to Automate the Management of Software Production," *Proc. National Computer Conference*, Chicago IL, May, pp231-234.

[Carmody et al., 68] Carmody, S, W. Gross, T. H. Nelson, D. E. Rice and A. van Dam, "A hypertext editing system for the /360," in *Pertinent Concepts in Computer Graphics*, M. Faiman and J. Nievergelt, Editors, The University of Illinois Press, Urbana, Il., 1969.

[Celentano, 78] Celentano, A., "Incremental LR Parsers," *Acta Informatica* 10, 1978.

[Conway and Constable, 76] Conway, R., and R. Constable, *PL/CS -- A Disciplined Subset of PL/I*, Technical Report 76-293, Dept. Computer Science, Cornell, 1976.

[Donzeau-Gouge et al., 75] Donzeau-Gouge, V., G. Huet, G. Kahn, B. Lang and J. J. Levy, *A Structure Oriented Program Editor: A first step towards computer assisted programming*, Technical Report 114, IRIA-LABORIA, Rocquencourt, France, April, 1975.

[Donzeau-Gouge et al., 79] Donzeau-Gouge, V., G. Huet, G. Kahn, and B. Lang, *The MENTOR Program Manipulation System*, IRIA-Laboria, Aug. 1979.

[Donzeau-Gouge et al., 80] Donzeau-Gouge, V., G. Huet, G. Kahn, and B. Lang, *Programming environments based on structure editors: the MENTOR experience*, Technical Report, INRIA, Rocquencourt, France, May, 1980.

[Englebart and English, 68] Englebart, D., and W. English, "A research center for augmenting human intellect," *Proc. AFIPS Fall Joint Computer Conference*, Vol. 33, Part 1, Arlington, VA., 1968.

[Fraser, 81] Fraser, C., "Syntax-Directed Editing of General Data Structures," *SIGPLAN Notices*, Vol. 16, No. 6, pp. 17–21, June, 1981.

[Ghezzi and Mandrioli, 80] Ghezzi, C. and D. Mandrioli, "Augmenting Parsers to Support Incrementality," *Journal of the ACM*, Vol. 27, No. 3, July, 1980.

[Ghezzi and Mandrioli, 79] Ghezzi, C., and D. Mandrioli, "Incremental Parsing," *ACM Transactions on Programming Languages and Systems*, Vol. 1, No. 1, July, 1979.

[Habermann, 79] Habermann, A. N., *An Overview of the Gandalf Project*, Computer Science Research Review 1978–79, Carnegie-Mellon University, 1979.

[Hansen, 70] Hansen, W. J., "Graphic Editing of Structured Text," *Proc., Computer Graphics 1970 International Symposium*, Brunel University, Uxbridge, Middlesex, England, V. 3, Sect. 7, April, 1970.

[Hansen, 71] Hansen, W. J., *Creation of Hierarchic Text with a Computer Display*, Ph. D. Thesis, Dept. Computer Science, Stanford University, reprinted as Argonne National Laboratory Report ANL-7818, Mathematics and Computers, June, 1971.

[Hopcroft and Ullman, 69] Hopcroft, J. E. and J. D. Ullman, *Formal Languages and Their Relation to Automata* Addison-Wesley, 1969.

[Hopcroft and Ullman, 79] Hopcroft, J. E. and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[Horton, 81] Horton, M. R., *Design of a Multi Language Editor with Static Error Detection Capabilities*, Ph. D. Thesis, Electronics Res. Lab. Tech. Report No. 81/53, Univ. of California, Berkeley, July, 1981.

[Jensen and Wirth, 74] Jensen, K. and N. Wirth, *Pascal User Manual and Report*, 2nd ed., Springer-Verlag, 1974.

[Johnson, 75] Johnson, S., *YACC -- Yet Another Compiler-Compiler*, Technical Report, Bell Labs, Murray Hill, N. J., 1975.

[Joy and Horton, 80] W. Joy and M. Horton, "An Introduction to Display Editing with Vi," *Berkeley UNIX Programmer's Manual*, September, 1980.

[Kaplan, 85] Kaplan, S., *Specification of Context Conditions of Programming Languages: The CFF/AML Approach*, Ph.D. dissertation, University of Cape Town, South Africa, 1985.

[Kernighan and Ritchie, 78] Kernighan, B., and D. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

[Kimball, 85] Kimball, J. J., *PCG: A Prototype Incremental Compilation Facility for the SAGA Environment*, M. S. Thesis, Dept. Computer Science, Univ. of Illinois at Urbana-Champaign, 1985.

[Kirslis et al., 85] Kirslis, P. A., R. B. Terwilliger, and R. H. Campbell, "The SAGA Approach to Large Program Development in an Integrated Modular Environment," *Proc. GTE Workshop on Software Engg. Environments for Programming-in-the-Large*, June, 1985.

[Knuth, 73] D. E. Knuth, *The Art of Computer Programming, Second Edition*, Vol. 1, Addison-Wesley, 1973.

[Lesk, 75] Lesk, M. E., *Lex: A Lexical Analyzer Generator*, Computer Science Tech. Report #39, Bell Laboratories, Murray Hill, N. J., October, 1975.

[Medina-Mora and Feiler, 81] Medina-Mora, R., and P. Feiler, "An Incremental Programming Environment," *IEEE Transactions on Software Engineering*, Vol. 7, No. 5, Sept. 1981.

[Meyrowitz and van Dam, 82] Meyrowitz, N., and A. van Dam, "Interactive Editing Systems," *ACM Computing Surveys*, Vol. 14, No. 3, Sept. 1982.

[Mickunas, 81] Mickunas, M. D., *The Illinois Parsing System Editor -- Working Documentation*, Dept. Computer Science, Univ. of Illinois at Urbana-Champaign, Oct. 1981 (unpublished).

[Mickunas, 86] Mickunas, M. D., *Illipse: The Illinois Parsing System Editor*, Technical Report UIUCDCS-R-86-1235, Dept. Computer Science, Univ. of Illinois at Urbana-Champaign, 1986.

[Morris and Schwartz, 81] Morris, J., and M. Schwartz, "The Design of a Language-Directed Editor for Block Structured Languages," *Proc. ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, Portland, OR., June, 1981, in SIGPLAN Notices, 16(6):28-33, June, 1981.

[Noonan and Collins, 84] Noonan, R. E. and W. R. Collins, *PARSER GENERATOR GUIDE: The Mystro System Version 7.0*, Dept. Computer Science, College of William and Mary, Williamsburg, Virginia, September 1984.

[Orailoglu, 83] Orailoglu, A., *Software Design Issues in the Implementation of Hierarchical, Display Editors*, Technical Report No. UIUCDCS-R-83-1139, Dept. Computer Science, Univ. of Illinois at Urbana-Champaign, September 1983.

[Osterweil, 83] Osterweil, L. J., "Toolpack - An Experimental Software Development Environment Research Project," *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 6, pp. 673-685, Nov. 1983.

[Osterweil and Cowell, 83] Osterweil, L. J., and W. R. Cowell, "The TOOLPACK/IST Programming Environment," *IEEE SOFTFAIR*, Arlington VA, July, 1983, pp. 326-333.

[Osterweil, 82] Osterweil, L. J., "Toolpack - An Experimental Software Development Environment Research Project," *IEEE 6th International Conference on Software Engineering*, Sept. 1982, pp. 166-175.

[Reid and Hanson, 81] Reid, Brian K., and David Hanson, "An Annotated Bibliography of Background Material on Text Manipulation," *SIGPLAN Notices*, Vol. 16, No. 6, pp. 157-160, June, 1981.

[Reiss, 84] Reiss, S., "Graphical Program Development with PECAN Program Development Systems", *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical*

Software Development Environments, Software Engg. Notes Vol 9., No. 3; SIGPLAN Notices Vol. 19, No. 5, May 1984.

[Reps, 82] Reps, T., *Generating Language-based Environments*, Ph.D. Thesis, Dept. Computer Science, Cornell Univ., August, 1982.

[Reps et al., 83] Reps, T., T. Teitelbaum, and A. Demers, "Incremental Context-Dependent Analysis for Language-Based Editors," *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 3, pp. 449-477, July, 1983.

[Reynolds, 70] Reynolds, J. C., "GEDANKEN - A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept," *Communications of the ACM*, Vol. 13, No. 5, May 1970, pp. 308-319.

[Richards, 84] Richards, P., *A Prototype Symbol Table Manager for the SAGA Environment*, Master's Thesis, Dept. Computer Science, Univ. Illinois at Urbana-Champaign, July 1984.

[Shilling, 85] Shilling, J., *FRED: A Program Development Tool*, Technical Report No. UIUCDCS-R-85-1224, Dept. Computer Science, Univ. of Illinois, Urbana, Illinois, September 1985.

[Teitelbaum, 79] Teitelbaum, T., *The Cornell Program Synthesizer: A Microcomputer Implementation of PL/CS*, Technical Report TR79-370, Dept. of Computer Science, Cornell University, June, 1979.

[Teitelbaum and Reps, 81] Teitelbaum, T; and T. Reps; "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Communications of the ACM*, Vol. 24, No. 9, Sept., 1981.

[Teitelman, 77] Teitelman, W., *A Display Oriented Programmer's Assistant*, Technical Report SSL-79-9, Xerox Palo Alto Research Center, Palo Alto, CA., March, 1977.

[Terwilliger and Campbell, 86] Terwilliger, R. B. and R. H. Campbell, "ENCOMPASS: A SAGA Based Environment for the Composition of Programs and Specifications." *Proc. 19th Hawaii International Conference on System Science*, January, 1986, to appear.

[Torek, 83] Torek, C., *The Maryland Window Library*, Dept. Computer Science, University of Maryland, College Park, Maryland, 1983.

[van Dam and Rice, 71] van Dam, A., and D. Rice, "On-line text editing: A Survey," *ACM Computing Surveys*, Vol. 3, No. 3, pp. 93-114, September, 1971.

[Waters, 82] Waters, R. C., "Program Editors Should Not Abandon Text Oriented Commands," *SIGPLAN Notices*, Vol. 17, No. 7, July, 1982.

[Wilcox et al., 76] Wilcox, T. R., A. M. Davis, and M. H. Tindall, "The Design and Implementation of a Table-driven, Interactive Diagnostic Programming System," *Communications of the ACM*, Vol. 19, No. 11, pp. 609-616, Nov. 1976.

APPENDIX A

LALR(1) GRAMMARS

The LALR(1) grammars used to produce SAGA editors for the Ada, FP, and Pascal programming languages are collected here. The grammars are presented as they appear in the listing file produced by the Mystro parser-generator. Some additional statistics about the parser generated by Mystro are also presented.

The grammar for Ada [ARM, 83] is based upon [Wetherell, 81], with some corrections. We have not yet tested our editor against the validation suite supplied by the Ada Joint Program Office, but we have run numerous tests, all of which the grammar has successfully passed. The grammar for the Functional Programming Language [Backus, 78] is based upon the 4.2BSD UNIX implementation [Baden, 83]. The Pascal grammar is based upon the description in [Jensen and Wirth, 74] and revised to include specific constructs which are permitted by the Berkeley 4.2 Pascal compiler.

For lexical analysis, the Mystro parser-generator requires Pascal code fragments to be written which recognize the generic classes of the terminal tokens of the language. (The reserved words, operators, and punctuation are collected and put into a table by the parser-generator.) These fragments are included in a lexical analysis module. Since the lexical classes of each of these languages are readily available in user's manuals for the languages, the code fragments giving the lexical specifications have been omitted here to conserve space. In the grammars presented here, these generic lexical classes are represented by non-terminal tokens of the form

<class...>, which appear on the right hand sides of productions and have no left hand side definition.

Mystro permits Pascal code fragments which perform semantic actions to accompany the production rules of the grammar. These fragments are collected into a case statement which is indexed by the production rule number. This case statement is included in the SAGA editor, and is executed each time a reduction is performed during the parse. No semantic actions are shown with the grammars presented here.

The binary parse tables for Ada take about 20k bytes of storage; for FP, about 4k bytes; and for Pascal, about 8k bytes. These figures include storage for the text names of the non-terminal tokens in the grammar.

When a grammar is analyzed, Mystro produces some additional statistics about the parser. These statistics are presented below, followed by the three grammars.

Ada Parser Statistics

```
A total of 432 rules containing 292 symbols were read from "Ada.g."
470 states and 10014 items have been constructed. Compute slr(1) follow set.
15 collisions are not slr(1)–resolvable, but all states are at least lalr(1).
6470 actions constructed for the actions array.
```

FP Parser Statistics

```
A total of 100 rules containing 97 symbols were read from "FP.g."
31 states and 1092 items have been constructed. Compute slr(1) follow set.
1 collision is not slr(1)–resolvable, but all states are at least lalr(1).
1029 actions constructed for the actions array.
```

Pascal Parser Statistics

```
A total of 217 rules containing 166 symbols were read from "Pascal.g."
209 states and 2537 items have been constructed. Compute slr(1) follow set.
3 collisions are not slr(1)–resolvable, but all states are at least lalr(1).
1929 actions constructed for the actions array.
```

```
 Input grammar.    Grammar option.    Default: on

  The goal symbol <system_goal_symbol> is found in rule 1.

  [  1] <system_goal_symbol> ::= <compilation_eof>
  [  2] <compilation_eof>    ::= <compilation> <eof>
  [  3] <compilation_eof>    ::= <eof>
  [  4] <compilation>        ::= <compilation_unit>
  [  5] <compilation>        ::= <compilation> <compilation_unit>
  [  6] <compilation_unit>   ::= <context_spec> <subpgm_decl>
  [  7] <compilation_unit>   ::= <context_spec> <subpgm_body>
  [  8] <compilation_unit>   ::= <context_spec> <pkg_decl>
  [  9] <compilation_unit>   ::= <context_spec> <pkg_body>
  [ 10] <compilation_unit>   ::= <context_spec> <subunit>
  [ 11] <context_spec>       ::= <with_use_list>
  [ 12] <with_use_list>      ::=
  [ 13] <with_use_list>      ::= <with_use_list> <with_clause>
  [ 14] <with_use_list>      ::= <with_use_list> <with_clause> <use_clause>
  [ 15] <with_use_list>      ::= <with_use_list> <pragma>
  [ 16] <with_clause>        ::= with <unit_name_list> ;
  [ 17] <unit_name_list>     ::= <unit_name>
  [ 18] <unit_name_list>     ::= <unit_name_list> , <unit_name>
  [ 19] <pragma>             ::= pragma <identifier> ;
  [ 20] <pragma>             ::= pragma <identifier> <arg_list> ;
  [ 21] <use_clause>         ::= use <pkg_name_list> ;
  [ 22] <pkg_name_list>      ::= <pkg_name>
  [ 23] <pkg_name_list>      ::= <pkg_name_list> , <pkg_name>
  [ 24] <subpgm_decl>        ::= <subpgm_spec> ;
  [ 25] <subpgm_decl>        ::= <gnrc_subpgm_decl>
  [ 26] <subpgm_decl>        ::= <gnrc_subpgm_inst>
  [ 27] <subpgm_spec>        ::= procedure <identifier>
  [ 28] <subpgm_spec>        ::= procedure <identifier> <frml_part>
  [ 29] <subpgm_spec>        ::= function <designator> return <subtype_ind>
  [ 30] <subpgm_spec>        ::= function <designator> <frml_part> return
                                 <subtype_ind>
  [ 31] <designator>         ::= <identifier>
  [ 32] <designator>         ::= <op_symbol>
  [ 33] <frml_part>          ::= ( <parm_decl_list> )
  [ 34] <parm_decl_list>     ::= <parm_decl>
  [ 35] <parm_decl_list>     ::= <parm_decl_list> ; <parm_decl>
  [ 36] <parm_decl>          ::= <identifier_list> : <mode> <subtype_ind>
  [ 37] <parm_decl>          ::= <identifier_list> : <mode> <subtype_ind> :=
                                 <expr>
  [ 38] <mode>               ::=
  [ 39] <mode>               ::= in
  [ 40] <mode>               ::= out
  [ 41] <mode>               ::= in out
  [ 42] <subpgm_body>        ::= <subpgm_spec> is <decl_part> begin
                                 <seq_of_stmts> <excepts_opt> end
                                 <designator_opt> ;
```

```
[ 43] <excepts_opt>          ::=
[ 44] <excepts_opt>          ::= exception <except_hand_list>
[ 45] <except_hand_list>     ::= <except_handler>
[ 46] <except_hand_list>     ::= <except_hand_list> <except_handler>
[ 47] <designator_opt>       ::=
[ 48] <designator_opt>       ::= <designator>
[ 49] <pkg_decl>             ::= <pkg_spec> ;
[ 50] <pkg_decl>             ::= <gnrc_pkg_decl>
[ 51] <pkg_decl>             ::= <gnrc_pkg_inst>
[ 52] <pkg_spec>             ::= package <identifier> is <decl_item_list>
                                 <private_part_opt> end
[ 53] <pkg_spec>             ::= package <identifier> is <decl_item_list>
                                 <private_part_opt> end <identifier>
[ 54] <pkg_body>             ::= package body <identifier> is <decl_part>
                                 <pkg_body_part_opt> end ;
[ 55] <pkg_body>             ::= package body <identifier> is <decl_part>
                                 <pkg_body_part_opt> end <identifier> ;
[ 56] <decl_item_list>       ::=
[ 57] <decl_item_list>       ::= <decl_item> <decl_item_list>
[ 58] <private_part_opt>     ::=
[ 59] <private_part_opt>     ::= private <decl_item_list>
[ 60] <repr_spec_list>       ::= <repr_spec_list> <pragma>
[ 61] <repr_spec_list>       ::= <repr_spec_list> <repr_spec>
[ 62] <repr_spec_list>       ::=
[ 63] <pkg_body_part_opt>    ::=
[ 64] <pkg_body_part_opt>    ::= begin <seq_of_stmts> <excepts_opt>
[ 65] <subunit>              ::= separate ( <unit_name> ) <body>
[ 66] <body_stub>            ::= <subpgm_spec> is separate ;
[ 67] <body_stub>            ::= package body <identifier> is separate ;
[ 68] <body_stub>            ::= task body <identifier> is separate ;
[ 69] <decl_part>            ::=
[ 70] <decl_part>            ::= <decl_part> <decl_item>
[ 71] <decl_part>            ::= <decl_part> <pgm_comp>
[ 72] <decl_item>            ::= <decl>
[ 73] <decl_item>            ::= <repr_spec>
[ 74] <decl_item>            ::= <use_clause>
[ 75] <decl_item>            ::= <pragma>
[ 76] <pgm_comp>             ::= <body>
[ 77] <pgm_comp>             ::= <body_stub>
[ 78] <body>                 ::= <subpgm_body>
[ 79] <body>                 ::= <pkg_body>
[ 80] <body>                 ::= <task_body>
[ 81] <task_decl>            ::= <task_spec>
[ 82] <task_spec>            ::= task <identifier> ;
[ 83] <task_spec>            ::= task type <identifier> ;
[ 84] <task_spec>            ::= task <identifier> <task_spec_part> ;
[ 85] <task_spec>            ::= task type <identifier> <task_spec_part> ;
[ 86] <task_spec_part>       ::= is <entry_decl_list> <repr_spec_list> end
[ 87] <task_spec_part>       ::= is <entry_decl_list> <repr_spec_list> end
                                 <identifier>
```

```
[ 88] <entry_decl_list>     ::=
[ 89] <entry_decl_list>     ::= <entry_decl_list> <entry_decl>
[ 90] <task_body>           ::= task body <identifier> is <decl_part> begin
                                <seq_of_stmts> <excepts_opt> end ;
[ 91] <task_body>           ::= task body <identifier> is <decl_part> begin
                                <seq_of_stmts> <excepts_opt> end <identifier> ;
[ 92] <decl>                ::= <object_decl>
[ 93] <decl>                ::= <type_decl>
[ 94] <decl>                ::= <subpgm_decl>
[ 95] <decl>                ::= <task_decl>
[ 96] <decl>                ::= <renaming_decl>
[ 97] <decl>                ::= <number_decl>
[ 98] <decl>                ::= <subtype_decl>
[ 99] <decl>                ::= <pkg_decl>
[100] <decl>                ::= <except_decl>
[101] <object_decl>         ::= <identifier_list> : <subtype_ind> <init_opt> ;
[102] <object_decl>         ::= <identifier_list> : <array_type_def> <init_opt>
                                ;
[103] <object_decl>         ::= <identifier_list> : constant <subtype_ind>
                                <init_opt> ;
[104] <object_decl>         ::= <identifier_list> : constant <array_type_def>
                                <init_opt> ;
[105] <init_opt>            ::=
[106] <init_opt>            ::= := <expr>
[107] <number_decl>         ::= <identifier_list> : constant := <literal_expr>
                                ;
[108] <identifier_list>     ::= <identifier>
[109] <identifier_list>     ::= <identifier_list> , <identifier>
[110] <type_decl>           ::= type <identifier> <discr_part_opt> is
                                <type_def> ;
[111] <type_decl>           ::= <incompl_type_decl>
[112] <discr_part_opt>      ::=
[113] <discr_part_opt>      ::= <discr_part>
[114] <type_def>            ::= <enum_type_def>
[115] <type_def>            ::= <real_type_def>
[116] <type_def>            ::= <record_type_def>
[117] <type_def>            ::= <derived_type_def>
[118] <type_def>            ::= <integer_type_def>
[119] <type_def>            ::= <array_type_def>
[120] <type_def>            ::= <access_type_def>
[121] <type_def>            ::= <private_type_def>
[122] <subtype_decl>        ::= subtype <identifier> is <subtype_ind> ;
[123] <subtype_ind>         ::= <name>
[124] <subtype_ind>         ::= <name> <range_constr>
[125] <subtype_ind>         ::= <name> <accuracy_constr>
[126] <derived_type_def>    ::= new <subtype_ind>
[127] <range_constr>        ::= range <range>
[128] <range>               ::= <simple_expr> .. <simple_expr>
[129] <enum_type_def>       ::= ( <enum_literal_list> )
[130] <enum_literal_list>   ::= <enum_literal>
```

```
[131] <enum_literal_list>   ::= <enum_literal_list> , <enum_literal>
[132] <enum_literal>        ::= <identifier>
[133] <enum_literal>        ::= <character>
[134] <integer_type_def>    ::= <range_constr>
[135] <real_type_def>       ::= <accuracy_constr>
[136] <accuracy_constr>     ::= <float_pt_constr>
[137] <accuracy_constr>     ::= <fixed_pt_constr>
[138] <float_pt_constr>     ::= digits <static_simple_expr>
[139] <float_pt_constr>     ::= digits <static_simple_expr> <range_constr>
[140] <fixed_pt_constr>     ::= delta <static_simple_expr>
[141] <fixed_pt_constr>     ::= delta <static_simple_expr> <range_constr>
[142] <array_type_def>      ::= array ( <index_list> ) of <comp_subtype_ind>
[143] <array_type_def>      ::= array <arg_list> of <comp_subtype_ind>
[144] <index_list>          ::= <index>
[145] <index_list>          ::= <index_list> , <index>
[146] <index>               ::= <name> range <>
[147] <discrete_range>      ::= <name>
[148] <discrete_range>      ::= <name> <range_constr>
[149] <discrete_range>      ::= <range>
[150] <record_type_def>     ::= record <comp_list> end record
[151] <comp_list>           ::= <comp_decl_list>
[152] <comp_list>           ::= <comp_decl_list> <variant_part>
[153] <comp_list>           ::= null ;
[154] <comp_decl_list>      ::=
[155] <comp_decl_list>      ::= <comp_decl_list> <comp_decl>
[156] <comp_decl>           ::= <identifier_list> : <subtype_ind> <init_opt> ;
[157] <comp_decl>           ::= <identifier_list> : <array_type_def> <init_opt>
                                ;
[158] <discr_part>          ::= ( <discr_decl_list> )
[159] <discr_decl_list>     ::= <discr_decl>
[160] <discr_decl_list>     ::= <discr_decl_list> ; <discr_decl>
[161] <discr_decl>          ::= <identifier_list> : <subtype_ind> <init_opt>
[162] <variant_part>        ::= case <name> is <variant_elt_list> end case ;
[163] <variant_elt_list>    ::=
[164] <variant_elt_list>    ::= <variant_elt_list> when <choice_list> =>
                                <comp_list>
[165] <choice_list>         ::= <choice>
[166] <choice_list>         ::= <choice_list> ! <choice>
[167] <choice>              ::= <simple_expr>
[168] <choice>              ::= <name> <range_constr>
[169] <choice>              ::= <range>
[170] <choice>              ::= others
[171] <access_type_def>     ::= access <subtype_ind>
[172] <incompl_type_decl>   ::= type <identifier> ;
[173] <incompl_type_decl>   ::= type <identifier> <discr_part> ;
[174] <expr>                ::= <rel>
[175] <expr>                ::= <rel_and_list>
[176] <expr>                ::= <rel_or_list>
[177] <expr>                ::= <rel_xor_list>
[178] <expr>                ::= <rel_and_then_list>
```

```
[179] <expr>                 ::= <rel_or_else_list>
[180] <expr>                 ::= <classplace>
[181] <rel_and_list>         ::= <rel> and <rel>
[182] <rel_and_list>         ::= <rel_and_list> and <rel>
[183] <rel_or_list>          ::= <rel> or <rel>
[184] <rel_or_list>          ::= <rel_or_list> or <rel>
[185] <rel_xor_list>         ::= <rel> xor <rel>
[186] <rel_xor_list>         ::= <rel_xor_list> xor <rel>
[187] <rel_and_then_list>    ::= <rel> and then <rel>
[188] <rel_and_then_list>    ::= <rel_and_then_list> and then <rel>
[189] <rel_or_else_list>     ::= <rel> or else <rel>
[190] <rel_or_else_list>     ::= <rel_or_else_list> or else <rel>
[191] <rel>                  ::= <simple_expr_list>
[192] <rel>                  ::= <simple_expr> in <subtype_ind>
[193] <rel>                  ::= <simple_expr> in <range>
[194] <rel>                  ::= <simple_expr> not in <subtype_ind>
[195] <rel>                  ::= <simple_expr> not in <range>
[196] <simple_expr_list>     ::= <simple_expr>
[197] <simple_expr_list>     ::= <simple_expr_list> <rel_op> <simple_expr>
[198] <simple_expr>          ::= <term_list>
[199] <simple_expr>          ::= <unary_op> <term_list>
[200] <term_list>            ::= <term>
[201] <term_list>            ::= <term_list> <add_op> <term>
[202] <term>                 ::= <factor_list>
[203] <factor_list>          ::= <factor>
[204] <factor_list>          ::= <factor_list> <mult_op> <factor>
[205] <factor>               ::= <primary> <primary_list>
[206] <primary_list>         ::= ** <primary>
[207] <primary_list>         ::=
[208] <primary>              ::= <literal>
[209] <primary>              ::= <aggregate>
[210] <primary>              ::= <name>
[211] <primary>              ::= <allocator>
[212] <primary>              ::= <qualified_expr>
[213] <rel_op>               ::= =
[214] <rel_op>               ::= /=
[215] <rel_op>               ::= <
[216] <rel_op>               ::= <=
[217] <rel_op>               ::= >
[218] <rel_op>               ::= >=
[219] <add_op>               ::= +
[220] <add_op>               ::= -
[221] <add_op>               ::= &
[222] <unary_op>             ::= +
[223] <unary_op>             ::= -
[224] <unary_op>             ::= not
[225] <mult_op>              ::= *
[226] <mult_op>              ::= /
[227] <mult_op>              ::= mod
[228] <mult_op>              ::= rem
```

```
[229] <name>               ::= <identifier>
[230] <name>               ::= <name> <arg_list>
[231] <name>               ::= <selected_comp>
[232] <name>               ::= <attr>
[233] <name>               ::= <op_symbol>
[234] <selected_comp>      ::= <name> . <identifier>
[235] <selected_comp>      ::= <name> . all
[236] <selected_comp>      ::= <name> . <op_symbol>
[237] <attr>               ::= <name> ' <identifier>
[238] <attr>               ::= <name> ' delta
[239] <attr>               ::= <name> ' digits
[240] <attr>               ::= <name> ' range
[241] <literal>            ::= <numeric_literal>
[242] <literal>            ::= <string>
[243] <literal>            ::= <character>
[244] <literal>            ::= null
[245] <aggregate>          ::= ( <comp_assoc_list> )
[246] <comp_assoc_list>    ::= <comp_assoc>
[247] <comp_assoc_list>    ::= <comp_assoc_list> , <comp_assoc>
[248] <comp_assoc>         ::= <expr>
[249] <comp_assoc>         ::= <choice_list> => <expr>
[250] <qualified_expr>     ::= <name> ' <aggregate>
[251] <allocator>          ::= new <name>
[252] <allocator>          ::= new <qualified_expr>
[253] <seq_of_stmts>       ::= <stmt>
[254] <seq_of_stmts>       ::= <seq_of_stmts> <stmt>
[255] <stmt>               ::= <simple_stmt>
[256] <stmt>               ::= <compound_stmt>
[257] <stmt>               ::= <pragma>
[258] <stmt>               ::= <label_list> <simple_stmt>
[259] <stmt>               ::= <label_list> <compound_stmt>
[260] <stmt>               ::= <classplace>
[261] <label_list>         ::= <label>
[262] <label_list>         ::= <label_list> <label>
[263] <simple_stmt>        ::= <null_stmt>
[264] <simple_stmt>        ::= <assign_stmt>
[265] <simple_stmt>        ::= <return_stmt>
[266] <simple_stmt>        ::= <proc_or_entry_call>
[267] <simple_stmt>        ::= <delay_stmt>
[268] <simple_stmt>        ::= <raise_stmt>
[269] <simple_stmt>        ::= <exit_stmt>
[270] <simple_stmt>        ::= <goto_stmt>
[271] <simple_stmt>        ::= <abort_stmt>
[272] <simple_stmt>        ::= <code_stmt>
[273] <compound_stmt>      ::= <if_stmt>
[274] <compound_stmt>      ::= <loop_stmt>
[275] <compound_stmt>      ::= <accept_stmt>
[276] <compound_stmt>      ::= <case_stmt>
[277] <compound_stmt>      ::= <block>
[278] <compound_stmt>      ::= <select_stmt>
```

```
[279] <label>              ::= << <identifier> >>
[280] <null_stmt>          ::= null ;
[281] <assign_stmt>        ::= <name> := <expr> ;
[282] <if_stmt>            ::= if <cond> then <seq_of_stmts> <elsif_list> end
                               if ;
[283] <if_stmt>            ::= if <cond> then <seq_of_stmts> end if ;
[284] <if_stmt>            ::= if <cond> then <seq_of_stmts> <elsif_list> else
                               <seq_of_stmts> end if ;
[285] <if_stmt>            ::= if <cond> then <seq_of_stmts> else
                               <seq_of_stmts> end if ;
[286] <elsif_list>         ::= elsif <cond> then <seq_of_stmts>
[287] <elsif_list>         ::= <elsif_list> elsif <cond> then <seq_of_stmts>
[288] <cond>               ::= <boolean_expr>
[289] <case_stmt>          ::= case <expr> is <when_list> end case ;
[290] <when_list>          ::=
[291] <when_list>          ::= <when_list> when <choice_list> =>
                               <seq_of_stmts>
[292] <loop_stmt>          ::= <basic_loop> ;
[293] <loop_stmt>          ::= <iteration_clause> <basic_loop> ;
[294] <loop_stmt>          ::= <identifier> : <basic_loop> <identifier> ;
[295] <loop_stmt>          ::= <identifier> : <iteration_clause> <basic_loop>
                               <identifier> ;
[296] <basic_loop>         ::= loop <seq_of_stmts> end loop
[297] <iteration_clause>   ::= for <loop_parm> in <discrete_range>
[298] <iteration_clause>   ::= for <loop_parm> in reverse <discrete_range>
[299] <iteration_clause>   ::= while <cond>
[300] <loop_parm>          ::= <identifier>
[301] <block>              ::= begin <seq_of_stmts> <excepts_opt> end ;
[302] <block>              ::= declare <decl_part> begin <seq_of_stmts>
                               <excepts_opt> end ;
[303] <block>              ::= <identifier> : begin <seq_of_stmts>
                               <excepts_opt> end <identifier> ;
[304] <block>              ::= <identifier> : declare <decl_part> begin
                               <seq_of_stmts> <excepts_opt> end <identifier> ;
[305] <exit_stmt>          ::= exit ;
[306] <exit_stmt>          ::= exit <loop_name> ;
[307] <exit_stmt>          ::= exit when <cond> ;
[308] <exit_stmt>          ::= exit <loop_name> when <cond> ;
[309] <return_stmt>        ::= return ;
[310] <return_stmt>        ::= return <expr> ;
[311] <goto_stmt>          ::= goto <label_name> ;
[312] <proc_or_entry_call> ::= <name> ;
[313] <entry_decl>         ::= entry <identifier> ;
[314] <entry_decl>         ::= entry <identifier> ( <discrete_range> ) ;
[315] <entry_decl>         ::= entry <identifier> <frml_part> ;
[316] <entry_decl>         ::= entry <identifier> ( <discrete_range> )
                               <frml_part> ;
[317] <accept_stmt>        ::= accept <entry_name> ;
[318] <accept_stmt>        ::= accept <entry_name> do <seq_of_stmts> end ;
[319] <accept_stmt>        ::= accept <entry_name> do <seq_of_stmts> end
```

```
                                  <identifier> ;
[320] <entry_name>            ::= <identifier> ( <entry_index> ) <frml_part>
[321] <entry_name>            ::= <identifier> <frml_part>
[322] <entry_name>            ::= <identifier> ( <entry_index> )
[323] <entry_name>            ::= <identifier>
[324] <entry_index>           ::= <expr>
[325] <delay_stmt>            ::= delay <simple_expr> ;
[326] <select_stmt>           ::= <selective_wait>
[327] <select_stmt>           ::= <cond_entry_call>
[328] <select_stmt>           ::= <timed_entry_call>
[329] <selective_wait>        ::= select <when_part_opt> <select_alt>
                                  <or_part_list> <else_part_opt> end select ;
[330] <when_part_opt>         ::=
[331] <when_part_opt>         ::= when <cond> =>
[332] <or_part_list>          ::=
[333] <or_part_list>          ::= <or_part_list> <or_part>
[334] <or_part>               ::= or <select_alt>
[335] <or_part>               ::= or when <cond> => <select_alt>
[336] <else_part_opt>         ::=
[337] <else_part_opt>         ::= else <seq_of_stmts>
[338] <select_alt>            ::= <accept_stmt> <seq_of_stmts_opt>
[339] <select_alt>            ::= <delay_stmt> <seq_of_stmts_opt>
[340] <select_alt>            ::= terminate ;
[341] <seq_of_stmts_opt>      ::=
[342] <seq_of_stmts_opt>      ::= <seq_of_stmts>
[343] <cond_entry_call>       ::= select <proc_or_entry_call> else <seq_of_stmts>
                                  end select ;
[344] <cond_entry_call>       ::= select <proc_or_entry_call> <seq_of_stmts> else
                                  <seq_of_stmts> end select ;
[345] <timed_entry_call>      ::= select <proc_or_entry_call> <seq_of_stmts_opt>
                                  or <delay_stmt> <seq_of_stmts_opt> end select ;
[346] <abort_stmt>            ::= abort <task_name_list> ;
[347] <task_name_list>        ::= <task_name>
[348] <task_name_list>        ::= <task_name_list> , <task_name>
[349] <raise_stmt>            ::= raise ;
[350] <raise_stmt>            ::= raise <except_name> ;
[351] <private_type_def>      ::= private
[352] <private_type_def>      ::= limited private
[353] <renaming_decl>         ::= <identifier_list> : <name> renames <name> ;
[354] <renaming_decl>         ::= <identifier_list> : exception renames <name> ;
[355] <renaming_decl>         ::= package <identifier> renames <name> ;
[356] <renaming_decl>         ::= task <identifier> renames <name> ;
[357] <renaming_decl>         ::= <subpgm_spec> renames <name> ;
[358] <except_decl>           ::= <identifier_list> : exception ;
[359] <except_handler>        ::= when <except_choice_list> => <seq_of_stmts>
[360] <except_choice_list>    ::= <except_choice>
[361] <except_choice_list>    ::= <except_choice_list> ! <except_choice>
[362] <except_choice>         ::= <except_name>
[363] <except_choice>         ::= others
[364] <gnrc_subpgm_decl>      ::= <gnrc_part> <subpgm_spec> ;
```

```
[365] <gnrc_pkg_decl>      ::= <gnrc_part> <pkg_spec> ;
[366] <gnrc_part>          ::= generic
[367] <gnrc_part>          ::= generic <gnrc_frml_parm_lst>
[368] <gnrc_frml_parm_lst> ::= <gnrc_frml_parm>
[369] <gnrc_frml_parm_lst> ::= <gnrc_frml_parm_lst> <gnrc_frml_parm>
[370] <gnrc_frml_parm>     ::= <parm_decl> ;
[371] <gnrc_frml_parm>     ::= type <identifier> is <gnrc_type_def> ;
[372] <gnrc_frml_parm>     ::= type <identifier> <discr_part> is
                               <gnrc_type_def> ;
[373] <gnrc_frml_parm>     ::= with <subpgm_spec> ;
[374] <gnrc_frml_parm>     ::= with <subpgm_spec> is <name> ;
[375] <gnrc_frml_parm>     ::= with <subpgm_spec> is <> ;
[376] <gnrc_type_def>      ::= ( <> )
[377] <gnrc_type_def>      ::= range <>
[378] <gnrc_type_def>      ::= delta <>
[379] <gnrc_type_def>      ::= digits <>
[380] <gnrc_type_def>      ::= <array_type_def>
[381] <gnrc_type_def>      ::= <access_type_def>
[382] <gnrc_type_def>      ::= <private_type_def>
[383] <gnrc_subpgm_inst>   ::= <subpgm_spec> is <gnrc_inst> ;
[384] <gnrc_subpgm_inst>   ::= function <designator> is <gnrc_inst> ;
[385] <gnrc_pkg_inst>      ::= package <identifier> is <gnrc_inst> ;
[386] <gnrc_inst>          ::= new <designator>
[387] <gnrc_inst>          ::= new <designator> <arg_list>
[388] <repr_spec>          ::= <len_or_enum_spec>
[389] <repr_spec>          ::= <record_type_repr>
[390] <repr_spec>          ::= <addr_spec>
[391] <len_or_enum_spec>   ::= for <name> use <expr> ;
[392] <record_type_repr>   ::= for <name> use record <align_clause_opt>
                               <loc_clause_list> end record ;
[393] <align_clause_opt>   ::=
[394] <align_clause_opt>   ::= <align_clause> ;
[395] <loc_clause_list>    ::=
[396] <loc_clause_list>    ::= <loc_clause_list> <comp_name> <loc> ;
[397] <loc>                ::= at <static_simple_expr> range <range>
[398] <align_clause>       ::= at mod <static_simple_expr>
[399] <addr_spec>          ::= for <name> use at <static_simple_expr> ;
[400] <code_stmt>          ::= <qualified_expr> ;
[401] <arg_list>           ::= ( <arg_part> )
[402] <arg_part>           ::= <arg_item>
[403] <arg_part>           ::= <arg_part> , <arg_item>
[404] <arg_item>           ::= <expr>
[405] <arg_item>           ::= <name> <range_constr>
[406] <arg_item>           ::= <range>
[407] <arg_item>           ::= <arg_stroke_list> => <expr>
[408] <arg_stroke_list>    ::= <name>
[409] <arg_stroke_list>    ::= <name> ! <arg_stroke_list>
[410] <pkg_name>           ::= <name>
[411] <unit_name>          ::= <name>
[412] <loop_name>          ::= <name>
```

```
[413] <label_name>         ::= <name>
[414] <task_name>          ::= <name>
[415] <except_name>        ::= <name>
[416] <comp_name>          ::= <name>
[417] <literal_expr>       ::= <expr>
[418] <boolean_expr>       ::= <expr>
[419] <static_simple_expr> ::= <simple_expr>
[420] <comp_subtype_ind>   ::= <subtype_ind>
[421] <numeric_literal>    ::= <real>
[422] <numeric_literal>    ::= <integer>
[423] <numeric_literal>    ::= <based_real>
[424] <numeric_literal>    ::= <based_int>
[425] <identifier>         ::= <classid>
[426] <character>          ::= <classchar>
[427] <string>             ::= <classstr>
[428] <op_symbol>          ::= <classop>
[429] <real>               ::= <classreal>
[430] <integer>            ::= <classint>
[431] <based_real>         ::= <classbreal>
[432] <based_int>          ::= <classbint>
```

List of tokens and their token numbers. Tokens option. Default: on

The reserved words and their token numbers are:

1	abort	2	accept	3	access
4	all	5	and	6	array
7	at	8	begin	9	body
10	case	11	constant	12	declare
13	delay	14	delta	15	digits
16	do	17	else	18	elsif
19	end	20	entry	21	exception
22	exit	23	for	24	function
25	generic	26	goto	27	if
28	in	29	is	30	limited
31	loop	32	mod	33	new
34	not	35	null	36	of
37	or	38	others	39	out
40	package	41	pragma	42	private
43	procedure	44	raise	45	range
46	record	47	rem	48	renames
49	return	50	reverse	51	select
52	separate	53	subtype	54	task
55	terminate	56	then	57	type
58	use	59	when	60	while
61	with	62	xor		

The angle-bracketed terminals and their token numbers are:

63	<classbint>	64	<classbreal>	65	<classchar>
66	<classid>	67	<classint>	68	<classop>
69	<classplace>	70	<classreal>	71	<classstr>
72	<eof>				

The special symbols and their token numbers are:

73	!	74	&	75	'
76	(	77	)	78	*
79	**	80	+	81	,
82	-	83	.	84	..
85	/	86	/=	87	:
88	:=	89	;	90	<
91	<<	92	<=	93	<>
94	=	95	=>	96	>
97	>=	98	>>		

```
 The non-terminals and their token numbers are:

   99 <abort_stmt>            100 <accept_stmt>           101 <access_type_def>
  102 <accuracy_constr>       103 <add_op>                104 <addr_spec>
  105 <aggregate>             106 <align_clause>          107 <align_clause_opt>
  108 <allocator>             109 <arg_item>              110 <arg_list>
  111 <arg_part>              112 <arg_stroke_list>       113 <array_type_def>
  114 <assign_stmt>           115 <attr>                  116 <based_int>
  117 <based_real>            118 <basic_loop>            119 <block>
  120 <body>                  121 <body_stub>             122 <boolean_expr>
  123 <case_stmt>             124 <character>             125 <choice>
  126 <choice_list>           127 <code_stmt>             128 <comp_assoc>
  129 <comp_assoc_list>       130 <comp_decl>             131 <comp_decl_list>
  132 <comp_list>             133 <comp_name>             134 <comp_subtype_ind>
  135 <compilation>           136 <compilation_eof>       137 <compilation_unit>
  138 <compound_stmt>         139 <cond>                  140 <cond_entry_call>
  141 <context_spec>          142 <decl>                  143 <decl_item>
  144 <decl_item_list>        145 <decl_part>             146 <delay_stmt>
  147 <derived_type_def>      148 <designator>            149 <designator_opt>
  150 <discr_decl>            151 <discr_decl_list>       152 <discr_part>
  153 <discr_part_opt>        154 <discrete_range>        155 <else_part_opt>
  156 <elsif_list>            157 <entry_decl>            158 <entry_decl_list>
  159 <entry_index>           160 <entry_name>            161 <enum_literal>
  162 <enum_literal_list>     163 <enum_type_def>         164 <except_choice>
  165 <except_choice_list>    166 <except_decl>           167 <except_hand_list>
  168 <except_handler>        169 <except_name>           170 <excepts_opt>
  171 <exit_stmt>             172 <expr>                  173 <factor>
  174 <factor_list>           175 <fixed_pt_constr>       176 <float_pt_constr>
  177 <frml_part>             178 <gnrc_frml_parm>        179 <gnrc_frml_parm_lst>
  180 <gnrc_inst>             181 <gnrc_part>             182 <gnrc_pkg_decl>
  183 <gnrc_pkg_inst>         184 <gnrc_subpgm_decl>      185 <gnrc_subpgm_inst>
  186 <gnrc_type_def>         187 <goto_stmt>             188 <identifier>
  189 <identifier_list>       190 <if_stmt>               191 <incompl_type_decl>
  192 <index>                 193 <index_list>            194 <init_opt>
  195 <integer>               196 <integer_type_def>      197 <iteration_clause>
  198 <label>                 199 <label_list>            200 <label_name>
  201 <len_or_enum_spec>      202 <literal>               203 <literal_expr>
  204 <loc>                   205 <loc_clause_list>       206 <loop_name>
  207 <loop_parm>             208 <loop_stmt>             209 <mode>
  210 <mult_op>               211 <name>                  212 <null_stmt>
  213 <number_decl>           214 <numeric_literal>       215 <object_decl>
  216 <op_symbol>             217 <or_part>               218 <or_part_list>
  219 <parm_decl>             220 <parm_decl_list>        221 <pgm_comp>
  222 <pkg_body>              223 <pkg_body_part_opt>     224 <pkg_decl>
  225 <pkg_name>              226 <pkg_name_list>         227 <pkg_spec>
  228 <pragma>                229 <primary>               230 <primary_list>
  231 <private_part_opt>      232 <private_type_def>      233 <proc_or_entry_call>
 234 <qualified_expr>      235 <raise_stmt>          236 <range>
 237 <range_constr>        238 <real>                239 <real_type_def>
 240 <record_type_def>     241 <record_type_repr>    242 <rel>
 243 <rel_and_list>        244 <rel_and_then_list>   245 <rel_op>
 246 <rel_or_else_list>    247 <rel_or_list>         248 <rel_xor_list>
 249 <renaming_decl>       250 <repr_spec>           251 <repr_spec_list>
 252 <return_stmt>         253 <select_alt>          254 <select_stmt>
 255 <selected_comp>       256 <selective_wait>      257 <seq_of_stmts>
 258 <seq_of_stmts_opt>    259 <simple_expr>         260 <simple_expr_list>
 261 <simple_stmt>         262 <static_simple_expr>  263 <stmt>
 264 <string>              265 <subpgm_body>         266 <subpgm_decl>
 267 <subpgm_spec>         268 <subtype_decl>        269 <subtype_ind>
 270 <subunit>             271 <system_goal_symbol>  272 <task_body>
 273 <task_decl>           274 <task_name>           275 <task_name_list>
 276 <task_spec>           277 <task_spec_part>      278 <term>
 279 <term_list>           280 <timed_entry_call>    281 <type_decl>
 282 <type_def>            283 <unary_op>            284 <unit_name>
 285 <unit_name_list>      286 <use_clause>          287 <variant_elt_list>
 288 <variant_part>        289 <when_list>           290 <when_part_opt>
 291 <with_clause>         292 <with_use_list>
```

```
Mystro Translator Writing System                    Version 7.0, June 1983
FP grammar                                                          Page 1

 Input grammar.    Grammar option.    Default: on

  The goal symbol <Goal> is found in rule 1.

  [  1] <Goal>          ::= <S> <eof>
  [  2] <S>             ::= <fpInputList>
  [  3] <S>             ::=
  [  4] <fpInputList>   ::= <fpInputList> <fpInput>
  [  5] <fpInputList>   ::= <fpInput>
  [  6] <fpInput>       ::= <fnDef>
  [  7] <fpInput>       ::= <application>
  [  8] <fpInput>       ::= <fpCmd>
  [  9] <fpInput>       ::= <classplace>
  [ 10] <fpInput>       ::= ^D
  [ 11] <fnDef>         ::= { <name> <funForm> }
  [ 12] <application>   ::= <funForm> : <object>
  [ 13] <name>          ::= <classident>
  [ 14] <nameList>      ::= <nameList> <name>
  [ 15] <nameList>      ::= <name>
  [ 16] <object>        ::= <atom>
  [ 17] <object>        ::= <fpSequence>
  [ 18] <object>        ::= ?
  [ 19] <fpSequence>    ::= < >
  [ 20] <fpSequence>    ::= < <objectList> >
  [ 21] <objectList>    ::= <objectList> , <object>
  [ 22] <objectList>    ::= <objectList> <object>
  [ 23] <objectList>    ::= <object>
  [ 24] <atom>          ::= T
  [ 25] <atom>          ::= F
  [ 26] <atom>          ::= <>
  [ 27] <atom>          ::= <classstrng>
  [ 28] <atom>          ::= <classident>
  [ 29] <atom>          ::= <classint>
  [ 30] <atom>          ::= <classreal>
  [ 31] <simpFn>        ::= <fpDefined>
  [ 32] <simpFn>        ::= <fpBuiltin>
  [ 33] <fpDefined>     ::= <name>
  [ 34] <fpBuiltin>     ::= <selectFn>
  [ 35] <fpBuiltin>     ::= tl
  [ 36] <fpBuiltin>     ::= id
  [ 37] <fpBuiltin>     ::= atom
  [ 38] <fpBuiltin>     ::= not
  [ 39] <fpBuiltin>     ::= eq
  [ 40] <fpBuiltin>     ::= <relFn>
  [ 41] <fpBuiltin>     ::= null
  [ 42] <fpBuiltin>     ::= reverse
  [ 43] <fpBuiltin>     ::= distl
  [ 44] <fpBuiltin>     ::= distr
  [ 45] <fpBuiltin>     ::= length
  [ 46] <fpBuiltin>     ::= <binaryFn>
[ 47] <fpBuiltin>    ::= trans
[ 48] <fpBuiltin>    ::= apndl
[ 49] <fpBuiltin>    ::= apndr
[ 50] <fpBuiltin>    ::= tlr
[ 51] <fpBuiltin>    ::= rotl
[ 52] <fpBuiltin>    ::= rotr
[ 53] <fpBuiltin>    ::= iota
[ 54] <fpBuiltin>    ::= pair
[ 55] <fpBuiltin>    ::= split
[ 56] <fpBuiltin>    ::= concat
[ 57] <fpBuiltin>    ::= last
[ 58] <fpBuiltin>    ::= <libFn>
[ 59] <selectFn>     ::= <classint>
[ 60] <relFn>        ::= <=
[ 61] <relFn>        ::= <
[ 62] <relFn>        ::= =
[ 63] <relFn>        ::= ~=
[ 64] <relFn>        ::= >
[ 65] <relFn>        ::= >=
[ 66] <binaryFn>     ::= +
[ 67] <binaryFn>     ::= -
[ 68] <binaryFn>     ::= *
[ 69] <binaryFn>     ::= /
[ 70] <binaryFn>     ::= or
[ 71] <binaryFn>     ::= and
[ 72] <binaryFn>     ::= xor
[ 73] <libFn>        ::= sin
[ 74] <libFn>        ::= cos
[ 75] <libFn>        ::= asin
[ 76] <libFn>        ::= acos
[ 77] <libFn>        ::= log
[ 78] <libFn>        ::= exp
[ 79] <libFn>        ::= mod
[ 80] <funForm>      ::= <funForm> @ <otherFun>
[ 81] <funForm>      ::= <otherFun>
[ 82] <otherFun>     ::= <classplace>
[ 83] <otherFun>     ::= <simpFn>
[ 84] <otherFun>     ::= <construction>
[ 85] <otherFun>     ::= <conditional>
[ 86] <otherFun>     ::= <while>
[ 87] <otherFun>     ::= <constantFn>
[ 88] <otherFun>     ::= <insertion>
[ 89] <otherFun>     ::= <alpha>
[ 90] <otherFun>     ::= ( <funForm> )
[ 91] <while>        ::= ( while <funForm> <funForm> )
[ 92] <conditional>  ::= ( <funForm> -> <funForm> ; <funForm> )
[ 93] <construction> ::= [ <formList> ]
[ 94] <construction> ::= [ ]
[ 95] <formList>     ::= <formList> , <funForm>
[ 96] <formList>     ::= <funForm>
[ 97] <constantFn>    ::= % <object>
[ 98] <insertion>     ::= ! <otherFun>
[ 99] <insertion>     ::= | <otherFun>
[100] <alpha>         ::= & <otherFun>
```

```
 List of tokens and their token numbers.   Tokens option.   Default: on


 The reserved words and their token numbers are:

   1 F                 2 T                 3 acos              4 and
   5 apndl             6 apndr             7 asin              8 atom
   9 concat           10 cos              11 distl            12 distr
  13 eq               14 exp              15 id               16 iota
  17 last             18 length           19 log              20 mod
  21 not              22 null             23 or               24 pair
  25 reverse          26 rotl             27 rotr             28 sin
  29 split            30 tl               31 tlr              32 trans
  33 while            34 xor


 The angle-bracketed terminals and their token numbers are:

  35 <classident>     36 <classint>       37 <classplace>     38 <classreal>
  39 <classstrng>     40 <eof>            41 <fpCmd>


 The special symbols and their token numbers are:

  42 !                43 %                44 &                45 (
  46 )                47 *                48 +                49 ,
  50 -                51 ->               52 /                53 :
  54 ;                55 <                56 <=               57 <>
  58 =                59 >                60 >=               61 ?
  62 @                63 [                64 ]                65 ^D
  66 {                67 |                68 }                69 ~=


 The non-terminals and their token numbers are:

  70 <Goal>           71 <S>              72 <alpha>          73 <application>
  74 <atom>           75 <binaryFn>       76 <conditional>    77 <constantFn>
  78 <construction>   79 <fnDef>          80 <formList>       81 <fpBuiltin>
  82 <fpDefined>      83 <fpInput>        84 <fpInputList>    85 <fpSequence>
  86 <funForm>        87 <insertion>      88 <libFn>          89 <name>
  90 <nameList>       91 <object>         92 <objectList>     93 <otherFun>
  94 <relFn>          95 <selectFn>       96 <simpFn>         97 <while>
```

```
Mystro Translator Writing System                    Version 7.0, June 1983
Pascal grammar                                                      Page 1

 Input grammar.   Grammar option.   Default: on

  The goal symbol <full_program> is found in rule 1.

  [  1] <full_program>  ::= <program> <eof>
  [  2] <program>       ::= <program_head> <block> .
  [  3] <program>       ::= <declarations>
  [  4] <program>       ::=
  [  5] <program_head>  ::= program <classident> ;
  [  6] <program_head>  ::= program <classident> ( <ext_file_part> ) ;
  [  7] <ext_file_part> ::= <external_file>
  [  8] <ext_file_part> ::= <ext_file_part> , <external_file>
  [  9] <external_file> ::= <classident>
  [ 10] <declarations>  ::= <decl_element>
  [ 11] <declarations>  ::= <declarations> <decl_element>
  [ 12] <decl_element>  ::= <include_part>
  [ 13] <decl_element>  ::= <label_decl>
  [ 14] <decl_element>  ::= <cnst_def_part>
  [ 15] <decl_element>  ::= <type_def_part>
  [ 16] <decl_element>  ::= <var_decl_part>
  [ 17] <decl_element>  ::= <proc_decl>
  [ 18] <decl_element>  ::= <fcn_decl>
  [ 19] <include_part>  ::= # include <classstrng>
  [ 20] <include_part>  ::= # include <classdqstr>
  [ 21] <label_decl>    ::= <label_symbol> <label_part> ;
  [ 22] <label_symbol>  ::= label
  [ 23] <label_part>    ::= <label>
  [ 24] <label_part>    ::= <label_part> , <label>
  [ 25] <label>         ::= <classint>
  [ 26] <cnst_def_part> ::= <const_symbol> <const_list>
  [ 27] <const_symbol>  ::= const
  [ 28] <const_list>    ::= <const_list> <const_def>
  [ 29] <const_list>    ::= <const_def>
  [ 30] <const_def>     ::= <classident> = <constant> ;
  [ 31] <const_def>     ::= <classplace> ;
  [ 32] <constant>      ::= <unsigned_num>
  [ 33] <constant>      ::= <classstrng>
  [ 34] <constant>      ::= <classident>
  [ 35] <constant>      ::= <sign> <unsigned_num>
  [ 36] <sign>          ::= +
  [ 37] <sign>          ::= -
  [ 38] <unsigned_num>  ::= <classint>
  [ 39] <unsigned_num>  ::= <classreal>
  [ 40] <type_def_part> ::= <type_symbol> <type_list>
  [ 41] <type_symbol>   ::= type
  [ 42] <type_list>     ::= <type_list> <type_def>
  [ 43] <type_list>     ::= <type_def>
  [ 44] <type_def>      ::= <classplace> ;
  [ 45] <type_def>      ::= <classident> = <type> ;
  [ 46] <type>          ::= <simple_type>
[ 47] <type>          ::= <struct_type>
[ 48] <type>          ::= packed <struct_type>
[ 49] <type>          ::= <point_type>
[ 50] <simple_type>   ::= <classident>
[ 51] <simple_type>   ::= ( <scalar_type> )
[ 52] <simple_type>   ::= <constant> .. <constant>
[ 53] <scalar_type>   ::= <classident>
[ 54] <scalar_type>   ::= <scalar_type> , <classident>
[ 55] <struct_type>   ::= <array_type>
[ 56] <struct_type>   ::= <record_type>
[ 57] <struct_type>   ::= <set_type>
[ 58] <struct_type>   ::= <file_type>
[ 59] <array_type>    ::= array [ <index_list> ] of <type>
[ 60] <index_list>    ::= <index_elt>
[ 61] <index_list>    ::= <index_list> , <index_elt>
[ 62] <index_elt>     ::= <simple_type>
[ 63] <record_type>   ::= record <field_list> <record_end>
[ 64] <record_end>    ::= end
[ 65] <field_list>    ::= <fixed_part>
[ 66] <field_list>    ::= <fixed_part> ; <variant_part>
[ 67] <field_list>    ::= <variant_part>
[ 68] <fixed_part>    ::= <record_sect>
[ 69] <fixed_part>    ::= <fixed_part> ; <record_sect>
[ 70] <record_sect>   ::= <variable_list> : <type>
[ 71] <record_sect>   ::=
[ 72] <variable_list> ::= <classident>
[ 73] <variable_list> ::= <variable_list> , <classident>
[ 74] <variant_part>  ::= case <tag> of <variant_list>
[ 75] <tag>           ::= <classident> : <classident>
[ 76] <tag>           ::= <classident>
[ 77] <variant_list>  ::= <variant>
[ 78] <variant_list>  ::= <variant_list> ; <variant>
[ 79] <variant>       ::= <case_lbl_list> : <fld_lst_part>
[ 80] <variant>       ::=
[ 81] <fld_lst_part>  ::= ( <field_list> )
[ 82] <case_lbl_list> ::= <case_label>
[ 83] <case_lbl_list> ::= <case_lbl_list> , <case_label>
[ 84] <case_label>    ::= <constant>
[ 85] <set_type>      ::= set of <simple_type>
[ 86] <file_type>     ::= file of <type>
[ 87] <point_type>    ::= ^ <classident>
[ 88] <var_decl_part> ::= <var_symbol> <var_decl_list>
[ 89] <var_symbol>    ::= var
[ 90] <var_decl_list> ::= <variable_list> : <type> ;
[ 91] <var_decl_list> ::= <var_decl_list> <variable_list> : <type> ;
[ 92] <var_decl_list> ::= <classplace> ;
[ 93] <var_decl_list> ::= <var_decl_list> <classplace> ;
[ 94] <proc_decl>     ::= <proc_heading> ; <proc_fcn_foll> ;
[ 95] <fcn_decl>      ::= <fcn_heading> ; <proc_fcn_foll> ;
[ 96] <proc_fcn_foll> ::= <block>
```

```
Mystro Translator Writing System                    Version 7.0, June 1983
Pascal grammar                                                      Page 3

 [ 97] <proc_fcn_foll> ::= forward
 [ 98] <proc_fcn_foll> ::= external
 [ 99] <proc_fcn_foll> ::= fortran
 [100] <proc_heading>  ::= procedure <proc_name> <parm_list>
 [101] <proc_name>     ::= <classident>
 [102] <fcn_heading>   ::= function <fcn_name> <parm_list> : <classident>
 [103] <fcn_heading>   ::= function <fcn_name>
 [104] <fcn_name>      ::= <classident>
 [105] <parm_list>     ::= ( <frml_parm_lst> )
 [106] <parm_list>     ::=
 [107] <frml_parm_lst> ::= <frml_parm_sct>
 [108] <frml_parm_lst> ::= <frml_parm_lst> ; <frml_parm_sct>
 [109] <frml_parm_sct> ::= var <variable_list> : <classident>
 [110] <frml_parm_sct> ::= <variable_list> : <classident>
 [111] <frml_parm_sct> ::= <proc_heading>
 [112] <frml_parm_sct> ::= <fcn_heading>
 [113] <block>         ::= <declarations> begin <stmt_list> end
 [114] <block>         ::= begin <stmt_list> end
 [115] <stmt_list>     ::= <statement>
 [116] <stmt_list>     ::= <stmt_list> ; <statement>
 [117] <statement>     ::= <S1>
 [118] <S1>            ::= if <expression> then <S1>
 [119] <S1>            ::= <label> : if <expression> then <S1>
 [120] <S1>            ::= if <expression> then <nested_ifstmt> else <S1>
 [121] <S1>            ::= <label> : if <expression> then <nested_ifstmt> else
                           <S1>
 [122] <S1>            ::= <non_ifstmt1>
 [123] <S1>            ::= <label> : <non_ifstmt1>
 [124] <nested_ifstmt> ::= if <expression> then <nested_ifstmt> else
                           <nested_ifstmt>
 [125] <nested_ifstmt> ::= <label> : if <expression> then <nested_ifstmt> else
                           <nested_ifstmt>
 [126] <nested_ifstmt> ::= <non_ifstmt2>
 [127] <nested_ifstmt> ::= <label> : <non_ifstmt2>
 [128] <non_ifstmt1>   ::= <for_stmt1>
 [129] <non_ifstmt1>   ::= <while_stmt1>
 [130] <non_ifstmt1>   ::= <with_stmt1>
 [131] <non_ifstmt1>   ::= <non_ifstmt>
 [132] <non_ifstmt2>   ::= <for_stmt2>
 [133] <non_ifstmt2>   ::= <while_stmt2>
 [134] <non_ifstmt2>   ::= <with_stmt2>
 [135] <non_ifstmt2>   ::= <non_ifstmt>
 [136] <non_ifstmt>    ::= <assign_stmt>
 [137] <non_ifstmt>    ::= <case_stmt>
 [138] <non_ifstmt>    ::= <classplace>
 [139] <non_ifstmt>    ::= <empty_stmt>
 [140] <non_ifstmt>    ::= goto <label>
 [141] <non_ifstmt>    ::= <proc_stmt>
 [142] <non_ifstmt>    ::= <repeat_stmt>
 [143] <non_ifstmt>    ::= begin <stmt_list> end
```

```
[144] <assign_stmt>    ::= <variable> := <expression>
[145] <variable>       ::= <classident>
[146] <variable>       ::= <variable> [ <express_list> ]
[147] <variable>       ::= <variable> . <classident>
[148] <variable>       ::= <variable> ^
[149] <case_stmt>      ::= case <expression> of <case_list> end
[150] <case_list>      ::= <case_element>
[151] <case_list>      ::= <case_list> ; <case_element>
[152] <case_element>   ::= <case_lbl_list> : <statement>
[153] <case_element>   ::=
[154] <empty_stmt>     ::=
[155] <for_stmt1>      ::= for <for_list> do <statement>
[156] <for_stmt2>      ::= for <for_list> do <nested_ifstmt>
[157] <for_list>       ::= <classident> := <expression> to <expression>
[158] <for_list>       ::= <classident> := <expression> downto <expression>
[159] <for_list>       ::= <classplace>
[160] <repeat_stmt>    ::= repeat <stmt_list> until <expression>
[161] <while_stmt1>    ::= while <expression> do <statement>
[162] <while_stmt2>    ::= while <expression> do <nested_ifstmt>
[163] <proc_stmt>      ::= <classident> ( <act_parm_list> )
[164] <proc_stmt>      ::= <classident>
[165] <proc_stmt>      ::= write ( <special_parms> )
[166] <proc_stmt>      ::= writeln ( <special_parms> )
[167] <proc_stmt>      ::= writeln
[168] <special_parms>  ::= <simple_expres> <field_width>
[169] <special_parms>  ::= <special_parms> , <simple_expres> <field_width>
[170] <field_width>    ::= : <simple_expres> : <simple_expres>
[171] <field_width>    ::= : <simple_expres>
[172] <field_width>    ::=
[173] <with_stmt1>     ::= with <rcd_var_list> do <statement>
[174] <with_stmt2>     ::= with <rcd_var_list> do <nested_ifstmt>
[175] <rcd_var_list>   ::= <variable>
[176] <rcd_var_list>   ::= <rcd_var_list> , <variable>
[177] <rcd_var_list>   ::= <classplace>
[178] <express_list>   ::= <expression>
[179] <express_list>   ::= <express_list> , <expression>
[180] <expression>     ::= <simple_expres>
[181] <expression>     ::= <simple_expres> <rel_op> <simple_expres>
[182] <expression>     ::= <classplace>
[183] <rel_op>         ::= =
[184] <rel_op>         ::= <>
[185] <rel_op>         ::= <
[186] <rel_op>         ::= <=
[187] <rel_op>         ::= >=
[188] <rel_op>         ::= >
[189] <rel_op>         ::= in
[190] <simple_expres>  ::= <classstrng>
[191] <simple_expres>  ::= <term>
[192] <simple_expres>  ::= <simple_expres> <add_op> <term>
[193] <add_op>         ::= +
[194] <add_op>         ::= -
[195] <add_op>         ::= or
[196] <term>           ::= <factor>
[197] <term>           ::= <term> <mult_op> <factor>
[198] <mult_op>        ::= *
[199] <mult_op>        ::= /
[200] <mult_op>        ::= div
[201] <mult_op>        ::= mod
[202] <mult_op>        ::= and
[203] <factor>         ::= <sign> <factor>
[204] <factor>         ::= <variable>
[205] <factor>         ::= <unsigned_num>
[206] <factor>         ::= nil
[207] <factor>         ::= ( <expression> )
[208] <factor>         ::= [ <element_list> ]
[209] <factor>         ::= [ ]
[210] <factor>         ::= <classident> ( <act_parm_list> )
[211] <factor>         ::= not <factor>
[212] <act_parm_list> ::= <expression>
[213] <act_parm_list> ::= <act_parm_list> , <expression>
[214] <element_list>  ::= <element>
[215] <element_list>  ::= <element_list> , <element>
[216] <element>       ::= <expression>
[217] <element>       ::= <expression> .. <expression>
```

```
 List of tokens and their token numbers.    Tokens option.    Default: on


 The reserved words and their token numbers are:

    1 and               2 array             3 begin
    4 case              5 const             6 div
    7 do                8 downto            9 else
   10 end              11 external         12 file
   13 for              14 fortran          15 forward
   16 function         17 goto             18 if
   19 in               20 include          21 label
   22 mod              23 nil              24 not
   25 of               26 or               27 packed
   28 procedure        29 program          30 record
   31 repeat           32 set              33 then
   34 to               35 type             36 until
   37 var              38 while            39 with
   40 write            41 writeln


 The angle-bracketed terminals and their token numbers are:

   42 <classdqstr>     43 <classident>     44 <classint>
   45 <classplace>     46 <classreal>      47 <classstrng>
   48 <eof>


 The special symbols and their token numbers are:

   49 #                50 (                51 )
   52 *                53 +                54 ,
   55 -                56 .                57 ..
   58 /                59 :                60 :=
   61 ;                62 <                63 <=
   64 <>               65 =                66 >
   67 >=               68 [                69 ]
   70 ^


 The non-terminals and their token numbers are:

   71 <S1>             72 <act_parm_list>  73 <add_op>
   74 <array_type>     75 <assign_stmt>    76 <block>
   77 <case_element>   78 <case_label>     79 <case_lbl_list>
   80 <case_list>      81 <case_stmt>      82 <cnst_def_part>
```

```
Mystro Translator Writing System                      Version 7.0, June 1983
Pascal grammar                                                        Page 7

  83 <const_def>          84 <const_list>         85 <const_symbol>
  86 <constant>           87 <decl_element>       88 <declarations>
  89 <element>            90 <element_list>       91 <empty_stmt>
  92 <express_list>       93 <expression>         94 <ext_file_part>
  95 <external_file>      96 <factor>             97 <fcn_decl>
  98 <fcn_heading>        99 <fcn_name>          100 <field_list>
 101 <field_width>       102 <file_type>         103 <fixed_part>
 104 <fld_lst_part>      105 <for_list>          106 <for_stmt1>
 107 <for_stmt2>         108 <frml_parm_lst>     109 <frml_parm_sct>
 110 <full_program>      111 <include_part>      112 <index_elt>
 113 <index_list>        114 <label>             115 <label_decl>
 116 <label_part>        117 <label_symbol>      118 <mult_op>
 119 <nested_ifstmt>     120 <non_ifstmt1>       121 <non_ifstmt2>
 122 <non_ifstmt>        123 <parm_list>         124 <point_type>
 125 <proc_decl>         126 <proc_fcn_foll>     127 <proc_heading>
 128 <proc_name>         129 <proc_stmt>         130 <program>
 131 <program_head>      132 <rcd_var_list>      133 <record_end>
 134 <record_sect>       135 <record_type>       136 <rel_op>
 137 <repeat_stmt>       138 <scalar_type>       139 <set_type>
 140 <sign>              141 <simple_expres>     142 <simple_type>
 143 <special_parms>     144 <statement>         145 <stmt_list>
 146 <struct_type>       147 <tag>               148 <term>
 149 <type>              150 <type_def>          151 <type_def_part>
 152 <type_list>         153 <type_symbol>       154 <unsigned_num>
 155 <var_decl_list>     156 <var_decl_part>     157 <var_symbol>
 158 <variable>          159 <variable_list>     160 <variant>
 161 <variant_list>      162 <variant_part>      163 <while_stmt1>
 164 <while_stmt2>       165 <with_stmt1>        166 <with_stmt2>
```

VITA

Peter A. Kirslis was born in [redacted], [redacted] in [redacted], [redacted]. He received his A.B. degree cum laude in Applied Mathematics from Harvard College, Cambridge, Massachusetts, in 1975, and his M.S. in Computer Science from the University of Illinois, Urbana, Illinois, in 1977. He has been employed as a Member of the Technical Staff of Bell Laboratories, Murray Hill, New Jersey, in the area of distributed operating systems. He is a member of the Association for Computing Machinery.

BIBLIOGRAPHIC DATA SHEET

1. Report No. UIUCDCS-R-85-1236

2.

3. Recipient's Accession No.

4. Title and Subtitle THE SAGA EDITOR:
A LANGUAGE-ORIENTED EDITOR BASED ON AN INCREMENTAL
LR(1) PARSER

5. Report Date December 1985

6.

7. Author(s) Peter Andre Christopher Kirslis

8. Performing Organization Rept. No. R-85-1236

9. Performing Organization Name and Address

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No. NASA NAG 1-138

12. Sponsoring Organization Name and Address

NASA Langley Research Center
Hampton, VA 23665

13. Type of Report & Period Covered Ph.D. Thesis

14.

15. Supplementary Notes

16. Abstracts The research described in this dissertation supports the thesis that a language oriented editor for full programming languages, and other languages specifiable with context-free LR(1) grammars, can be based upon an incremental LR(1) parser employing incremental analysis techniques. The resulting editor is flexible, supporting a higher-level command interface which includes structure-oriented commands involving tokens and sub-trees, while retaining common text editing commands which operate on arbitrary groups of characters and lines. The editor can be used to develop practical programs which incorporate software engineering principles concerning the design and construction of software systems. In this dissertation, an incremental parsing algorithm suitable for use with an interactive editor is developed. A new solution to the handling of comments in syntax trees is proposed, and an error-recovery algorithm which permits editing of the parse tree in the midst of syntax errors is presented. The resulting editor, its commands, and environment are described. The editor can be retargeted to other languages, and can use any parser-generating system which can meet its interface. A prototype editor which employs these algorithms has been implemented as a part of the SAGA project as a demonstration of the practicality and flexibility of this approach; this editor has been in experimental use during the past couple of years at the University of Illinois at Urbana-Champaign.

17. Key Words and Document Analysis. 17a. Descriptors

language-oriented editing
software engineering
parsing
incremental parsing
syntax analysis

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement unlimited

19. Security Class (This Report) UNCLASSIFIED

20. Security Class (This Page UNCLASSIFIED

21. No. of Pages 163

22. Price

An Example of a Stepwise Development Methodology

Lee A. Benzinger

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, Illinois

Submitted to 9th Annual International Conference on Software Engineering,
March 30-April 2, 1987, Monterey, CA.

An Example of a Stepwise Development Methodology

LEE A. BENZINGER*

Abstract– We give an example of a stepwise development methodology for the development of software which uses the Hoare calculus and a notion of partial correctness of programs with respect to specifications. We prove that this example falls within the framework provided by an abstract mathematical model for software development. Since the model possesses some of the basic properties that we would expect of an idealised development, it follows that the example also possesses these properties. This paper uses the technique of comparing an example of a software development methodology with a abstract model for software development in order to gain insight into the methodology.

Index Terms– Hoare logic, partial correctness, stepwise development.

1. Introduction

The task of developing software which meets a given specification is very difficult. Various approaches have been suggested to make the task more tractable. In [10] the problem of designing an algorithm which meets a specification is considered. In [13] an axiomatic approach to the problem of program of correctness proofs for programs is given, while in [22] and [11] stepwise approaches to program development are considered. The Vienna Development Method (VDM) [14] is a software development method which combines the notions of stepwise refinement with proofs of correctness at each step. In [16] a stepwise approach to software design is discussed which includes the notion of correctness of a software component with respect to a specification at each step. The purpose of this paper is to construct a mathematically rigorous foundation for the stepwise approach to the development of software. This work of part of the SAGA (Software Automation, Generation and Administration) project, which is concerned with providing an environment to support the theory and practice of software development [3-7,12,15,20,21].

In order to compare different stepwise design methodologies, to study the properties of a particular design methodology, or to develop new design methodologies, it is valuable to have an abstract mathematical model of the stepwise development process. A model serves as a standard for comparison of design methodologies. If we can prove that design methodology A has the properties of an abstract model and design methodology B either does not have these properties or it is not known whether B possesses these properties, then we have a basis for choosing A over B. In developing a new design methodology, a proof that it satisfies the properties of an abstract model is a guarantee that the software component obtained as a result of using the methodology will at least possess the properties of the abstract model. This is a significant improvement over the situation in which we implicitly assume that a design methodology has desirable properties because it seems intuitively reasonable.

In [2] an abstract mathematical model for the stepwise development of software is presented. The model is quite simple in that it describes an idealized development. Issues such as backtracking or the effect on a development of changing the original specification are not considered. The model is fairly general since it is independent of the notions of specification, correctness, and implementation. These notions are dependent upon a particular design methodology, not the abstract model.

In this paper we present an abstract model, an example of a stepwise development methodology that has the properties of the abstract model, and sketch the proof that the properties are satisfied by the example. Section 2 contains an overview of the abstract model. In section 3 we give an outline of the construction of the example and the proof that it satisfies the requirements of the model. Section 4 contains definitions which are

*This research is supported in part by NASA grant NAG 1-138, Roy H. Campbell principal investigator.

used in the construction of the example. Sections 5 and 6 contain proofs which indicate in more detail the methods used for showing that the example does have the properties of the abstract model. Section 7 contains the conclusion.

2. The Abstract Model

In this section we present an informal discussion of the abstract model. See [2] for further details and proofs. We define an *abstract program* $\mathcal{A}$ as an ordered pair, (S, C), where S is a specification and C is the set of all implementations which are correct with respect to S. The set C may be empty. This can occur, for example, when S is inconsistent and there exists no implementation which is correct with respect to S. As already noted, the notions of specification, implementation, and correctness are left undefined in the discussion of the abstract model. We are primarily interested in a model for stepwise design methodologies which allows us to study those properties which are intrinsic to an idealized stepwise development process, independent of the notions of specification, implementation, and correctness.

A *development* D with respect to a specification S_0 is an $(n + 1)$-tuple of abstract programs, $(\mathcal{A}_0, \mathcal{A}_1, ..., \mathcal{A}_n)$, for some nonnegative integer n such that for each i, $0 \leq i \leq n$, $\mathcal{A}_i = (S_i, C_i)$. Let C be a set. By $|C|$ we mean the cardinality of C. D is *correct* if $C_{i+1} \subseteq C_i$, $0 \leq i < n$. D is *complete* if $|C_n| = 1$. D is *incomplete* if $|C_n| > 1$. Correct and complete developments are those which start out with an abstract program $\mathcal{A}_0 = (S_0, C_0)$, as the first member of the ordered $(n + 1)$-tuple which is the development. S_0 is the original specification. The last member of the development is (S_n, C_n). C_n is a set which contains a single implementation and S_n is the last specification in the development. The sets of implementations form a nested family; that is, for each integer i, $0 \leq i \leq n$, $C_{i+1} \subseteq C_i$. Because the sets of implementations have this property, it follows that any implementation which is correct with respect to a given specification in a development is also correct with respect to all preceding specifications in the development. This property ensures that the implementation obtained from a development is correct with respect to the original specification. Correct and incomplete developments are developments that are, intuitively, correct so far, but are not finished. The last abstract program in a correct and incomplete development is an ordered pair, (S_n, C_n). C_n is a set with more than a single implementation which is correct with respect to the specification S_n. S_n specifies a family of implementations rather than a single implementation.

In a stepwise development, developments are formed from steps. A development step is the result of a process of going from one abstract program to another. A *development step* with respect to a specification S_i is an ordered pair of abstract programs, $(\mathcal{A}_i, \mathcal{A}_{i+1})$, such that $\mathcal{A}_i = (S_i, C_i)$ and $\mathcal{A}_{i+1} = (S_{i+1}, C_{i+1})$. Let $D = ((S_0, C_0), (S_1, C_1), ..., (S_n, C_n))$ be a development with respect to a specification S_0. Let $((S_j, C_j), (S_{j+1}, C_{j+1}))$ be a development step with respect to the specification S_j. The development D *contains the development step* if $j = i$ for some integer i, $0 \leq i \leq n - 1$; that is, the development step is $((S_i, C_i), (S_{i+1}, C_{i+1}))$, where (S_i, C_i) and (S_{i+1}, C_{i+1}) are successive members of the $(n + 1)$-tuple which is the development with respect to the specification S_0. A development step with respect to a specification S_i for some nonnegative integer i, $((S_i, C_i), (S_{i+1}, C_{i+1}))$, is *correct* if the following hold:

(1) $C_i, C_{i+1} \neq \emptyset$

(2) $C_{i+1} \subseteq C_i$.

A development step with respect to a specification S_i, $((S_i, C_i),(S_{i+1}, C_{i+1}))$, is *complete* if $|C_{i+1}| = 1$. A development step with respect to a specification S_i, $((S_i, C_i),(S_{i+1}, C_{i+1}))$, is *incomplete* if $|C_{i+1}| > 1$.

Developments can be extended by development steps to form new developments. We state a result about extensions of developments by development steps.

Theorem: Let D be a correct and incomplete development, $((S_0, C_0), (S_1, C_1), ..., (S_n, C_n))$, with respect to the specification S_0. Suppose that $((S_n, C_n), (S_{n+1}, C_{n+1}))$ is a complete and correct development step with respect to S_n. Let D_1 be $((S_0, C_0), (S_1, C_1), ..., (S_{n+1}, C_{n+1}))$. D_1 is a correct and complete development with respect to the specification S_0, which contains the given development step.

Developments can be constructed from development steps. The properties of the resulting developments depend upon the properties of the development steps used in the construction of the developments. The following result shows that development steps can be viewed as "building blocks" for the construction of developments.

Theorem: Let $((S_0, C_0), (S_1, C_1))$, $((S_1, C_1), (S_2, C_2))$, ..., $((S_{n-1}, C_{n-1}), (S_n, C_n))$ be a collection of n correct development steps with respect to the specifications $\mathcal{S}_0$, $\mathcal{S}_1$, ..., $\mathcal{S}_n$ respectively, for some positive integer n. Furthermore, suppose that $((\mathcal{S}_{n-1}, C_{n-1}), (\mathcal{S}_n, C_n))$ is a complete development step. Let $D = ((\mathcal{S}_0, C_0), (\mathcal{S}_1, C_1), ..., (\mathcal{S}_n, C_n))$. Then D is a correct and complete development with respect to the specification $\mathcal{S}_0$.

3. A Stepwise Design Methodology

In this section we outline the approach we use to construct an example of a stepwise design methodology and to prove that the methodology actually does satisfy the constraints of the abstract model. Initially, we need to define the concepts of an implementation, a specification, and correctness with respect to a specification. In the example, an implementation is a while-program, a program in a programming language which allows assignment statements, composed statements, conditional statements, and while statements. A specification is in terms of pre- and post-conditions and the constructs of the while-programming language. By correctness with respect to a specification we mean an extension of the notion of partial correctness with respect to formulas from first order logic. Most of the notation and terminology which we use is in [18]. We try to be consistent with [18] when we introduce new concepts and notation.

3.1. A Correct Development Step

In a stepwise design methodology, we must be able to construct correct development steps. In addition to a notion of correctness, it is necessary to have a deductive system within which we can prove that implementations are correct with respect to specifications. For the example, we use the axiomatic method of Hoare. The Hoare method is used in program verification to prove while-programs partially correct, but in the example the method is extended so that it is used to prove implementations partially correct with respect to specifications at each step in a development. Generally, at each step in a development except for the last, a specification will specify a family of implementations rather than a single while-program. In the terminology of the abstract model, the problem of program verification is the following: Given an abstract program, $\mathcal{A} = (\mathcal{S}, C)$, and a program W, prove that $W \in C$; that is, given a program W, show that it is correct with respect to the specification $\mathcal{S}$. In a stepwise methodology, given an abstract program, $\mathcal{A}_i = (\mathcal{S}_i, C_i)$, we must be able to construct a new abstract program, $\mathcal{A}_{i+1} = (\mathcal{S}_{i+1}, C_{i+1})$, so that $(\mathcal{A}_i, \mathcal{A}_{i+1})$ is a correct development step. In the example, given an abstract program, $\mathcal{A}_i = (\mathcal{S}_i, C_i)$, we construct a new abstract program, $\mathcal{A}_{i+1} = (\mathcal{S}_{i+1}, C_{i+1})$. We also prove that $C_{i+1} \subseteq C_i$ or $W \in C_{i+1}$ implies that $W \in C_i$. The construction of the new abstract program $\mathcal{A}_{i+1}$ and the proof that the pair of abstract programs $(\mathcal{A}_i, \mathcal{A}_{i+1})$ is a correct development step depends upon specification transformations,

$$T: \mathcal{S}_i \rightarrow \mathcal{S}_{i+1}.$$

These transformations are defined explicitly.

3.2. A Development

From the abstract model we know that to construct a correct and complete development it is sufficient to construct a series of correct development steps followed by a single correct and complete development step. A correct and complete development step, is a correct development step, $(\mathcal{A}_i, \mathcal{A}_{i+1}) = ((\mathcal{S}_i, C_i), (\mathcal{S}_{i+1}, C_{i+1}))$, with the additional property that $|C_{i+1}| = 1$. The specification $\mathcal{S}_{i+1}$ must be detailed enough so that it specifies exactly one while-program. We use the concept of an *annotated program* to describe such a specification. In [2] we prove that for an abstract program $\mathcal{A} = (\mathcal{S}, C)$ such that $\mathcal{S}$ is an annotated program and $C \neq \emptyset$, it follows that $|C| = 1$.

3.3. Stepwise Verification

Given a correct development step, $(\mathcal{A}_i, \mathcal{A}_{i+1}) = ((\mathcal{S}_i, C_i), (\mathcal{S}_{i+1}, C_{i+1}))$, and a while-program $W \in C_i$, we introduce proof rules which, when satisfied, enable us to prove that $W \in C_{i+1}$. It is necessary to have additional constraints other than $W \in C_i$, since C_{i+1} is a subset of C_i. The theorems which use these proof rules formalize the stepwise verification process. The proofs of these theorems clarify the stepwise verification process. For any correct development step, $((\mathcal{S}_i, C_i), (\mathcal{S}_{i+1}, C_{i+1}))$, and any $W \in C_i$, the conditions under which we can prove $W \in C_{i+1}$ depend upon the assumption that we will be able to prove the "incompletely specified parts" of W correct with respect to $\mathcal{S}_{i+1}$. Because it is only under this assumption that we can prove $W \in \mathcal{S}_{i+1}$, we do not have a verification of an implementation in as strong a sense as the verification of a program until we reach the last development step, which is correct and complete. At this point, no additional assumptions concerning the implementation W and the specification $\mathcal{S}_{i+1}$ are necessary to prove $W \in \mathcal{S}_{i+1}$, since W is completely specified by S_{i+1}.

These proof rules are somewhat similar to the rules in [14] for control refinement. Unlike [14], the rules we use are embedded in a methodology which uses the Hoare calculus for obtaining derivations. In section 6 we give an example of a lemma which uses the proof rules in a special case of a composed statement specification transformation.

4. Basic Definitions

In this section we give a precise definition of the syntax of while-programs, the syntax of specifications in terms of pre- and post-conditions, partial correctness of a while-program with respect to a specification, and the syntax of annotated programs. The definition of partial correctness of a while-program with respect to a specification is an extension of the notion of partial correctness of a while-program with respect to formulas. We need to define some terms which are used in these definitions. Let B be a basis for predicate logic, V the set of variables, T_B the set of terms, QFF_B the set of quantifier free formulas, and WFF_B the set of all well-formed formulas of first-order predicate logic over the basis B.

Definition: (*Syntax of $\mathcal{L}_W$*) The set, L_W^B, of *while-programs* for the basis B is defined inductively as follows:

a) *Assignment statement* If x is a variable from V and t is a term from T_B, then

$$x := t$$

is a while-program.

b) *Composed statement* If W_1, W_2 are while-programs then

$$W_1 ; W_2$$

is a while-program.

c) *Conditional statement* If W_1, W_2 are while-programs and e is a quantifier free formula from QFF_B, then

$$\textit{if}\ e\ \textit{then}\ W_1\ \textit{else}\ W_2\ \textit{fi}$$

is a while-program.

d) *While statement* If W_1 is a while-program and e is a quantifier free formula from QFF_B, then

$$\textit{while}\ e\ \textit{do}\ W_1\ \textit{od}$$

is a while-program.

Definition: (*Syntax of $\mathcal{L}_S$*) The set, L_S^B, of *specifications*, for the basis B is defined inductively as follows:

a) *Simple specification* If p, q are formulas from WFF_B, then

$$\{p\}\ \{q\}$$

is a specification.

b) *Assignment specification* If x is a variable from V, t is a term from T_B and p, q are formulas from WFF_B, then

$$\{p\}\ x := t\ \{q\}$$

is a specification.

c) *Composed specification* If S_1, S_2 are specifications and p, q are formulas from WFF_B, then

$$\{p\}\ S_1\ ;\ S_2\ \{q\}$$

is a specification.

d) *Conditional specification* If S_1, S_2 are specifications, e is a quantifier free formula from QFF_B, and p, q are formulas from WFF_B, then

$$\{p\}\ \textit{if}\ e\ \textit{then}\ S_1\ \textit{else}\ S_2\ \textit{fi}\ \{q\}$$

is a specification.

e) *While specification* If S_1 is a specification, e is a quantifier free formula from QFF_B, and p, q are formulas from WFF_B, then

$$\{p\}\ \textit{while}\ e\ \textit{do}\ S_1\ \textit{od}\ \{q\}$$

is a specification.

We call specifications which are not simple *structured specifications.* An operational semantics for $\mathcal{L}_W$ in terms of an interpretation $\mathcal{I}$ for the basis B and a definition of partial correctness with respect to formulas is given in [18].

Definition: (*Correctness with Respect to Specifications*) Let W be a while-program from L_W^B. The notion that W is *partially correct with respect to the specification S* (in the interpretation $\mathcal{I}$) is defined inductively (the induction being on the specification, S) as follows:

a) If S is a simple specification,

$$\{p\}\ \{q\},$$

where p, q are formulas from WFF_B, then W is partially correct with respect to S if

(i) W is partially correct with respect to p and q (in the interpretation $\mathcal{I}$).

b) If S is an assignment specification,

$$\{p\}\ x := t\ \{q\},$$

where x is a variable from V, t is a term from T_B and p, q are formulas from WFF_B, then W is partially correct with respect to S if the following hold:

(i) W is $x := t$.

(ii) W is partially correct with respect to p and q.

c) If S is a composed specification,

$$\{p\}\ S_1\ ;\ S_2\ \{q\},$$

where S_1, S_2 are specifications from L_S^B, and p, q are formulas from WFF_B, then W is partially correct with respect to S if the following hold:

(i) W is W_1 ; W_2 for some $W_1, W_2 \in L_W^B$.

(ii) W is partially correct with respect to p and q.

(iii) W_1 is partially correct with respect to the specification S_1.

(iv) W_2 is partially correct with respect to the specification S_2.

d) If S is a conditional specification,

$$\{p\}\ \textit{if}\ e\ \textit{then}\ S_1\ \textit{else}\ S_2\ \textit{fi}\ \{q\},$$

where S_1, S_2 are specifications from L_S^B, e is a quantifier free formula from QFF_B, and p, q are formulas from WFF_B, then W is partially correct with respect to S if the following hold:

(i) W is *if* e *then* W_1 *else* W_2 *fi* for some $W_1, W_2 \in L_W^B$.

(ii) W is partially correct with respect to p and q.

(iii) W_1 is partially correct with respect to the specification S_1.

(iv) W_2 is partially correct with respect to the specification S_2.

e) If S is a while specification,

$$\{p\}\ \textit{while}\ e\ \textit{do}\ S_1\ \textit{od}\ \{q\},$$

where S_1 is a specification from L_S^B, e is a quantifier free formula from QFF_B, and p, q are formulas from WFF_B, then W is partially correct with respect to S if the following hold:

(i) W is *while* e *do* W_1 *od* for some $W_1 \in L_W^B$.

(ii) W is partially correct with respect to p and q.

(iii) W_1 is partially correct with respect to the specification S_1.

Definition: Let W, I, S, p and q be as in the preceding definition. Then the formulas p and q are called, respectively, the *pre-condition* and *post-condition associated with the specification S*.

For example, if S is the simple specification,

$$\{p\}\ \{q\},$$

then the pre- and post-conditions associated with S are p and q.

Definition: (*Syntax of $\mathcal{L}_A$*) The set, L_A^B, of *annotated programs* for the basis B is defined inductively as follows:

a) *Assignment statement* If x is a variable from V , t is a term from T_B, and p, q are formulas from WFF_B, then

$$\{p\}\ x := t\ \{q\}$$

is an annotated program.

b) *Composed statement* If A_1, A_2 are annotated programs, and p, q are formulas from WFF_B, then

$$\{p\}\ A_1\ ;\ A_2\ \{q\}$$

is an annotated program.

c) *Conditional statement* If A_1, A_2 are annotated programs, p, q are formulas from WFF_B, and e is a quantifier free formula from QFF_B, then

$$\{p\}\ \textit{if}\ e\ \textit{then}\ A_1\ \textit{else}\ A_2\ \textit{fi}\ \{q\}$$

is an annotated program.

d) *While statement* If A_1 is an annotated program p, q are formulas from WFF_B, and e is a quantifier free formula from QFF_B, then

$$\{p\}\ \textit{while}\ e\ \textit{do}\ A_1\ \textit{od}\ \{q\}$$

is an annotated program.

We make a distinction in the preceding definitions between the sets of all while-programs, L_W^B, specifications, L_S^B, and annotated programs, L_A^B, and the corresponding sets along with an interpretation, which we denote by $\mathcal{L}_W$, $\mathcal{L}_S$, $\mathcal{L}_A$, respectively.

5. Derivations and Partial Correctness

In this section we assume some definitions and results concerning Hoare logic and calculus. See [18] for more details and [1] for a survey of Hoare logic. We denote by HF_B the set of all Hoare formulas,

$$\{p\}\ W\ \{q\},$$

where p, q $\in WFF_B$ and W $\in L_W^B$ is a while-program. A theory of an interpretation $\mathcal{I}$ of a basis B for predicate logic (denoted by $\mathrm{Th}(\mathcal{I})$) is the set of all formulas which are valid in $\mathcal{I}$. Proofs appear in [2] for the following two lemmas. The first lemma shows the connection between partial correctness with respect to a simple specification and the existence of a derivation from a theory of an interpretation. The second lemma shows how to construct an abstract program from a simple specification.

Lemma: (*Derivations from a Theory and Partial Correctness*) Let B be a basis for predicate logic and $\mathcal{I}$ an interpretation of B. Let S be the simple statement specification,

$$\{p\}\ \{q\}.$$

It follows that for each Hoare formula $\{p\}\ W\ \{q\} \in \mathrm{HF_B}$, if $\mathrm{Th}(\mathcal{I}) \vdash \{p\}\ W\ \{q\}$, then W is partially correct with respect to the specification S.

Lemma: Let S be the simple specification,

$$\{p\}\ \{q\},$$

and let

$$C = \{\ W \in L_W^B \mid \mathrm{Th}(\mathcal{I}) \vdash \{p\}\ W\ \{q\}\ \}.$$

Then (S, C) is an abstract program.

We introduce a definition which is an extension of the notion of the deduction of a Hoare formula from a theory. This definition is used to associate a set C of implementations with a specification S from L_S^B. This section also contains a theorem which shows that the pair, (S, C), is an abstract program. This extends a similar result for simple specifications.

Definition: (*Deduction Consistent with a Specification*) Let B be a basis for predicate logic, W a while-program from L_W^B, $\mathcal{I}$ an interpretation of the basis B, S a specification from L_S^B, and p′, q′, respectively, the pre- and post-conditions associated with the specification S. The notion that there is a *deduction from Th(I) to the Hoare formula* {p′} W {q′} *consistent with* S, denoted by:

$$\mathrm{Th}(\mathcal{I}) \vdash^{S} \{p'\}\ W\ \{q'\},$$

is defined inductively (the induction being on the specification, S) as follows:

a) If S is a simple specification,

$$\{p'\}\ \{q'\},$$

then

$$\mathrm{Th}(\mathcal{I}) \vdash^{S} \{p'\}\ W\ \{q'\}$$

if

(i) $\mathrm{Th}(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

b) If S is an assignment specification,

$$\{p'\}\ x := t\ \{q'\},$$

where x is a variable from V, t is a term from $\mathrm{T_B}$ then

$$\mathrm{Th}(\mathcal{I}) \vdash^{S} \{p'\}\ W\ \{q'\}$$

if the following hold:

(i) W is $x := t$

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

c) If S is a composed specification,

$$\{p'\}\ S_1\ ;\ S_2\ \{q'\},$$

where S_1, S_2 are specifications from L_S^B, p_1, q_1 and p_2, q_2 are the pre- and post-conditions associated with S_1, and S_2, respectively, then

$$\mathrm{Th}(\mathcal{I}) \vdash^{S} \{p'\}\ W\ \{q'\}$$

if the following hold:

(i) W is W_1 ; W_2 for some $W_1, W_2 \in L_W^B$

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$

(iii) $\mathrm{Th}(\mathcal{I}) \vdash^{S_1} \{p_1\}\ W_1\ \{q_1\}$

(iv) $\mathrm{Th}(\mathcal{I}) \vdash^{S_2} \{p_2\}\ W_2\ \{q_2\}$.

d) If S is a conditional specification,

$$\{p'\}\ \mathit{if}\ e\ \mathit{then}\ S_1\ \mathit{else}\ S_2\ \mathit{fi}\ \{q'\},$$

where S_1, S_2 are specifications from L_S^B, e is a quantifier free formula from $\mathrm{QFF_B}$, p_1, q_1 and p_2, q_2 are the pre- and post-conditions associated with S_1, and S_2, respectively, then

$$\mathrm{Th}(\mathcal{I}) \vdash^{S} \{p'\}\ W\ \{q'\}$$

if the following hold:

(i) W is *if* e *then* W_1 *else* W_2 *fi* for some $W_1, W_2 \in L_W^B$.

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$

(iii) $\mathrm{Th}(\mathcal{I}) \vdash^{S_1} \{p_1\}\ W_1\ \{q_1\}$

(iv) $\mathrm{Th}(\mathcal{I}) \vdash^{S_2} \{p_2\}\ W_2\ \{q_2\}$.

e) If S is a while specification,

$$\{p'\}\ \mathit{while}\ e\ \mathit{do}\ S_1\ \mathit{od}\ \{q'\},$$

where S_1 is a specification from L_S^B, e is a quantifier free formula from $\mathrm{QFF_B}$, and p_1, q_1 are the pre- and post-conditions associated with S_1, then

$$\mathrm{Th}(\mathcal{I}) \vdash^{S} \{p'\}\ W\ \{q'\}$$

if the following hold:

(i) W is *while* e *do* W_1 *od* for some $W_1 \in L_W^B$.

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$

(iii) $\mathrm{Th}(\mathcal{I}) \vdash^{S_1} \{p_1\}\ W_1\ \{q_1\}$.

Lemma: Let $W \in L_W^B$, $S \in L_S^B$, and let p′, q′ be the pre- and post-conditions associated with S. If

$$\mathrm{Th}(\mathcal{I}) \vdash^{S} \{p'\}\ W\ \{q'\},$$

then W is partially correct with respect to the specification $\mathcal{S}$.

Proof: This is an immediate consequence of the preceding definition, the definition of correctness with respect to specifications, and the lemma on derivations from a theory and partial correctness.

Note that in the case that $\mathcal{S}$ is the simple specification,

$$\{p\}\ \{q\},$$

$\mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}} \{p\}\ W\ \{q\}$, reduces to $\mathrm{Th}(\mathcal{I}) \vdash \{p\}\ W\ \{q\}$.

Just as the notion of partial correctness with respect to specifications is an extension of the notion of partial correctness with respect to formulas, the notion of a deduction from a theory of an interpretation to a Hoare formula consistent with a specification is an extension of the notion of a deduction from a theory of an interpretation to a Hoare formula. From the preceding lemma, we have the connection between derivations consistent with specifications and partial correctness of while-programs with respect to specifications. We use the next theorem in the construction of abstract programs from specifications.

Theorem: Let $\mathcal{S} \in L_S^B$, and let p′, q′ be the pre- and post-conditions associated with $\mathcal{S}$. If C is

$$\{\ W \in L_W^B \mid \mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}} \{p'\}\ W\ \{q'\}\ \},$$

then ($\mathcal{S}$, C) is an abstract program.

Proof: We need to show that for each W ∈ C, W is partially correct with respect to $\mathcal{S}$. This follows from the preceding lemma.

6. A Special Case of a Correct Development Step

We consider a somewhat simplified situation in which we wish to construct a correct development step. This is actually part of the basis step for an induction proof that the example does have the properties of the abstract model. We start with an abstract program $\mathcal{A} = (\mathcal{S}, C)$ for which $\mathcal{S}$ has the form,

$$\{p\}\ \{q\},$$

where p and q are formulas from $\mathrm{WFF_B}$; that is, $\mathcal{S}$ is a simple specification. C is the set of while-programs, $W \in L_W^B$, for which there exists a deduction in the Hoare calculus from the theory of the interpretation of the predicate logic to the Hoare formula {p} W {q} consistent with $\mathcal{S}$; that is,

$$C = \{\ W \in L_W^B \mid \mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}} \{p\}\ W\ \{q\}\ \}.$$

From the abstract program, ($\mathcal{S}$, C), we construct a new abstract program,

$$(\mathcal{S}', C'),$$

in which the specification, $\mathcal{S}'$, and the set of while-programs, C′, are related to $\mathcal{S}$ and C. The relationship involves the transformation of $\mathcal{S}$ by changing the simple specification into another specification. Using the notation of the abstract model, we have a transformation on the specifications,

$$T: \mathcal{S} \rightarrow \mathcal{S}'.$$

In terms of the example of the formal development the transformation can be expressed as

$$T: \{p\}\ \{q\} \rightarrow \{p\}\ \mathcal{S}_1\ \{q\}$$

where $\mathcal{S}_1 \in L_S^B$ is either an assignment statement specification, composed statement specification, conditional statement specification, or a while statement specification. We give a formal definition of these transformations in this section.

Let $\mathcal{S}'$ be $\{p\}\ \mathcal{S}_1\ \{q\}$. C′ is a set of while-programs for which there exists a deduction in the Hoare calculus from the theory of the interpretation of the predicate logic to the Hoare formula {p} W {q} consistent with $\mathcal{S}'$; that is, C′ is

$$\{\ W \in L_W^B \mid \mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}'} \{p\}\ W\ \{q\}\ \}.$$

We assume that both C and C′ ≠ Ø. This is an assumption that there exist while-programs which satisfy the specifications $\mathcal{S}$ and $\mathcal{S}'$. Since we are constructing an example of an idealized development, these assumptions are reasonable restrictions on the specifications. There are four possibilities for C′, depending upon the four kinds of transformation from {p} {q} to $\{p\}\ \mathcal{S}_1\ \{q\}$. In this section we will introduce conditions under which it is possible to guarantee that a while-program $W \in L_W^B$ is in C ∩ C′ for the case that T is a composed statement transformation. As a consequence of these conditions being satisfied, for each such transformation, T, and for each such while-program W, W is partially correct with respect to $\mathcal{S}'$ and $\mathcal{S}$.

Definition: (*Specification Transformations*) A transformation, T, from a simple specification, $\mathcal{S}$, which is {p} {q}, where p, q are formulas from $\mathrm{WFF_B}$, to another specification, $\mathcal{S}'$, which is the image under T, of $\mathcal{S}$, is defined as follows:

a) *Assignment statement transformation* If x is a variable from V, and t is a term from $\mathrm{T_B}$, then

$$T: \{p\}\ \{q\} \rightarrow \{p\}\ x := t\ \{q\}.$$

b) *Composed statement transformation* If p_1, p_2, q_1, q_2 are formulas from $\mathrm{WFF_B}$, and $\{p_1\}\ \{q_1\}$ and $\{p_2\}\ \{q_2\}$ are specifications, then

$$T : \{p\}\ \{q\} \rightarrow \{p\}\ \{p_1\}\ \{q_1\}\ ;\ \{p_2\}\ \{q_2\}\ \{q\}.$$

c) *Conditional statement transformation* If p_1, p_2, q_1, q_2 are formulas from WFF_B, and $\{p_1\}\ \{q_1\}$ and $\{p_2\}\ \{q_2\}$ are specifications, and e is a quantifier free formula from QFF_B, then

$$T : \{p\}\ \{q\} \rightarrow$$

$$\{p\}\ \textit{if}\ e\ \textit{then}\ \{p_1\}\ \{q_1\}\ \textit{else}\ \{p_2\}\ \{q_2\}\ \textit{fi}\ \{q\}.$$

d) *While statement transformation* If p_1, q_1 are formulas from WFF_B, $\{p_1\}\ \{q_1\}$ is a specification, and e is a quantifier free formula from QFF_B, then

$$T : \{p\}\ \{q\} \rightarrow \{p\}\ \textit{while}\ e\ \textit{do}\ \{p_1\}\ \{q_1\}\ \textit{od}\ \{q\}.$$

We note that the pre- and post-conditions associated with both S and S' are p and q. Thus, the transformation,

$$T: S \rightarrow S',$$

preserves pre- and post-conditions.

The lemma which follows gives conditions under which it is possible to have a derivation of a specific kind of Hoare formula. This Hoare formula is closely related to the composed statement specification transformation. We call these conditions *proof rules*, since they are sufficient to guarantee the existence of derivations in the Hoare calculus which will lead to a correct development step. In [2] proofs for the other three kinds of specification transformation are given. Because of the way in which specifications are defined, these transformations are very similar to program transformations. See [19] for a general survey of program transformations.

Lemma: (*Composed Statement Derivation*) Let T: $S \rightarrow S'$ be a composed statement transformation,

$$T: \{p\}\ \{q\} \rightarrow \{p\}\ \{p_1\}\ \{q_1\}\ ;\ \{p_2\}\ \{q_2\}\ \{q\}.$$

Let $W \in L^B_W$. Suppose that W is

$$W_1\ ;\ W_2$$

for some $W_1, W_2 \in L^B_W$. Let p, p_1, p_2, q, q_1, q_2 be formulas from WFF_B, and $\{p\}\ \{q\}$, $\{p_1\}\ \{q_1\}$, and $\{p_2\}\ \{q_2\}$ be specifications from L^B_S. Furthermore, assume that there exists a derivation of the following formulas from the theory of the interpretation $\mathcal{I}$:

a) $p \Rightarrow p_1$

b) $q_1 \Rightarrow p_2$

c) $q_2 \Rightarrow q$

d) $\{p_1\}\ W_1\ \{q_1\}$ for some $W_1 \in L^B_W$

e) $\{p_2\}\ W_2\ \{q_2\}$ for some $W_2 \in L^B_W$.

Then $W \in C \cap C'$.

Proof: As a consequence of a) – e) there exists the following deduction in the Hoare calculus:

$$Th(\mathcal{I}) \vdash \{p\}\ W_1\ ;\ W_2\ \{q\}.$$

It follows that $W \in C$.

Let S_1 be $\{p_1\}\ \{q_1\}$ and S_2 be $\{p_2\}\ \{q_2\}$. If the following hold

i) W is W_1 ; W_2 for some $W_1, W_2 \in L^B_W$

ii) $Th(\mathcal{I}) \vdash \{p\}\ W\ \{q\}$

iii) $Th(\mathcal{I}) \vdash^{S_1} \{p_1\}\ W_1\ \{q_1\}$

iv) $Th(\mathcal{I}) \vdash^{S_2} \{p_2\}\ W_2\ \{q_2\}$

then

$$Th(\mathcal{I}) \vdash^{S'} \{p\}\ W\ \{q\}$$

and $W \in C'$. Condition i) holds by assumption. Condition ii) is a consequence of a) – e). Condition iii) follows from d) and the fact that

$$Th(\mathcal{I}) \vdash^{S_1} \{p_1\}\ W_1\ \{q_1\}$$

is

$$Th(\mathcal{I}) \vdash \{p_1\}\ W_1\ \{q_1\}.$$

Condition iv) follows from e) and the fact that

$$Th(\mathcal{I}) \vdash^{S_2} \{p_2\}\ W_2\ \{q_2\}$$

is

$$Th(\mathcal{I}) \vdash \{p_2\}\ W_2\ \{q_2\}.$$

In this section we have shown that the existence of a simple specification transformation and the satisfaction of the proof rules implies that $W \in C' \cap C$. In [2] it is proved that given $W \in C$, a transformation of a structured specification and the satisfaction of the proof rules then $W \in C'$. This gives an explicit method for going from the higher level specification in a development step to the lower level (more detailed) specification in a development step. The fact that $C' \subseteq C$ follows from the existence of a specification transformation from S to S'. The proof appears in [2].

7. Conclusion

We have presented an example of a stepwise development methodology and have outlined the proof that it has the properties of an abstract model for stepwise development. We have also given some details of the approach used to prove that the example does have the properties of the model. Section 6 contains the proofs for only one of four cases needed in the basis for an induction proof of the correctness of a development step. We have not even considered the induction step for the proof of the correctness of a development step in this paper although in [2] a complete proof is presented. In order to apply the methodology we do need a completeness result. If we use an expressive interpretation for the Hoare logic [9], [18], then we have the required completeness. The expressive interpretations are basically the finite interpretation and the interpretation of the usual arithmetic of nonnegative integers. The existence of expressive interpretations is considered in [17] and discussed in [8].

We are primarily interested in using the technique of an abstract model as an aid in constructing and reasoning about stepwise development methods. The example we have given shows that even for the simple model we introduced rather deep results concerning the deductive system (such as the existence of expressive interpretations in Hoare logic in the example presented) may be needed to prove a methodology has the properties of the model.

Acknowledgement The author would like to acknowledge many helpful discussions with Roy Campbell and Bob Terwilliger. The definitions of the sets of specifications and annotated programs are due to Bob Terwilliger.

8. References

1. Apt, Krzysztof R. *Ten Years of Hoare's Logic: A Survey - Part I.* **ACM Transactions on Programming Languages and Systems** (October 1981) vol. 3, no. 4, pp. 431–483.

2. Benzinger, Lee A. "Toward a Theory of the Stepwise Development of Programs", In preparation, Dept. of Computer Science, University of Illinois, 1986.

3. Beshers, George M. and Roy H. Campbell. *Maintained and Constructor Attributes.* **Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments** (June 1985) pp. 34–42.

4. Campbell, Roy H. *SAGA: A Project to Automate the Management of Software Production Systems.* In: **Software Engineering Environments**, Ian Sommerville, ed. Peter Perigrinus Ltd, 1986.

5. Campbell, Roy H. and Peter A. Kirslis. *The SAGA Project: A System for Software Development.* **Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments** (April 1984) pp. 73–80.

6. Campbell, Roy H. and Paul G. Richards. *SAGA: A system to automate the management of software production.* **Proceedings of the National Computer Conference** (May 1981) pp. 231–234.

7. Campbell, Roy H. and Robert B. Terwilliger. *The SAGA Approach to Automated Project Management.* In: **International Workshop on Advanced Programming Environments**, Lynn R. Carter, ed. Springer-Verlag Lecture Notes in Computer Science, New York, 1986.

8. Clarke, E. M., Jr., S. M. German and J. Y. Halpern. *On Effective Axiomatizations of Hoare Logics.* **Proceedings of the 9th ACM Symposium on Principles of Programming Languages** (January 1982) pp. 309–321.

9. Cook, S. A. *Soundness and Completeness of an Axiom System for Program Verification.* **SIAM Journal of Computing** (February 1978) vol. 7, no. 1, pp. 70–90.

10. Dijkstra, Edsger W. *A Constructive Approach to the Problem of Program Correctness.* BIT (1968) pp. 174–186.

11. Gries, David. **The Science of Programming.** Springer-Verlag, New York, 1981.

12. Hammerslag, David H., Samuel N. Kamin and Roy H. Campbell. *Tree-Oriented Interactive Processing with an Application to Theorem-Proving.* **Proceedings of the Second ACM/IEEE Conference on Software Development Tools, Techniques, and Alternatives** (December, 1985).

13. Hoare, C. A. R. *An Axiomatic Basis for Computer Programming.* **Communications of the ACM** (October 1969) vol. 12, no. 10, pp. 576–580.

14. Jones, Cliff B. **Software Development: A Rigorous Approach**. Prentice-Hall International, Engelwood Cliffs, N.J., 1980.

15. Kirslis, Peter A., Robert B. Terwilliger and Roy H. Campbell. *The SAGA Approach to Large Program Development in an Integrated Modular Environment.* **Proceedings of the GTE Workshop on Software Engineering Environments for Programming-in-the-Large** (June 1985) pp. 44–53.

16. Lehman, M. M., V. Stenning and W. M. Turski. *Another Look at Software Design Methodology.*

Software Engineering Notes (April 1984) vol. 9, no. 2, pp. 38–53.

17. Lipton, R. J. *A Necessary and Sufficient Condition for the Existence of Hoare Logics.* **Proceedings of the 18th IEEE Symposium on Foundations of Computer Science** (October 1977) pp. 1–6.

18. Loeckx, Jacques and Kurt Sieber. **The Foundations of Program Verification.** John Wiley & Sons, New York, 1984.

19. Partsch, H. and R. Steinbruggen. *Program Transformation Systems.* **Computing Surveys** (September 1983) vol. 15, no. 3, pp. 199–236.

20. Terwilliger, Robert B. and Roy H. Campbell. *ENCOMPASS: a SAGA Based Environment for the Composition of Programs and Specifications.* **Proceedings of the 19th Hawaii International Conference on System Sciences** (January 1986) pp. 436–447.

21. ——. *PLEASE: Predicate Logic based ExecutAble SpEcifications.* **Proceedings of the 1986 ACM Computer Science Conference** (February, 1986) pp. 349–358.

22. Wirth, Niklaus. *Program Development by Stepwise Refinement.* **Communications of the ACM** (April 1971) vol. 14, no. 4, pp. 221–227.

An Abstract Model for the Stepwise Development of Programs

Lee A. Benzinger

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, Illinois

Submitted to the 4th International Workshop on Software Specifications
and Design, April 3-4, 1987, Monterey, CA.

An Abstract Model for the Stepwise Development of Programs

Lee A. Benzinger*
Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield Ave.
Urbana, Illinois 61801

Abstract

We present an abstract model for the stepwise development of programs. The model describes an idealized development which is independent of specification method and notion of correctness. We prove several results about the abstract model. These results show that the model possesses many of the properties that we would expect of an idealized stepwise development.

1. Introduction

Many approaches have been suggested to ease the difficulty of the task of producing software components which satisfy given specifications; for example, in [4] the problem of designing an algorithm which meets a specification is discussed. In [6] an axiomatic approach to correctness proofs for programs is given, while in [11], [5] stepwise approaches to program development are presented. The Vienna Development Method (VDM) [7] combines the notions of stepwise development and correctness proofs. In [8] a stepwise approach to software design methodology, which incorporates a notion of program correctness with respect to a specification, is discussed.

The stepwise development of programs which satisfy given specifications has been presented as a methodology. In order to reason about a particular design methodology, to compare different design methodologies, or to construct new design methodologies, it would be valuable to have a abstract mathematical model for the design process of developing verified software components by using a stepwise development method. The purpose of this paper is to construct such a model and to show that the model possesses the properties that one would intuitively expect of an idealized stepwise development method. The model does not merely provide a unifying conceptual foundation for stepwise program development methods, but the properties of the model give insight into the stepwise development methods described by the model. For example, implicit assumptions contained within certain stepwise development methods are actually theorems which are true for the abstract model or consequences of definitions which are used to construct the abstract model. In the consideration of the example of a software component development method, it becomes clear exactly what properties the example must have in order to agree with the model. This is extremely useful, since a stepwise development method which does not have these properties cannot be guaranteed to behave like the abstract model.

2. The Abstract Model

We give basic definitions and results which we use to construct the abstract model in section 2.1. In section 2.2 we introduce definitions of an abstract program and three classes of developments. In section 2.3 we obtain some results about these classes of developments. In section 2.4 we introduce definitions of classes of development steps and we show how developments can be extended with development steps in

*This research is supported in part by NASA grant NAG 1-138, Roy H. Campbell principle investigator.

section 2.5. In section 2.6 we show that developments can be constructed from development steps, so that development steps can be viewed as "building blocks" for developments.

2.1. Foundation for the Abstract Model

In this section we present some basic definitions and a theorem which we use to construct the model. Most of these appear in [10]. We mention the notion of an implementation being correct with respect to a specification. For the purposes of constructing the abstract model and investigating the properties of the abstract model we do not explicitly define correctness with respect to a specification. Also we do not precisely define what we mean by a specification for a software component. The basic idea is to investigate the properties of an abstract model for stepwise development which are independent of the specification method and notion of correctness. The purpose of the abstract model is to provide a framework to study an incremental development method for a particular approach to specification and a particular notion of correctness. This enables us to distinguish between those properties which are characteristic of an incremental development and those properties which are intrinsic to a specific incremental development method.

2.1.1. Notation:
Let

$Bool = \{ \text{true}, \text{false} \}$
SPEC = $\{ \mathcal{S} \mid \mathcal{S}$ is a specification $\}$
IMPL = $\{ p \mid p$ is an implementation $\}$.

2.1.2. Definition: Let f: SPEC × IMPL → *Bool* be defined as follows:

$$f(\mathcal{S}, p) = \begin{cases} \text{true} & \text{if p is correct with respect to } \mathcal{S} \\ \text{false} & \text{otherwise.} \end{cases}$$

2.1.3. Definition: Let $\mathcal{S}$ be an element of SPEC. Let $C = \{ p \in \text{IMPL} \mid f(\mathcal{S}, p) = \text{true} \}$.

It may be that $C = \emptyset$, that is, $\mathcal{S}$ is a specification for which there exists no correct implementation. This can occur if, for example, the specification $\mathcal{S}$ is inconsistent.

2.1.4. Definition: A *partial order* is a pair (P, R) where P is a set and R is a relation on P which is reflexive (for all a ∈ P, aRa), antisymmetric (for all a, b ∈ P, aRb and bRa implies that a = b), and transitive (for all a, b, c ∈ P, aRb and bRc implies that aRc).

2.1.5. Definition: Let (P, R) be a partial order and S a nonempty subset of P. S is called a *chain* if aRb or bRa (or both) holds for all a, b ∈ S. This simply means that the relation "R" restricted to S is total.

2.1.6. Definition: Let (P, R) be a partial order and S be a subset of P. Let a ∈ S. The element a is a *least element* of S if, for all b ∈ S, aRb.

2.1.7. Definition: Let (P, R) be a partial order and S a subset of P. An element a of P is an *upper bound* of S (in P) if bRa for all b ∈ S. An element a of P is the *least upper bound* (lub) of S if a is the least element of the set of upper bounds of S in P.

2.1.8. Definition: A partial order (P, R) is a *complete partial order*, denoted by *cpo*, if the following two conditions hold:

(1) The set P has a least element.
(2) For every chain S in P the least upper bound, lub S exists.

2.1.9. Theorem: Every partial order which contains a least element and contains only finite chains is a cpo.

2.1.10. Notation: Let S be a set. By $\mathcal{P}(S)$ we mean the power set of S.

2.1.11. Lemma: If C_i is a set of implementations which are correct with respect to the specification $\mathcal{S}_i$ for some integer i, $i \geq 0$, then the ordered pair $(\mathcal{P}(C_i), \subseteq)$ is a partial order.
Proof: This follows from the fact that the relation "$\subseteq$" is reflexive, antisymmetric, and transitive.

2.2. Classes of Developments

In this section we introduce definitions of an abstract program, a development with respect to a specification, a correct development, a complete development, and an incomplete development. The formal definitions concerning a development and the various classifications of developments correspond to the intuitive notions of development and kinds of developments which occur in a stepwise development process.

2.2.1. Definition: An *abstract program* $\mathcal{A}$ is an ordered pair, $(\mathcal{S}, C)$, such that the first member of the pair, $\mathcal{S}$, is a specification, and the second member of the pair, C, is a set of implementations which are correct with respect to $\mathcal{S}$.

2.2.2. Definition: A *development* with respect to a specification $\mathcal{S}_0$ is an (n + 1)-tuple of abstract programs, $(\mathcal{A}_0, \mathcal{A}_1, ..., \mathcal{A}_n)$, for some nonnegative integer n such that for each i, $0 \leq i \leq n$, $\mathcal{A}_i = (\mathcal{S}_i, C_i)$.

In the discussion of developments, we shall usually write out the abstract programs explicitly as ordered pairs of specifications and sets of implementations, since it is the interaction of the components of the ordered pair rather than the abstract programs themselves which are of interest.

2.2.3. Notation: Let S be a set. By $|S|$ we mean the cardinality of S.

2.2.4. Definition: A development with respect to a specification $\mathcal{S}_0$, $((\mathcal{S}_0, C_0), (\mathcal{S}_1, C_1), ..., (\mathcal{S}_n, C_n))$ is *correct* if $C_{i+1} \subseteq C_i$, $0 \leq i < n$.

It is possible that a correct development with respect to a specification, $\mathcal{S}_0$, will have a set of implementations, C_i, for which $C_i = \emptyset$. Then all C_j, for $j \geq i$, will also be equal to the empty set. These kinds of developments will not be of interest in themselves, since they do not lead to an implementation which is correct with respect to the original specification, $\mathcal{S}_0$. What is needed is an additional property for developments, which will ensure that the sets of implementations associated with the specifications in the development will all be nonempty. There are two properties of developments which enable us to describe the kinds of developments that we wish to consider. These two properties are independent of the correctness of a development, but will be used only in conjunction with correct developments. A correct development, which is *complete*, is a development ending in a single implementation which is correct with respect to the specification with which it is associated. A correct development, which is *incomplete*, is a development which may extended (in a sense which will be made precise later) to form a correct and complete development. We define the notions of a complete development and an incomplete development more precisely.

2.2.5. Definition: A development with respect to a specification $\mathcal{S}_0$, $((\mathcal{S}_0, C_0), (\mathcal{S}_1, C_1), ..., (\mathcal{S}_n, C_n))$ is *complete* if $|C_n| = 1$.

2.2.6. Definition: A development with respect to a specification $\mathcal{S}_0$, $((\mathcal{S}_0, C_0), (\mathcal{S}_1, C_1), ..., (\mathcal{S}_n, C_n))$ is *incomplete* if $|C_n| > 1$.

2.3. Properties of Classes of Developments

In this section we show that the sets of implementations associated with correct developments and correct and complete developments form nonempty, finite, nested families. This leads to theorem 2.4.2 which states that an implementation, which is correct with respect to a specification, is also correct with respect to all preceding specifications in a development. The last result of this section states that the sets of implementations associated with a correct and complete development form a cpo when ordered by set inclusion.

2.3.1. Theorem: Let D be a correct and complete development, $((\mathcal{S}_0, C_0), (\mathcal{S}_1, C_1), ..., (\mathcal{S}_n, C_n))$, with respect to the specification $\mathcal{S}_0$. It follows that:

(1) for each integer i, $0 \leq i \leq n$, C_i is a subset of the set of all implementations which are correct with respect to the specification $\mathcal{S}_i$

(2) $C_{i+1} \subseteq C_i$, $0 \leq i < n$

(3) $|C_n| = 1$.

Proof: Property (1) follows from the fact that D is a development. Property (2) follows from the fact that D is correct, and property (3) holds because D is complete.

As a direct consequence of the preceding theorem, if D is a correct and complete development or a correct and incomplete development then it follows that for each i, $0 \leq i \leq n$, $C_i \neq \emptyset$.

2.3.2. Theorem: Let $((\mathcal{S}_0, C_0), (\mathcal{S}_1, C_1), ..., (\mathcal{S}_n, C_n))$ be a correct development with respect to the specification $\mathcal{S}_0$. If $p \in C_i$ for some integer i, $0 \leq i \leq n$, then p is correct with respect to all specifications $\mathcal{S}_j$, $0 \leq j \leq i$.
Proof: If $p \in C_i$ for some integer i, $0 \leq i \leq n$, then $p \in C_j$ for all integers j, $0 \leq j < i$ from the definition of a correct development.

2.3.3. Lemma: Let $((\mathcal{S}_0, C_0), (\mathcal{S}_1, C_1), ..., (\mathcal{S}_n, C_n))$ be a correct development with respect to a specification $\mathcal{S}_0$. Let $S = \{C_0, C_1, ..., C_n\}$. Then S is a finite chain in $(\mathcal{P}(C_0), \subseteq)$.
Proof: Clearly, S is finite and the order relation "$\subseteq$" restricted to the set S is total.

2.3.4. Definition: We call the set S a *finite chain associated with the correct development.*

2.3.5. Notation: Let DV be the union of $\{\emptyset\}$ with the collection of all finite chains associated with all correct and complete developments with respect to a specification $\mathcal{S}_0$.

2.3.6. Theorem: The ordered pair (DV, $\subseteq$) is a cpo.
Proof: By lemma 2.1.11, $(\mathcal{P}(C_0), \subseteq)$ is a partial order. It follows that (DV, $\subseteq$) is a partial order. The least element of DV is $\emptyset$. By the preceding lemma and the fact that any correct and complete development with respect to the specification $\mathcal{S}_0$ is also a finite chain in (DV, $\subseteq$), (DV, $\subseteq$) contains only finite chains. By theorem 2.1.9, (DV, $\subseteq$) is a cpo.

2.4. Classes of Development Steps

In this section we introduce the notion of a development step and several classifications of development steps. We classify development steps as correct, incomplete, and complete. Correct development steps are those development steps that have properties which make these steps suitable for use in constructing correct developments. Incomplete development steps are used in constructing all but the final step in a development, while a complete development step is used as the final step in the construction of a development. The notion of a development step is fundamental, since it is the concept which describes the result of a process of going from one abstract program to another.

2.4.1. Definition: A *development step* with respect to a specification $\mathcal{S}_i$ is an ordered pair of abstract programs, $(\mathcal{A}_i, \mathcal{A}_{i+1})$, such that $\mathcal{A}_i = (\mathcal{S}_i, C_i)$ and $\mathcal{A}_{i+1} = (\mathcal{S}_{i+1}, C_{i+1})$.

For the same reasons presented in the discussion of developments, in the discussion of development steps, we shall usually write out the abstract programs explicitly as ordered pairs of specifications and sets of implementations.

2.4.2. Definition: Let $((\mathcal{S}_0, C_0), (\mathcal{S}_1, C_1), ..., (\mathcal{S}_n, C_n))$ be a development with respect to a specification $\mathcal{S}_0$. Let $((\mathcal{S}_j, C_j), (\mathcal{S}_{j+1}, C_{j+1}))$ be a development step with respect to the specification $\mathcal{S}_j$. The development *contains the development step* if j = i for some integer i, $0 \leq i \leq n - 1$, that is, the development step is $((\mathcal{S}_i, C_i), (\mathcal{S}_{i+1}, C_{i+1}))$, where $(\mathcal{S}_i, C_i)$ and $(\mathcal{S}_{i+1}, C_{i+1})$ are successive members of the (n + 1)-tuple which is the development with respect to the specification $\mathcal{S}_0$.

2.4.3. Definition: A development step with respect to a specification $\mathcal{S}_i$ for some nonnegative integer i, $((\mathcal{S}_i, C_i), (\mathcal{S}_{i+1}, C_{i+1}))$, is *correct* if the following hold:
(1) $C_i, C_{i+1} \neq \emptyset$
(2) $C_{i+1} \subseteq C_i$.

2.4.4. Definition: A development step with respect to a specification $\mathcal{S}_i$, $((\mathcal{S}_i, C_i),(\mathcal{S}_{i+1}, C_{i+1}))$, is *complete* if $|C_{i+1}| = 1$.

2.4.5. Definition: A development step with respect to a specification $\mathcal{S}_i$, $((\mathcal{S}_i, C_i),(\mathcal{S}_{i+1}, C_{i+1}))$, is *incomplete* if $|C_{i+1}| > 1$.

2.5. The Extension of Developments with Development Steps

The results in this section show that developments can be extended by development steps to form new developments. The resulting new developments have properties which depend upon the original developments and the development steps.

2.5.1. Lemma: Let D be a development, $((S_0, C_0), (S_1, C_1), \ldots, (S_n, C_n))$, with respect to the specification S_0. Suppose that $((S_n, C_n), (S_{n+1}, C_{n+1}))$ is a development step with respect to S_n. Let D_1 be the ordered (n + 2)-tuple, $((S_0, C_0), (S_1, C_1), \ldots, (S_{n+1}, C_{n+1}))$. Then D_1 is a development with respect to the specification S_0, which contains the given development step.

Proof: The ordered (n + 2)-tuple, D_1, is a development with respect to the specification, S_0, since it can be shown that

(1) for each i, $0 \leq i \leq n$, C_i is the set of all implementations which are correct with respect to the specification S_i

(2) C_{n+1} is the set of all implementations which are correct with respect to the specification S_{n+1}.

Property (1) follows from the assumption that D is a development, while property (2) follows from the assumption that $((S_n, C_n), (S_{n+1}, C_{n+1}))$ is a development step with respect to S_n. The development D_1 is clearly a development which contains the given development step, $((S_n, C_n), (S_{n+1}, C_{n+1}))$.

2.5.2. Lemma: Let D be a correct development, $((S_0, C_0), (S_1, C_1), \ldots, (S_n, C_n))$, with respect to the specification S_0. Suppose that $((S_n, C_n), (S_{n+1}, C_{n+1}))$ is a correct development step with respect to S_n. Then D_1, the ordered (n + 2)-tuple, $((S_0, C_0), (S_1, C_1), \ldots, (S_{n+1}, C_{n+1}))$, is a correct development with respect to the specification S_0, which contains the given development step.

Proof: From the preceding lemma, D_1 is a development with respect to the specification S_0 which contains the given development step. D_1 is a correct development since it can be shown that

(1) $C_{i+1} \subseteq C_i, 0 \leq i \leq n$

(2) $C_n \subseteq C_{n+1}$.

Property (1) follows from the assumption that D is correct and property (2) follows from the assumption that the development step, $((S_n, C_n), (S_{n+1}, C_{n+1}))$, is a correct development step with respect to S_n.

2.5.3. Theorem: Let D be a correct development, $((S_0, C_0), (S_1, C_1), \ldots, (S_n, C_n))$, with respect to the specification S_0. Suppose that $((S_n, C_n), (S_{n+1}, C_{n+1}))$ is a complete and correct development step with respect to S_n. Let D_1 be $((S_0, C_0), (S_1, C_1), \ldots, (S_{n+1}, C_{n+1}))$. D_1 is a correct and complete development with respect to the specification S_0, which contains the given development step.

Proof: From the preceding lemma, D_1 is a correct development with respect to the specification S_0 which contains the given development step. Because $((S_n, C_n), (S_{n+1}, C_{n+1}))$ is a complete development step with respect to S_n, it follows that $|C_{n+1}| = 1$. This shows that the development is complete.

2.5.4. Corollary: Let D be a correct and incomplete development, $((S_0, C_0), (S_1, C_1), \ldots, (S_n, C_n))$, with respect to the specification S_0. Suppose that $((S_n, C_n), (S_{n+1}, C_{n+1}))$ is a complete and correct development step with respect to S_n. Let D_1 be $((S_0, C_0), (S_1, C_1), \ldots, (S_{n+1}, C_{n+1}))$. D_1 is a correct and complete development with respect to the specification S_0, which contains the given development step.

2.6. The Construction of Developments from Development Steps

In this section we show that developments can be constructed from development steps. The properties of the resulting developments are dependent upon the properties of the development steps used in the construction of the developments.

2.6.1. Lemma: Let $((S_0, C_0), (S_1, C_1)), ((S_1, C_1), (S_2, C_2)), \ldots, ((S_{n-1}, C_{n-1}), (S_n, C_n))$ be a collection of n correct development steps with respect to the specifications $S_0, S_1, \ldots, S_n$ respectively, for some positive integer n. Let $D = ((S_0, C_0), (S_1, C_1), \ldots, (S_n, C_n))$. Then D is a correct development with respect to the specification S_0.

Proof: D is a development with respect to the specification S_0 from the definition of a development step. Since $((S_i, C_i), S_{i+1}, C_{i+1}))$ is a correct development step with respect to the specification S_i for each integer i, $0 \leq i < n$, $C_{i+1} \subseteq C_i$. It follows that D is a correct development.

2.6.2. Theorem: Let $((S_0, C_0), (S_1, C_1)), ((S_1, C_1), (S_2, C_2)), \ldots, ((S_{n-1}, C_{n-1}), (S_n, C_n))$ be a collection of n correct development steps with respect to the specifications $S_0, S_1, \ldots, S_n$ respectively, for some positive integer n. Furthermore, suppose that $((S_{n-1}, C_{n-1}), (S_n, C_n))$ is a complete development step. Let $D = ((S_0, C_0), (S_1, C_1), \ldots, (S_n, C_n))$. Then D is a correct and complete development with respect to the specification S_0.

Proof: From the preceding lemma, D is a correct development with respect to the specification S_0. Since $((S_{n-1}, C_{n-1}), (S_n, C_n))$ is a complete development step, $|C_n| = 1$. It follows that D is a complete development.

3. Conclusion

The abstract model describes the process of starting with a specification for a software component and, through a series of steps, ending with an implementation which is correct with respect to the original specification. The model has been used to construct an example of a stepwise development method in [1] This example uses specifications in terms of pre– and post–conditions [10]. The notion of correctness which is used is an extension of partial correctness with respect to formulas from predicate logic. In the construction of the example, an extension of the Hoare calculus from a program verification method to a stepwise program development method was required. Conceptually, this extension was fairly easy to make because of the existence of the abstract model. On the other hand, the proof that an example of a stepwise development method actually does satisfy the requirements of the abstract model may depend upon non-trivial properties of the example. For the example that we have studied, the proof depends upon the relative completeness of the Hoare calculus [3] and the existence of expressive interpretations for the Hoare logic [10], [2], [9].

Acknowledgement The author would like to acknowledge many helpful discussions with Bob Terwilliger.

4. References

1. Benzinger, Lee A. "Toward a Theory of the Stepwise Development of Programs", In preparation, Dept. of Computer Science, University of Illinois, 1986.

2. Clarke, E. M., Jr., S. M. German and J. Y. Halpern. *On Effective Axiomatizations of Hoare Logics.* **Proceedings of the 9th ACM Symposium on Principles of Programming Languages** (January 1982) pp. 309-321.

3. Cook, S. A. *Soundness and Completeness of an Axiom System for Program Verification.* **SIAM Journal of Computing** (February 1978) vol. 7, no. 1, pp. 70–90.

4. Dijkstra, Edsger W. *A Constructive Approach to the Problem of Program Correctness.* **BIT** (1968) pp. 174–186.

5. Gries, David. **The Science of Programming.** Springer–Verlag, New York, 1981.

6. Hoare, C. A. R. *An Axiomatic Basis for Computer Programming.* **Communications of the ACM** (October 1969) vol. 12, no. 10, pp. 576–580.

7. Jones, Cliff B. **Software Development: A Rigorous Approach.** Prentice–Hall International, Engelwood Cliffs, N.J., 1980.

8. Lehman, M. M., V. Stenning and W. M. Turski. *Another Look at Software Design Methodology.* **Software Engineering Notes** (April 1984) vol. 9, no. 2, pp. 38–53.

9. Lipton, R. J. *A Necessary and Sufficient Condition for the Existence of Hoare Logics.* **Proceedings of the 18th IEEE Symposium on Foundations of Computer Science** (October 1977) pp. 1–6.

10. Loeckx, Jacques and Kurt Sieber. **The Foundations of Program Verification.** John Wiley & Sons, New York, 1984.

11. Wirth, Niklaus. *Program Development by Stepwise Refinement.* **Communications of the ACM** (April 1971) vol. 14, no. 4, pp. 221–227.

SAGA Project Mid-Year Report 1986 Appendix H

Toward a Theory of the Stepwise Development of Programs

Lee A. Benzinger

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, Illinois

Toward a Theory of the Stepwise Development of Programs

PAPER

Lee A. Benzinger

Department of Computer Science
University of Illinois at Urbana–Champaign
1304 W. Springfield Ave.
Urbana, Illinois 61801
217–333–8426

Abstract

We present an abstract model for the stepwise development of programs. The model describes an idealized development which is independent of specification

method and notion of correctness. We illustrate the abstract model with an example of a program development method using Hoare calculus and partial correctness of programs with respect to specifications which are in terms of pre- and post-conditions.

This research is supported in part by NASA grant NAG 1-138.

Table of Contents

1.

Introduction

It is a difficult task to develop a software component which satisfies a given specification. If the specification is not precise, as in the case of a specification in terms of natural language, the ambiguities in the specification can create confusion as to the meaning of the specification and the intent of the specifier. The introduction of a formal specification, which uses well–defined notation, can eliminate ambiguities. A disadvantage of such a formal specification is that it may be more difficult to understand, simply due to notation, than a less formal specification.

The matter of showing that a software component actually satisfies a specification can be accomplished through testing or verification. The testing approach does not, in general, provide a guarantee that a software component satisfies a specification, while verification quickly becomes a formidable problem as the size and complexity of a component increases. In addition, verification requires that a specification be expressed in some formal manner.

The task of developing a software component which satisfies a given specification can be simplified by breaking the task into a series of subtasks or "steps". Associated with each step is a specification and a software component. Initially, the software component is nothing more than the original specification. At each step, the specification associated with the step is more detailed than the specifications associated with preceding steps. In addition, the specification associated with a particular step is consistent with the specifications associated with preceding steps. The software components corresponding to these specifications become increasingly more detailed as the stepwise process proceeds. At the final step in the process, the result is a final specification and a final software component. This final specification is consistent with all preceding specifications, including the original specification. The corresponding final software component is an implementation which not only satisfies the specification associated with the final step, but also satisfies the original specification. This technique is used by the Vienna Development Method [2].

In the Vienna Development Method the processes of developing a software component and verifying that it actually satisfies the requirements of the original specification proceed hand in hand. This has the dual advantage of the development process aiding the verification process and the verification process, in turn, aiding the development process. The development process, because of its incremental nature, breaks the verification process into smaller parts, each of which is a piece of the total problem of verifying that the implementation which results from the development actually satisfies the original specification. The verification process, because of its incremental nature, aids in the development process. Indeed, the part of a software component under development which may not satisfy the requirements of its specification is at most one logical step away from a component under development which is known to satisfy its specification. Each step in the development will either eventually yield an implementation which will satisfy the original specification or backtracking occurs which itself eventually yields an implementation which will satisfy the original specification. This approach can reduce the time and cost of developing reliable software since design decisions can be checked for correctness in the middle of the development process and can be changed precisely at the point in the development of a software component which is affected by these decisions.

In order to reason about a particular design approach or to compare different design methods, it would be valuable to have an abstract model for the design process of developing verified software components by using a stepwise development method. The purpose of this paper is to construct such a model, to show that the model possesses the properties that one would expect of a stepwise development method, and finally, to show an example of a particular software component development method that falls within the framework provided by the model.

The model does not merely provide a unifying conceptual foundation for stepwise program development methods, but the properties of the model give insight into the stepwise development methods described by the model. For example, implicit assumptions contained within certain stepwise development methods are actually theorems which are true for the abstract model or consequences of definitions which are used to construct the abstract model. In the consideration of the example of a software component development method, it becomes clear exactly what properties the example must have in order to agree with the model. This is extremely useful, since a stepwise development method which does not have these properties cannot be guaranteed to behave like the abstract model.

2. The Abstract Model

In this section we introduce definitions which we use to construct the model. We also obtain some results about the model. Definitions and theorems which are standard are not numbered. Most of the standard definitions and theorems appear in [3]. Definitions introduced in the construction of the model, and theorems, lemmas, and corollaries, which we prove concerning the model, are numbered. The definitions which we introduce include the notions of an abstract program, a development with respect to a given specification, a•correct development, a complete development, an incomplete development, a development step, a correct development step, and a complete development step.

There are five main results in this section. The first result is Theorem 1 which gives three properties of a correct and complete development. Starting with a specification and ending with a verified implementation, these are properties that one would intuitively expect to have in an idealized development of a software component.

Given a specification for a software component, we can associate with it a set of implementations, which are correct with respect to the specification. We define another set, denoted by DV, of subsets of the set of implementations. The second result is Theorem 2, that the ordered pair (DV, $\subseteq$), where "$\subseteq$" is the set inclusion relation on the elements of DV, is a complete partial order. This result shows that (DV, $\subseteq$) has a well-understood structure, in addition to the more obvious "chain structure" of sets of implementations restricted to a particular development.

The third result is that correct and complete developments with respect to a given specification can be obtained from a correct development followed by a correct and complete development step. This shows the relationship between a correct development and a new development which is obtained from the original development by adding a correct and complete development step. The relationship is an immediate consequence of Theorem 4.

The fourth result of this section shows that the correctness of an implementation of a software component with respect to the original specification is maintained throughout the development process, provided that the development is correct. This is the result of the Corollary to Theorem 2.

The fifth result shows that development steps can be viewed as "building blocks" for developments. We obtain correct and complete developments by building them from correct development steps and a single correct and complete development step. This is the result of Theorem 5.

2.1. Preliminary Definitions

In the following, we mention the notion of an implementation being correct with respect to a specification. For the present we choose not to consider issues which arise in discussions of correctness; for example, partial versus total correctness. Also, we do not precisely define what we mean by a specification for a software component. We defer these matters until later when we discuss in more detail a specific method of software component specification and a specific example of this method. The basic idea is to investigate the properties of an abstract model for stepwise development which are independent of the specification method and notion of correctness. We then use the abstract model to study an incremental development for a particular specification method and a particular notion of correctness. This enables us to distinguish between those properties which are characteristic of an incremental development and those properties which are intrinsic to a specific incremental development method. We give some basic definitions, most of which are standard, which are used in the construction of the abstract model.

Notation: Let

$$Bool = \{ \text{true, false} \}$$

SPEC = { $\mathcal{S}$ | $\mathcal{S}$ is a specification }

IMPL = { p | p is an implementation }.

Definition: Let f: SPEC × IMPL → *Bool* be defined as follows:

$$f(\mathcal{S}, p) = \begin{cases} \text{true} & \text{if p is correct with respect to } \mathcal{S} \\ \text{false} & \text{otherwise.} \end{cases}$$

Definition: Let $\mathcal{S}$ be an element of SPEC. Let C = { p ∈ IMPL | f($\mathcal{S}$, p) = true }.

It may be that C = ∅, that is, $\mathcal{S}$ is a specification for which there exists no correct implementation. This can occur if, for example, the specification $\mathcal{S}$ is inconsistent.

Definition: A *partial order* is a pair (P, R) where P is a set and R is a relation on P which is reflexive (for all a ∈ P, aRa), antisymmetric (for all a, b ∈ P, aRb and bRa implies that a = b), and transitive (for all a, b, c ∈ P, aRb and bRc implies that aRc).

Definition: Let (P, R) be a partial order and S a nonempty subset of P. S is called a *chain* if aRb or bRa (or both) holds for all a, b ∈ S. This simply means that the relation "R" restricted to S is total.

Definition: Let (P, R) be a partial order and S be a subset of P. Let a ∈ S. The element a is a least element of S if, for all b ∈ S, aRb.

We note that if a and b are each least elements of S it follows that aRb and bRa. Since the relation R is antisymmetric, a = b, so that a least element, if it exists, is unique.

Definition: Let (P, R) be a partial order and S a subset of P. An element a of P is an *upper bound* of S (in P) if bRa for all b ∈ S. An element a of P is the ***least upper bound*** (lub) of S if a is the least element of the set of upper bounds of S in P.

Definition: A partial order (P, R) is a *complete partial order*, denoted by *cpo*, if the following two conditions hold:

(1) The set P has a least element.

(2) For every chain S in P the least upper bound, lub S exists.

Theorem: Every partial order which contains a least element and contains only finite chains is a cpo.

2.2. The Construction of the Model

Notation: Let S be a set. By $\mathcal{P}(S)$ we mean the power set of S.

Lemma: If C_i is a set of implementations which are correct with respect to the specification $\mathcal{S}_i$ for some integer i, $i \geq 0$, then the ordered pair $(\mathcal{P}(C_i), \subseteq)$ is a partial order.

Proof: The relation "$\subseteq$" is reflexive since $a \in \mathcal{P}(C_i)$ implies that $a \subseteq a$. The relation "$\subseteq$" is antisymmetric since for all a, $b \in \mathcal{P}(C_i)$, if $a \subseteq b$ and $b \subseteq a$ then $a = b$. The relation "$\subseteq$" is transitive, since for all a, b, $c \in \mathcal{P}(C_i)$, if $a \subseteq b$ and $b \subseteq c$ then $a \subseteq c$.

2.3. Classes of Developments

In this section we introduce definitions of an abstract program, a development with respect to a specification, a correct development, a complete development, and an incomplete development. The formal definitions concerning a development and the various classifications of developments correspond to the intuitive notions of development and kinds of developments which occur in a stepwise development process.

Definition: An *abstract program* $\mathcal{A}$ is an ordered pair, $(\mathcal{S}, C)$, such that the first member of the pair, $\mathcal{S}$, is a specification, and the second member of the pair, C, is a set of implementations which are correct with respect to $\mathcal{S}$.

Definition: A *development* with respect to a specification $\mathcal{S}_0$ is an (n + 1)-tuple of abstract programs, $(\mathcal{A}_0, \mathcal{A}_1, ..., \mathcal{A}_n)$, for some nonnegative integer n such that for each i, $0 \leq i \leq n$, $\mathcal{A}_i = (\mathcal{S}_i,$

C_i).

In the discussion of developments, we shall usually write out the abstract programs explicitly as ordered pairs of specifications and sets of implementations, since it is the interaction of the components of the ordered pair rather than the abstract programs themselves which are of interest.

Notation: Let S be a set. By $|S|$ we mean the cardinality of S.

Definition: A development with respect to a specification S_0, $((S_0, C_0), (S_1, C_1), ..., (S_n, C_n))$ is *correct* if $C_{i+1} \subseteq C_i$, $0 \leq i < n$.

It is possible that a correct development with respect to a specification, S_0, will have a set of implementations, C_i, for which $C_i = \emptyset$. Then all C_j, for $j \geq i$, will also be equal to the empty set. These kinds of developments will not be of interest in themselves, since they do not lead to an implementation which is correct with respect to the original specification, S_0. What is needed is an additional property for developments, which will ensure that the sets of implementations associated with the specifications in the development will all be nonempty. There are two properties of developments which enable us to describe the kinds of developments that we wish to consider. These two properties are independent of the correctness of a development, but will be used only in conjunction with correct developments. A correct development, which is *complete*, is a development ending in a single implementation which is correct with respect to the specification with which it is associated. A correct development, which is *incomplete*, is a development which may extended (in a sense which will be made precise later) to form a correct and complete development. We define the notions of a complete development and an incomplete development more precisely.

Definition: A development with respect to a specification S_0, $((S_0, C_0), (S_1, C_1), ..., (S_n, C_n))$ is *complete* if $|C_n| = 1$.

Definition: A development with respect to a specification S_0, $((S_0, C_0), (S_1, C_1), ..., (S_n, C_n))$ is *incomplete* if $|C_n| > 1$.

2.4. Properties of Classes of Developments

In this section we show that correct developments and correct and complete developments have the properties that we would expect of developments which resulted in implementations which satisfy the original specification for a software component. The sets of implementations associated with all correct and complete developments with respect to a given specification have a particularly nice structure when ordered by the relation of set inclusion. The result is that the collection of all such sets of implementations with the order relation of set inclusion is a complete partial order.

Theorem 1: Let D be a correct and complete development, $((S_0, C_0), (S_1, C_1), ..., (S_n, C_n))$, with respect to the specification S_0. It follows that:

(1) for each integer i, $0 \leq i \leq n$, C_i is a subset of the set of all implementations which are correct with respect to the specification S_i

(2) $C_{i+1} \subseteq C_i$, $0 \leq i < n$

(3) $|C_n| = 1$.

Proof: Property (1) follows from the fact that D is a development. Property (2) follows from the fact that D is correct, and property (3) holds because D is complete.

Corollary: Let D be a correct and complete development, $((S_0, C_0), (S_1, C_1), ..., (S_n, C_n))$, with respect to the specification S_0. It follows that for each i, $0 \leq i \leq n$, $C_i \neq \emptyset$.

Proof: Property (2) implies the nesting of all of the C_i's with respect to set inclusion, starting with C_0 down to C_n. Property (3) states that, at the last stage of the development, the set of all implementations which satisfy $\mathcal{S}_n$, that is, the most deeply nested of the C_i's, is a singleton set. A direct consequence of property (2) and property (3) is the following:

$$C_i \neq \emptyset \text{ for } 1 \leq i \leq n.$$

Corollary: Let D be a correct and incomplete development, $((\mathcal{S}_0, C_0), (\mathcal{S}_1, C_1), ..., (\mathcal{S}_n, C_n))$, with respect to the specification $\mathcal{S}_0$. It follows that for each i, $0 \leq i \leq n$, $C_i \neq \emptyset$.

Proof: The proof is similar to the previous corollary except that the most deeply nested of the C_i's is a set with cardinality greater that 1.

Theorem 2: Let $((\mathcal{S}_0, C_0), (\mathcal{S}_1, C_1), ..., (\mathcal{S}_n, C_n))$ be a correct development with respect to the specification $\mathcal{S}_0$. If $p \in C_i$ for some integer i, $0 \leq i \leq n$, then p is correct with respect to all specifications $\mathcal{S}_j$, $0 \leq j \leq i$.

Proof: If $p \in C_i$ for some integer i, $0 \leq i \leq n$, then $p \in C_j$ for all integers j, $0 \leq j < i$ from the definition of a correct development.

Corollary: Let $((\mathcal{S}_0, C_0), (\mathcal{S}_1, C_1), ..., (\mathcal{S}_n, C_n))$ be a correct and complete development with respect to the specification $\mathcal{S}_0$. If $p \in C_n$ then p is correct with respect to all specifications $\mathcal{S}_i$, $0 \leq i \leq n$.

Lemma 2: Let $((\mathcal{S}_0, C_0), (\mathcal{S}_1, C_1), ..., (\mathcal{S}_n, C_n))$ be a correct development with respect to a specification $\mathcal{S}_0$. Let $S = \{C_0, C_1, ..., C_n\}$. Then S is a finite chain in $(\mathcal{P}(C_0), \subseteq)$.

Proof: Clearly, S is finite and the order relation "$\subseteq$" restricted to the set S is total.

Definition: We call the set S a ***finite chain associated with the correct development.***

Notation: Let DV be the union of $\{\emptyset\}$ with the collection of all finite chains associated with all correct and complete developments with respect to a specification $\mathcal{S}_0$.

Theorem 3: The ordered pair (DV, $\subseteq$) is a cpo.

Proof: By the lemma of section 2.2, ($\mathcal{P}(C_0)$, $\subseteq$) is a partial order. With respect to the same order relation "$\subseteq$", but on the subset DV of $\mathcal{P}(C_0)$, (DV, $\subseteq$) is a partial order. The least element of DV is Ø. By the preceding lemma and the fact that any correct and complete development with respect to the specification $\mathcal{S}_0$ is also a finite chain in (DV, $\subseteq$), (DV, $\subseteq$) contains only finite chains. By the theorem of section 2.1, (DV, $\subseteq$) is a cpo.

2.5. Classes of Development Steps

In this section we introduce the notion of a development step and several classifications of development steps. We classify development steps as correct, incomplete, and complete. Correct development steps are those development steps that have properties which make these steps suitable for use in constructing correct developments. Incomplete development steps are used in constructing all but the final step in a development, while a complete development step is used as the final step in the construction of a development. The notion of a development step is fundamental, since it is the concept which describes the result of a process of going from one abstract program to another.

Definition: A ***development step*** with respect to a specification $\mathcal{S}_i$ is an ordered pair of abstract programs, $(\mathcal{A}_i, \mathcal{A}_{i+1})$, such that $\mathcal{A}_i = (\mathcal{S}_i, C_i)$ and $\mathcal{A}_{i+1} = (\mathcal{S}_{i+1}, C_{i+1})$.

For the same reasons presented in the discussion of developments, in the discussion of development steps, we shall usually write out the abstract programs explicitly as ordered pairs of specifications and sets of implementations.

Definition: Let $((\mathcal{S}_0, C_0), (\mathcal{S}_1, C_1), ..., (\mathcal{S}_n, C_n))$ be a development with respect to a specification $\mathcal{S}_0$. Let $((\mathcal{S}_j, C_j), (\mathcal{S}_{j+1}, C_{j+1}))$ be a development step with respect to the specification $\mathcal{S}_j$. The development ***contains the development step*** if $j = i$ for some integer i, $0 \leq i \leq n - 1$, that is, the development step is $((S_i, C_i), (S_{i+1}, C_{i+1}))$, where (S_i, C_i) and (S_{i+1}, C_{i+1}) are successive members of the (n + 1)–tuple which is the development with respect to the specification S_0.

Definition: A development step with respect to a specification S_i for some nonnegative integer i, $((S_i, C_i), (S_{i+1}, C_{i+1}))$, is *correct* if the following hold:

(1) $C_i, C_{i+1} \neq \emptyset$

(2) $C_{i+1} \subseteq C_i$.

Definition: A development step with respect to a specification S_i, $((S_i, C_i),(S_{i+1}, C_{i+1}))$, is *complete* if $|C_{i+1}| = 1$.

Definition: A development step with respect to a specification S_i, $((S_i, C_i),(S_{i+1}, C_{i+1}))$, is *incomplete* if $|C_{i+1}| > 1$.

2.6. The Extension of Developments with Development Steps

The results in this section show that developments can be extended by development steps to form new developments. The resulting new developments have properties which depend upon the original developments and the development steps.

Lemma: Let D be a development, $((S_0, C_0), (S_1, C_1), ..., (S_n, C_n))$, with respect to the specification S_0. Suppose that $((S_n, C_n), (S_{n+1}, C_{n+1}))$ is a development step with respect to S_n. Let D_1 be the ordered (n + 2)–tuple, $((S_0, C_0), (S_1, C_1), ..., (S_{n+1}, C_{n+1}))$. Then D_1 is a development with respect to the specification S_0, which contains the given development step.

Proof: The ordered (n + 2)–tuple, D_1, is a development with respect to the specification, S_0, since it can be shown that

(1) for each i, $0 \leq i \leq n$, C_i is the set of all implementations which are correct with respect to the specification S_i

(2) C_{n+1} is the set of all implementations which are correct with respect to the specification

S_{n+1}.

Property (1) follows from the assumption that D is a development, while property (2) follows from the assumption that ((S_n, C_n), (S_{n+1}, C_{n+1})) is a development step with respect to S_n. The development D_1 is clearly a development which contains the given development step, ((S_n, C_n), (S_{n+1}, C_{n+1})).

Lemma: Let D be a correct development, ((S_0, C_0), (S_1, C_1), ..., (S_n, C_n)), with respect to the specification S_0. Suppose that ((S_n, C_n), (S_{n+1}, C_{n+1})) is a correct development step with respect to S_n. Then D_1, the ordered (n + 2)-tuple, ((S_0, C_0), (S_1, C_1), ..., (S_{n+1}, C_{n+1})), is a correct development with respect to the specification S_0, which contains the given development step.

Proof: From the preceding lemma, D_1 is a development with respect to the specification S_0 which contains the given development step. D_1 is a correct development since it can be shown that

(1) $C_{i+1} \subseteq C_i$, $0 \leq i \leq n$

(2) $C_n \subseteq C_{n+1}$.

Property (1) follows from the assumption that D is correct and property (2) follows from the assumption that the development step, ((S_n, C_n), (S_{n+1}, C_{n+1})), is a correct development step with respect to S_n.

Theorem 4: Let D be a correct development, ((S_0, C_0), (S_1, C_1), ..., (S_n, C_n)), with respect to the specification S_0. Suppose that ((S_n, C_n), (S_{n+1}, C_{n+1})) is a complete and correct development step with respect to S_n. Let D_1 be ((S_0, C_0), (S_1, C_1), ..., (S_{n+1}, C_{n+1})). D_1 is a correct and complete development with respect to the specification S_0, which contains the given development step.

Proof: From the preceding lemma, D_1 is a correct development with respect to the specification S_0 which contains the given development step. Because ((S_n, C_n), (S_{n+1}, C_{n+1})) is a complete development step with respect to S_n, it follows that $|C_{n+1}| = 1$. This shows that the development is complete.

Corollary: Let D be a correct and incomplete development, $((S_0, C_0), (S_1, C_1), ..., (S_n, C_n))$, with respect to the specification S_0. Suppose that $((S_n, C_n), (S_{n+1}, C_{n+1}))$ is a complete and correct development step with respect to S_n. Let D_1 be $((S_0, C_0), (S_1, C_1), ..., (S_{n+1}, C_{n+1}))$. D_1 is a correct and complete development with respect to the specification S_0, which contains the given development step.

2.7. The Construction of Developments from Development Steps

In this section we show that developments can be constructed from development steps. The properties of the resulting developments are dependent upon the properties of the development steps used in the construction of the developments.

Lemma: Let $((S_0, C_0), (S_1, C_1))$, $((S_1, C_1), (S_2, C_2))$, ..., $((S_{n-1}, C_{n-1}), (S_n, C_n))$ be a collection of n development steps with respect to the specifications S_0, S_1, ..., S_n respectively, for some positive integer n. Let $D = ((S_0, C_0), (S_1, C_1), ..., (S_n, C_n))$. Then D is a development with respect to the specification S_0.

Proof: This follows immediately from the definition of a development step.

Lemma: Let $((S_0, C_0), (S_1, C_1))$, $((S_1, C_1), (S_2, C_2))$, ..., $((S_{n-1}, C_{n-1}), (S_n, C_n))$ be a collection of n correct development steps with respect to the specifications S_0, S_1,..., S_n respectively, for some positive integer n. Let $D = ((S_0, C_0), (S_1, C_1), ..., (S_n, C_n))$. Then D is a correct development with respect to the specification S_0.

Proof: D is a development with respect to the specification S_0 from the preceding lemma. Since $((S_i, C_i), S_{i+1}, C_{i+1}))$ is a correct development step with respect to the specification S_i for each integer i, $0 \leq i < n$, $C_{i+1} \subseteq C_i$. It follows that D is a correct development.

Theorem 5: Let $((\mathcal{S}_0, C_0), (\mathcal{S}_1, C_1)), ((\mathcal{S}_1, C_1), (\mathcal{S}_2, C_2)), \ldots, ((\mathcal{S}_{n-1}, C_{n-1}), (\mathcal{S}_n, C_n))$ be a collection of n correct development steps with respect to the specifications $\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_n$ respectively, for some positive integer n. Furthermore, suppose that $((\mathcal{S}_{n-1}, C_{n-1}), (\mathcal{S}_n, C_n))$ is a complete development step. Let $D = ((\mathcal{S}_0, C_0), (\mathcal{S}_1, C_1), \ldots, (\mathcal{S}_n, C_n))$. Then D is a correct and complete development with respect to the specification $\mathcal{S}_0$.

Proof: From the preceding lemma, D is a correct development with respect to the specification $\mathcal{S}_0$. Since $((\mathcal{S}_{n-1}, C_{n-1}), (\mathcal{S}_n, C_n))$ is a complete development step, $|C_n| = 1$. It follows that D is a complete development.

3. An Example of a Formal Development

In this section we relate the abstract model to an example of a formal development. In this example an implementation is a *while-program*; that is, the programming language allows assignment statements, composed statements, conditional statements, and while statements. Correctness means partial correctness of a while–program with respect to specifications. This is an extension of the concept of partial correctness of a while–program with respect to pre– and post–conditions, in which the pre– and post–conditions are well–formed formulas from first order predicate logic. A specification is based upon the notion of pairs of pre– and post–conditions and the statements allowed by the while–programming language. An abstract program $\mathcal{A}$ is a pair ($\mathcal{S}$, C) for which C is a set of while–programs which are partially correct with respect to the specification $\mathcal{S}$.

In a formal development we have an (n + 1)–tuple of abstract programs, ($\mathcal{A}_0$, $\mathcal{A}_1$, ..., $\mathcal{A}_n$), such that $\mathcal{A}_i = (\mathcal{S}_i, C_i)$ for $0 \leq i \leq n$. For the first abstract program in the development $\mathcal{A}_0$, which is ($\mathcal{S}_0$, C_0), the specification $\mathcal{S}_0$ is the original specification and the set C_0 is a set of while–programs which are partially correct with respect to S_0. The final abstract program in the development is ($\mathcal{S}_n$, C_n) where $\mathcal{S}_n$ is a specification which specifies a single while–program. We call such a specification an ***annotated program***. The set C_n is a singleton set {W} where W is a while–program which is partially correct with respect to S_n.

From the abstract model the successsive pairs of abstract programs in a correct development must be related to one another in the sense that each of the successive pairs are correct development steps. In the example, we obtain constraints which ensure that the successive pairs of abstract programs are correct development steps and that the last abstract program in the development ($\mathcal{S}_n$, C_n) has the property that $|C_n| = 1$. These constraints are consequences of the definitions which we introduce and the properties of the Hoare calculus.

3.1. Preliminary Definitions

The following definitions and notation provide the basic framework used in the discussion of the example.

Definition: A set is *recursively enumerable* if there exists an algorithm which recognizes elements of the set but which may not terminate for elements not in the set.

Definition: The *logical symbols* are exactly the following:

the *connectives* $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\equiv$

the *equality symbol* $=$

the *existential quantifier* $\exists$ and the *universal quantifier* $\forall$

the four punctuation marks ., (,), and ,

the *variables* x, y, z, x_1, ..., x', ...

the *truth symbols true* and *false.*

Notation: The set of symbols in the language of first–order predicate logic which are to be variables is denoted by V. We assume that V is infinite but recursively enumerable.

Notation: The *extralogical symbols* are taken from two arbitrarily chosen sets, which are disjoint from one another as well as disjoint from the set of all logical symbols. These two sets are:

F, the set of *function symbols.*

P, the set of *predicate symbols.* We assume that the sets F and P are both recursively enumerable.

Definition: A *basis* for predicate logic is a pair B = (F, P) of sets of symbols, where F and P are understood to be the sets of function and predicate symbols previously described.

Definition: The set T_B of all *terms* of (first-order) predicate logic over a basis B = (F, P) is defined inductively by:

a) Every variable from V and every constant from F is a term.

b) If $t_1, ..., t_n$ are terms for $n \geq 1$ and $f \in F$ is an n-ary function symbol, then $f(t_1, ..., t_n)$ is a term.

Definition: (*Syntax of Predicate Logic*) The set WFF_B of all (*well-formed*) *formulas* of (first-order) predicate logic over a basis B = (F, P) is defined inductively by:

(a) The truth symbols *true* and *false* are formulas.

Every propositional constant from P is a formula.

If t_1 and t_2 are terms, then $t_1 = t_2$ is a formula.

If $t_1, ..., t_n$ for $n \geq 1$ are terms and $p \in P$ is an n-ary predicate symbol, then $p(t_1, ..., t_n)$ is a formula.

(b) If w is a formula, then $(\neg w)$ is a formula.

If w is a formula and x is a variable, then $(\forall x.w)$ and $(\exists x.w)$ are also formulas.

If w_1 and w_2 are formulas, then so are $(w_1 \wedge w_2)$, $(w_1 \vee w_2)$, $(w_1 \Rightarrow w_2)$, and $(w_1 \equiv w_2)$.

Notation: The set of all well-formed formulas which do not have any quantifier is denoted by QFF_B. A well-formed formula which does not have any quantifier is called *quantifier free*.

Definition: (*Interpretation*) Let B = (F, P) be a basis for predicate logic. An *interpretation* of B is a pair $I = (D, I_0)$, where D is a non-empty set (called the *domain* of I) and I_0 is a mapping which assigns

(1) To every constant $c \in F$ an element $I_0(c) \in D$;

(2) To every function symbol f $\in$ F of arity $n \geq 1$ a total function $\mathcal{I}_0(f)$: $D^n \rightarrow D$;

(3) To every propositional constant a $\in$ P an element $\mathcal{I}_0 \in$ Bool;

(4) To every predicate symbol p $\in$ P of arity $n \geq 1$ a predicate $\mathcal{I}_0(p)$: $D^n \rightarrow$ Bool.

Definition: A total function σ: V $\rightarrow$ D mapping variables to the domain D of some interpretation is called an *assignment* or *state*. The set of all assignments for some interpretation $\mathcal{I}$ is denoted by $\Sigma_{\mathcal{I}}$ or just by Σ.

Definition: (*Semantics of Predicate Logic*) Let $\mathcal{I} = (D, \mathcal{I}_0)$ be an interpretation for a basis B = (F, P). To $\mathcal{I}$ is associated a functional, also denoted by $\mathcal{I}$, which maps every term t $\in T_B$ to a function $\mathcal{I}(t)$: $\Sigma \rightarrow$ D and every formula w $\in WFF_B$ to a function $\mathcal{I}(w)$: $\Sigma \rightarrow$ Bool; the functions $\mathcal{I}(t)$ and $\mathcal{I}(w)$ are defined as follows:

Semantics of terms

(a) If c $\in$ F is a constant, then

$$\mathcal{I}(c)(\sigma) = \mathcal{I}_0(c) \text{ for all assignments } \sigma \in \Sigma.$$

If x $\in$ V is a variable, then

$$\mathcal{I}(x)(\sigma) = \sigma(x) \text{ for all assignments } \sigma \in \Sigma.$$

(b) If $t_1, \ldots, t_n$ for $n \geq 1$ are terms and f $\in$ F is an n–ary function symbol, then

$$\mathcal{I}(f(t_1,\ldots,t_n))(\sigma) = \mathcal{I}_0(f)(\mathcal{I}(t_1)(\sigma),\ldots,\mathcal{I}(t_n)(\sigma)) \text{ for all assignments } \sigma \in \Sigma.$$

Semantics of formulas

(a) $\mathcal{I}(\text{true})(\sigma)$ = true for all $\sigma \in \Sigma$.

$\mathcal{I}(\text{false})(\sigma)$ = false for all $\sigma \in \Sigma$. If a $\in$ P is a propositional constant, then

$$\mathcal{I}(a)(\sigma) = \mathcal{I}_0(a) \text{ for all } \sigma \in \Sigma.$$

If t_1, t_2 are terms, then

$$\mathcal{I}(t_1 = t_2)(\sigma) = \begin{cases} \text{true} & \text{if } \mathcal{I}(t_1)(\sigma) = \mathcal{I}(t_2)(\sigma) \\ \text{false} & \text{otherwise, for all } \sigma \in \Sigma. \end{cases}$$

If $t_1, \ldots, t_n$ for $n \geq 1$ are terms and P is a n–ary predicate symbol, then

$$\mathcal{I}(p(t_{1,}\ldots,t_n))(\sigma) = \mathcal{I}_0(p)(\mathcal{I}(t_1)(\sigma),\ldots,\mathcal{I}(t_n)(\sigma)) \text{ for all } \sigma \in \Sigma.$$

(b) If $w \in WFF_B$ is a formula, then

$$\mathcal{I}((\neg w))(\sigma) = \begin{cases} \text{true} & \text{if } \mathcal{I}(w)(\sigma) = \text{false} \\ \text{false} & \text{otherwise, for all } \sigma \in \Sigma. \end{cases}$$

If w_1, $w_2 \in WFF_B$ then analogous statements hold for $(w_1 \wedge w_2)$, $(w_1 \vee w_2)$, $(w_1 \Rightarrow w_2)$, and $(w_1 \equiv w_2)$.

If $w \in WFF_B$ and $x \in V$, then

$$\mathcal{I}((\exists x.w))(\sigma) = \begin{cases} \text{true} & \text{if there exists } d \in D \text{ such that } \mathcal{I}(w)(\sigma[x/d]) = \text{true} \\ \text{false} & \text{otherwise, for all } \sigma \in \Sigma. \end{cases}$$

If $w \in WFF_B$ and $x \in V$, then

$$\mathcal{I}((\forall x.w))(\sigma) = \begin{cases} \text{true} & \text{if for all } d \in D\ \mathcal{I}(w)(\sigma[x/d]) = \text{true} \\ \text{false} & \text{otherwise, for all } \sigma \in \Sigma. \end{cases}$$

Definition: A formula w is called *valid* in an interpretation $\mathcal{I}$, denoted by $\models_{\mathcal{I}} w$, if $\mathcal{I}(w)(\sigma) =$ true for all assignments $\sigma \in \Sigma_{\mathcal{I}}$. The set of all formulas valid in $\mathcal{I}$ is denoted by $Th(\mathcal{I})$.

Definition: A formula w is called *logically valid*, denoted by $\models w$, if it is valid in all interpretations.

Definition: Let W be a subset of WFF_B of well-formed formulas of predicate logic. An interpretation $\mathcal{I}$ is called a *model* of W, if $\models_{\mathcal{I}} w$ for every formula $w \in W$. A formula $w \in WFF_B$ is called a *logical consequence* of W, denoted by $W \models w$, if $\models_{\mathcal{I}} w$ for every model $\mathcal{I}$ of W. The set of all logical consequences of W is denoted by Cn(W).

Definition: (*Calculi*) Let SO be some set of syntactic objects. A *calculus* or *axiomatic system* over SO is a pair $\mathcal{K} = (\mathcal{A}, \mathcal{R})$, where

$\mathcal{A}$ is a finite set of *axiom schemes*, which are decidable subsets of SO; the elements of an axiom scheme are called *axioms*

$\mathcal{R}$ is a finite set of *inference rules*, which are decidable subsets of $SO^n \times SO$, $n \geq 1$.

Definition: Let X be a (possibly empty) subset of a set SO of syntactic objects. The set of all syntactic objects which are *derivable* from X in calculus $\mathcal{K} = (\mathcal{A}, \mathcal{R})$ over SO is defined inductively by:

a) the basis set $X \cup \left(\bigcup_{A \in \mathcal{A}} A \right)$, and

b) the constructor set $\mathcal{R}$.

If a syntactic object s is derivable from a set of syntactic objects X in a calculus $\mathcal{K}$, we write $X \vdash_{\mathcal{K}} s$ or, $X \vdash s$. If X is the empty set, we write $\vdash s$. A construction sequence of s is called a *deduction* for s from X in $\mathcal{K}$.

Axiom schemes

(A1) $\neg w \vee w$ for all $w \in WFF_B$

(A2) $w_x^t \Rightarrow \exists x.w$ for all $w \in WFF_B$, $x \in V$, $t \in T_B$

(A3) $x = x$ for all $x \in V$

(A4) $x = y \Rightarrow y = x$ for all $x, y \in V$

(A5) $x = y \wedge y = z \Rightarrow x = z$ for all $x, y, z \in V$

(A6) $x_1 = y_1 \Rightarrow \ldots \Rightarrow x_n = y_n \Rightarrow p(x_1, \ldots, x_n) \Rightarrow p(y_1, \ldots, y_n)$ for all $x_1, \ldots, x_n, y_1, \ldots, y_n \in V$ for $n \geq 1$ and all n–ary predicates $p \in P$

(A7) $x_1 = y_1 \Rightarrow \ldots \Rightarrow x_n = y_n \Rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$ for all $x_1, \ldots, x_n, y_1, \ldots, y_n \in V$ for $n \geq 1$ and all n–ary function symbols $f \in F$

Inference rules

(R1) $\dfrac{w \lor w}{w}$ for all $w \in WFF_B$

(R2) $\dfrac{w_2}{w_1 \lor w_2}$ for all $w_1, w_2 \in WFF_B$

(R3) $\dfrac{w_1 \lor (w_2 \lor w_3)}{(w_1 \lor w_2) \lor w_3}$ for all $w_1, w_2, w_3 \in WFF_B$

(R4) $\dfrac{w_1 \Rightarrow w_2}{(\exists w.w_1) \Rightarrow w_2}$ for all $w_1, w_2 \in WFF_B$, $x \in V$, such that x is not free in w_2

(R5) $\dfrac{w_1 \lor w_2,\ \neg w_1 \lor w_3}{w_2 \lor w_3}$ for all $w_1, w_2, w_3 \in WFF_B$

3.2. The Construction of the Example

In this section we give a precise definition of the syntax of while–programs, the syntax of specifications in terms of pre– and post–conditions, an operational semantics for while–programs, partial correctness of a while–program with respect to a specification, and the syntax of annotated programs. The definition of partial correctness of a while–program with respect to a specification is an extension of the notion of partial correctness of a while–program with respect to formulas.

Definition: (*Syntax of* $\mathcal{L}_W$) The set, L_W^B, of ***while programs*** for the basis B is defined inductively as follows:

a) *Assignment statement* If x is a variable from V and t is a term from T_B, then

$$x := t$$

is a while program.

b) *Composed statement* If W_1, W_2 are while programs then

$$W_1 \,;\, W_2$$

is a while program.

c) *Conditional statement* If W_1, W_2 are while programs and e is a quantifier free formula from QFF_B, then

$$\textit{if}\ e\ \textit{then}\ W_1\ \textit{else}\ W_2\ \textit{fi}$$

is a while program.

d) *While statement* If W_1 is a while program and e is a quantifier free formula from QFF_B, then

$$\textit{while}\ e\ \textit{do}\ W_1\ \textit{od}$$

is a while program.

Definition: (*Syntax of* $\mathcal{L}_S$) The set, L_S^B, of *specifications*, for the basis B is defined inductively as follows:

a) *Unknown specification* If p, q are formulas from WFF_B, then

$$\{p\}\ \{q\}$$

is a specification.

b) *Assignment specification* If x is a variable from V, t is a term from T_B and p, q are formulas from WFF_B, then

$$\{p\}\ x := t\ \{q\}$$

is a specification.

c) *Composed specification* If S_1, S_2 are specifications and p, q are formulas from WFF_B, then

$$\{p\}\ S_1\ ;\ S_2\ \{q\}$$

is a specification.

d) *Conditional specification* If S_1, S_2 are specifications, e is a quantifier free formula from $\mathrm{QFF_B}$, and p, q are formulas from $\mathrm{WFF_B}$, then

$$\{p\}\ \textit{if}\ e\ \textit{then}\ S_1\ \textit{else}\ S_2\ \textit{fi}\ \{q\}$$

is a specification.

e) *While specification* If S_1 is a specification, e is a quantifier free formula from $\mathrm{QFF_B}$, and p, q are formulas from $\mathrm{WFF_B}$, then

$$\{p\}\ \textit{while}\ e\ \textit{do}\ S_1\ \textit{od}\ \{q\}$$

is a specification.

Definition: Let S be an arbitrary set of symbols. A sequence of symbols from S is called a *string* over S. A set of strings over S is called a *formal language* over S. The number of symbols in a finite string s is called its length. The sequence with no symbols, which has length 0, is called the *empty string* and is denoted by ϵ.

Definition: A *configuration* for a basis B and an interpretation I of B is a pair,

$$(W, \sigma) \in (L_W^B \cup \{\ \epsilon\ \}) \times \Sigma_I.$$

The first member of the ordered pair, W, represents the rest of the program to be executed, and σ represents the contents of the variables.

Definition: (*Transition Relation*) For every basis B and every interpretation I of B, the relation $\Rightarrow_c$ on the set of configurations $(L_W^B \cup \{\ \epsilon\ \}) \times \Sigma_I$ is defined by:

$(W_1, \sigma_1) \Rightarrow_c (W_2, \sigma_2)$ iff one of the following six conditions holds:

(1) There is a variable $x \in V$ and a term $t \in T_B$ such that

$$W_1 \text{ is } x := t\ ;\ W_2$$

and

$$\sigma_2 = \sigma_1[x/I(t)(\sigma_1)];$$

(2) There are while–programs W_1', W_2', W_3' and there is a quantifier free formula e such that

$$W_1 \text{ is } \textit{if}\ e\ \textit{then}\ W_1'\ \textit{else}\ W_2'\ \textit{fi}\ ;\ W_3'$$

and

$$W_2 \text{ is } \begin{cases} W_1'\ ;\ W_3' & \text{if } \mathcal{I}(e)(\sigma_1) = \text{true} \\ W_2'\ ;\ W_3' & \text{if } \mathcal{I}(e)(\sigma_1) = \text{false;} \end{cases}$$

(3) There are while–programs W_1', W_2' and there is a quantifier free formula e such that

$$W_1 \text{ is } \textit{while}\ e\ \textit{do}\ W_1'\ \textit{od}\ ;\ W_2'$$

$$\sigma_2 = \sigma_1$$

and

$$W_2 \text{ is } \begin{cases} W_1'\ ;\ W_1 & \text{if } \mathcal{I}(e)(\sigma_1) = \text{true} \\ W_2' & \text{if } \mathcal{I}(e)(\sigma_1) = \text{false;} \end{cases}$$

(4) There is a variable $x \in V$ and a term $t \in T_B$ such that

$$W_1 \text{ is } x := t$$

$$W_2 \text{ is } \epsilon$$

and

$$\sigma_2 = \sigma_1[x/\mathcal{I}(t)(\sigma_1)];$$

(5) and (6) are similar to (4) for (2) and (3).

Definition: A *computation sequence* for a state σ, which is called an *input state*, is a sequence of configurations

$$(W_1, \sigma_1), (W_2, \sigma_2), \ldots$$

such that $W_1 = W$, $\sigma_1 = \sigma$ and for every pair of consecutive configurations in the sequence

$$(W_i, \sigma_i) \Rightarrow_c (W_{i+1}, \sigma_{i+1})$$

for $i \geq 1$.

A computation sequence which is either infinite or ends with a configuration (W_k, σ_k) such that $W_k = \epsilon$ is called a *computation.*

A while–program W is said to terminate for an input state σ, if there is a finite computation

$$(W_1, \sigma_1), ..., (W_k, \sigma_k)$$

for this input state. The state σ_k is called the *output state.*

Definition: (*Operational Semantics of* $\mathcal{L}_W$) Let W be a while–program from L_W^B over the basis B and let $\mathcal{I}$ be an interpretation of this basis. The *meaning* of W (in the interpretation $\mathcal{I}$) is the function $\mathcal{M}_{\mathcal{I}}(W)$: $\Sigma \rightarrow_p \Sigma$, or $\mathcal{M}(W)$, defined by the following:

$$\mathcal{M}_{\mathcal{I}}(W) = \begin{cases} \sigma' & \text{if W terminates for the input state } \sigma \text{ with output state } \sigma'; \\ \text{undefined} & \text{if W does not terminate for the input state } \sigma. \end{cases}$$

Definition: (*Correctness with Respect to Formulas*) Let B be a basis for predicate logic, $\mathcal{I}$ an interpretation of this basis, and Σ the corresponding set of states. Let W be a while–program from L_W^B and let $\mathcal{M}_{\mathcal{I}}(W)$ be the meaning of the program W. Let p, q be formulas from WFF_B. The program W is ***partially correct with respect to p and q*** (in the interpretation $\mathcal{I}$) if for all states $\sigma \in \Sigma$ it follows that if $\mathcal{I}(p)(\sigma)$ = true and $\mathcal{M}_{\mathcal{I}}(W)(\sigma)$ is defined then $\mathcal{I}(q)(\mathcal{M}_{\mathcal{I}}(W))(\sigma)$ is true.

Definition: Let B, $\mathcal{I}$, Σ, W, p, q be as in the preceding definition. Then the formulas p and q are called the *pre–condition* and *post–condition*, respectively.

Definition: (*Correctness with Respect to Specifications*) Let W be a while–program from L_W^B. The notion that W is ***partially correct with respect to the specification*** $\mathcal{S}$ (in the interpretation $\mathcal{I}$) is defined inductively (the induction being on the specification, $\mathcal{S}$) as follows:

a) If $\mathcal{S}$ is an unknown specification,

$$\{p\} \{q\},$$

where p, q are formulas from WFF_B, then W is partially correct with respect to $\mathcal{S}$ if

(i) W is partially correct with respect to p and q (in the interpretation $\mathcal{I}$).

b) If $\mathcal{S}$ is an assignment specification,

$$\{p\}\ x := t\ \{q\},$$

where x is a variable from V, t is a term from T_B and p, q are formulas from WFF_B, then W is partially correct with respect to S if

(i) W is $x := t$.

(ii) W is partially correct with respect to p and q.

c) If S is a composed specification,

$$\{p\}\ S_1\ ;\ S_2\ \{q\},$$

where S_1, S_2 are specifications from L_S^B, and p, q are formulas from WFF_B, then W is partially correct with respect to S if

(i) W is W_1 ; W_2 for some $W_1, W_2 \in L_W^B$.

(ii) W is partially correct with respect to p and q.

(iii) W_1 is partially correct with respect to the specification S_1.

(iv) W_2 is partially correct with respect to the specification S_2.

d) If S is a conditional specification,

$$\{p\}\ \textit{if}\ e\ \textit{then}\ S_1\ \textit{else}\ S_2\ \textit{fi}\ \{q\},$$

where S_1, S_2 are specifications from L_S^B, e is a quantifier free formula from QFF_B, and p, q are formulas from WFF_B, then W is partially correct with respect to S if

(i) W is *if* e *then* W_1 *else* W_2 *fi* for some $W_1, W_2 \in L_W^B$.

(ii) W is partially correct with respect to p and q.

(iii) W_1 is partially correct with respect to the specification S_1.

(iv) W_2 is partially correct with respect to the specification S_2.

e) If S is a while specification,

$$\{p\}\ \textit{while}\ e\ \textit{do}\ S_1\ \textit{od}\ \{q\},$$

where S_1 is a specification from L_S^B, e is a quantifier free formula from QFF_B, and p, q are formulas from WFF_B, then W is partially correct with respect to S if

(i) W is *while* e *do* W_1 *od* for some $W_1 \in L_W^B$.

(ii) W is partially correct with respect to p and q.

(iii) W_1 is partially correct with respect to the specification S_1.

Definition: Let W, I, S, p and q be as in the preceding definition. Then the formulas p and q are called, respectively, the *pre–condition* and *post–condition associated with the specification S.*

If S is the unknown specification,

$$\{p\}\ \{q\},$$

then the pre– and post–conditions associated with S are p and q.

Definition: (*Syntax of* L_A) The set, L_A^B, of *annotated programs* for the basis B is defined inductively as follows:

a) *Assignment statement* If x is a variable from V , t is a term from T_B, and p, q are formulas from WFF_B, then

$$\{p\}\ x := t\ \{q\}$$

is an annotated program.

b) *Composed statement* If A_1, A_2 are annotated programs, and p, q are formulas from WFF_B, then

$$\{p\}\ A_1\ ;\ A_2\ \{q\}$$

is an annotated program.

c) *Conditional statement* If A_1, A_2 are annotated programs, p, q are formulas from WFF_B, and e is a quantifier free formula from QFF_B, then

$$\{p\}\ \textit{if}\ e\ \textit{then}\ A_1\ \textit{else}\ A_2\ \textit{fi}\ \{q\}$$

is an annotated program.

d) *While statement* If A_1 is an annotated program p, q are formulas from WFF_B, and e is a quantifier free formula from QFF_B, then

$$\{p\}\ \textit{while}\ e\ \textit{do}\ A_1\ \textit{od}\ \{q\}$$

is an annotated program.

We make a distinction in the preceding definitions between the sets of all while-programs, L_W^B, specifications, L_S^B, and annotated programs, L_A^B, and the corresponding sets along with an interpretation, which we denote by $\mathcal{L}_W$, $\mathcal{L}_S$, $\mathcal{L}_A$, respectively.

3.3. The Hoare Logic and Calculus

Given that an implementation is a while-program, a specification is in terms of pre- and post-conditions, and correctness is partial correctness of while-programs with respect to these specifications, it is necessary to have a logic and a calculus for a discussion of a formal development within this framework. The Hoare logic and calculus provide a natural means for reasoning about such a formal development. We give some basic definitions and state some results concerning Hoare logic and Hoare calculus which we use in later sections to discuss the example of a formal development. These are from [3]. For a survey of Hoare logic, see [1].

Definition: (*Syntax of Hoare Logic*) Let B be a basis for predicate logic. A *Hoare formula* over the basis B is an expression of the form

$$\{p\}\ W\ \{q\}$$

where p, $q \in WFF_B$ are formulas of the predicate logic and $W \in L_W^B$ is a while program.

Definition: (*Semantics of Hoare Logic*) Let an interpretation $\mathcal{I}$ of a basis B for predicate logic be given, and let Σ be the corresponding set of states. Every Hoare formula $\{p\}\ W\ \{q\} \in$

$\mathrm{HF_B}$ is mapped by a semantic functional, also denoted $\mathcal{I}$, to a function

$$\mathcal{I}(\{p\}\ W\ \{q\}) \colon \Sigma \rightarrow \mathit{Bool}$$

defined as follows:

$$\mathcal{I}(\{p\}\ W\ \{q\})(\sigma) = \text{true iff} \begin{cases} \text{if } \mathcal{I}(p)(\sigma) = \text{true} \\ \text{and if } \mathcal{M}(W)(\sigma) \text{ is defined,} \\ \text{then } \mathcal{I}(q)(\mathcal{M}_{\mathcal{I}}(W)(\sigma)) = \text{true.} \end{cases}$$

Definition: A Hoare formula, $\{p\}\ W\ \{q\}$, is said to be *valid* in an interpretation $\mathcal{I}$, denoted by

$$\models_{\mathcal{I}} \{p\}\ W\ \{q\}$$

if $\mathcal{I}(\{p\}\ W\ \{q\})(\sigma) = \text{true}$ for all states $\sigma \in \Sigma$.

We note that to say a Hoare formula, $\{p\}\ W\ \{q\}$, is valid in an interpretation $\mathcal{I}$ is a restatement within the context of Hoare logic of the fact that W is partially correct with respect to the formulas p and q in the interpretation $\mathcal{I}$. More generally, $\{p\}\ W\ \{q\}$ is valid in an interpretation $\mathcal{I}$ if and only if W is partially correct with respect to the unknown specification, $\{p\}\ \{q\}$.

Definition: A Hoare formula, $\{p\}\ W\ \{q\}$, is said to be *logically valid*, denoted by

$$\models \{p\}\ W\ \{q\}$$

if $\models_{\mathcal{I}} \{p\}\ W\ \{q\}$ for all interpretations $\mathcal{I}$.

Definition: A Hoare formula, $\{p\}\ W\ \{q\}$, is called a *logical consequence* of a set $F \subseteq \mathrm{WFF_B}$ of formulas of the predicate logic, denoted by

$$F \models \{p\}\ W\ \{q\}$$

if $\models_{\mathcal{I}} \{p\}\ W\ \{q\}$ holds for all models $\mathcal{I}$ of F.

Definition: The *Hoare calculus* (over a basis B for a predicate logic) is a calculus over the union of the set $\mathrm{HF_B}$ of Hoare formulas and the set $\mathrm{WFF_B}$ of formulas of the predicate logic and consists of an axiom (scheme) and five inference rules.

(i) *Assignment axiom*

$$\{p_x^t\}\ x := t\ \{p\}$$

for all $p \in WFF_B$, $x \in V$, $t \in T_B$

(ii) *Composition rule*

$$\frac{\{p\}\ W_1\ \{r\},\ \{r\}\ W_2\ \{q\}}{\{p\}\ W_1\ ;\ W_2\ \{q\}}$$

for all p, q, $r \in WFF_{B,}\ W_{1,}\ W_2 \in L_W^B$.

(iii) *Conditional rule*

$$\frac{\{p \land e\}\ W_1\ \{q\},\ \{p \land \neg e\}\ W_2\ \{q\}}{\{p\}\ \textit{if}\ e\ \textit{then}\ W_1\ \textit{else}\ W_2\ \textit{fi}\ \{q\}}$$

for all p, $q \in WFF_B$, $w \in QFF_B$, W_1, $W_2 \in L_W^B$.

(iv) *While rule*

$$\frac{\{p \land e\}\ W_1\ \{p\}}{\{p\}\ \textit{while}\ e\ \textit{do}\ W_1\ \textit{od}\ \{p \land \neg e\}}$$

for all $p \in WFF_B$, $e \in QFF_B$, $W_1 \in L_2^B$.

(v) *Consequence rule*

$$\frac{p \Rightarrow q,\ \{q\}\ W\ \{r\},\ r \Rightarrow s}{\{p\}\ W\ \{s\}}$$

for all p, q, r, $s \in WFF_B$, $W \in L_W^B$.

Lemma: (*Derived Rule*) For all p, $q \in WFF_B$, $x \in V$, and $t \in T_B$ it follows that:

$$\frac{p \Rightarrow q_x^t}{\{p\}\ x := t\ \{q\}}$$

The following theorem states that the formulas which are derivable in the Hoare calculus are logical consequences of subsets of WFF_B; that is, in an intuitive sense, the derivable formulas are true. In particular, this theorem holds when the subset of $\mathrm{WFF_B}$ is a theory.

Theorem: (*Soundness of the Hoare Logic*) Let B be a basis for predicate logic, let p, q $\in$ $\mathrm{WFF_B}$, and W $\in$ $\mathrm{L_W^B}$. Then for each subset F $\subseteq$ $\mathrm{WFF_B}$ and each Hoare formula {p} W {q} $\in$ $\mathrm{HF_B}$:

$$\text{if } F \vdash \{p\}\ W\ \{q\}, \text{ then } F \models \{p\}\ W\ \{q\}.$$

Lemma: (*Hoare Logic is a First-order Logic*) Let B be a basis for predicate logic and $\mathcal{I}$ an interpretation of B. Then for each Hoare formula h $\in$ $\mathrm{HF_B}$

$$\models_{\mathcal{I}} h \text{ iff } \mathrm{Th}(\mathcal{I}) \models h.$$

4. A Development

It follows from Theorem 5 that a complete and correct development can be obtained from a finite sequence of correct development steps, if the finite sequence meets the additional requirement that the final step in the development is complete. This reduces the problem of constructing a correct and complete development to the problem of constructing a finite sequence of correct development steps, the last step also being complete.

Within the framework of the Hoare calculus, we construct an example of a development as a finite sequence of abstract programs. Given a specification, S, we need to be able to associate with it a set of while–programs which are partially correct with respect to the given specification. This will enable us to construct an abstract program from the specification. We have the notion of partial correctness of a while–program with respect to a specification. We need, however, a notion in terms of a derivation within the Hoare calculus, which will imply partial correctness of a while–program with respect to a specification.

4.1. Derivations and Partial Correctness

In this section we present some preliminary results which show the connection between derivations from a theory of an interpretation in the Hoare calculus and partial correctness. The following lemma connects a Hoare formula which is derivable from the theory of an interpretation with the notion of partial correctness of unknown specifications.

Lemma: (*Derivations from a Theory and Valid Hoare Formulas*) Let B be a basis for predicate logic and $\mathcal{I}$ an interpretation of B. It follows that for each Hoare formula $h \in HF_B$, if $Th(\mathcal{I}) \vdash h$ then $\models_{\mathcal{I}} h$.

Proof: From the soundness of the Hoare calculus, if $Th(\mathcal{I}) \vdash h$, then $Th(\mathcal{I}) \models h$. From the lemma that Hoare logic is a first–order logic, it follows that $\models_{\mathcal{I}} h$.

As an immediate consequence of this lemma, we see that if there exists a derivation of the Hoare formula,

$$\{p\}\ W\ \{q\},$$

from the theory of an interpretation, then W is partially correct with respect to an unknown specification {p} {q}. This relationship between derivations from a theory and partial correctness is the result of the next lemma.

Lemma: (*Derivations from a Theory and Partial Correctness*) Let B be a basis for predicate logic and $\mathcal{I}$ an interpretation of B. Let $\mathcal{S}$ be the unknown statement specification,

$$\{p\}\ \{q\}.$$

It follows that for each Hoare formula $\{p\}\ W\ \{q\} \in \mathrm{HF}_B$, if $\mathrm{Th}(\mathcal{I}) \vdash \{p\}\ W\ \{q\}$, then W is partially correct with respect to the specification $\mathcal{S}$.

Proof: From the preceding lemma, it follows that $\models_{\mathcal{I}} \{p\}\ W\ \{q\}$. Therefore, W is partially correct with respect to the specification $\mathcal{S}$.

It is possible to associate with unknown specifications sets of while–programs, which are defined in terms of derivations within the Hoare calculus. These sets have the property that any element is a while–program which is partially correct with respect to the unknown specification with which it is associated. The following lemma constructs an abstract program from an unknown specification.

Lemma: Let $\mathcal{S}$ be the unknown specification,

$$\{p\}\ \{q\},$$

and let

$$C = \{\ W \in L_W^B \mid \mathrm{Th}(\mathcal{I}) \vdash \{p\}\ W\ \{q\}\ \}.$$

Then ($\mathcal{S}$, C) is an abstract program.

Proof: We need to show that for each W $\in$ C, W is partially correct with respect to S. This follows from the preceding lemma.

4.2. The Construction of an Abstract Program

In this section we introduce a definition which is an extension of the notion of the deduction of a Hoare formula from a theory. This definition is used to associate a set C of implementations with a specification S from L_S^B. This section also contains a theorem which shows that the pair, (S, C), is an abstract program. This extends a similar result for unknown specifications.

Definition: (*Deduction Consistent with a Specification*) Let B be a basis for predicate logic, W a while–program from L_W^B, I an interpretation of the basis B, S a specification from L_S^B, and p$'$, q$'$, respectively, the pre– and post–conditions associated with the specification S. The notion that there is a ***deduction from Th(I) to the Hoare formula*** {p$'$} W {q$'$} ***consistent with*** S, denoted by:

$$\mathrm{Th}(I) \vdash^{S} \{p'\}\ W\ \{q'\},$$

is defined inductively (the induction being on the specification, S) as follows:

a) If S is an unknown specification,

$$\{p'\}\ \{q'\},$$

then

$$\mathrm{Th}(I) \vdash^{S} \{p'\}\ W\ \{q'\}$$

if

(i) $\mathrm{Th}(I) \vdash \{p'\}\ W\ \{q'\}$.

b) If S is an assignment specification,

$$\{p'\}\ x := t\ \{q'\},$$

where x is a variable from V, t is a term from T_B then

$$\mathrm{Th}(\mathcal{I}) \vdash^{S} \{p'\}\ W\ \{q'\}$$

if

(i) W is $x := t$

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

c) If S is a composed specification,

$$\{p'\}\ S_1\ ;\ S_2\ \{q'\},$$

where S_1, S_2 are specifications from L_S^B, p_1, q_1 and p_2, q_2 are the pre– and post–conditions associated with S_1, and S_2, respectively, then

$$\mathrm{Th}(\mathcal{I}) \vdash^{S} \{p'\}\ W\ \{q'\}$$

if

(i) W is W_1 ; W_2 for some W_1, $W_2 \in L_W^B$

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$

(iii) $\mathrm{Th}(\mathcal{I}) \vdash^{S_1} \{p_1\}\ W_1\ \{q_1\}$

(iv) $\mathrm{Th}(\mathcal{I}) \vdash^{S_2} \{p_2\}\ W_2\ \{q_2\}$.

d) If S is a conditional specification,

$$\{p'\}\ \textit{if}\ e\ \textit{then}\ S_1\ \textit{else}\ S_2\ \textit{fi}\ \{q'\},$$

where S_1, S_2 are specifications from L_S^B, e is a quantifier free formula from QFF_B, p_1, q_1 and p_2, q_2 are the pre– and post–conditions associated with S_1, and S_2, respectively, then

$$\mathrm{Th}(\mathcal{I}) \vdash^{S} \{p'\}\ W\ \{q'\}$$

if

(i) W is *if* e *then* W_1 *else* W_2 *fi* for some W_1, $W_2 \in L_W^B$.

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$

(iii) $\mathrm{Th}(\mathcal{I}) \vdash^{S_1} \{p_1\}\ W_1\ \{q_1\}$

(iv) $\mathrm{Th}(\mathcal{I}) \vdash^{S_2} \{p_2\}\ W_2\ \{q_2\}$.

e) If S is a while specification,

$$\{p'\}\ \textit{while}\ e\ \textit{do}\ S_1\ \textit{od}\ \{q'\},$$

where S_1 is a specification from L_S^B, e is a quantifier free formula from QFF_B, and p_1, q_1 are the pre- and post-conditions associated with S_1, then

$$\mathrm{Th}(\mathcal{I}) \vdash^{S} \{p'\}\ W\ \{q'\}$$

if

(i) W is *while* e *do* W_1 *od* for some $W_1 \in L_W^B$.

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$

(iii) $\mathrm{Th}(\mathcal{I}) \vdash^{S_1} \{p_1\}\ W_1\ \{q_1\}$.

Lemma: Let $W \in L_W^B$, $S \in L_S^B$, and let p′, q′ be the pre- and post-conditions associated with S. If

$$\mathrm{Th}(\mathcal{I}) \vdash^{S} \{p'\}\ W\ \{q'\},$$

then W is partially correct with respect to the specification S.

Proof: This is an immediate consequence of the preceding definition, the definition of correctness with respect to specifications, and the lemma on derivations from a theory and partial correctness.

Note that in the case that S is the unknown specification,

$$\{p\}\ \{q\},$$

$\mathrm{Th}(\mathcal{I}) \vdash^{S} \{p\}\ W\ \{q\}$, reduces to $\mathrm{Th}(\mathcal{I}) \vdash \{p\}\ W\ \{q\}$.

Just as the notion of partial correctness with respect to specifications is an extension of the notion of partial correctness with respect to formulas, the notion of a deduction from a theory of an interpretation to a Hoare formula consistent with a specification is an extension of the notion of a deduction from a theory of an interpretation to a Hoare formula. From the preceding lemma, we have the connection between derivations consistent with specifications and partial correctness of while-programs with respect to specifications. We use the next theorem in the construction of abstract programs from specifications.

Theorem: Let $\mathcal{S} \in L^B_S$, and let p′, q′ be the pre- and post-conditions associated with $\mathcal{S}$. If C is

$$\{ W \in L^B_W \mid Th(\mathcal{I}) \vdash^{\mathcal{S}} \{p'\}\ W\ \{q'\} \},$$

then ($\mathcal{S}$, C) is an abstract program.

Proof: We need to show that for each W ∈ C, W is partially correct with respect to $\mathcal{S}$. This follows from the preceding lemma.

4.3. The Construction of a Development

We recall that a development with respect to a specification $\mathcal{S}_0$ is an (n + 1)-tuple of abstract programs, ($\mathcal{A}_0$, $\mathcal{A}_1$, ..., $\mathcal{A}_n$), for some nonnegative integer n such that for each i, $0 \leq i \leq n$, $\mathcal{A}_i = (\mathcal{S}_i, C_i)$. Let $\mathcal{S}_0$ be a given specification and let p′, q′ be the pre- and post-conditions associated with $\mathcal{S}_0$. We can form an abstract program $\mathcal{A}_0$ by defining C_0 to be

$$\{ W \in L^B_W \mid Th(\mathcal{I}) \vdash^{\mathcal{S}_0} \{p'\}\ W\ \{q'\} \}.$$

The fact that $\mathcal{A}_0$ is an abstract program is the main result of the preceding section. If the specification $\mathcal{S}_0$ is itself an annotated program, then $|C_0| = 1$ and $\mathcal{A}_0$ is a correct and complete development.

If $\mathcal{S}_0$ is not an annotated program, then it "contains" an unknown specification. We prove this fact in the course of constructing a correct development step. The notion that a specification contains an unknown specification will be defined precisely in the section on a correct development step. Assume that we have a way of constructing a correct development step; that is, from an abstract program $\mathcal{A}_0$, which is $(\mathcal{S}_0, C_0)$, we can construct a new abstract program $\mathcal{A}_1$, which is $(\mathcal{S}_1, C_1)$, such that $C_1 \subseteq C_0$. C_1 is defined to be

$$\{ W \in L_W^B \mid Th(\mathcal{I}) \vdash^{\mathcal{S}_1} \{p'\}\ W\ \{q'\} \}.$$

If $\mathcal{S}_1$ is an annotated program then $|C_1| = 1$. It follows that the pair, $(\mathcal{A}_0, \mathcal{A}_1)$, which is

$$((\mathcal{S}_0, C_0), (\mathcal{S}_1, C_1)),$$

is a correct and complete development.

In general, if we have an incomplete development, $(\mathcal{A}_0, \mathcal{A}_1, ..., \mathcal{A}_{i-1})$, then, assuming that we have a way of constructing a correct development step, we can construct $(\mathcal{A}_{i-1}, \mathcal{A}_i)$, where $\mathcal{A}_i$ is $(\mathcal{S}_i, C_i)$ and C_i is

$$\{ W \in L_W^B \mid Th(\mathcal{I}) \vdash^{\mathcal{S}_i} \{p'\}\ W\ \{q'\} \}.$$

If $\mathcal{S}_i$ is an annotated program, then $|C_i| = 1$ and $(\mathcal{A}_0, \mathcal{A}_1, ..., \mathcal{A}_i)$ is a correct and complete development; otherwise, we continue by constructing a new correct development step, $(\mathcal{A}_i, \mathcal{A}_{i+1})$. Since the abstract model describes an idealized development which always ends with an implementation which is correct with respect to the specification with which it is associated, by assumption, in the example of a development within the framework of the Hoare calculus, we restrict ourselves to a consideration of those cases for which there exists a nonnegative integer n, and abstract programs, $\mathcal{A}_0, \mathcal{A}_1, ..., \mathcal{A}_n$, such that $(\mathcal{A}_0, \mathcal{A}_1, ..., \mathcal{A}_n)$ is a correct and complete development. In short, the example we present gives an explicit construction of a development, which we prove to be correct and complete, under the assumption that a correct and complete development exists.

In the next section, we give a construction of correct development step. We define a specification transformation,

$$T: \mathcal{S}_i \rightarrow \mathcal{S}_{i+1},$$

and we give conditions under which it is possible to have a transformation, T, which preserves partial correctness. More precisely, we prove that for each nonnegative integer i, if there exists a suitable specification transformation,

$$T: \mathcal{S}_i \rightarrow \mathcal{S}_{i+1},$$

then $W \in C_i$ implies that $W \in C_{i+1}$. This result can be restated as follows: If we have a while–program which is partially correct with respect to $\mathcal{S}_i$ and several other constraints are satisfied concerning the transformation T, then this while–program is partially correct with respect to the new specification, $\mathcal{S}_{i+1}$.

5. A Correct Development Step

In the abstract model a development step with respect to a specification S is a pair of abstract programs $((S, C), (S', C'))$. In the example, it is necessary to precisely define a new abstract program,

$$(S', C'),$$

given an abstract program,

$$(S, C).$$

Given a specification, S, we define a transformation, T, from S to S'. We associate with S' a set of implementations, C', such that (S', C') is an abstract program. In order to define a specification transformation we need to define the notion that a specification "contains" an unknown specification. We also prove two theorems which depend upon this definition. The first theorem relates specifications and annotated programs. The second theorem is a result about the cardinality of the set of implementations which are partially correct with respect to a specification which is also an annotated program. The definition and theorems are in section 5.1. In section 5.2 we define a specification transformation,

$$T: S \rightarrow S',$$

for the special case in which S is an unknown specification. We introduce proof rules which are sufficient for the construction of a correct development step,

$$((S, C), (S', C')).$$

In section 5.3 we extend the definition of a specification transformation to include a larger class of specifications than the unknown specifications. In section 5.4 we extend the notion of proof rules to include this larger class of specifications.

It is also necessary to prove that each step in the development is correct for this larger class of specifications. In terms of the abstract model this involves proving that $C' \subseteq C$. In section 5.5

we show that under the generalized specification transformation,

$$T: \mathcal{S} \rightarrow \mathcal{S}',$$

the pair of abstract programs,

$$((\mathcal{S}, C), (\mathcal{S}', C')),$$

is a development step with the property that $C' \subseteq C$. In section 5.6 we prove that under the generalized specification transformation for which the proof rules hold, if there is a $W \in C$, then $W \in C'$. Thus, given the existence of an appropriate specification transformation for which the proof rules hold, it is possible to prove that a while–program, which is partially correct with respect to the specification $\mathcal{S}$, is also partially correct with respect to the transformed (and more detailed) specification $\mathcal{S}'$.

5.1. Specifications and Annotated Programs

In this section we lay the foundation for the construction of a correct development step. We formally define the notion that a specification "contains" an unknown statement specification. This formal definition corresponds to the meaning that one would intuitively expect for the idea that one specification contains another specification. We use this definition in the proofs which occur in the construction of a correct development step.

Definition: (*Syntax of* $\mathcal{L}_{\{p\}\,\{q\}}$) The set, $L^{B}_{\{p\}\,\{q\}}$, of ***specifications which contain the unknown specification***,

$$\{p\}\ \{q\},$$

for formulas p, q from WFF_B for the basis B is defined inductively, the induction being on a specification $\mathcal{S}$ which has pre– and post–conditions p′ and q′, respectively, as follows:

Basis

a) *Unknown statement specification* If $\mathcal{S}$ is the unknown specification,

$$\{p\}\ \{q\},$$

then S contains the unknown specification, $\{p\}\ \{q\}$.

Induction step

b) *Composed statement specification* Let S_1, S_2 be specifications from L_S^B, and suppose that either S_1 or S_2 contains the unknown specification, $\{p\}\ \{q\}$. If S is the composed statement specification,

$$\{p'\}\ S_1\ ;\ S_2\ \{q'\},$$

then S contains the unknown specification, $\{p\}\ \{q\}$.

c) *Conditional statement specification* Let S_1, S_2 be specifications from L_S^B, e a quantifier free formula from QFF_B, and suppose that either S_1 or S_2 contains the unknown specification, $\{p\}\ \{q\}$. If S is the conditional statement specification,

$$\{p'\}\ \textit{if}\ e\ \textit{then}\ ,S_1\ \textit{else}\ S_2\ \textit{fi}\ \{q'\}$$

then S contains the unknown specification, $\{p\}\ \{q\}$.

d) *While statement specification* Let S_1 be a specification from L_S^B, e be a quantifier free formula from QFF_B, and suppose that S_1 contains the unknown specification, $\{p\}\ \{q\}$. If S is the while statement specification,

$$\{p'\}\ \textit{while}\ e\ \textit{do}\ S_1\ \textit{od}\ \{q'\},$$

then S contains the unknown specification, $\{p\}\ \{q\}$.

The theorem which follows shows the relationship between specifications which do not contain unknown specifications and annotated programs.

Theorem: (*Specifications and Annotated Programs*) If $S \in L_S^B$ does not contain any unknown statement specification, then S is an annotated program.

Proof: The proof is by induction on the specification S.

a) If S is an assignment specification,

$$\{p\}\ x := t\ \{q\},$$

where x is a variable from V, t is a term from T_B and p, q are formulas from WFF_B, then S is an annotated program by definition.

b) If S is an composed specification,

$$\{p\}\ S_1\ ;\ S_2\ \{q\},$$

where S_1 and S_2 are specifications, p, q are formulas from WFF_B, and S is a specification which does not contain any unknown statement specification, it follows that S_1 and S_2 also do not contain any unknown statement specification. By the induction hypothesis, S_1 and S_2 must be annotated programs. It follows by definition that S is an annotated program.

c) If S is a conditional specification,

$$\{p\}\ \textit{if}\ e\ \textit{then}\ S_1\ \textit{else}\ S_2\ \textit{fi}\ \{q\},$$

where S_1, S_2 are specifications, e is a quantifier free formula from QFF_B, p, q are formulas from WFF_B, and S is a specification which does not contain any unknown statement specification, it follows that S_1 and S_2 also do not contain any unknown statement specification. By the induction hypothesis, S_1 and S_2 must be annotated programs. It follows by definition that S is an annotated program.

d) If S is while specification,

$$\{p\}\ \textit{while}\ e\ \textit{do}\ S_1\ \textit{od}\ \{q\},$$

where S_1 is a specification, e is a quantifier free formula from QFF_B, p, q are formulas from $\mathrm{WFF_B}$, and S is a specification which does not contain any unknown statement specification, it follows that S_1 also does not contain any unknown statement specification. By the induction hypothesis, S_1 must be an annotated program. It follows by definition that S is an annotated program.

Intuitively, specifications which contain unknown specifications, may not fully specify programs. We can consider such specifications as being in some sense "incomplete." On the other hand, specifications which do not contain any unknown specifications are not "incomplete", but can be associated with a specific program. The following theorem makes these ideas precise.

Theorem: Let (S, C) be an abstract program. If S does not contain any unknown specification and $\mathrm{C} \neq \emptyset$, then $|\mathrm{C}| = 1$.

Proof: The proof is by induction on the specification S.

a) If S is an assignment statement specification,

$$\{\mathrm{p}\}\ x := t\ \{\mathrm{q}\},$$

where x is a variable from V, t is a term from $\mathrm{T_B}$ and p, q are formulas from $\mathrm{WFF_B}$, then

$$\mathrm{C} = \{\ x := t \in \mathrm{L^B_W} \mid \mathrm{Th}(\mathcal{I}) \vdash \{\mathrm{p}\}\ x := t\ \{\mathrm{q}\}\ \}.$$

Since $\mathrm{C} \neq \emptyset$, there exists a $\mathrm{W} \in \mathrm{C}$ and W is $x := t$. It follows that $|\mathrm{C}| = 1$.

b) If S is a composed specification,

$$\{\mathrm{p}\}\ S_1\ ;\ S_2\ \{\mathrm{q}\},$$

where S_1 and S_2 are specifications, p, $\mathrm{p_1}$, $\mathrm{p_2}$, q, $\mathrm{q_1}$, $\mathrm{q_2}$ are formulas from $\mathrm{WFF_B}$, $\mathrm{p_1}$, $\mathrm{q_1}$ are the pre– and post–conditions associated with S_1, $\mathrm{p_2}$, $\mathrm{q_2}$ are the pre– and post–conditions associated with S_2, p, q are the pre– and post–conditions associated with S, and S is a specification which does not contain any unknown statement specification, it follows that S_1 and S_2 also do not contain any unknown statement specification. Let

$$C_1 = \{\ W \in L_W^B \mid Th(\mathcal{I}) \vdash^{S_1} \{p_1\}\ W\ \{q_1\}\ \}.$$

Let

$$C_2 = \{\ W \in L_W^B \mid Th(\mathcal{I}) \vdash^{S_2} \{p_2\}\ W\ \{q_2\}\ \}.$$

Since $C \neq \emptyset$, both C_1 and $C_2 \neq \emptyset$. By the induction hypothesis, it follows that $|C_1| = 1$ and $|C_2| = 1$. If $W \in C$, then W is W_1 ; W_2 for $W_1 \in C_1$ and $W_2 \in C_2$. Therefore $|C| = 1$.

c) If S is a conditional specification,

$$\{p\}\ \textit{if}\ e\ \textit{then}\ S_1\ \textit{else}\ S_2\ \textit{fi}\ \{q\},$$

where S_1 and S_2 are specifications, e is a quantifier free formula from QFF_B, p, p_1, p_2, q, q_1, q_2 are formulas from WFF_B, p_1, q_1 are the pre– and post–conditions associated with S_1, p_2, q_2 are the pre– and post–conditions associated with S_2, p, q are the pre– and post–conditions associated with S, and S is a specification which does not contain any unknown statement specification, it follows that S_1 and S_2 also do not contain any unknown statement specification. Let

$$C_1 = \{\ W \in L_W^B \mid Th(\mathcal{I}) \vdash^{S_1} \{p_1\}\ W\ \{q_1\}\ \}.$$

Let

$$C_2 = \{\ W \in L_W^B \mid Th(\mathcal{I}) \vdash^{S_2} \{p_2\}\ W\ \{q_2\}\ \}.$$

Since $C \neq \emptyset$, both C_1 and $C_2 \neq \emptyset$. By the induction hypothesis, it follows that $|C_1| = 1$ and $|C_2| = 1$. If $W \in C$, then W is

$$\textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ W_1\ \textbf{\textit{else}}\ W_2\ \textbf{\textit{fi}}$$

for $W_1 \in C_1$ and $W_2 \in C_2$. Therefore $|C| = 1$.

d) If S is while specification,

$$\{p\}\ \textit{while}\ e\ \textit{do}\ S_1\ \textit{od}\ \{q\},$$

where $\mathcal{S}_1$ is a specification, e is a quantifier free formula from $\mathrm{QFF_B}$, p, p_1, q, q_1 are formulas from $\mathrm{WFF_B}$, p_1, q_1 are the pre- and post-conditions associated with $\mathcal{S}_1$, p, q are the pre- and post-conditions associated with $\mathcal{S}$, and $\mathcal{S}$ is a specification which does not contain any unknown statement specification, it follows that $\mathcal{S}_1$ also does not contain any unknown statement specification. Let

$$C_1 = \{ W \in L_W^B \mid \mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}_1} \{p_1\}\ W\ \{q_1\} \}.$$

Since $C \neq \emptyset$, $C_1 \neq \emptyset$. By the induction hypothesis, it follows that $|C_1| = 1$. If $W \in C$, then W is

while e *do* W_1 *od*

for $W_1 \in C_1$. Therefore $|C| = 1$.

5.2. A Special Case of a Correct Development Step

Initially, we consider a somewhat simplified situation in which we wish to construct a correct development step. Let us consider the ordered pair, ($\mathcal{S}$, C) for which $\mathcal{S}$ has the form,

$$\{p\}\ \{q\},$$

where p and q are formulas from $\mathrm{WFF_B}$. C is the set of while–programs, $W \in L_W^B$, for which there exists a deduction in the Hoare calculus from the theory of the interpretation of the predicate logic to the Hoare formula {p} W {q} consistent with $\mathcal{S}$; that is,

$$C = \{ W \in L_W^B \mid \mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}} \{p\}\ W\ \{q\} \}.$$

From the abstract program, ($\mathcal{S}$, C), we construct a new abstract program,

$$(\mathcal{S}', C'),$$

in which the specification, $\mathcal{S}'$, and the set of while–programs, C′, are related to $\mathcal{S}$ and C. The relationship involves the transformation of $\mathcal{S}$ by changing the unknown specification into a another specification. Using the notation of the abstract model, we have a transformation on the specifications,

$$T: S \rightarrow S'.$$

In terms of the example of the formal development the transformation can be expressed as

$$T: \{p\}\ \{q\} \rightarrow \{p\}\ S_1\ \{q\}$$

where $S_1 \in L_S^B$ is either an assignment statement specification, composed statement specification, conditional statement specification, or a while statement specification. We give a formal definition of these transformations in this section.

Let S' be $\{p\}\ S_1\ \{q\}$. C′ is a set of while-programs for which there exists a deduction in the Hoare calculus from the theory of the interpretation of the predicate logic to the Hoare formula $\{p\}\ W\ \{q\}$ consistent with S'; that is, C′ is

$$\{\ W \in L_W^B \mid Th(\mathcal{I}) \vdash^{S'} \{p\}\ W\ \{q\}\ \}.$$

We assume that both C and $C' \neq \emptyset$. This is an assumption that there exist while-programs which satisfy the specifications S and S'. Since we are constructing an example of an idealized development, these assumptions are reasonable restrictions on the specifications. There are four possibilities for C′, depending upon the four kinds of transformation from $\{p\}\ \{q\}$ to $\{p\}\ S_1\ \{q\}$. In this section we will introduce conditions under which it is possible to guarantee that a while-program $W \in L_W^B$ is in $C \cap C'$. As a consequence of these conditions being satisfied, for each transformation, T, and for each such while-program W, W is partially correct with respect to S' and S.

Definition: (*Specification Transformations -- special case*) A transformation, T, from a specification, S, which is an unknown statement specification, $\{p\}\ \{q\}$, where p, q are formulas from WFF_B, to another specification, S', which is the image under T, of S, is defined as follows:

a) *Assignment statement transformation* If x is a variable from V, and t is a term from T_B, then

$$T: \{p\}\ \{q\} \rightarrow \{p\}\ x := t\ \{q\}.$$

b) *Composed statement transformation* If p_1, p_2, q_1, q_2 are formulas from WFF_B, and $\{p_1\}$ $\{q_1\}$ and $\{p_2\}$ $\{q_2\}$ are specifications, then

$$T: \{p\}\ \{q\} \rightarrow \{p\}\ \{p_1\}\ \{q_1\}\ ;\ \{p_2\}\ \{q_2\}\ \{q\}.$$

c) *Conditional statement transformation* If p_1, p_2, q_1, q_2 are formulas from WFF_B, and $\{p_1\}$ $\{q_1\}$ and $\{p_2\}$ $\{q_2\}$ are specifications, and e is a quantifier free formula from QFF_B, then

$$T: \{p\}\ \{q\} \rightarrow \{p\}\ \textit{if}\ e\ \textit{then}\ \{p_1\}\ \{q_1\}\ \textit{else}\ \{p_2\}\ \{q_2\}\ \textit{fi}\ \{q\}.$$

d) *While statement transformation* If p_1, q_1 are formulas from WFF_B, $\{p_1\}$ $\{q_1\}$ is a specification, and e is a quantifier free formula from QFF_B, then

$$T: \{p\}\ \{q\} \rightarrow \{p\}\ \textit{while}\ e\ \textit{do}\ \{p_1\}\ \{q_1\}\ \textit{od}\ \{q\}.$$

We note that the pre- and post-conditions associated with both S and S' are p and q. Thus, the transformation,

$$T: S \rightarrow S',$$

preserves pre- and post-conditions.

The four lemmas which follow give conditions under which it is possible to have derivations of specific kinds of Hoare formulas. Each of these Hoare formulas is closely related to one of the four kinds of specification transformations. We call these conditions *proof rules*, since they are sufficient to guarantee the existence of derivations in the Hoare calculus which will lead to a correct development step.

Lemma: (*Assignment Statement Derivation*) Let T: $S \rightarrow S'$ be an assignment statement transformation,

$$T: \{p\}\ \{q\} \rightarrow \{p\}\ x := t\ \{q\}.$$

Let $W \in L^B_W$. Suppose that W is $x := t$ for some $x \in V$ and $t \in T_B$. Let p, q be formulas from WFF_B and let $\{p\}\ \{q\}$ be a specification from L^B_W. Furthermore, assume that there exists a derivation of the following formula from the theory of the interpretation $\mathcal{I}$:

a) $p \Rightarrow q^t_x$.

Then $W \in C \cap C'$.

Proof: We first prove that $W \in C$. Since $p \Rightarrow q^t_x$, it is a consequence of the derived rule that $\{p\}\ x := t\ \{q\}$; that is,

$$Th(\mathcal{I}) \vdash \{p\}\ x := t\ \{q\}.$$

Therefore $W \in C$.

If the following two conditions are satisfied

i) W is $x := t$

ii) $Th(\mathcal{I}) \vdash \{p\}\ W\ \{q\}$

then

$$Th(\mathcal{I}) \vdash^{S'} \{p\}\ W\ \{q\}$$

and $W \in C'$. Condition i) holds by assumption. Condition ii) is a consequence of a).

Definition: (***Assignment Statement Proof Rule -- special case***) Let T, W, x, t, p, q, $\mathcal{I}$, and condition a) be as in the preceding lemma. Then a) is called an ***assignment statement proof rule.***

The preceding lemma shows that partial correctness with respect to specifications is preserved by assignment statement transformations if the assignment statement proof rule holds.

Lemma: (*Composed Statement Derivation*) Let T: $S \rightarrow S'$ be a composed statement transformation,

$$T: \{p\}\ \{q\} \rightarrow \{p\}\ \{p_1\}\ \{q_1\}\ ;\ \{p_2\}\ \{q_2\}\ \{q\}.$$

Let W $\in L_W^B$. Suppose that W is

$$W_1\ ;\ W_2$$

for some W_1, $W_2 \in L_W^B$. Let p, p_1, p_2, q, q_1, q_2 be formulas from WFF_B, and $\{p\}\ \{q\}$, $\{p_1\}\ \{q_1\}$, and $\{p_2\}\ \{q_2\}$ be specifications from L_S^B. Furthermore, assume that there exists a derivation of the following formulas from the theory of the interpretation $\mathcal{I}$:

a) $p \Rightarrow p_1$

b) $q_1 \Rightarrow p_2$

c) $q_2 \Rightarrow q$

d) $\{p_1\}\ W_1\ \{q_1\}$ for some $W_1 \in L_W^B$

e) $\{p_2\}\ W_2\ \{q_2\}$ for some $W_2 \in L_W^B$.

Then $W \in C \cap C'$.

Proof: From the formulas $q_1 \Rightarrow q_1$, $p \Rightarrow p_1$, and $\{p\}\ W_1\ \{q\}$, it follows from rule (v) that $\{p\}\ W_1\ \{q_1\}$. Similarly, from $q_1 \Rightarrow p_2$, $q_2 \Rightarrow q$, and $\{p_2\}\ W_2\ \{q_2\}$, it follows from rule (v) that $\{q_1\}\ W_2\ \{q\}$. From $\{p\}\ W_1\ \{q_1\}$ and $\{q_1\}\ W_2\ \{q\}$ it follows from rule (ii) that $\{p\}\ W_1\ ;\ W_2\ \{q\}$; that is,

$$Th(\mathcal{I}) \vdash \{p\}\ W_1\ ;\ W_2\ \{q\}.$$

It follows that $W \in C$.

Let S_1 be $\{p_1\}\ \{q_1\}$ and S_2 be $\{p_2\}\ \{q_2\}$. If the following hold

i) W is $W_1\ ;\ W_2$ for some W_1, $W_2 \in L_W^B$

ii) $Th(\mathcal{I}) \vdash \{p\}\ W\ \{q\}$

iii) $\mathrm{Th}(\mathcal{I}) \vdash^{S_1} \{p_1\}\ W_1\ \{q_1\}$

iv) $\mathrm{Th}(\mathcal{I}) \vdash^{S_2} \{p_2\}\ W_2\ \{q_2\}$

then

$$\mathrm{Th}(\mathcal{I}) \vdash^{S'} \{p\}\ W\ \{q\}$$

and $W \in C'$. Condition i) holds by assumption. Condition ii) is a consequence of a) – e). Condition iii) follows from d) and the fact that

$$\mathrm{Th}(\mathcal{I}) \vdash^{S_1} \{p_1\}\ W_1\ \{q_1\}$$

is

$$\mathrm{Th}(\mathcal{I}) \vdash \{p_1\}\ W_1\ \{q_1\}.$$

Condition iv) follows from e) and the fact that

$$\mathrm{Th}(\mathcal{I}) \vdash^{S_2} \{p_2\}\ W_2\ \{q_2\}$$

is

$$\mathrm{Th}(\mathcal{I}) \vdash \{p_2\}\ W_2\ \{q_2\}.$$

Definition: (*Composed Statement Proof Rules -- special case*) Let T, W, W_1, W_2, p, p_1, p_2, q, q_1, q_2, $\mathcal{I}$, and conditions a) – e) be as in the preceding lemma. Then a) – e) are called *composed statement proof rules.*

The preceding lemma shows that partial correctness with respect to specifications is preserved by composed statement transformations if the composed statement proof rules hold.

Lemma: (*Conditional Statement Derivation*) Let T: $S \rightarrow S'$ be a conditional statement transformation,

$$T\colon \{p\}\ \{q\} \rightarrow \{p\}\ \textit{if}\ e\ \textit{then}\ \{p_1\}\ \{q_1\}\ \textit{else}\ \{p_2\}\ \{q_2\}\ \textit{fi}\ \{q\}.$$

Let $W \in L_W^B$. Suppose that W is

$$\textit{if}\ e\ \textit{then}\ W_1\ \textit{else}\ W_2\ \textit{fi}$$

for some quantifier free formula e from QFF_B and some W_1, $W_2 \in L^B_W$. Let p, p_1, p_2, q, q_1, q_2 be formulas from WFF_B, and let {p} {q}, {p_1} {q_1}, and {p_2} {q_2} be specifications from L^B_S. Furthermore, assume that there exists a derivation of the following formulas from the theory of the interpretation $\mathcal{I}$:

a) $p \wedge e \Rightarrow p_1$

b) $q_1 \Rightarrow q$

c) $p \wedge \neg e \Rightarrow p_2$

d) $q_2 \Rightarrow q$

e) $\{p_1\}\ W_1\ \{q_1\}$

f) $\{p_2\}\ W_2\ \{q_2\}$.

Then $W \in C \cap C'$.

Proof: Since $p \wedge \neg e \Rightarrow p_2$, $\{p_2\}\ W_2\ \{q_2\}$, and $q_2 \Rightarrow q$, it follows from rule (v) that $\{p \wedge \neg e\}$ $W_2\ \{q\}$. Similarly, $\{p \wedge e\}\ W_1\ \{q\}$ follows from $p \wedge e \Rightarrow p_1$, $\{p_1\}\ W_1\ \{q_1\}$, $q_1 \Rightarrow q$, and rule (v). Using the fact that $\{p \wedge e\}\ W_1\ \{q\}$ and $\{p \wedge \neg e\}\ W_2\ \{q\}$, it follows from rule (iii) that

$$\{p\}\ \textit{if e then}\ W_1\ \textit{else}\ W_2\ \{q\};$$

that is,

$$Th(\mathcal{I}) \vdash \{p\}\ \textit{if e then}\ W_1\ \textit{else}\ W_2\ \{q\}.$$

Therefore $W \in C$.

Let S_1 be {p_1} {q_1} and S_2 be {p_2} {q_2}. If the following four conditions hold

i) W is *if* e *then* W_1 *else* W_2 *fi* for some W_1, $W_2 \in L^B_W$

ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p\}\ W\ \{q\}$

iii) $\mathrm{Th}(\mathcal{I}) \vdash^{S_1} \{p_1\}\ W_1\ \{q_1\}$

iv) $\mathrm{Th}(\mathcal{I}) \vdash^{S_2} \{p_2\}\ W_2\ \{q_2\}$

then

$$\mathrm{Th}(\mathcal{I}) \vdash^{S'} \{p\}\ W\ \{q\}$$

and $W \in C'$. Condition i) holds by assumption. Condition ii) is a consequence of a) – f). Condition iii) follows from e) and the fact that

$$\mathrm{Th}(\mathcal{I}) \vdash^{S_1} \{p_1\}\ W_1\ \{q_1\}$$

is

$$\mathrm{Th}(\mathcal{I}) \vdash \{p_1\}\ W_1\ \{q_1\}.$$

Condition iv) follows from f) and the fact that

$$\mathrm{Th}(\mathcal{I}) \vdash^{S_2} \{p_2\}\ W_2\ \{q_2\}$$

is

$$\mathrm{Th}(\mathcal{I}) \vdash \{p_2\}\ W_2\ \{q_2\}.$$

Definition: (*Conditional Statement Proof Rules -- special case*) Let T, W, W_1, W_2, p, p_1, p_2, q, q_1, q_2, e, $\mathcal{I}$, and conditions a) – f) be as in the preceding lemma. Then a) – f) are called *conditional statement proof rules.*

The preceding lemma shows that partial correctness with respect to specifications is preserved by conditional statement transformations if the conditional statement proof rules hold.

Lemma: (*While Statement Derivation*) Let T: $S \rightarrow S'$ be a while statement transformation,

$$\mathrm{T}\colon \{p\}\ \{q\} \rightarrow \{p\}\ \textit{while}\ e\ \textit{do}\ \{p_1\}\ \{q_1\}\ \textit{od}\ \{q\}.$$

Let $W \in L_W^B$. Suppose that W is

$$\textit{while}\ \mathrm{e}\ \textit{do}\ \mathrm{W}_1\ \textit{od}$$

for some quantifier free formula e from QFF_B, and some $W_1 \in L^B_W$. Let p, p_1, q, q_1 be formulas from WFF_B, and let {p} {q}, and {p_1} {q_1} be specifications from L^B_S. Furthermore, assume that there exists a derivation of the following formulas from the theory of the interpretation $\mathcal{I}$:

a) $p \wedge \neg e \Rightarrow q$

b) $p \wedge e \Rightarrow p_1$

c) $q_1 \Rightarrow p$

d) $\{p_1\}\ W_1\ \{q_1\}$ for some $W_1 \in L^B_W$.

Then $W \in C \cap C'$.

Proof: We first prove that $W \in C$. From the formulas $p \wedge e \Rightarrow p_1$, $q_1 \Rightarrow p$, and $\{p_1\}\ W_1\ \{q_1\}$, it follows from rule (v) that $\{p \wedge e\}\ W_1\ \{p\}$. From $\{p \wedge e\}\ W_1\ \{p\}$ and rule (iii) we obtain

$$\{p\}\ \textit{while e do}\ W_1\ \textit{od}\ \{p \wedge \neg e\}.$$

Since $p \Rightarrow p$, $p \wedge \neg e \Rightarrow q$, and $\{p\}\ \textit{while e do}\ W_1\ \textit{od}\ \{p \wedge \neg e\}$, it follows from rule (v) that

$$\{p\}\ \textit{while e do}\ W_1\ \textit{od}\ \{q\}.$$

Therefore, we have

$$Th(\mathcal{I}) \vdash \{p\}\ \textit{while e do}\ W_1\ \textit{od}\ \{q\}.$$

It follows that $W \in C$.

Let S_1 be $\{p_1\}\ \{q_1\}$. If the following hold

i) W is *while* e *do* W_1 *od* for some $W_1 \in L^B_W$.

ii) $Th(\mathcal{I}) \vdash \{p\}\ W\ \{q\}$

iii) $\mathrm{Th}(\mathcal{I}) \vdash^{S_1} \{p_1\}\ W_1\ \{q_1\}$

then

$$\mathrm{Th}(\mathcal{I}) \vdash^{S'} \{p\}\ W\ \{q\}$$

and $W \in C'$. Condition i) holds by assumption. Condition iii) follows from d) and the fact that

$$\mathrm{Th}(\mathcal{I}) \vdash^{S_1} \{p_1\}\ W_1\ \{q_1\}$$

is

$$\mathrm{Th}(\mathcal{I}) \vdash \{p_1\}\ W_1\ \{q_1\}.$$

Condition ii) is a consequence of a) – d).

Definition: (*While Statement Proof Rules -- special case*) Let T, W, W_1, e, p, p_1, q, q_1, $\mathcal{I}$, and conditions a) – d) be as in the preceding lemma. Then a) – d) are called *while statement proof rules.*

The preceding lemma shows that partial correctness with respect to specifications is preserved by while statement transformations if the while statement proof rules hold.

Definition: (*Proof Rules -- special case*) The assignment statement, composed statement, conditional statement, and while statement proof rules are called *proof rules* (for the specification {p} {q}).

We combine the results of the lemmas of this section to obtain the following two theorems.

Theorem: (*Development Step -- special case*) Let S be an unknown statement specification, {p} {q}. Let T be any one of the four possible kinds of transformations,

$$T: S \rightarrow S',$$

such that S' is the specification, $\{p\}\ S_1\ \{q\}$, and S_1 is either an assignment statement specification, a composed statement specification, a conditional statement specification, or a while statement specification. Let C be

$$\{\ W \in L_W^B \mid Th(\mathcal{I}) \vdash^{S} \{p\}\ W\ \{q\}\ \}$$

and let C′ be

$$\{\ W \in L_W^B \mid Th(\mathcal{I}) \vdash^{S'} \{p\}\ W\ \{q\}\ \}.$$

Assume that $C \neq \emptyset$ and $C' \neq \emptyset$ and that the proof rules (for the specification {p} {q}) hold. Then (S, C) and (S', C′) are abstract programs and for some $W \in L_W^B$, $W \in C \cap C'$.

Proof: From the theorem of section 4.2 it follows that (S, C), (S', C′) are abstract programs. The fact that $W \in C \cap C'$ for some $W \in L_W^B$ follows from the four preceding lemmas.

Theorem: (*Development Step Correctness -- special case*) Let S be an unknown statement specification, {p} {q}. Let T be any one of the four possible kinds of transformations,

$$T\colon S \rightarrow S',$$

such that S' is the specification, {p} S_1 {q}, and S_1 is either an assignment statement specification, a composed statement specification, a conditional statement specification, or a while statement specification. Let C be

$$\{\ W \in L_W^B \mid Th(\mathcal{I}) \vdash^{S} \{p\}\ W\ \{q\}\ \}$$

and let C′ be

$$\{\ W \in L_W^B \mid Th(\mathcal{I}) \vdash^{S'} \{p\}\ W\ \{q\}\ \}.$$

Assume that $C' \subseteq C$, $C' \neq \emptyset$, and that the proof rules (for the specification {p} {q}) hold. Then (S, C) and (S', C′) are abstract programs and

$$((S,\ C),\ (S',\ C'))$$

is a correct development step.

Proof: From the preceding theorem (S, C) and (S', C′) are abstract programs. Since C′ is a subset of C and $C' \neq \emptyset$, the theorem follows from the definition of a correct development step.

In section 5.5 we prove that the existence of a specification transformation implies that $C' \subseteq C$, not only for the class of specification transformations which we consider in this section, but for a more general class of specification transformations.

5.3. Specification Transformations

Let $S \in L^{B}_{\{p\}\{q\}}$, so that S is a specification which contains the unknown specification $\{p\}\ \{q\}$. The specification transformations, which we define next, are transformations of specifications, which contain unknown specifications, to specifications. These are a generalization of the specification transformations defined for unknown specifications.

Definition: (*Specification Transformations -- general case*) Let $S \in L_{\{p\}\ \{q\}}$ and let p′, q′ be the pre– and post–conditions asssociated with S. A transformation, T, of the specification, S, which is a specification containing the unknown specification,

$$\{p\}\ \{q\},$$

where p, q are formulas from WFF_B, to another specification, S', which is the image of S under T is defined inductively as follows:

a) *Assignment statement transformation* Let x be a variable from V, and t a term from T_B.

(i) If S is the unknown specification,

$$\{p\}\ \{q\},$$

then S' is

$$\{p\}\ x := t\ \{q\}.$$

(ii) If S is a composed specification,

$$\{p'\}\ S_1\ ;\ S_2\ \{q'\},$$

for some specifications S_1, S_2 from L^{B}_{S}, then either S_1 or S_2 contains the unknown specification,

$$\{p\}\ \{q\}.$$

If S_1 contains $\{p\}\ \{q\}$, then by the induction hypothesis, there exists an assignment statement transformation,

$$T_1 \colon S_1 \rightarrow S_1'.$$

Define T as an extension of T_1 from S_1 to S as follows:

$$T \colon \{p'\}\ S_1\ ;\ S_2\ \{q'\} \rightarrow \{p'\}\ T_1(S_1)\ ;\ S_2\ \{q'\}.$$

If S_2 contains $\{p\}\ \{q\}$, then by the induction hypothesis there exists an assignment statement transformation T_2 on S_2. Let the transformation T on S be defined as the extension of the transformation T_2 on S_2 to S.

(iii) If S is a conditional specification,

$$\{p'\}\ \textit{if}\ e\ \textit{then}\ S_1\ \textit{else}\ S_2\ \textit{fi}\ \{q'\},$$

for some quantifier free formula e from QFF_B, and some specifications S_1, S_2 from L_S^B, then either S_1 or S_2 contains the unknown specification,

$$\{p\}\ \{q\}.$$

If S_1 contains $\{p\}\ \{q\}$, then by the induction hypothesis, there exists an assignment statement transformation,

$$T_1 \colon S_1 \rightarrow S_1'.$$

Define T as an extension of T_1 from S_1 to S as follows:

$$T \colon \{p'\}\ \textit{if}\ e\ \textit{then}\ S_1\ \textit{else}\ S_2\ \textit{fi}\ \{q'\} \rightarrow \{p'\}\ \textit{if}\ e\ \textit{then}\ T_1(S_1)\ \textit{else}\ S_2\ \textit{fi}\ \{q'\}.$$

If S_2 contains $\{p\}\ \{q\}$, then by the induction hypothesis there exists an assignment statement transformation T_2 on S_2. Let the transformation T on S be defined as the extension of the transformation T_2 on S_2 to S.

(iv) If S is a while specification,

$$\{p'\}\ \textit{while}\ e\ \textit{do}\ S_1\ \textit{od}\ \{q'\},$$

for some quantifier free formula e from QFF_B, and some specification S_1 from L_S^B, then S_1 contains the unknown specification,

$$\{p\}\ \{q\}.$$

By the induction hypothesis, there exists an assignment statement transformation,

$$T_1: S_1 \rightarrow S_1'.$$

Define T as an extension of T_1 from S_1 to S as follows:

$$T: \{p'\}\ \textit{while}\ e\ \textit{do}\ S_1\ \textit{od}\ \{q'\} \rightarrow \{p'\}\ \textit{while}\ e\ \textit{do}\ T_1(S_1)\ \textit{od}\ \{q'\}.$$

b) *Composed statement transformation* Let p_1, p_2, q_1, q_2 be formulas from WFF_B, and let $\{p_1\}\ \{q_1\}$ and $\{p_2\}\ \ \{q_2\}$ be specifications from L_S^B. This part is similar to part a) except that the basis for the induction is the composed statement transformation,

$$T: \{p\}\ \{q\} \rightarrow \{p\}\ \{p_1\}\ \{q_1\}\ ;\ \{p_2\}\ \{q_2\}\ \{q\}.$$

c) *Conditional statement transformation* Let p_1, p_2, q_1, q_2 be formulas from WFF_B, let $\{p_1\}$ $\{q_1\}$ and $\{p_2\}\ \ \{q_2\}$ be specifications from L_S^B, and let e_1 be a quantifier free formula from QFF_B. This part is similar to part a) except that the basis for the induction is the conditional statement transformation,

$$T: \{p\}\ \{q\} \rightarrow \{p\}\ \textit{if}\ e_1\ \textit{then}\ \{p_1\}\ \{q_1\}\ \textit{else}\ \{p_2\}\ \{q_2\}\ \textit{fi}\ \{q\}.$$

d) *While statement transformation* Let p_1, q_1 be formulas from WFF_B, let $\{p_1\}\ \{q_1\}$ be a specification from L_S^B, and let e_1 be a quantifier free formula from QFF_B. This part is similar to part a) except that the basis for the induction is the while statement transformation,

$$T: \{p\}\ \{q\} \rightarrow \{p\}\ \textit{while}\ e_1\ \textit{do}\ \{p_1\}\ \{q_1\}\ \textit{od}\ \{q\}.$$

The definition just given for specification transformations is not quite precise enough, since we really need a definition which defines a unique specification transformation for each occurrence of the unknown specification,

$$\{p\}\ \{q\},$$

in the specification S. One way to handle this is to distinguish between the occurrences of the unknown specification in S. For example, if there were n occurrences of the specification,

$$\{p\}\ \{q\},$$

label them $\{p_1\}\ \{q_1\}$, $\{p_2\}\ \{q_2\}$, ..., $\{p_n\}\ \{q_n\}$. For each i, $1 \leq i \leq n$, a specification transformation of S is defined using the preceding definition, where S contains a single occurrence of the unknown specification,

$$\{p_i\}\ \{q_i\}.$$

We note that if p' and q' are the pre– and post–conditions associated with the specification transformation,

$$T: S \rightarrow S',$$

then the pre– and post–conditions associated with S' are also p' and q'.

5.4. The General Case for Transformation Proof Rules

In this section we generalize the notion of proof rules for the unknown specification,

$$\{p\}\ \{q\},$$

to proof rules for specifications which contain the unknown specification $\{p\}\ \{q\}$. The definitions for each of the four kinds of specification transformations are inductive and all are very similar to one another. We include the definitions for proof rules for each kind of specification transformation for the sake of completeness.

Definition: (*Assignment Statement Proof Rule -- general case*) Let S be a specification from L_S^B with pre– and post–conditions p' and q'. Suppose that S contains the unknown specification,

$$\{p\}\ \{q\}.$$

The *assignment statement proof rule (for the specification* $\{p\}\ \{q\}$*) holds for* S is defined inductively, the induction being on the specification S.

Basis

a) If S is the unknown specification,

$$\{p\}\ \{q\},$$

then, if the assignment statement proof rule (for the specification $\{p\}\ \{q\}$) holds, the assignment proof rule holds for S.

Induction step

b) If S is the composed specification,

$$\{p'\}\ S_1\ ;\ S_2\ \{q'\},$$

for some specifications S_1, S_2 from L_S^B, then either S_1 or $S_2 \in L_{\{p\}\ \{q\}}$. Assume that $S_1 \in L_{\{p\}\ \{q\}}$. If there exists an assignment statement specification transformation,

$$T_1\text{: } S_1 \rightarrow S_1',$$

such that the assignment statement proof rule holds for S_1, then the assignment statement proof rule holds for S. If $S_2 \in L_{\{p\}\ \{q\}}$, then the definition is similar.

c) If S is the conditional specification,

$$\{p'\}\ \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ S_1\ \textbf{\textit{else}}\ S_2\ \textbf{\textit{fi}}\ \{q'\},$$

for some quantifier free formula e from QFF_B, and for some specifications S_1, S_2 from L_S^B, then either S_1 or $S_2 \in L_{\{p\}\ \{q\}}$. Assume that $S_1 \in L_{\{p\}\ \{q\}}$. If there exists an assignment statement specification transformation,

$$T_1\text{: } S_1 \rightarrow S_1',$$

such that the assignment statement proof rule holds for S_1, then the assignment statement proof rule holds for S. If $S_2 \in L_{\{p\}\ \{q\}}$, then the definition is similar.

d) If S is the while specification,

$$\{p'\}\ \textit{while}\ e\ \textit{do}\ S_1\ \textit{od}\ \{q'\},$$

for some quantifier free formula e from QFF_B, and for some specification S_1 from L_S^B, then $S_1 \in L_{\{p\}\ \{q\}}$. If there exists an assignment statement specification transformation,

$$T_1\colon S_1 \rightarrow S_1',$$

such that the assignment statement proof rule holds for S_1, then the assignment statement proof rule holds for S.

Definition: (*Composed Statement Proof Rules -- general case*) Let S be a specification from L_S^B with pre– and post–conditions p′ and q′. Suppose that S contains the unknown specification,

$$\{p\}\ \{q\}.$$

The ***composed statement proof rules (for the specification*** $\{p\}\ \{q\}$***) hold for*** S is defined inductively, the induction being on the specification S.

Basis

a) If S is the unknown specification,

$$\{p\}\ \{q\},$$

then, if the composed statement proof rules (for the specification $\{p\}\ \{q\}$) hold, the composed proof rules hold for S.

Induction step

b) If S is the composed specification,

$$\{p'\}\ S_1\ ;\ S_2\ \{q'\},$$

for some specifications S_1, S_2 from L_S^B, then either S_1 or $S_2 \in L_{\{p\}\ \{q\}}$. Assume that $S_1 \in L_{\{p\}\ \{q\}}$. If there exists an composed statement specification transformation,

$$T_1\colon S_1 \rightarrow S_1',$$

such that the composed statement proof rules hold for S_1, then the composed statement proof rules hold for S. If $S_2 \in L_{\{p\}\ \{q\}}$, then the definition is similar.

c) If S is the conditional specification,

$$\{p'\}\ \textit{if}\ e\ \textit{then}\ S_1\ \textit{else}\ S_2\ \textit{fi}\ \{q'\},$$

for some quantifier free formula e from QFF_B, and for some specifications S_1, S_2 from L_S^B, then either S_1 or $S_2 \in L_{\{p\}\ \{q\}}$. Assume that $S_1 \in L_{\{p\}\ \{q\}}$. If there exists an composed statement specification transformation,

$$T_1: S_1 \rightarrow S_1',$$

such that the composed statement proof rules hold for S_1, then the composed statement proof rules hold for S. If $S_2 \in L_{\{p\}\ \{q\}}$, then the definition is similar.

d) If S is the while specification,

$$\{p'\}\ \textit{while}\ e\ \textit{do}\ S_1\ \textit{od}\ \{q'\},$$

for some quantifier free formula e from QFF_B, and for some specification S_1 from L_S^B, then $S_1 \in L_{\{p\}\ \{q\}}$. If there exists an composed statement specification transformation,

$$T_1: S_1 \rightarrow S_1',$$

such that the composed statement proof rules hold for S_1, then the composed statement proof rules hold for S.

Definition: (*Conditional Statement Proof Rules — general case*) Let S be a specification from L_S^B with pre– and post–conditions p' and q'. Suppose that S contains the unknown specification,

$$\{p\}\ \{q\}.$$

The *conditional statement proof rules (for the specification* $\{p\}\ \{q\}$*) hold for* S is defined inductively, the induction being on the specification S.

Basis

a) If S is the unknown specification,

$$\{p\}\ \{q\},$$

then, if the conditional statement proof rules (for the specification $\{p\}\ \{q\}$) hold, the conditional proof rules hold for S.

Induction step

b) If S is the composed specification,

$$\{p'\}\ S_1\ ;\ S_2\ \{q'\},$$

for some specifications S_1, S_2 from L_S^B, then either S_1 or $S_2 \in L_{\{p\}\ \{q\}}$. Assume that $S_1 \in L_{\{p\}\ \{q\}}$. If there exists an conditional statement specification transformation,

$$T_1\colon S_1 \rightarrow S_1',$$

such that the conditional statement proof rules hold for S_1, then the conditional statement proof rules hold for S. If $S_2 \in L_{\{p\}\ \{q\}}$, then the definition is similar.

c) If S is the conditional specification,

$$\{p'\}\ \textit{if}\ e\ \textit{then}\ S_1\ \textit{else}\ S_2\ \textit{fi}\ \{q'\},$$

for some quantifier free formula e from QFF_B, and for some specifications S_1, S_2 from L_S^B, then either S_1 or $S_2 \in L_{\{p\}\ \{q\}}$. Assume that $S_1 \in L_{\{p\}\ \{q\}}$. If there exists an conditional statement specification transformation,

$$T_1\colon S_1 \rightarrow S_1',$$

such that the conditional statement proof rules hold for S_1, then the conditional statement proof rules hold for S. If $S_2 \in L_{\{p\}\ \{q\}}$, then the definition is similar.

d) If S is the while specification,

$$\{p'\}\ \textit{while}\ e\ \textit{do}\ S_1\ \textit{od}\ \{q'\},$$

for some quantifier free formula e from QFF_B, and for some specification S_1 from L_S^B, then $S_1 \in L_{\{p\}\,\{q\}}$. If there exists an conditional statement specification transformation,

$$T_1 \colon S_1 \rightarrow S_1',$$

such that the conditional statement proof rules hold for S_1, then the conditional statement proof rules hold for S.

Definition: (*While Statement Proof Rules -- general case*) Let S be a specification from L_S^B with pre- and post-conditions p′ and q′. Suppose that S contains the unknown specification,

$$\{p\}\ \{q\}.$$

The *while statement proof rules (for the specification* {p} {q}*) hold for* S is defined inductively, the induction being on the specification S.

Basis

a) If S is the unknown specification,

$$\{p\}\ \{q\},$$

then, if the while statement proof rules (for the specification {p} {q}) hold, the while proof rules hold for S.

Induction step

b) If S is the composed specification,

$$\{p'\}\ S_1\ ;\ S_2\ \{q'\},$$

for some specifications S_1, S_2 from L_S^B, then either S_1 or $S_2 \in L_{\{p\}\,\{q\}}$. Assume that $S_1 \in L_{\{p\}\,\{q\}}$. If there exists an while statement specification transformation,

$$T_1 \colon S_1 \rightarrow S_1',$$

such that the while statement proof rules hold for S_1, then the while statement proof rules hold for S. If $S_2 \in L_{\{p\}\,\{q\}}$, then the definition is similar.

c) If S is the conditional specification,

$$\{p'\}\ \textit{if}\ e\ \textit{then}\ S_1\ \textit{else}\ S_2\ \textit{fi}\ \{q'\},$$

for some quantifier free formula e from QFF_B, and for some specifications S_1, S_2 from L_S^B, then either S_1 or $S_2 \in L_{\{p\}\ \{q\}}$. Assume that $S_1 \in L_{\{p\}\ \{q\}}$. If there exists an while statement specification transformation,

$$T_1: S_1 \rightarrow S_1',$$

such that the while statement proof rules hold for S_1, then the while statement proof rules hold for S. If $S_2 \in L_{\{p\}\ \{q\}}$, then the definition is similar.

d) If S is the while specification,

$$\{p'\}\ \textit{while}\ e\ \textit{do}\ S_1\ \textit{od}\ \{q'\},$$

for some quantifier free formula e from QFF_B, and for some specification S_1 from L_S^B, then $S_1 \in L_{\{p\}\ \{q\}}$. If there exists an while statement specification transformation,

$$T_1: S_1 \rightarrow S_1',$$

such that the while statement proof rules hold for S_1, then the while statement proof rules hold for S.

Definition: (*Proof Rules -- general case*) Let S be a specification from L_S^B with pre– and post–conditions p' and q'. Suppose that S contains the unknown specification,

$$\{p\}\ \{q\}.$$

If there is an assignment statement (composed statement, conditional statement, while statement, respectively) transformation on S such that the assignment statement (composed statement, conditional statement, while statement, respectively) proof rules (for the specification $\{p\}$ $\{q\}$) hold for S, then the *proof rules (for the specification* $\{p\}$ $\{q\}$*) hold for* S.

5.5. Sets of Implementations Related by Set Inclusion

Let (S, C) be an abstract program and let (S', C$'$) be the the abstract program obtained from (S, C) by a specification transformation T from S to S'. In this section we prove that the sets C and C$'$ have the property that $C' \subseteq C$. This set inclusion relation on the implementations is one of the requirements for a correct development step in the abstract model. This set inclusion relation is an immediate consequence of the theorem which we prove in this section. The theorem requires four lemmas and each lemma depends upon the kind of specification transformation, T, which is used to transform S to S'. Even though the proof of each of the lemmas is rather involved due to the induction on the specifications, the basic idea for the proofs is simple. Each proof can be summarized as follows: Any while–program which is partially correct with respect to a given specification must also be partially correct with respect to a less detailed specification, which is consistent with the given specification.

Lemma: (*Correctness of Assignment Statement Implementations*) Let T be an assignment statement transformation of the specification S, which contains the unknown specification,

$$\{p\}\ \{q\},$$

where p, q are formulas from WFF_B. Let S' be the image of S under T and let p$'$, q$'$ be the pre– and post–conditions associated with S and S'. Let C be

$$\{\ W \in L_W^B \mid Th(\mathcal{I}) \vdash^{S} \{p'\}\ W\ \{q'\}\ \}$$

and let C$'$ be

$$\{\ W \in L_W^B \mid Th(\mathcal{I}) \vdash^{S'} \{p'\}\ W\ \{q'\}\ \}.$$

For each $W \in C'$, $W \in C$; that is, $C' \subseteq C$.

Proof: The proof is by induction on the specification S. Associated with the specific assignment statement transformation T is a variable $x \in V$, and a term t from T_B.

a) If S is the unknown specification,

$$\{p\}\ \{q\},$$

then S' is

$$\{p\}\ x := t\ \{q\}.$$

If W $\in$ C$'$, then

$$\mathrm{Th}(\mathcal{I}) \vdash^{S'} \{p\}\ W\ \{q\}.$$

It follows that

(i) W is $x := t$.

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p\}\ W\ \{q\}$.

Conditions (i) and (ii) imply that

$$\mathrm{Th}(\mathcal{I}) \vdash^{S} \{p\}\ W\ \{q\}$$

or W $\in$ C.

b) If S is a composed specification,

$$\{p'\}\ S_1\ ;\ S_2\ \{q'\},$$

for some specifications S_1, S_2 from L_S^B, then either S_1 or $S_2 \in L_{\{p\}\ \{q\}}$. If $S_1 \in L_{\{p\}\ \{q\}}$, then S' is

$$\{p'\}\ S_1'\ ;\ S_2\ \{q'\},$$

where S_1' is the specification which is the image of S_1 under an assignment statement transformation,

$$T_1\colon S_1 \rightarrow S_1'.$$

Let W $\in$ C$'$. Since

$$\mathrm{Th}(\mathcal{I}) \vdash^{S'} \{p'\}\ W\ \{q'\},$$

it follows that for some pre- and post-conditions p_1, q_1, and p_2, q_2 associated with S_1 and S_2, respectively, that

(i) W is W_1 ; W_2 for some W_1, $W_2 \in L^B_W$.

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

(iii) $\mathrm{Th}(\mathcal{I}) \vdash^{S_1'} \{p_1\}\ W_1\ \{q_1\}$.

(iv) $\mathrm{Th}(\mathcal{I}) \vdash^{S_2} \{p_2\}\ W_2\ \{q_2\}$.

Using the induction hypothesis, if $W_1 \in L^B_W$ satisfies (iii), then

(v) $\mathrm{Th}(\mathcal{I}) \vdash^{S_1} \{p_1\}\ W_1\ \{q_1\}$.

It follows from (i), (ii), (iv), and (v) that

$$\mathrm{Th}(\mathcal{I}) \vdash^{S} \{p'\}\ W\ \{q'\}$$

or $W \in C$. If we assume that $S_2 \in L_{\{p\}\ \{q\}}$, then the proof is similar.

c) If S is a conditional specification,

$$\{p'\}\ \textit{if}\ e\ \textit{then}\ S_1\ \textit{else}\ S_2\ \textit{fi}\ \{q'\},$$

for some quantifier free formula e from QFF_B, and some specifications S_1, S_2 from L^B_S, then either S_1 or $S_2 \in L_{\{p\}\ \{q\}}$. If $S_1 \in L_{\{p\}\ \{q\}}$, then S' is

$$\{p'\}\ \textit{if}\ e\ \textit{then}\ S_1'\ \textit{else}\ S_2\ \textit{fi}\ \{q'\},$$

where S_1' is the specification which is the image of S_1 under an assignment statement transformation,

$$T_1\colon S_1 \rightarrow S_1'.$$

Let $W \in C'$. Since

$$\mathrm{Th}(\mathcal{I}) \vdash^{S'} \{p'\}\ W\ \{q'\},$$

it follows that for some pre– and post–conditions p_1, q_1, and p_2, q_2 associated with S_1 and S_2, respectively, that

(i) W is *if* e *then* W_1 *else* W_2 *fi* for some W_1, $W_2 \in L^B_W$.

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

(iii) $\mathrm{Th}(\mathcal{I}) \vdash^{S_1'} \{p_1\}\ W_1\ \{q_1\}$.

(iv) $\mathrm{Th}(\mathcal{I}) \vdash^{S_2} \{p_2\}\ W_2\ \{q_2\}$.

Using the induction hypothesis, if $W_1 \in L_W^B$ satisfies (iii), then

(v) $\mathrm{Th}(\mathcal{I}) \vdash^{S_1} \{p_1\}\ W_1\ \{q_1\}$.

It follows from (i), (ii), (iv), and (v) that

$$\mathrm{Th}(\mathcal{I}) \vdash^{S} \{p'\}\ W\ \{q'\}$$

or $W \in C$. If we assume that $S_2 \in L_{\{p\}\{q\}}$, then the proof is similar.

d) If S is a while specification,

$$\{p'\}\ \textit{while}\ e\ \textit{do}\ S_1\ \textit{od}\ \{q'\},$$

for some specification $S_1 \in L_{\{p\}\{q\}}$, and some quantifier free formula e from QFF_B, then S' is

$$\{p'\}\ \textit{while}\ e\ \textit{do}\ S_1'\ \textit{od}\ \{q'\},$$

where S_1' is the specification which is the image of S_1 under an assignment statement transformation,

$$T_1 \colon S_1 \rightarrow S_1'.$$

Let $W \in C'$. Since

$$\mathrm{Th}(\mathcal{I}) \vdash^{S'} \{p'\}\ W\ \{q'\},$$

it follows that for some pre- and post-conditions p_1, q_1 associated with S_1 that

(i) W is *while* e *do* W_1 *od* for some $W_1 \in L_W^B$.

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

(iii) $\mathrm{Th}(\mathcal{I}) \vdash^{S_1'} \{p_1\}\ W_1\ \{q_1\}$.

Using the induction hypothesis, if $W_1 \in L_W^B$ satisfies (iii), then

(iv) $\mathrm{Th}(\mathcal{I}) \vdash^{S_1} \{p_1\}\ W_1\ \{q_1\}$.

It follows from (i), (ii), and (iv) that

$$\mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}} \{p'\}\ W\ \{q'\}$$

or $W \in C$.

Lemma: (*Correctness of Composed Statement Implementations*) Let T be an composed statement transformation of the specification $\mathcal{S}$, which contains the unknown specification,

$$\{p\}\ \{q\},$$

where p, q are formulas from $\mathrm{WFF_B}$. Let $\mathcal{S}'$ be the image of $\mathcal{S}$ under T and let p′, q′ be the pre– and post–conditions associated with $\mathcal{S}$ and $\mathcal{S}'$. Let C be

$$\{\ W \in L_W^B \mid \mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}} \{p'\}\ W\ \{q'\}\ \}$$

and let C′ be

$$\{\ W \in L_W^B \mid \mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}'} \{p'\}\ W\ \{q'\}\ \}.$$

For each $W \in C'$, $W \in C$; that is, $C' \subseteq C$.

Proof: The proof is by induction on the specification $\mathcal{S}$. Associated with the composed statement transformation T are formulas p_1, p_2, q_1, q_2 from $\mathrm{WFF_B}$, and the specifications, $\{p_1\}\ \{q_1\}$ and $\{p_2\}\ \{q_2\}$, from L_S^B.

a) If $\mathcal{S}$ is the unknown specification,

$$\{p\}\ \{q\},$$

then $\mathcal{S}'$ is

$$\{p\}\ \mathcal{S}_1\ ;\ \mathcal{S}_2\ \{q\},$$

where $\mathcal{S}_1$ is $\{p_1\}\ \{q_1\}$ and $\mathcal{S}_2$ is $\{p_2\}\ \{q_2\}$. If $W \in C'$, then

$$\mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}'} \{p\}\ W\ \{q\}.$$

It follows that

(i) W is W_1 ; W_2 for some $W_1, W_2 \in L_W^B$.

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p\}\ W\ \{q\}$.

(iii) $\mathrm{Th}(\mathcal{I}) \vdash^{S_1} \{p_1\}\ W_1\ \{q_1\}$.

(iv) $\mathrm{Th}(\mathcal{I}) \vdash^{S_2} \{p_2\}\ W_2\ \{q_2\}$.

As a consequence of (i) – (iv), it follows that

$$\mathrm{Th}(\mathcal{I}) \vdash^{S} \{p\}\ W\ \{q\}$$

or $W \in C$.

b) If S is a composed specification,

$$\{p'\}\ S_3\ ;\ S_4\ \{q'\},$$

for some specifications S_3, S_4 from L_S^B, then either S_3 or $S_4 \in L_{\{p\}\ \{q\}}$. If $S_3 \in L_{\{p\}\ \{q\}}$, then S' is

$$\{p'\}\ S_3'\ ;\ S_4\ \{q'\},$$

where S_3' is the specification which is the image of S_3 under a composed statement transformation,

$$T_3\colon S_3 \rightarrow S_3'.$$

Let $W \in C'$. Since

$$\mathrm{Th}(\mathcal{I}) \vdash^{S'} \{p'\}\ W\ \{q'\},$$

it follows that for some pre– and post–conditions p_3, q_3, and p_4, q_4 associated with S_3 and S_4, respectively,

(i) W is $W_3\ ;\ W_4$ for some W_3, $W_4 \in L_W^B$.

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

(iii) $\mathrm{Th}(\mathcal{I}) \vdash^{S_3'} \{p_3\}\ W_3\ \{q_3\}$.

(iv) $\mathrm{Th}(\mathcal{I}) \vdash^{S_4} \{p_4\}\ W_4\ \{q_4\}$.

Using the induction hypothesis, if $W_3 \in L_W^B$ satisfies (iii), then

(v) $\mathrm{Th}(\mathcal{I}) \vdash^{S_3} \{p_3\}\ W_3\ \{q_3\}$.

It follows from (i), (ii), (iv), and (v) that

$$\mathrm{Th}(\mathcal{I}) \vdash^{S} \{p'\}\ W\ \{q'\}.$$

or $W \in C$. If we assume that $S_4 \in L_{\{p\}\,\{q\}}$, then the proof is similar.

c) If S is a conditional specification,

$$\{p'\}\ \textit{if}\ e_1\ \textit{then}\ S_3\ \textit{else}\ S_4\ \textit{fi}\ \{q'\},$$

for some specifications S_3, S_4 from L^B_S, and some quantifier free formula e_1 from QFF_B, then either S_3 or $S_4 \in L_{\{p\}\,\{q\}}$. If $S_3 \in L_{\{p\}\,\{q\}}$, then S' is

$$\{p'\}\ \textit{if}\ e_1\ \textit{then}\ S_3{}'\ \textit{else}\ S_4\ \textit{fi}\ \{q'\},$$

where $S_3{}'$ is the specification which is the image of S_3 under the composed statement transformation,

$$T_3\colon S_3 \rightarrow S_3{}'.$$

Let $W \in C'$. Since

$$\mathrm{Th}(\mathcal{I}) \vdash^{S'} \{p'\}\ W\ \{q'\},$$

it follows that for some pre– and post–conditions p_3, q_3, and p_4, q_4 associated with S_3 and S_4, respectively,

(i) W is $\textit{if}\ e_1\ \textit{then}\ W_3\ \textit{else}\ W_4\ \textit{fi}$ for some W_3, $W_4 \in L^B_W$.

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

(iii) $\mathrm{Th}(\mathcal{I}) \vdash^{S_3{}'} \{p_3\}\ W_3\ \{q_3\}$.

(iv) $\mathrm{Th}(\mathcal{I}) \vdash^{S_4} \{p_4\}\ W_4\ \{q_4\}$.

Using the induction hypothesis, if $W_3 \in L^B_W$ satisfies (iii), then

(v) $\mathrm{Th}(\mathcal{I}) \vdash^{S_3} \{p_3\}\ W_3\ \{q_3\}$.

It follows from (i), (ii), (iv), and (v) that

$$\mathrm{Th}(\mathcal{I}) \vdash^{S} \{p'\}\ W\ \{q'\}.$$

or $W \in C$. If we assume that $S_4 \in L_{\{p\}\,\{q\}}$, then the proof is similar.

d) If S is a while specification,

$$\{p'\}\ \textit{while}\ e_1\ \textit{do}\ S_3\ \textit{od}\ \{q'\},$$

for some specification S_3 from L_S^B, and some quantifier free formula e_1 from QFF_B, then S' is

$$\{p'\}\ \textit{while}\ e_1\ \textit{do}\ S_3'\ \textit{od}\ \{q'\},$$

where S_3' is the specification which is the image of S_3 under the composed statement transformation,

$$T_3\colon S_3 \rightarrow S_3'.$$

Let $W \in C'$. Since

$$\mathrm{Th}(\mathcal{I}) \vdash^{S'} \{p'\}\ W\ \{q'\},$$

it follows that for pre- and post-conditions p_3, q_3 associated with S_3

(i) W is *while* e_1 *do* W_3 *od* for some $W_3 \in L_W^B$.

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

(iii) $\mathrm{Th}(\mathcal{I}) \vdash^{S_3'} \{p_3\}\ W_3\ \{q_3\}$.

Using the induction hypothesis, if $W_3 \in L_W^B$ satisfies (iii), then

(iv) $\mathrm{Th}(\mathcal{I}) \vdash^{S_3} \{p_3\}\ W_3\ \{q_3\}$.

It follows from (i), (ii), and (iv) that

$$\mathrm{Th}(\mathcal{I}) \vdash^{S} \{p'\}\ W\ \{q'\}.$$

or $W \in C$.

Lemma: (*Correctness of Conditional Statement Implementations*) Let T be a conditional statement transformation of the specification S, which contains the unknown specification,

$$\{p\}\ \{q\},$$

where p, q are formulas from WFF_B. Let S' be the image of S under T and let p′, q′ be the pre- and post-conditions associated with S and S'. Let C be

$$\{\ W \in L_W^B \mid Th(\mathcal{I}) \vdash^{S} \{p'\}\ W\ \{q'\}\ \}$$

and let C′ be

$$\{\ W \in L_W^B \mid Th(\mathcal{I}) \vdash^{S'} \{p'\}\ W\ \{q'\}\ \}.$$

For each $W \in C'$, $W \in C$; that is, $C' \subseteq C$.

Proof: The proof is by induction on the specification S. Associated with the conditional statement transformation T are the quantifier free formula e from QFF_B, the formulas p_1, p_2, q_1, q_2 from WFF_B, and the specifications, $\{p_1\}\ \{q_1\}$ and $\{p_2\}\ \{q_2\}$, from L_S^B. Let S_1 be $\{p_1\}\ \{q_1\}$ and let S_2 be $\{p_2\}\ \{q_2\}$.

a) If S is the unknown specification,

$$\{p\}\ \{q\},$$

then S' is

$$\{p\}\ \textit{if}\ e\ \textit{then}\ S_1\ \textit{else}\ S_2\ \textit{fi}\ \{q\}.$$

If $W \in C'$, then

$$Th(\mathcal{I}) \vdash^{S'} \{p\}\ W\ \{q\}.$$

It follows that

(i) W is *if* e *then* W_1 *else* W_2 *fi* for some W_1, $W_2 \in L_W^B$.

(ii) $Th(\mathcal{I}) \vdash \{p\}\ W\ \{q\}$.

(iii) $Th(\mathcal{I}) \vdash^{S_1} \{p_1\}\ W_1\ \{q_1\}$.

(iv) $Th(\mathcal{I}) \vdash^{S_2} \{p_2\}\ W_2\ \{q_2\}$.

As a consequence of (i) – (iv), it follows that

$$\mathrm{Th}(\mathcal{I}) \vdash^{S} \{p\}\ W\ \{q\}$$

or $W \in C$.

b) If S is a composed specification,

$$\{p'\}\ S_3\ ;\ S_4\ \{q'\},$$

for some specifications S_3, S_4 from L_S^B, then either S_3 or $S_4 \in L_{\{p\}\ \{q\}}$. If $S_4 \in L_{\{p\}\ \{q\}}$, then S' is

$$\{p'\}\ S_3\ ;\ S_4'\ \{q'\},$$

where S_4' is the specification which is the image of S_4 under the conditional statement transformation,

$$T_4\colon S_4 \rightarrow S_4'.$$

Let $W \in C'$. Since

$$\mathrm{Th}(\mathcal{I}) \vdash^{S'} \{p'\}\ W\ \{q'\},$$

it follows that for some pre- and post-conditions p_3, q_3, and p_4, q_4 associated with S_3 and S_4, respectively,

(i) W is $W_3\ ;\ W_4$ for some W_3, $W_4 \in L_W^B$.

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

(iii) $\mathrm{Th}(\mathcal{I}) \vdash^{S_3} \{p_3\}\ W_3\ \{q_3\}$.

(iv) $\mathrm{Th}(\mathcal{I}) \vdash^{S_4'} \{p_4\}\ W_4\ \{q_4\}$.

Using the induction hypothesis, if $W_4 \in L_W^B$ satisfies (iv), then

(v) $\mathrm{Th}(\mathcal{I}) \vdash^{S_4} \{p_4\}\ W_4\ \{q_4\}$.

It follows from (i), (ii), (iii), and (v) that

$$\mathrm{Th}(\mathcal{I}) \vdash^{S} \{p'\}\ W\ \{q'\}.$$

or $W \in C$. If we assume that $S_3 \in L_{\{p\}\ \{q\}}$, then the proof is similar.

c) If S is a conditional specification,

$$\{p'\}\ \textit{if}\ e_1\ \textit{then}\ S_3\ \textit{else}\ S_4\ \textit{fi}\ \{q'\},$$

for some specifications S_3, S_4 from L_S^B, and some quantifier free formula e_1 from QFF_B, then either S_3 or $S_4 \in L_{\{p\}\,\{q\}}$. If $S_4 \in L_{\{p\}\,\{q\}}$, then S' is

$$\{p'\}\ \textit{if}\ e_1\ \textit{then}\ S_3\ \textit{else}\ S_4'\ \textit{fi}\ \{q'\},$$

where S_4' is the specification which is the image of S_4 under the conditional statement transformation,

$$T_4\colon S_4 \rightarrow S_4'.$$

Let $W \in C'$. Since

$$Th(\mathcal{I}) \vdash^{S'} \{p'\}\ W\ \{q'\},$$

it follows that for some pre– and post–conditions p_3, q_3, and p_4, q_4 associated with S_3 and S_4, respectively,

(i) W is *if* e_1 *then* W_3 *else* W_4 *fi* for some W_3, $W_4 \in L_W^B$.

(ii) $Th(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

(iii) $Th(\mathcal{I}) \vdash^{S_3} \{p_3\}\ W_3\ \{q_3\}$.

(iv) $Th(\mathcal{I}) \vdash^{S_4'} \{p_4\}\ W_4\ \{q_4\}$.

Using the induction hypothesis, if $W_4 \in L_W^B$ satisfies (iv), then

(v) $Th(\mathcal{I}) \vdash^{S_4} \{p_4\}\ W_4\ \{q_4\}$.

It follows from (i), (ii), (iii), and (v) that

$$Th(\mathcal{I}) \vdash^{S} \{p'\}\ W\ \{q'\}.$$

or $W \in C$. If we assume that $S_3 \in L_{\{p\}\,\{q\}}$, then the proof is similar.

d) If S is a while specification,

$$\{p'\}\ \textit{while}\ e_1\ \textit{do}\ S_3\ \textit{od}\ \{q'\},$$

for some specification $\mathcal{S}_3$ from L_S^B, and some quantifier free formula e_1 from QFF_B, then $\mathcal{S}'$ is

$$\{p'\}\ \textit{while}\ e_1\ \textit{do}\ \mathcal{S}_3{}'\ \textit{od}\ \{q'\},$$

where $\mathcal{S}_3{}'$ is the specification which is the image of $\mathcal{S}_3$ under the conditional statement transformation,

$$T_3\colon \mathcal{S}_3 \rightarrow \mathcal{S}_3{}'.$$

Let $W \in C'$. Since

$$Th(\mathcal{I}) \vdash^{\mathcal{S}'} \{p'\}\ W\ \{q'\},$$

it follows that for pre- and post-conditions p_3, q_3 associated with $\mathcal{S}_3$

(i) W is *while* e_1 *do* W_3 *od* for some $W_3 \in L_W^B$.

(ii) $Th(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

(iii) $Th(\mathcal{I}) \vdash^{\mathcal{S}_3{}'} \{p_3\}\ W_3\ \{q_3\}$.

Using the induction hypothesis, if $W_3 \in L_W^B$ satisfies (iii), then

(iv) $Th(\mathcal{I}) \vdash^{\mathcal{S}_3} \{p_3\}\ W_3\ \{q_3\}$.

It follows from (i), (ii), and (iv) that

$$Th(\mathcal{I}) \vdash^{\mathcal{S}} \{p'\}\ W\ \{q'\}.$$

or $W \in C$.

Lemma: (*Correctness of While Statement Implementations*) Let T be a while statement transformation of the specification $\mathcal{S}$, which contains the unknown specification,

$$\{p\}\ \{q\},$$

where p, q are formulas from WFF_B. Let $\mathcal{S}'$ be the image of $\mathcal{S}$ under T and let p′, q′ be the pre- and post-conditions associated with $\mathcal{S}$ and $\mathcal{S}'$. Let C be

$$\{\ W \in L_W^B \mid Th(\mathcal{I}) \vdash^{\mathcal{S}} \{p'\}\ W\ \{q'\}\ \}$$

and let C′ be

$$\{ W \in L_W^B \mid Th(\mathcal{I}) \vdash^{S'} \{p'\}\ W\ \{q'\} \}.$$

For each $W \in C'$, $W \in C$; that is, $C' \subseteq C$.

Proof: The proof is by induction on the specification S. Associated with the specific while statement transformation T are formulas p_1, q_1 from WFF_B, a specification $\{p_1\}\ \{q_1\}$ from L_S^B, and a quantifier free formula e from QFF_B. Let S_1 be $\{p_1\}\ \{q_1\}$.

a) If S is the unknown specification,

$$\{p\}\ \{q\},$$

then S' is

$$\{p\}\ \textit{while}\ e\ \textit{do}\ S_1\ \textit{od}\ \{q\}.$$

If $W \in C'$, then

$$Th(\mathcal{I}) \vdash^{S'} \{p\}\ W\ \{q\}.$$

It follows that

(i) W is *while* e *do* W_1 *od* $\{q\}$ for some $W_1 \in L_W^B$.

(ii) $Th(\mathcal{I}) \vdash \{p\}\ W\ \{q\}$.

(iii) $Th(\mathcal{I}) \vdash^{S_1} \{p_1\}\ W_1\ \{q_1\}$.

As a consequence of (i) – (iii), it follows that

$$Th(\mathcal{I}) \vdash^{S} \{p\}\ W\ \{q\}$$

or $W \in C$.

b) If S is a composed specification,

$$\{p'\}\ S_3\ ;\ S_4\ \{q'\},$$

for some specifications S_3, S_4 from L_S^B, then either S_3 or $S_4 \in L_{\{p\}\ \{q\}}$. If $S_3 \in L_{\{p\}\ \{q\}}$, then S' is

$$\{p'\}\ S_3{}'\ ;\ S_4\ \{q'\},$$

where S_3' is the specification which is the image of S_3 under a while statement transformation,

$$T_3\colon S_3 \rightarrow S_3'.$$

Let $W \in C'$. Since

$$\mathrm{Th}(\mathcal{I}) \vdash^{S'} \{p'\}\ W\ \{q'\},$$

it follows that for some pre- and post-conditions p_3, q_3, and p_4, q_4 associated with S_3 and S_4, respectively,

(i) W is W_3 ; W_4 for some W_3, $W_4 \in L_W^B$.

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

(iii) $\mathrm{Th}(\mathcal{I}) \vdash^{S_3'} \{p_3\}\ W_3\ \{q_3\}$.

(iv) $\mathrm{Th}(\mathcal{I}) \vdash^{S_4} \{p_4\}\ W_4\ \{q_4\}$.

Using the induction hypothesis, if $W_3 \in L_W^B$ satisfies (iii), then

(v) $\mathrm{Th}(\mathcal{I}) \vdash^{S_3} \{p_3\}\ W_3\ \{q_4\}$.

It follows from (i), (ii), (iv), and (v) that

$$\mathrm{Th}(\mathcal{I}) \vdash^{S} \{p'\}\ W\ \{q'\}.$$

or $W \in C$. If we assume that $S_4 \in L_{\{p\}\,\{q\}}$, then the proof is similar.

c) If S is a conditional specification,

$$\{p'\}\ \textit{if}\ e_1\ \textit{then}\ S_3\ \textit{else}\ S_4\ \textit{fi}\ \{q'\},$$

for some specifications S_3, S_4 from L_S^B, and some quantifier free formula e_1 from QFF_B, then either S_3 or $S_4 \in L_{\{p\}\,\{q\}}$. If $S_3 \in L_{\{p\}\,\{q\}}$, then S' is

$$\{p'\}\ \textit{if}\ e_1\ \textit{then}\ S_3'\ \textit{else}\ S_4\ \textit{fi}\ \{q'\},$$

where S_3' is the specification which is the image of S_3 under a while statement transformation,

$$T_3\colon \mathcal{S}_3 \rightarrow \mathcal{S}_3'.$$

Let $W \in C'$. Since

$$Th(\mathcal{I}) \vdash^{\mathcal{S}'} \{p'\}\ W\ \{q'\},$$

it follows that for some pre– and post–conditions p_3, q_3, and p_4, q_4 associated with $\mathcal{S}_3$ and $\mathcal{S}_4$, respectively,

(i) W is *if* e_1 *then* W_3 *else* W_4 *fi* for some W_3, $W_4 \in L^B_W$.

(ii) $Th(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

(iii) $Th(\mathcal{I}) \vdash^{\mathcal{S}_3'} \{p_3\}\ W_3\ \{q_3\}$.

(iv) $Th(\mathcal{I}) \vdash^{\mathcal{S}_4} \{p_4\}\ W_4\ \{q_4\}$.

Using the induction hypothesis, if $W_3 \in L^B_W$ satisfies (iv), then

(v) $Th(\mathcal{I}) \vdash^{\mathcal{S}_3} \{p_3\}\ W_3\ \{q_3\}$.

It follows from (i), (ii), (iv), and (v) that

$$Th(\mathcal{I}) \vdash^{\mathcal{S}} \{p'\}\ W\ \{q'\}.$$

or $W \in C$. If we assume that $\mathcal{S}_4 \in L_{\{p\}\ \{q\}}$, then the proof is similar.

d) If $\mathcal{S}$ is a while specification,

$$\{p'\}\ \textit{while}\ e_1\ \textit{do}\ \mathcal{S}_3\ \textit{od}\ \{q'\},$$

for some specification $\mathcal{S}_3$ from L^B_S, and some quantifier free formula e_1 from QFF_B, then $\mathcal{S}'$ is

$$\{p'\}\ \textit{while}\ e_1\ \textit{do}\ \mathcal{S}_3'\ \textit{od}\ \{q'\},$$

where $\mathcal{S}_3'$ is the specification which is the image of $\mathcal{S}_3$ under a while statement transformation,

$$T_3\colon \mathcal{S}_3 \rightarrow \mathcal{S}_3'.$$

Let $W \in C'$. Since

$$\mathrm{Th}(\mathcal{I}) \vdash^{S'} \{p'\}\ W\ \{q'\},$$

it follows that for pre– and post–conditions p_3, q_3 associated with S_3

(i) W is *while* e_1 *do* W_3 *od* for some $W_3 \in L^B_W$.

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

(iii) $\mathrm{Th}(\mathcal{I}) \vdash^{S_3'} \{p_3\}\ W_3\ \{q_3\}$.

Using the induction hypothesis, if $W_3 \in L^B_W$ satisfies (iii), then

(iv) $\mathrm{Th}(\mathcal{I}) \vdash^{S_3} \{p_3\}\ W_3\ \{q_3\}$.

It follows from (i), (ii), and (iv) that

$$\mathrm{Th}(\mathcal{I}) \vdash^{S} \{p'\}\ W\ \{q'\}.$$

or $W \in C$.

Theorem: (*Transformations on Specifications Containing Unknown Specifications*) Let (S, C) be an abstract program. Assume that S is a specification from $L^B_{\{p\}\{q\}}$; that is, S contains the unknown specification,

$$\{p\}\ \{q\},$$

and that p', q' are the pre– and post–conditions associated with S. Let C be the set

$$\{\ W \in L^B_W \mid \mathrm{Th}(\mathcal{I}) \vdash^{S} \{p'\}\ W\ \{q'\}\ \}.$$

Let T be a transformation from S to S' which is either an assignment statement transformation, a composed statement transformation, a conditional statement transformation, or a while statement transformation. Let C′ be the set

$$\{\ W \in L^B_W \mid \mathrm{Th}(\mathcal{I}) \vdash^{S'} \{p'\}\ W\ \{q'\}\ \}.$$

For each $W \in C'$, $W \in C$; that is, $C' \subseteq C$.

Proof: The proof is an immediate consequence of the preceding four lemmas.

Theorem: (*Development Step -- general case*) Let (S, C) be an abstract program. Assume that S is a specification from $L^B_{\{p\}\{q\}}$; that is, S contains the unknown specification,

$$\{p\}\ \{q\},$$

and C is

$$\{\ W \in L^{B}_{W} \mid Th(\mathcal{I}) \vdash^{\mathcal{S}} \{p'\}\ W\ \{q'\}\ \}.$$

Let T be a transformation from $\mathcal{S}$ to $\mathcal{S}'$ which is either an assignment statement transformation, a composed statement transformation, a conditional statement transformation, or a while statement transformation. Let C$'$ be the set

$$\{\ W \in L^{B}_{W} \mid Th(\mathcal{I}) \vdash^{\mathcal{S}'} \{p'\}\ W\ \{q'\}\ \}.$$

Then ($\mathcal{S}'$, C$'$) is an abstract program and the pair of abstract programs,

$$((\mathcal{S},\ C),\ (\mathcal{S}',\ C')),$$

is a development step with the property that $C' \subseteq C$.

Proof: This is an immediate consequence of the theorem on the construction of a new abstract program and the preceding theorem.

Theorem: (*Development Step Correctness -- general case*) Let ($\mathcal{S}$, C) be an abstract program. Assume that $\mathcal{S}$ is a specification from $L^{B}_{\{p\}\{q\}}$; that is, $\mathcal{S}$ contains the unknown specification,

$$\{p\}\ \{q\},$$

and C is

$$\{\ W \in L^{B}_{W} \mid Th(\mathcal{I}) \vdash^{\mathcal{S}} \{p'\}\ W\ \{q'\}\ \}.$$

Let T be a transformation from $\mathcal{S}$ to $\mathcal{S}'$ which is either an assignment statement transformation, a composed statement transformation, a conditional statement transformation, or a while statement transformation. Let C$'$ be the set

$$\{\ W \in L^{B}_{W} \mid Th(\mathcal{I}) \vdash^{\mathcal{S}'} \{p'\}\ W\ \{q'\}\ \}.$$

and suppose that $C' \neq \emptyset$. Then ($\mathcal{S}'$, C$'$) is an abstract program and the pair of abstract programs,

$$((S, C), (S', C')),$$

is a correct development step.

Proof: This is an immediate consequence of the preceding theorem and the definition of a correct development step.

5.6. Obtaining an Implementation Using Proof Rules

In the preceding section, given an abstract program,

$$(S, C),$$

and a specification transformation,

$$T: S \rightarrow S',$$

we have a new abstract program,

$$(S', C'),$$

for which

$$((S, C), (S', C'))$$

is a development step with the property that $C' \subseteq C$. In order to use a development step in the development of a program we need to start with a $W \in C$, a transformation,

$$T: S \rightarrow S',$$

and conditions ,which when satisfied, guarantee that $W \in C$. This is the main result of this section. The conditions are the proof rules which are given in section 5.4. In general, given $W \in C$, $W \notin C'$ since the set inclusion relation from section 5.5 is $C' \subseteq C$. The main theorem that we prove in this section requires four lemmas, each lemma obtains the result for one of the four kinds of specification transformations.

Lemma: (*Assignment Statement Implementations -- general case*) Let T be an assignment statement transformation of the specification S, which contains the unknown specification,

$$\{p\} \{q\},$$

where p, q are formulas from $\mathrm{WFF_B}$. Let S' be the image of S under T and let p′, q′ be the pre– and post–conditions associated with S and S'. Associated with the specific assignment statement transformation T is a variable $x \in V$, and a term t from $\mathrm{T_B}$. Let C be

$$\{ W \in L^{B}_{W} \mid \mathrm{Th}(\mathcal{I}) \vdash^{S} \{p'\}\ \{q'\} \}$$

and let C′ be

$$\{ W \in L^{B}_{W} \mid \mathrm{Th}(\mathcal{I}) \vdash^{S'} \{p'\}\ \{q'\} \}$$

If $W \in C$ and the assignment statement proof rule holds for S, then $W \in C'$.

Proof: The proof is by induction on the specification S.

a) If S is the unknown specification,

$$\{p\}\ \{q\},$$

then S' is

$$\{p\}\ x := t\ \{q\}.$$

If

(i) W is $x := t$

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p\}\ x := t\ \{q\}$,

then $W \in C'$. Condition (i) follows from the fact that T is an assignment statement transformation from S to S'. Condition (ii) follows from the assumption that the assignment statement proof rule holds for S.

b) If S is a composed specification,

$$\{p'\}\ S_1\ ;\ S_2\ \{q'\},$$

for some specifications S_1, S_2 from L^{B}_{S}, then either S_1 or $S_2 \in L_{\{p\}\ \{q\}}$. Assume that $S_1 \in L_{\{p\}\ \{q\}}$. The specification S' is

$$\{p'\}\ T_1(S_1)\ ;\ S_2\ \{q'\},$$

where

$$T_1\colon S_1 \rightarrow S_1'$$

is an assignment statement transformation for which the assignment statement proof rule holds for S_1. Since $W \in C$, it follows that for some pre- and post-conditions p_1, q_1, and p_2, q_2 associated with S_1 and S_2, respectively, that

(i) W is W_1 ; W_2 for some $W_1, W_2 \in L^B_W$.

(ii) $Th(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

(iii) $Th(\mathcal{I}) \vdash^{S_1} \{p_1\}\ W_1\ \{q_1\}$.

(iv) $Th(\mathcal{I}) \vdash^{S_2} \{p_2\}\ W_2\ \{q_2\}$.

From (iii) and the induction hypothesis,

(v) $Th(\mathcal{I}) \vdash^{S_1'} \{p_1\}\ W_1\ \{q_1\}$.

It follows from (i), (ii), (iv), and (v) that

$$Th(\mathcal{I}) \vdash^{S'} \{p'\}\ W\ \{q'\}.$$

Therefore, $W \in C'$. If we assume that $S_2 \in L_{\{p\}\{q\}}$, then the proof is similar.

c) If S is a conditional specification,

$$\{p'\}\ \textit{if}\ e\ \textit{then}\ S_1\ \textit{else}\ S_2\ \textit{fi}\ \{q'\},$$

for some quantifier free formula e from QFF_B, and some specifications S_1, S_2 from L^B_S, then either S_1 or $S_2 \in L_{\{p\}\{q\}}$. If $S_1 \in L_{\{p\}\{q\}}$, then S' is

$$\{p'\}\ \textit{if}\ e\ \textit{then}\ T_1(S_1)\ \textit{else}\ S_2\ \textit{fi}\ \{q'\},$$

where

$$T_1\colon S_1 \rightarrow S_1'$$

is an assignment statement transformation for which the assignment statement proof rule holds for S_1. Since $W \in C$, it follows that for some pre- and post-conditions p_1, q_1, and

p_2, q_2 associated with $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively, that

(i) W is *if* e *then* W_1 *else* W_2 *fi* for some $W_1, W_2 \in L^B_W$.

(ii) $Th(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

(iii) $Th(\mathcal{I}) \vdash^{\mathcal{S}_1} \{p_1\}\ W_1\ \{q_1\}$.

(iv) $Th(\mathcal{I}) \vdash^{\mathcal{S}_2} \{p_2\}\ W_2\ \{q_2\}$.

From (iii) and the induction hypothesis,

(v) $Th(\mathcal{I}) \vdash^{\mathcal{S}_1'} \{p_1\}\ W_1\ \{q_1\}$.

It follows from (i), (ii), (iv), and (v) that

$$Th(\mathcal{I}) \vdash^{\mathcal{S}'} \{p'\}\ W\ \{q'\}.$$

Therefore, $W \in C'$. If we assume that $\mathcal{S}_2 \in L_{\{p\}\ \{q\}}$, then the proof is similar.

d) If $\mathcal{S}$ is a while specification,

$$\{p'\}\ \textit{while}\ e\ \textit{do}\ \mathcal{S}_1\ \textit{od}\ \{q'\},$$

for some specification $\mathcal{S}_1$ from L^B_S, and some quantifier free formula e from QFF_B, then $\mathcal{S}'$ is

$$\{p'\}\ \textit{while}\ e\ \textit{do}\ T_1(\mathcal{S}_1)\ \textit{od}\ \{q'\},$$

where

$$T_1\colon \mathcal{S}_1 \rightarrow \mathcal{S}_1'$$

is an assignment statement transformation for which the assignment statement proof rule holds for $\mathcal{S}_1$. Since $W \in C$, it follows that for some pre- and post-conditions p_1, q_1 associated with $\mathcal{S}_1$, that

(i) W' is *while* e *do* W_1 *od* for some $W_1 \in L^B_W$.

(ii) $Th(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

p_2, q_2 associated with S_1 and S_2, respectively, that

(i) W is *if* e *then* W_1 *else* W_2 *fi* for some $W_1, W_2 \in L_W^B$.

(ii) $Th(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

(iii) $Th(\mathcal{I}) \vdash^{S_1} \{p_1\}\ W_1\ \{q_1\}$.

(iv) $Th(\mathcal{I}) \vdash^{S_2} \{p_2\}\ W_2\ \{q_2\}$.

From (iii) and the induction hypothesis,

(v) $Th(\mathcal{I}) \vdash^{S_1'} \{p_1\}\ W_1\ \{q_1\}$.

It follows from (i), (ii), (iv), and (v) that

$$Th(\mathcal{I}) \vdash^{S'} \{p'\}\ W\ \{q'\}.$$

Therefore, $W \in C'$. If we assume that $S_2 \in L_{\{p\}\,\{q\}}$, then the proof is similar.

d) If S is a while specification,

$$\{p'\}\ \textit{while}\ e\ \textit{do}\ S_1\ \textit{od}\ \{q'\},$$

for some specification S_1 from L_S^B, and some quantifier free formula e from QFF_B, then S' is

$$\{p'\}\ \textit{while}\ e\ \textit{do}\ T_1(S_1)\ \textit{od}\ \{q'\},$$

where

$$T_1\colon S_1 \rightarrow S_1'$$

is an assignment statement transformation for which the assignment statement proof rule holds for S_1. Since $W \in C$, it follows that for some pre– and post–conditions p_1, q_1 associated with S_1, that

(i) W' is *while* e *do* W_1 *od* for some $W_1 \in L_W^B$.

(ii) $Th(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

(iii) $Th(\mathcal{I}) \vdash^{S_1} \{p_1\}\ W_1\ \{q_1\}$.

From (iii) and the induction hypothesis,

(iv) $\mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}_1'} \{p_1\}\ W_1\ \{q_1\}$.

It follows from (i), (ii), and (iv) that

$$\mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}'} \{p'\}\ W\ \{q'\}$$

or $W \in C'$.

Lemma: (*Composed Statement Implementations -- general case*) Let T be a composed statement transformation of the specification $\mathcal{S}$, which contains the unknown specification,

$$\{p\}\ \{q\},$$

where p, q are formulas from WFF_B. Let $\mathcal{S}'$ be the image of $\mathcal{S}$ under T and let p', q' be the pre- and post-conditions associated with $\mathcal{S}$ and $\mathcal{S}'$. Associated with the specific composed statement transformation, T, are formulas p_1, p_2, q_1, q_2 from WFF_B, and the specifications, $\{p_1\}\ \{q_1\}$ and $\{p_2\}\ \{q_2\}$, from L_S^B. Let C be

$$\{\ W \in L_W^B \mid \mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}} \{p'\}\ W\ \{q'\}\ \}$$

and let C' be

$$\{\ W \in L_W^B \mid \mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}'} \{p'\}\ W\ \{q'\}\ \}.$$

If $W \in C$ and the composed statement proof rules hold for $\mathcal{S}$, then $W \in C'$.

Proof: The proof is by induction on the specification $\mathcal{S}$.

a) If $\mathcal{S}$ is the unknown specification,

$$\{p\}\ \{q\},$$

then $\mathcal{S}'$ is

$$\{p\}\ \mathcal{S}_1\ ;\ \mathcal{S}_2\ \{q\},$$

where $\mathcal{S}_1$ is $\{p_1\}\ \{q_1\}$ and $\mathcal{S}_2$ is $\{p_2\}\ \{q_2\}$. If

(i) W is W_1 ; W_2 for some $W_1, W_2 \in L_W^B$

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p\}\ W\ \{q\}$

(iii) $\mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}_1} \{p_1\}\ W_1\ \{q_1\}$

(iv) $\mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}_2} \{p_2\}\ W_2\ \{q_2\}$,

then $W \in C'$. Condition (i) follows from the fact that T is a composed statement transformation from $\mathcal{S}$ to $\mathcal{S}'$. Conditions (ii) – (iv) are consequences of the composed statement proof rules.

b) If $\mathcal{S}$ is a composed specification,

$$\{p'\}\ \mathcal{S}_3\ ;\ \mathcal{S}_4\ \{q'\},$$

for some specifications $\mathcal{S}_3$, $\mathcal{S}_4$ from L^B_S, then either $\mathcal{S}_3$ or $\mathcal{S}_4 \in L_{\{p\}\,\{q\}}$. Assume that $\mathcal{S}_3 \in L_{\{p\}\,\{q\}}$. The specification $\mathcal{S}'$ is

$$\{p'\}\ T_3(\mathcal{S}_3)\ ;\ \mathcal{S}_4\ \{q'\},$$

where

$$T_3\colon \mathcal{S}_3 \rightarrow \mathcal{S}_3'$$

is a composed statement transformation for which the composed statement proof rules hold for $\mathcal{S}_3$. Since $W \in C$, it follows that for some pre– and post–conditions p_3, q_3 and p_4, q_4 associated with $\mathcal{S}_3$ and $\mathcal{S}_4$, respectively, that

(i) W is $W_3\ ;\ W_4$ for some $W_3, W_4 \in L^B_W$.

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

(iii) $\mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}_3} \{p_3\}\ W_3\ \{q_3\}$.

(iv) $\mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}_4} \{p_4\}\ W_4\ \{q_4\}$.

Using the induction hypothesis, it follows from (iii) that

(v) $\mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}_3'} \{p_3\}\ W_3\ \{q_3\}$.

It follows from conditions (i), (ii), (iv), and (v) that

$$\mathrm{Th}(\mathcal{I}) \vdash^{S'} \{p'\}\ W\ \{q'\}.$$

Therefore, $W \in C'$. If $S_4 \in L_{\{p\}\ \{q\}}$, then the proof is similar.

c) If S is a conditional specification,

$$\{p'\}\ \textit{if}\ e_1\ \textit{then}\ S_3\ \textit{else}\ S_4\ \textit{fi}\ \{q'\},$$

for some quantifier free formula e_1 from QFF_B, and for some specifications S_3, S_4 from L_S^B, then either S_3 or $S_4 \in L_{\{p\}\ \{q\}}$. Assume that $S_3 \in L_{\{p\}\ \{q\}}$. The specification S' is

$$\{p'\}\ \textit{if}\ e_1\ \textit{then}\ T_3(S_3)\ \textit{else}\ S_4\ \textit{fi}\ \{q'\},$$

where

$$T_3\colon S_3 \rightarrow S_3'$$

is a composed statement transformation for which the composed statement proof rules hold for S_3. Since $W \in C$, it follows that for some pre- and post-conditions p_3, q_3 and p_4, q_4 associated with S_3 and S_4, respectively, that

(i) W is $\textit{if}\ e_1\ \textit{then}\ W_3\ \textit{else}\ W_4\ \textit{fi}$ for some $W_3, W_4 \in L_W^B$.

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

(iii) $\mathrm{Th}(\mathcal{I}) \vdash^{S_3} \{p_3\}\ W_3\ \{q_3\}$.

(iv) $\mathrm{Th}(\mathcal{I}) \vdash^{S_4} \{p_4\}\ W_4\ \{q_4\}$.

Using the induction hypothesis, it follows from (iii) that

(v) $\mathrm{Th}(\mathcal{I}) \vdash^{S_3'} \{p_3\}\ W_3\ \{q_3\}$.

It follows from conditions (i), (ii), (iv), and (v) that

$$\mathrm{Th}(\mathcal{I}) \vdash^{S'} \{p'\}\ W\ \{q'\}.$$

Therefore, $W \in C'$. If $S_4 \in L_{\{p\}\ \{q\}}$, then the proof is similar.

d) If S is a while specification,

$$\{p'\}\ \textit{while}\ e_1\ \textit{do}\ S_3\ \textit{od}\ \{q'\},$$

and let C′ be

$$\{\ W \in L_W^B \mid Th(\mathcal{I}) \vdash^{\mathcal{S}'} \{p'\}\ W\ \{q'\}\ \}.$$

If W ∈ C and the conditional statement proof rules hold for $\mathcal{S}$, then W ∈ C′.

Proof: The proof is by induction on the specification $\mathcal{S}$.

a) If $\mathcal{S}$ is the unknown specification,

$$\{p\}\ \{q\},$$

then $\mathcal{S}'$ is

$$\{p\}\ \textit{if}\ e\ \textit{then}\ \mathcal{S}_1\ \textit{else}\ \mathcal{S}_2\ \textit{fi}\ \{q\},$$

where $\mathcal{S}_1$ is $\{p_1\}\ \{q_1\}$ and $\mathcal{S}_2$ is $\{p_2\}\ \{q_2\}$. If

(i) W is *if* e *then* W_1 *else* W_2 *fi* for some W_1, $W_2 \in L_W^B$

(ii) $Th(\mathcal{I}) \vdash \{p\}\ W\ \{q\}$

(iii) $Th(\mathcal{I}) \vdash^{\mathcal{S}_1} \{p_1\}\ W_1\ \{q_1\}$

(iv) $Th(\mathcal{I}) \vdash^{\mathcal{S}_2} \{p_2\}\ W_2\ \{q_2\}$,

then W ∈ C′. Condition (i) follows from the fact that T is a conditional statement transformation from $\mathcal{S}$ to $\mathcal{S}'$. Conditions (ii) – (iv) are consequences of the conditional statement proof rules.

b) If $\mathcal{S}$ is a composed specification,

$$\{p'\}\ \mathcal{S}_3\ ;\ \mathcal{S}_4\ \{q'\},$$

for some specifications $\mathcal{S}_3$, $\mathcal{S}_4$ from L_S^B, then either $\mathcal{S}_3$ or $\mathcal{S}_4 \in L_{\{p\}\ \{q\}}$. Assume that $\mathcal{S}_3 \in L_{\{p\}\ \{q\}}$. The specification $\mathcal{S}'$ is

$$\{p'\}\ T_3(\mathcal{S}_3)\ ;\ \mathcal{S}_4\ \{q'\},$$

where

$$T_3\colon \mathcal{S}_3 \rightarrow \mathcal{S}_3{}'$$

is a conditional statement transformation for which the conditional statement proof rules hold for $\mathcal{S}_3$. Since $\mathrm{W} \in \mathrm{C}$, it follows that for some pre– and post–conditions p_3, q_3 and p_4, q_4 associated with $\mathcal{S}_3$ and $\mathcal{S}_4$, respectively, that

(i) W is W_3 ; W_4 for some $\mathrm{W}_3, \mathrm{W}_4 \in \mathrm{L}^{\mathrm{B}}_{\mathrm{W}}$.

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{\mathrm{p}'\}\ \mathrm{W}\ \{\mathrm{q}'\}$.

(iii) $\mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}_3} \{\mathrm{p}_3\}\ \mathrm{W}_3\ \{\mathrm{q}_3\}$.

(iv) $\mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}_4} \{\mathrm{p}_4\}\ \mathrm{W}_4\ \{\mathrm{q}_4\}$.

Using the induction hypothesis, it follows from (iii) that

(v) $\mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}_3'} \{\mathrm{p}_3\}\ \mathrm{W}_3\ \{\mathrm{q}_3\}$.

It follows from conditions (i), (ii), (iv), and (v) that

$$\mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}'} \{\mathrm{p}'\}\ \mathrm{W}\ \{\mathrm{q}'\}.$$

Therefore, $\mathrm{W} \in \mathrm{C}'$. If $\mathcal{S}_4 \in \mathrm{L}_{\{\mathrm{p}\}\,\{\mathrm{q}\}}$, then the proof is similar.

c) If $\mathcal{S}$ is a conditional specification,

$$\{\mathrm{p}'\}\ \textit{if}\ \mathrm{e}_1\ \textit{then}\ \mathcal{S}_3\ \textit{else}\ \mathcal{S}_4\ \textit{fi}\ \{\mathrm{q}'\},$$

for some quantifier free formula e_1 from $\mathrm{QFF}_{\mathrm{B}}$, and for some specifications $\mathcal{S}_3$, $\mathcal{S}_4$ from $\mathrm{L}^{\mathrm{B}}_{\mathrm{S}}$, then either $\mathcal{S}_3$ or $\mathcal{S}_4 \in \mathrm{L}_{\{\mathrm{p}\}\,\{\mathrm{q}\}}$. Assume that $\mathcal{S}_3 \in \mathrm{L}_{\{\mathrm{p}\}\,\{\mathrm{q}\}}$. The specification $\mathcal{S}'$ is

$$\{\mathrm{p}'\}\ \textit{if}\ \mathrm{e}_1\ \textit{then}\ \mathrm{T}_3(\mathcal{S}_3)\ \textit{else}\ \mathcal{S}_4\ \textit{fi}\ \{\mathrm{q}'\},$$

where

$$\mathrm{T}_3\colon \mathcal{S}_3 \rightarrow \mathcal{S}_3'$$

is a conditional statement transformation for which the conditional statement proof rules hold for $\mathcal{S}_3$. Since $\mathrm{W} \in \mathrm{C}$, it follows that for some pre– and post–conditions p_3, q_3 and p_4, q_4 associated with $\mathcal{S}_3$ and $\mathcal{S}_4$, respectively, that

(i) W is $\textit{if}\ \mathrm{e}_1\ \textit{then}\ \mathrm{W}_3\ \textit{else}\ \mathrm{W}_4\ \textit{fi}$ for some $\mathrm{W}_3, \mathrm{W}_4 \in \mathrm{L}^{\mathrm{B}}_{\mathrm{W}}$.

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

(iii) $\mathrm{Th}(\mathcal{I}) \vdash^{S_3} \{p_3\}\ W_3\ \{q_3\}$.

(iv) $\mathrm{Th}(\mathcal{I}) \vdash^{S_4} \{p_4\}\ W_4\ \{q_4\}$.

Using the induction hypothesis, it follows from (iii) that

(v) $\mathrm{Th}(\mathcal{I}) \vdash^{S_3'} \{p_3\}\ W_3\ \{q_3\}$.

It follows from conditions (i), (ii), (iv), and (v) that

$$\mathrm{Th}(\mathcal{I}) \vdash^{S'} \{p'\}\ W\ \{q'\}.$$

Therefore, $W \in C'$. If $S_4 \in L_{\{p\}\,\{q\}}$, then the proof is similar.

d) If S is a while specification,

$$\{p'\}\ \textit{while}\ e_1\ \textit{do}\ S_3\ \textit{od}\ \{q'\},$$

for some specification S_3 from $L_{\{p\}\,\{q\}}$, and some quantifier free formula e_1 from QFF_B, then S' is

$$\{p'\}\ \textit{while}\ e_1\ \textit{do}\ T_3(S_3)\ \textit{od}\ \{q'\},$$

where

$$T_3\colon S_3 \rightarrow S_3'$$

is a conditional statement transformation for which the conditional statement proof rules hold for S_3. Since $W \in C$, it follows that for some pre– and post–conditions p_3, q_3 associated with S_3 that

(i) W is ***while*** e_1 ***do*** W_3 ***od*** for some $W_3 \in L_W^B$.

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

(iii) $\mathrm{Th}(\mathcal{I}) \vdash^{S_3} \{p_3\}\ W_3\ \{q_3\}$.

Using the induction hypothesis, it follows from (iii) that

(iv) $\mathrm{Th}(\mathcal{I}) \vdash^{S_3'} \{p_3\}\ W_3\ \{q_3\}$.

for some specification S_3 from $L_{\{p\}\ \{q\}}$, and some quantifier free formula e_1 from QFF_B, then S' is

$$\{p'\}\ \textit{while}\ e_1\ \textit{do}\ T_3(S_3)\ \textit{od}\ \{q'\},$$

where

$$T_3\colon S_3 \rightarrow S_3'$$

is a composed statement transformation for which the composed statement proof rules hold for S_3. Since $W \in C$, it follows that for some pre– and post–conditions p_3, q_3 associated with S_3 that

(i) W is *while* e_1 *do* W_3 *od* for some $W_3 \in L^B_W$.

(ii) $Th(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

(iii) $Th(\mathcal{I}) \vdash^{S_3} \{p_3\}\ W_3\ \{q_3\}$.

Using the induction hypothesis, it follows from (iii) that

(iv) $Th(\mathcal{I}) \vdash^{S_3'} \{p_3\}\ W_3\ \{q_3\}$.

It follows from conditions (i), (ii), and (iv) that

$$Th(\mathcal{I}) \vdash^{S'} \{p'\}\ W\ \{q'\}.$$

Therefore, $W \in C'$.

Lemma: (*Conditional Statement Implementations -- general case*) Let T be a conditional statement transformation of the specification S, which contains the unknown specification,

$$\{p\}\ \{q\},$$

where p, q are formulas from WFF_B. Let S' be the image of S under T and let p', q' be the pre– and post–conditions associated with S and S'. Associated with the specific composed statement transformation, T, are formulas p_1, p_2, q_1, q_2 from WFF_B, the specifications, $\{p_1\}\ \{q_1\}$ and $\{p_2\}\ \{q_2\}$ from L^B_S, and the quantifier free formula e from QFF_B. Let C be

$$\{\ W \in L^B_W \mid Th(\mathcal{I}) \vdash^{S} \{p'\}\ W\ \{q'\}\ \}$$

It follows from conditions (i), (ii), and (iv) that

$$\mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}'} \{p'\}\ W\ \{q'\}.$$

Therefore, $W \in C'$.

Lemma: (*While Statement Implementations -- general case*) Let T be a while statement transformation of the specification $\mathcal{S}$, which contains the unknown specification,

$$\{p\}\ \{q\},$$

where p, q are formulas from WFF_B. Let $\mathcal{S}'$ be the image of $\mathcal{S}$ under T and let p', q' be the pre– and post–conditions associated with $\mathcal{S}$ and $\mathcal{S}'$. Associated with the specific while statement transformation, T, are formulas p_1, p_2 from WFF_B, the specification, $\{p_1\}\ \{q_1\}$, from L_S^B, and the quantifier free formula e from QFF_B. Let C be

$$\{\ W \in L_W^B \mid \mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}} \{p'\}\ W\ \{q'\}\ \}$$

and let C' be

$$\{\ W \in L_W^B \mid \mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}'} \{p'\}\ W\ \{q'\}\ \}.$$

If $W \in C$ and the while statement proof rules hold for $\mathcal{S}$, then $W \in C'$.

Proof: The proof is by induction on the specification $\mathcal{S}$.

a) If $\mathcal{S}$ is the unknown specification,

$$\{p\}\ \{q\},$$

then $\mathcal{S}'$ is

$$\{p\}\ \textit{while}\ e\ \textit{do}\ \mathcal{S}_1\ \textit{od}\ \{q\},$$

where $\mathcal{S}_1$ is $\{p_1\}\ \{q_1\}$. If

(i) W is *while* e *do* W_1 *od* for some $W_1 \in L_W^B$

(ii) $\mathrm{Th}(\mathcal{I}) \vdash \{p\}\ W\ \{q\}$

(iii) $\mathrm{Th}(\mathcal{I}) \vdash^{\mathcal{S}_1} \{p_1\}\ W_1\ \{q_1\}$, then $W \in C'$. Condition (i) follows from the fact that T is a while statement transformation from $\mathcal{S}$ to $\mathcal{S}'$. Conditions (ii) and (iii) are consequences of the while statement proof rules.

b) If $\mathcal{S}$ is a composed specification,

$$\{p'\}\ \mathcal{S}_3\ ;\ \mathcal{S}_4\ \{q'\},$$

for some specifications $\mathcal{S}_3$, $\mathcal{S}_4$ from L^B_S, then either $\mathcal{S}_3$ or $\mathcal{S}_4 \in L_{\{p\}\ \{q\}}$. Assume that $\mathcal{S}_3 \in L_{\{p\}\ \{q\}}$. The specification $\mathcal{S}'$ is

$$\{p'\}\ T_3(\mathcal{S}_3)\ ;\ \mathcal{S}_4\ \{q'\},$$

where

$$T_3\colon \mathcal{S}_3 \rightarrow \mathcal{S}_3{}'$$

is a while statement transformation for which the while statement proof rules hold for $\mathcal{S}_3$. Since $W \in C$, it follows that for some pre– and post–conditions p_3, q_3 and p_4, q_4 associated with $\mathcal{S}_3$ and $\mathcal{S}_4$, respectively, that

(i) W is W_3 ; W_4 for some W_3, $W_4 \in L^B_W$.

(ii) $Th(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

(iii) $Th(\mathcal{I}) \vdash^{\mathcal{S}_3} \{p_3\}\ W_3\ \{q_3\}$.

(iv) $Th(\mathcal{I}) \vdash^{\mathcal{S}_4} \{p_4\}\ W_4\ \{q_4\}$.

Using the induction hypothesis, it follows from (iii) that

(v) $Th(\mathcal{I}) \vdash^{\mathcal{S}_3'} \{p_3\}\ W_3\ \{q_3\}$.

It follows from conditions (i), (ii), (iv), and (v) that

$$Th(\mathcal{I}) \vdash^{\mathcal{S}'} \{p'\}\ W\ \{q'\}.$$

Therefore, $W \in C'$. If $\mathcal{S}_4 \in L_{\{p\}\ \{q\}}$, then the proof is similar.

c) If $\mathcal{S}$ is a conditional specification,

$$\{p'\}\ \textit{if}\ e_1\ \textit{then}\ S_3\ \textit{else}\ S_4\ \textit{fi}\ \{q'\},$$

for some quantifier free formula e_1 from QFF_B, and for some specifications S_3, S_4 from L_S^B, then either S_3 or $S_4 \in L_{\{p\}\,\{q\}}$. Assume that $S_3 \in L_{\{p\}\,\{q\}}$. The specification S' is

$$\{p'\}\ \textit{if}\ e_1\ \textit{then}\ T_3(S_3)\ \textit{else}\ S_4\ \textit{fi}\ \{q'\},$$

where

$$T_3\colon S_3 \rightarrow S_3'$$

is a while statement transformation for which the while statement proof rules hold for S_3. Since $W \in C$, it follows that for some pre- and post-conditions p_3, q_3 and p_4, q_4 associated with S_3 and S_4, respectively, that

(i) W is $\textit{if}\ e_1\ \textit{then}\ W_3\ \textit{else}\ W_4\ \textit{fi}$ for some $W_3, W_4 \in L_W^B$.

(ii) $Th(\mathcal{I}) \vdash \{p'\}\ W\ \{q'\}$.

(iii) $Th(\mathcal{I}) \vdash^{S_3} \{p_3\}\ W_3\ \{q_3\}$.

(iv) $Th(\mathcal{I}) \vdash^{S_4} \{p_4\}\ W_4\ \{q_4\}$.

Using the induction hypothesis, it follows from (iii) that

(v) $Th(\mathcal{I}) \vdash^{S_3'} \{p_3\}\ W_3\ \{q_3\}$.

It follows from conditions (i), (ii), (iv), and (v) that

$$Th(\mathcal{I}) \vdash^{S'} \{p'\}\ W\ \{q'\}.$$

Therefore, $W \in C'$. If $S_4 \in L_{\{p\}\,\{q\}}$, then the proof is similar.

d) If S is a while specification,

$$\{p'\}\ \textit{while}\ e_1\ \textit{do}\ S_3\ \textit{od}\ \{q'\},$$

for some specification S_3 from $L_{\{p\}\,\{q\}}$, and some quantifier free formula e_1 from QFF_B, then S' is

$$\{p'\}\ \textit{while}\ e_1\ \textit{do}\ T_3(S_3)\ \textit{od}\ \{q'\},$$

where

$$T_3: S_3 \rightarrow S_3'$$

is a while statement transformation for which the while statement proof rules hold for S_3. Since W $\in$ C, it follows that for some pre- and post-conditions p_3, q_3 associated with S_3 that

(i) W is *while* e_1 *do* W_3 *od* for some $W_3 \in L^B_W$.

(ii) $Th(I) \vdash \{p'\}\ W\ \{q'\}$.

(iii) $Th(I) \vdash^{S_3} \{p_3\}\ W_3\ \{q_3\}$.

Using the induction hypothesis, it follows from (iii) that

(iv) $Th(I) \vdash^{S_3'} \{p_3\}\ W_3\ \{q_3\}$.

It follows from conditions (i), (ii), and (iv) that

$$Th(I) \vdash^{S'} \{p'\}\ W\ \{q'\}.$$

Therefore, W $\in$ C$'$.

Theorem: (*Construction of a New Abstract Program*) Let (S, C) be an abstract program. Assume that S is a specification from $L^B_{\{p\}\{q\}}$; that is, S contains the unknown specification,

$$\{p\}\ \{q\}.$$

Let T be a transformation from S to S' which is either an assignment statement transformation, a composed statement transformation, a conditional statement transformation, or a while statement transformation. Let W $\in$ C be such that the proof rules corresponding for S corresponding to the transformation T hold. Let C$'$ be the set of implementations associated with S'. Then W $\in$ C$'$.

Proof: There are four cases. Either T is an assignment statement transformation, a composed statement transformation, a conditional statement transformation, or a while statement transformation. In each case, it follows from one of the four preceding lemmas that W $\in$ C$'$.

6. Conclusions

Need work here, especially with the implications of the proof rules.

7. References

1. Apt, Krzysztof R. *Ten Years of Hoare's Logic: A Survey – Part I.* **ACM Transactions on Programming Languages and Systems** (October 1981) vol. 3, no. 4, pp. 431–483.

2. Jones, Cliff B. **Software Development: A Rigorous Approach.** Prentice–Hall International, Engelwood Cliffs, N.J., 1980.

3. Loeckx, Jacques and Kurt Sieber. **The Foundations of Program Verification.** John Wiley & Sons, New York, 1984.

Incremental Software Development using Executable, Logic-based Specifications

Robert B. Terwilliger

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, Illinois

THESIS PROPOSAL

Incremental Software Development using Executable, Logic–based Specifications

Robert B. Terwilliger

Department of Computer Science
1304 W. Springfield Ave.
University of Illinois at Urbana–Champaign
Urbana, Illinois 61801
217–333–4428

This research is supported by NASA Grant NAG 1–138

THESIS PROPOSAL: Incremental Software Development using Executable, Logic-based Specifications†

Robert B. Terwilliger

Department of Computer Science
University of Illinois at Urbana-Champaign
252 Digital Computer Laboratory
1304 West Springfield Avenue
Urbana, IL 61801
(217) 333-4428

†This research is supported by NASA grant NAG 1-138.

Abstract

The Vienna Development Method (VDM) supports the top-down development of software specified in a notation suitable for formal verification. Components are first written using a combination of conventional programming languages and predicate logic. These abstract components are then incrementally refined into components in an implementation language. Each refinement is verified before another is applied; therefore, the final components produced by the development satisfy the original specifications. VDM has been used in industrial applications to enhance the development process. In such environments VDM is applied in an informal, non-automated manner; verification conditions are generated and certified without the aid of specialized tools, and data types are not formally axiomatized. We propose that an automated environment supporting a formal development method similar to VDM can be constructed, and that the environment will enhance the development method. For the thesis, we will design and build a prototype environment, and demonstrate that it enhances the VDM style development process. The environment will support the use of executable specifications and mechanical theorem proving, as well as providing simple facilities for configuration control and project management.

1. Introduction

It is widely acknowledged that producing correct software is both difficult and expensive. To help remedy this situation, many methods for specifying and verifying software have been developed[10,17]. The SAGA (Software Automation, Generation and Administration) project is investigating both the formal and practical aspects of providing automated support for the full range of software engineering activities[4,5]. ENCOMPASS[22,23] is an integrated environment to support the construction of software in a manner similar to the Vienna Development Method[13]. PLEASE is the wide-spectrum, executable specification and design language used in ENCOMPASS[24,25]. For the thesis, we will design and implement prototype versions of PLEASE and ENCOMPASS and demonstrate that they enhance the software development process.

The first step in the production of a software system is usually the creation of a *specification* which describes the functions and properties of the desired system. We say that a specification is *validated* when it is shown to correctly reflect the users' desires[8]. Producing a valid specification is a difficult task. The users of the system may not really know what they want, and they may be unable to communicate their desires to the development team. If the specification is in a formal notation it may be an ineffective medium for communication with the customers, but natural language specifications are notoriously ambiguous and incomplete. *Prototyping*[11,16] and the use of executable specification languages[14,27] have been suggested as partial solutions to these problems. Providing the customers with prototypes for experimentation and evaluation early in the development process may increase customer/developer communication and enhance the validation and design processes.

Even with a validated specification, producing a correct implementation is not an easy task. We say that an implementation is *verified* when it is shown to satisfy the specification[8]. Many methodologies for the design and development of correct implementations have been proposed[1,2,13,19]. For example, it has been suggested that *top-down development* can help control the complexity of program construction. By using *stepwise refinement*[26] to create a concrete implementation from an abstract specification we divide the decisions necessary into smaller, more comprehensible groups.

The Vienna Development Method (VDM) supports the top-down development of programs specified in a notation suitable for mathematical verification[13,21]. In this method, programs are first written in a language combining elements from conventional programming languages and mathematics. A procedure or function may be specified using *pre-* and *post-conditions* written in predicate logic; similarly, a data type may have an *invariant.* These *abstract programs* are then incrementally refined into programs in an implementation language. The refinements are performed one at a time, and each is verified before another is applied; therefore, the final program produced by the development satisfies the original specification.

ENCOMPASS is an environment being created by the SAGA project to provide automated support for all aspects of a development method similar to VDM. We believe that neither testing[9,18], technical review[7], or formal verification[17] alone can guarantee program correctness; therefore, ENCOMPASS provides a framework in which all three methods can be used as needed. ENCOMPASS includes a number of different tools including: a language-oriented editor; a test harness; a configuration control and project management system; and a user interface package. ENCOMPASS is in the early stages of development; an initial prototype has been constructed and used to develop small programs. ENCOMPASS is described in more detail in [23], which is also Appendix A of this paper; early reports on the environment can be found in[3,15,22].

PLEASE is the wide-spectrum, executable specification language used in ENCOMPASS. PLEASE extends its underlying implementation, or *base*, language so that a procedure or function may be specified with pre- and post-conditions and an implementation may be completely annotated. At present, all our efforts involve Ada[1] as the base language. PLEASE specifications may be used in proofs of correctness; they also may be transformed into prototypes which use Prolog[6] to "execute" pre- and post-conditions and may interact with other modules written in the base language. We believe that the early production of executable prototypes for experimentation and evaluation will enhance the software development process. PLEASE is described in more detail in [24], which is also Appendix B to this paper; a preliminary report on the language can be found in[25].

IDEAL is the programming-in-the-small environment used within ENCOMPASS[23]. IDEAL supports the specification, construction, validation, and verification of single modules. It includes ISLET, a simple language-oriented editor which supports the creation of PLEASE specifications and their refinement into Ada implementations. As the specifications are created and refined, the syntax and semantics are constantly checked. From IDEAL, the user can invoke commands to create Ada/Prolog prototypes from PLEASE specifications. IDEAL also includes an interface to the ENCOMPASS test harness and TED, a proof management system which is interfaced to a number of theorem provers[12].

In section two of this paper, we describe the development methodology which PLEASE, IDEAL, and ENCOMPASS are designed to support and in section three, we present a proposed thesis outline. In sec-

[1] Ada is a trademark of the US Government, Ada Joint Program Office.

tion four, we give completion criteria for the thesis in section five, we summarize the proposed research and expected results.

2. Software Development in ENCOMPASS

ENCOMPASS is based on a *traditional* or *phased*[8] life-cycle model extended to support executable specifications and formal verification. In ENCOMPASS, a development passes through the phases: planning, requirements definition, validation, refinement and system integration. In the *requirements definition phase*, the functions and properties of the software to be produced by the development are determined[8]. In ENCOMPASS, software requirements specifications are a combination of natural language and components specified in PLEASE. Although a software system may be shown to meet its specification, this does not imply that the system satisfies the customers' requirements. In ENCOMPASS, we extend the traditional life-cycle to include a separate phase for customer validation.

The *validation phase* attempts to show that any system which satisfies the specification will also satisfy the customers' requirements, that is, that the requirements specification is valid. If not, then the requirements specification should be corrected before the development proceeds any further. To aid in the validation process, the PLEASE components in the specification may be transformed into executable prototypes which satisfy the specifications. These prototypes may be used in interactions with the customers; they may be subjected to a series of tests, be delivered to the customers for experimentation and evaluation, or be installed for production use on a trial basis. The use of prototypes may increase customer/developer communication and enhance the validation process. If it is found that the specification does not satisfy the customers, then it is revised, new prototypes are produced, and the validation process is reinitiated; this cycle is repeated until a validated specification is produced.

In general, this process does not guarantee that the specification is valid. The fact that the prototype does satisfy the customers means only that at least one implementation which satisfies the specification is acceptable. For example, the post-condition for a procedure may hold true for an infinite number of values while the prototype will only return one. We say the specification of a component is *complete* if, for any input state, it is satisfied by only one output state. Although in some cases it is possible to require and verify that the specification of a component is complete, this is difficult in practice. We believe that while prototypes may enhance the validation process, they do not replace communication with the customers and review of the specification.

In the *refinement phase*, the validated specification is incrementally transformed into a program in the implementation language; this process is viewed as the construction of a proof in the Hoare calculus. In ENCOMPASS, the refinement process is supported by a language oriented editor similar to[20]. As the specification is transformed into an implementation (and the proof is constructed) the syntax and semantics are constantly checked. Many steps in the refinement will generate verification conditions in the underlying first-order logic. These are algebraically simplified and then subjected to a number of simple proof tactics. If these fail, the verification conditions are passed to TED, a proof management system which is interfaced to a number of theorem provers[12]. In our experience, it is too expensive to mechanically certify all of the verification conditions; therefore, the implementor can simply "check off" the verification conditions for a refinement and continue. The verification conditions are recorded by ENCOMPASS for use in project monitoring, management and debugging.

PLEASE specifications enhance the verification of system components using either testing or proof techniques. The specification of a component can be transformed into a prototype. This prototype may be used as a test oracle against which the implementation can be compared. Since the specification is formal, proof techniques may be used which range from a very detailed, completely formal proof using mechanical theorem proving to a development "annotated" with unproven verification conditions. PLEASE provides a framework for the *rigorous*[13] development of programs. Although detailed mechanical proofs are not required at every step, the framework is present so that they can be constructed if necessary. Parts of a project may use detailed mechanical verification while other, less critical parts may be handled using less expensive techniques.

3. Proposed Thesis Outline

Figure 1 shows the proposed thesis outline. After the introductory comments, enough information on first-order predicate logic and the resolution principle is given to make the thesis self contained. In the

1. Introduction
2. Mathematical Preliminaries
    - a. First-order Predicate Logic
    - b. Decidable, Axiomatizable Theories
    - c. Automatic Theorem Proving (The Resolution Principle)
3. Previous Work
    - a. Specification Methods (Algebraic, State Transition)
    - b. Program Verification (Hoare Calculus, Partial and Total Correctness)
    - c. Logic Programming (Prolog)
    - d. Development Methods (Top-down, Transformational, Proofs as Programs)
    - e. Life-cycle Models (traditional, operational, automatic programming)
    - f. Software Engineering Environments
4. PLEASE (Statements, Pre-defined and User Types)
5. Producing Prototypes from Pre- and Post-conditions
6. Using PLEASE Prototypes in Software Validation
7. Refinement of IDEAL Specifications (Incremental Verification)
8. IDEAL (Goals, Development Paradigm, Components)
9. ENCOMPASS (Goals, Life-Cycle, Limitations, Components)
10. Implementation
11. Summary and Conclusions
12. References

Figure 1. Proposed Thesis Outline

previous work section, results on program specification and verification, logic programming, development methods, life-cycle models and software engineering environments are given. In section four, both the abstract syntax and semantics of the PLEASE language are defined. In section five, the methods used to produce prototypes from PLEASE specifications are discussed, while in section six the use of these prototypes in software validation is explored. Section seven discusses the incremental refinement of PLEASE specifications into Ada implementations, while sections eight and nine discuss IDEAL and ENCOMPASS respectively. Section ten briefly describes the implementation and section eleven contains a summary and conclusions.

4. Completion Criteria

In the completed thesis, a prototype implementation of ENCOMPASS with the following features will be described:

- Rudimentary systems for object-oriented configuration control and project management.
- Tools to automatically translate PLEASE specifications into Ada/Prolog prototypes.
- A test harness compatible with both Ada implementations and PLEASE prototypes.
- A language-oriented editor to support the creation and refinement of PLEASE specifications.

This prototype will support a preliminary subset of PLEASE with the following features:

- A small, fixed set of types including natural numbers, lists, booleans and characters.
- The *if-then-else*, *while* and assignment statements.
- Procedure calls with *in*, *out* and *in out* parameters.
- User defined functions (without side effects).
- A facility supporting user-defined types specified using predicate logic.

Throughout the thesis, the emphasis will be placed on the theoretical basis and design of these components, rather than on the creation of production-quality implementations. The emphasis will also be on the programming-in-the-small aspects of the environment, rather than on the programming-in-the-large; only an architecture for ENCOMPASS will be given, while IDEAL and PLEASE will be explained in greater detail.

5. Summary

The Vienna Development Method (VDM) supports the top-down development of software specified in a notation suitable for formal verification. Components are first written using a combination of conventional programming languages and predicate logic. These abstract components are then incrementally refined into components in an implementation language. Each refinement is verified before another is applied; therefore, the final components produced by the development satisfy the original specifications. VDM has been used in industrial applications to enhance the development process. In such environments VDM is applied in an informal, non-automated manner; verification conditions are generated and certified without the aid of specialized tools, and data types are not formally axiomatized. We propose that an automated environment supporting a formal development method similar to VDM can be constructed, and that the environment will enhance the development method. For the thesis, we will design and build a prototype environment, and demonstrate that it enhances the VDM style development process. The environment will support the use of executable specifications and mechanical theorem proving, as well as providing simple facilities for configuration control and project management.

6. References

1. Balzer, Robert, Thomas E. Cheatham and Cordell Green. *Software Technology in the 1990's: Using a New Paradigm.* IEEE **Computer** (November 1983) vol. 16, no. 11, pp. 39–45.

2. Bates, Joseph L. and Robert L. Constable. *Proofs as Programs.* ACM **Transactions on Programming Languages and Systems** (January 1985) vol. 7, no. 1, pp. 113–136.

3. Campbell, Roy H. and Robert B. Terwilliger,. *The SAGA Approach to Automated Project Management.* In: **International Workshop on Advanced Programming Environments**, Lynn R. Carter, ed. Springer–Verlag Lecture Notes in Computer Science, New York, 1986, pp. 145–159.

4. Campbell, Roy H. *SAGA: A Project to Automate the Management of Software Production Systems.* In: **Software Engineering Environments**, Ian Sommerville, ed. Peter Perigrinus Ltd, 1986.

5. Campbell, Roy H. and Peter A. Kirslis. *The SAGA Project: A System for Software Development.* **Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments** (April 1984) pp. 73–80.

6. Clocksin, W. F. and C. S. Mellish. **Programming in Prolog**. Springer–Verlag, New York, 1981.

7. Fagan, Michael E. *Advances in Software Inspections.* **IEEE Transactions on Software Engineering** (July 1986) vol. SE–12, no. 7, pp. 744-751.

8. Fairley, Richard. **Software Engineering Concepts**. McGraw-Hill, New York, 1985.

9. Gannon, John, Paul McMullin and Richard Hamlet. *Data-Abstraction Implementation, Specification, and Testing.* ACM **Transactions on Programming Languages and Systems** (July 1981) vol. 3, no. 3, pp. 211–223.

10. Gehani, Narain and Andrew D. McGettrick (eds.). **Software Specification Techniques**. Addison Wesley, Reading, Massachusetts, 1986.

11. Goguen, Joseph and Jose Meseguer. *Rapid Prototyping in the OBJ Ezececutable Specification Laguage.* **Software Engineering Notes** (December 1982) vol. 7, no. 5, pp. 75–84.

12. Hammerslag, David H., Samuel N. Kamin and Roy H. Campbell. *Tree-Oriented Interactive Processing with an Application to Theorem-Proving.* **Proceedings of the Second ACM/IEEE Conference on Software Development Tools, Techniques, and Alternatives** (December, 1985).

13. Jones, Cliff B. **Software Development: A Rigorous Approach**. Prentice–Hall International, Engelwood Cliffs, N.J., 1980.

14. Kamin, S. N., S. Jefferson and M. Archer. *The Role of Executable Specifications: The FASE System.* **Proceedings of the IEEE Symposium on Application and Assessment of Automated Tools for Software Development** (November 1983).

15. Kirslis, Peter A., Robert B. Terwilliger and Roy H. Campbell. *The SAGA Approach to Large Program Development in an Integrated Modular Environment.* **Proceedings of the GTE Workshop on Software Engineering Environments for Programming-in-the-Large** (June 1985) pp. 44–53.

16. Kruchten, Philippe, Edmond Schonberg and Jacob Schwartz. *Software Prototyping Using the SETL Programming Language.* **IEEE Software** (October 1984) vol. 1, no. 4, pp. 66–75.

17. Loeckx, Jacques and Kurt Sieber. **The Foundations of Program Verification**. John Wiley & Sons, New York, 1984.

18. Meyers, G. J. **The Art of Software Testing**. John Wiley & Sons, New York, 1979.

19. Mills, Harlan D. and Richard C. Linger. *Data Structured Programming: Program Design without Arrays and Pointers.* IEEE **Transactions on Software Engineering** (February 1986) vol. SE–12, no. 2, pp. 192–197.

20. Reps, Thomas and Bowen Alpern. *Interactive Proof Checking.* **Proceedings of the 11th ACM Symposium on the Principles of Programming Languages** (January 1984) pp. 36–45.

21. Shaw, R. C., P. N. Hudson and N. W. Davis. *Introduction of A Formal Technique into a Software Development Environment (Early Observations).* **Software Engineering Notes** (April 1984) vol. 9, no. 2, pp. 54–79.

22. Terwilliger, Robert B. and Roy H. Campbell. *ENCOMPASS: a SAGA Based Environment for the Composition of Programs and Specifications.* **Proceedings of the 19th Hawaii International Conference on System Sciences** (January 1986) pp. 436–447.

23. ——. "ENCOMPASS: an Environment for the Incremental Development of Software", Report No. UIUCDCS-R-86-1296, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1986.

24. ——. "PLEASE: Executable Specifications for Incremental Software Development", Report No. UIUCDCS-R-86-1295, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1986.

25. ——. *PLEASE: Predicate Logic based ExecutAble SpEcifications.* **Proceedings of the 1986 ACM Computer Science Conference** (February, 1986) pp. 349-358.

26. Wirth, Niklaus. *Program Development by Stepwise Refinement.* **Communications of the ACM** (April 1971) vol. 14, no. 4, pp.

221–227.

27. Zave, Pamela and William Schnell. *Salient Features of an Executable Specification Language and Its Environment.* IEEE Transactions on Software Engineering (February 1986) vol. SE–12, no. 2, pp. 312–325.

Appendix A

ENCOMPASS: An Environment for the Incremental Development of Software

Robert B. Terwilliger
Roy H. Campbell

Report No. UIUCDCS-R-86-1296
Department of Computer Science
1304 W. Springfield Ave.
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
217-333-4428

Preprint September 15, 1986

This research is supported by NASA Grant NAG 1-138

ENCOMPASS: an Environment for the Incremental Development of Software[*]

Robert B. Terwilliger
Roy H. Campbell

Department of Computer Science
University of Illinois at Urbana-Champaign
252 Digital Computer Laboratory
1304 West Springfield Avenue
Urbana, IL 61801
(217) 333-4428

[*]This research is supported by NASA grant NAG 1-138.

Abstract

ENCOMPASS is an integrated environment being constructed by the SAGA project to support incremental software development in a manner similar to the Vienna Development Method. In this paper, we describe the architecture of ENCOMPASS and give an example of software development in the environment. In ENCOMPASS, software is modeled as entities which may have relationships between them. These entities can be structured into complex hierarchies which may be seen through different views. The configuration management system stores and structures the components developed and used in a project, as well as providing a mechanism for controlling access. The project management system implements a milestone-based policy using the mechanism provided. In ENCOMPASS, software is first specified using a combination of natural language and PLEASE, a wide-spectrum, executable specification and design language. Components specified in PLEASE are then incrementally refined into components written in Ada[1]; this process can be viewed as the construction of a proof in the Hoare calculus. Each refinement is verified before another is applied; therefore, the final components produced by the development satisfy the original specifications. PLEASE specifications may be used in formal proofs of correctness; they may also be transformed into executable prototypes which can be used in the validation and design processes. ENCOMPASS provides automated support for all aspects of software development using PLEASE. We believe the use of ENCOMPASS will enhance the software development process.

1. Introduction

It is both difficult and expensive to produce high-quality software. One solution to this problem is the use of *software engineering environments* which integrate a number of tools, methods, and data structures to provide support for program development and/or maintenance[2,17,29,34,43,54,66,79,90,93-97,108,111]. The SAGA (Software Automation, Generation and Administration) project is investigating both the formal and practical aspects of providing automated support for the full range of software engineering activities[10,18-21,49,63,98-100]. ENCOMPASS[98] is an integrated environment being

[1] Ada is a trademark of the US Government, Ada Joint Program Office.

created by the SAGA project to support the incremental development of software using the PLEASE[99,100] executable specification language. In this paper, we describe the architecture of ENCOMPASS and give an example of software development in the environment.

A *life-cycle model* describes the sequence of distinct stages through which a software product passes during its lifetime[37]. There is no single, universally accepted model of the software life-cycle[3,6,13,112]. The stages of the life-cycle generate *software components*, such as code written in programming languages, test data or results, and many types of documentation. In many models, a *specification* of the system to be built is created early in the life-cycle (many methods for specifying software have been proposed[39,42,46,47,60,76,82]). As components are produced, they are *verified*[37] for correctness with respect to their specifications. A specification is *validated*[37] when it is shown to correctly state the customers' requirements.

Producing a valid specification is a difficult task. The users of the system may not really know what they want, and they may be unable to communicate their desires to the development team. If the specification is in a formal notation, it may be an ineffective medium for communication with the customers, but natural language specifications are notoriously ambiguous and incomplete. *Prototyping* and the use of *executable specification languages* have been suggested as partial solutions to these problems[28,41,50,61,62,65,103,113]. Providing the customers with prototypes for experimentation and evaluation early in the development process may increase customer/developer communication and enhance the validation and design processes.

Even given a validated specification, it may be difficult to determine if an implementation is correct. Many techniques for verifying the correctness of implementations have been proposed. For example, *testing* can be used to check the operation of an implementation on a representative set of input data[38,74]. In a *technical review* process, the specification and implementation are inspected, discussed and compared by a group of knowledgeable personnel[36,106]. If the specification is in a suitable notation, formal methods can be used to verify the correctness of an implementation[48,51,52,58,73,109]. Many feel that no one technique alone can insure the production of correct software[31,32]; therefore, methods which combine a number of techniques have been proposed[86].

To help control the complexity of software design and construction, many different *development methods* have been proposed[5,44,56,58,75,110]. Many of these methods are based on a model of the software development process; they combine standard representations, intellectual disciplines, and well defined techniques in a unified framework. For example, it has been suggested that that the development process be viewed as a sequence of *transformations* between different, but somehow equivalent, specifications[6,7,23,70,77,83].

Others have suggested that *modular programming*[81,101,104] and the *top-down development* of programs[33,44,58,107] can help reduce the difficulty of program construction and maintenance. By logically dividing a monolithic program into a number of modules, we reduce the knowledge required to change fragments of the system and decrease the apparent complexity. By using *stepwise refinement* to create a concrete implementation from an abstract specification, we divide the decisions necessary for an implementation into smaller, more comprehensible groups. A number of modern programming languages support modular programming[30,69,72], and environments to support such methods have been both proposed and constructed[17,93,94,111]. Methods to support the top-down development of programs have been both devised and put into use[12,14,15,27,58,75,87,88].

The Vienna Development Method (VDM) supports the top-down development of software specified in a notation suitable for formal verification[11,12,27,57-59,88]. In this method, components are first written in a language combining elements from conventional programming languages and mathematics. A procedure or function may be specified using *pre-* and *post-conditions* written in predicate logic; similarly, a data type may have an *invariant*. These *abstract components* are then incrementally refined into components in an implementation language. The refinements are performed one at a time, and each is verified before another is applied; therefore, the final components produced by the development satisfy the original specifications.

PLEASE is a wide-spectrum, executable specification language which supports a development method similar to VDM. PLEASE extends its underlying implementation, or *base*, language so that a procedure or function may be specified with pre- and post-conditions, a data type may have an invariant, and an implementation may be completely annotated. At present, we are using Ada[30,105] as the base language. PLEASE specifications may be used in proofs of correctness; they also may be transformed into prototypes which use Prolog[26,64] to "execute" pre- and post-conditions, and may interact with other modules written in the base language. We believe that the early production of executable prototypes for experimentation and evaluation will enhance the software development process.

ENCOMPASS is an integrated environment being constructed by the SAGA project to support incremental software development using PLEASE. In ENCOMPASS, software is modeled as entities which have relationships between them. These entities can be structured into complex hierarchies which may be seen through different views. The configuration management system stores and structures the components developed and used in a project, as well as providing a mechanism for controlling access. The project management system implements a milestone-based policy using the mechanism provided. In ENCOMPASS, software is first specified using a combination of natural language and PLEASE. Components specified in PLEASE are then incrementally refined into components written in Ada; this process can be viewed as the construction of a proof in the Hoare calculus[51,73]. Each refinement is verified before another is applied; therefore, the final components produced by the development satisfy the original specifications. ENCOMPASS provides automated support for all aspects of this development process.

In section two of this paper we describe the ENCOMPASS environment, both its architecture and the life-cycle model on which it is based. In section three we describe IDEAL, the programming-in-the-small environment used within ENCOMPASS, and in section four, we give an example of software development using ENCOMPASS. In section five, we briefly describe the current status of the system and in section six, we summarize the support ENCOMPASS provides for incremental software development.

2. ENCOMPASS

ENCOMPASS is designed to support a particular model of the software life-cycle; this is basically Fairley's *phased* or *waterfall* life-cycle[37], extended to support the use of executable specifications and the Vienna Development Method. In ENCOMPASS, a development passes through the phases planning, requirements definition, validation, refinement and system integration.

In the *planning phase*, the problem to be solved is defined and it is determined if a computer solution is feasible and cost effective, while in the *requirements definition* phase, the functions and qualities of the software to be produced by the development are precisely described[37]. In ENCOMPASS, software requirements specifications are a combination of natural language documents and components specified in PLEASE. Although the requirements specification describes a software system, it is not known if any system which satisfies the specification will satisfy the customers. In ENCOMPASS, we extend Fairley's phased life-cycle model to include a separate phase for customer validation.

The *validation phase* attempts to show that any system which satisfies the software requirements specification will also satisfy the customers, that is, that the requirements specification is valid. If not, then the requirements specification should be corrected before the development proceeds to the costly phases of refinement and system integration. To aid in the validation process, the PLEASE components in the specification may be transformed into executable prototypes which satisfy the specification. These prototypes may be used in interactions with the customers; they may be subjected to a series of tests, be delivered to the customers for experimentation and evaluation, or be installed for production use on a trial basis. We feel the use of prototypes will increase customer/developer communication and enhance the validation process.

In the *refinement phase*, the PLEASE specifications are incrementally transformed into Ada implementations. The refinement phase can be decomposed into a number of steps, each of which consists of a *design transformation* and its associated *verification phase*. The design transformation may produce annotated components in the base language as well as an updated requirements specification. Components which have been implemented need not be refined further, but components which are only specified will undergo further refinements until a complete implementation is produced. Each design transformation creates a new specification, whose relationship to the original is unknown. Before further refinements are performed, a *verification phase* must show that any implementation which satisfies the lower level specification will also satisfy the upper level one. In our model, this is accomplished using a combination of testing, technical review, and formal verification.

PLEASE specifications enhance the verification of system components using either testing or proof techniques. The specification of a component can be transformed into a prototype; this prototype may be used as a test oracle against which the implementation can be compared. Since the specification is formal, proof techniques may be used which range from a very detailed, completely formal proof using mechanical theorem proving, to a development "annotated" with unproven verification conditions. ENCOMPASS is an environment for the *rigorous*[58] development of programs. Although detailed mechanical proofs are not required at every step, the framework is present so that they can be constructed if necessary. Parts of a project may use detailed mechanical verification while other, less critical parts may be handled using less expensive techniques.

The planning, requirements definition, and validation phases are sequential in nature, but during the refinement phase, some tasks may be performed in parallel. For example, suppose a specification is refined to produce a more detailed specification which contains a number of independent components. These components may be refined concurrently to produce more detailed specifications and finally implementations. These independently developed implementations must then be integrated into a complete system. In the *system integration phase*, separately implemented modules are integrated into successively larger units, each of which is shown to satisfy the specifications[37]. When the final integration has been performed, the acceptance tests are performed, the product is delivered and the development is complete.

In ENCOMPASS, a phase may contain a sub-development just as a development contains a number of phases. For example, if a system is very large and complex, the production of a prototype in the validation phase may in itself be a complete development. If the system is composed of several major components, the production of each component from its specification during the refinement phase might also be considered a complete development. By dividing the development process into small steps using hierarchical composition, ENCOMPASS allows each step to be smaller and more comprehensible and thereby increases management's ability to trace and control the project.

2.1. System Architecture

Figure 1 shows the top-level architecture of ENCOMPASS. The *user* accesses and modifies components using a set of *software development tools*. These include ISLET, a language-oriented editor for the construction and refinement of PLEASE specifications, and Ted[49], a proof management system which is interfaced to a number of theorem provers. The *configuration management system* structures the software components developed by a project and stores them in a *project data base*. The configuration management system also provides a primative form of *software capabilities* to control access to components. The *project management system* distributes these capabilities to implement a *management by objectives*[45] approach to software development; each phase in the life-cycle satisfies an objective by producing a *mile-*

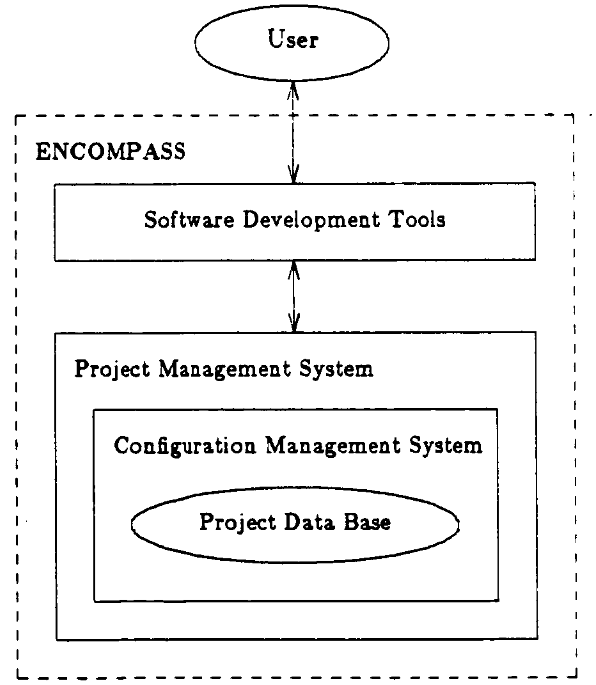

User
ENCOMPASS
Software Development Tools
Project Management System
Configuration Management System
Project Data Base

Figure 1. Architecture of ENCOMPASS

stone which can be recognized by the system.

Configuration management is concerned with the identification, control, auditing, and accounting of components produced and used in software development and maintenance[1,8,9,16]. Configuration control systems and models of software configurations have been suggested as aids to configuration management[4,35,40,53,55,67,68,71,78,89,102,114]. In ENCOMPASS, software configurations are modeled using variant of the *entity-relationship model*[24,25,80] which incorporates the concepts of *aggregation* and *generalization*[91,92].

An *entity* is a distinct, named component; an entity may have *attributes* which describe its properties or qualities. Two or more entities may have a *relationship* between them; a relationship may also have attributes. A group of entities with a relationship between them may be abstracted into an *aggregate* entity. This entity would have entities as the value of some or all of its attributes. A *view* is a mapping from names to components. A project under development has a unique *base view* or *project library* which describes the components of the system being developed and the primitive relationships between them. Other views can be include *images* of entities in this base view. In ENCOMPASS, access to components is controlled through the use of views.

The project management system is organized around *work trays*[18], which provide a mechanism to manage and record the allocation, progress, and completion of work within a software development project. In ENCOMPASS, each user may have a number of work trays, each of which may contain a number of *tasks* that contain software *products*. Project libraries are one type of task. There are four types of trays: *input trays*, *output trays*, *in-progress trays*, and *file trays*. Each user receives tasks in one or more input trays. The user may then transfer these tasks to an in-progress tray where he will perform the actions required of him and produce new products. The user may then return the task via a conceptual output tray to an input tray for the originator of the task. A user may also create new tasks in in-progress trays that he owns. These tasks may then be transferred to another user's input tray. A task that has been transferred back into the in-progress tray of the user who created the task may be marked as complete and transferred to a file tray for long term storage.

3. IDEAL

ENCOMPASS may be used to develop programs which consist of many interacting modules; in this sense, it is an environment for programming-in-the-large[84,108]. IDEAL is an environment concerned with the specification, prototyping, implementation and verification of single modules; it is the programming-in-the-small environment used within ENCOMPASS.

Figure 2 shows the top-level architecture of IDEAL, which contains four tools: TED, a proof management system which is interfaced to a number of theorem provers; ISLET (Incredibly Simple Language-oriented Editing Tool), a prototype program/proof editor; a tool to support the construction of executable prototypes from PLEASE specifications; and a test harness. The user interacts with these tools through a common interface. The tools in IDEAL operate on components which are stored in a *module data base.* The module data base is stored as part of a project data base by the configuration control system; IDEAL receives a capability to the module data base from the project management system. The module data base contains five types of components: symbol tables, proofs, source code, load modules and test cases.

A set of *symbol tables* represent the PLEASE specifications and Ada programs being developed. These symbol tables are displayed and manipulated by ISLET, a prototype program/proof editor. ISLET can be used to create PLEASE specifications and incrementally refine them into Ada programs; this process can also be viewed as the construction of a proof in the Hoare calculus[51,73]. Some steps in the proof may generate verification conditions in the underlying first-order logic; these can be reformated as *proofs* which serve as input for TED. Using TED, the user can structure the proof into a number of lemmas and bring in pre-existing theories.

The symbol tables also serve as input for the prototyping tool, which uses them to produce executable prototypes from PLEASE specifications. The *source code* for the prototypes is written in a combination of Prolog and Ada and utilizes a number of run-time support routines in both languages. The *load modules* produced from both prototypes and final implementations are used by the test harness. From the test harness, the user can invoke commands to manipulate *test cases.* Commands are available to: edit or

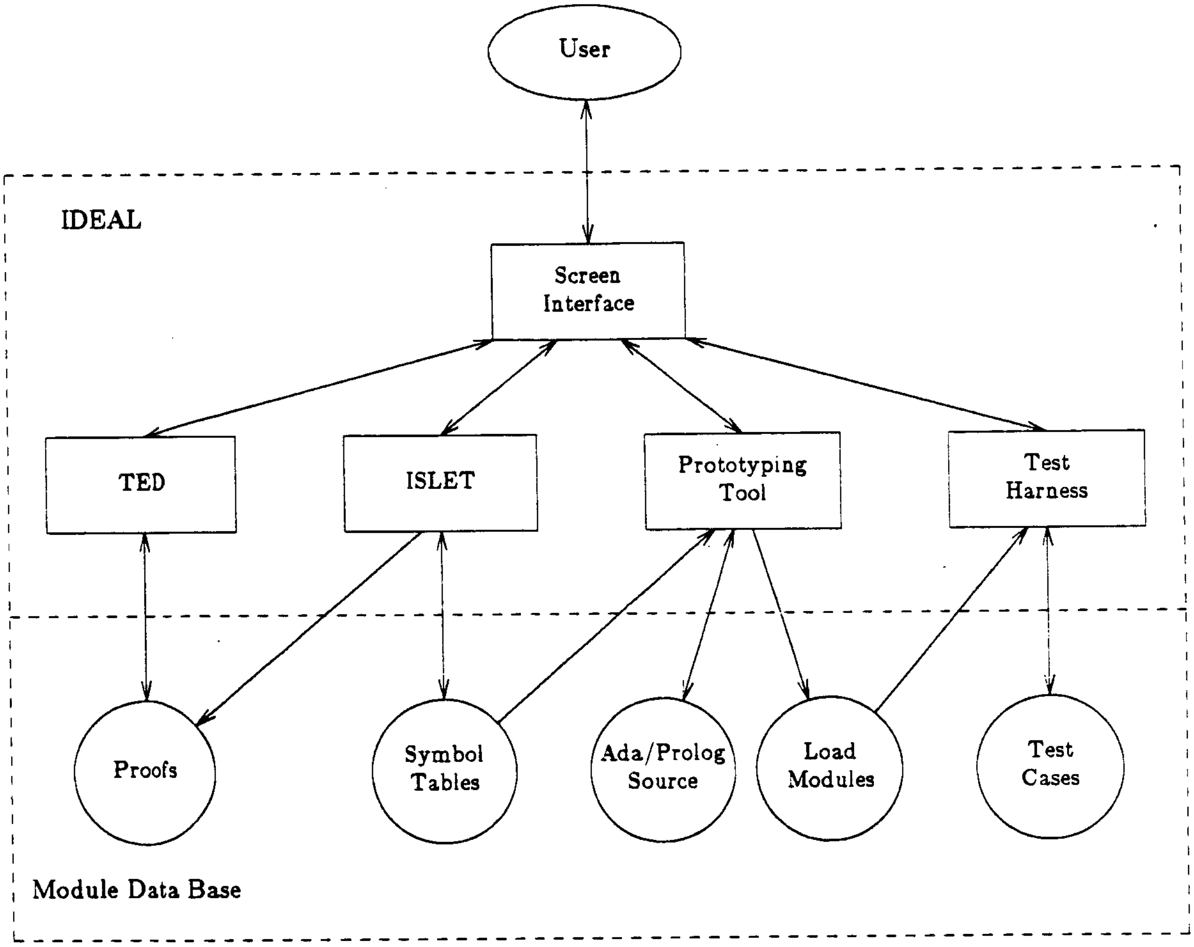


Figure 2. Architecture of IDEAL

browse the input for a test case; generate output for a test case; or run a program and compare the results with output that has been previously checked for correctness.

The central tool in IDEAL is ISLET. It not only manipulates the symbol tables representing specifications and implementations, but provides a user interface and, in a sense, controls the entire development process.

3.1. ISLET

ISLET supports both the creation of PLEASE specifications and their incremental refinement into annotated Ada implementations. This process can be viewed in two ways: as the development of a program, or as the construction of a proof in the Hoare calculus[51,73]. The refinement process consists of a number of *atomic transformations*. From the program view, an atomic transformation changes an unknown statement into a particular language construct; from the proof view, an atomic transformation adds another step to an incomplete proof. From the program view, defining a predicate adds a new construct to the program; from the proof view, defining a predicate adds new axioms to the first-order theory on which the proof is based.

Figure 3 shows the architecture of ISLET. The user interacts with ISLET through a simple language-oriented editor similar to[85]. The editor provides commands to add, delete, and refine constructs; as the program/proof is incrementally constructed, the syntax and semantics are constantly checked. The editor also controls the other components: an algebraic simplifier, a number of simple proof procedures, and an interface to TED. Many steps in the refinement process generate verification conditions in the underlying first-order logic. These verification conditions are first simplified algebraically and then subjected to a number of simple proof tactics. These methods can handle a large percentage of the verification conditions generated. If a set of verification conditions can not be proved using these methods alone, the TED interface is invoked to create a proof in the proper format.

TED can then be invoked in an attempt to prove the verification conditions. Using TED is very expensive, both in system resources and user time; however, many complex theorems can be proved with its aid. The algebraic simplification and simple proof tactics used in ISLET are very inexpensive; however, they are not very powerful. The combined use of these two methods supports the *rigorous*[58] development of programs. Most of the verification conditions will be proven using inexpensive methods; those that are expensive to verify may be proven immediately, or deferred until a later time. Parts of a system may be developed using completely mechanical methods, while other, less critical parts may use less expensive techniques.

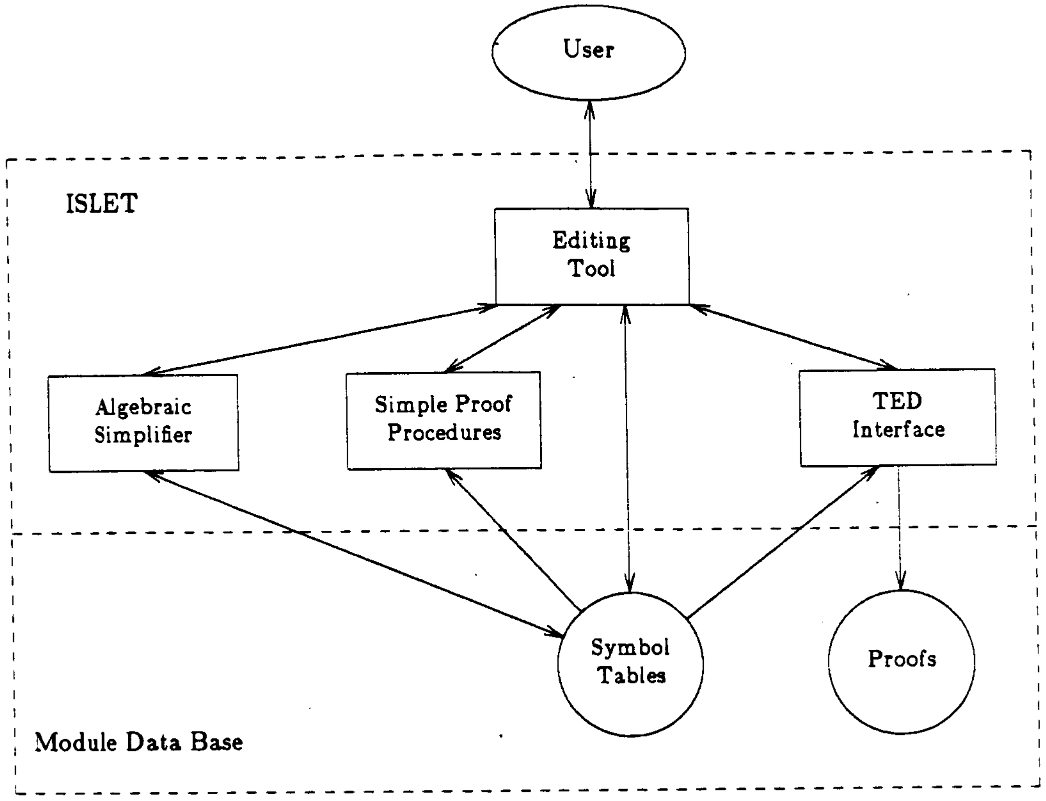


Figure 3. Architecture of ISLET

To further clarify the concepts and operation of ENCOMPASS and show how ENCOMPASS can enhance the software development process, we will consider an example of software development. We will follow the development from receipt of the assignment by the team leader through delivery of a verified and validated implementation.

4. An Example of Software Development

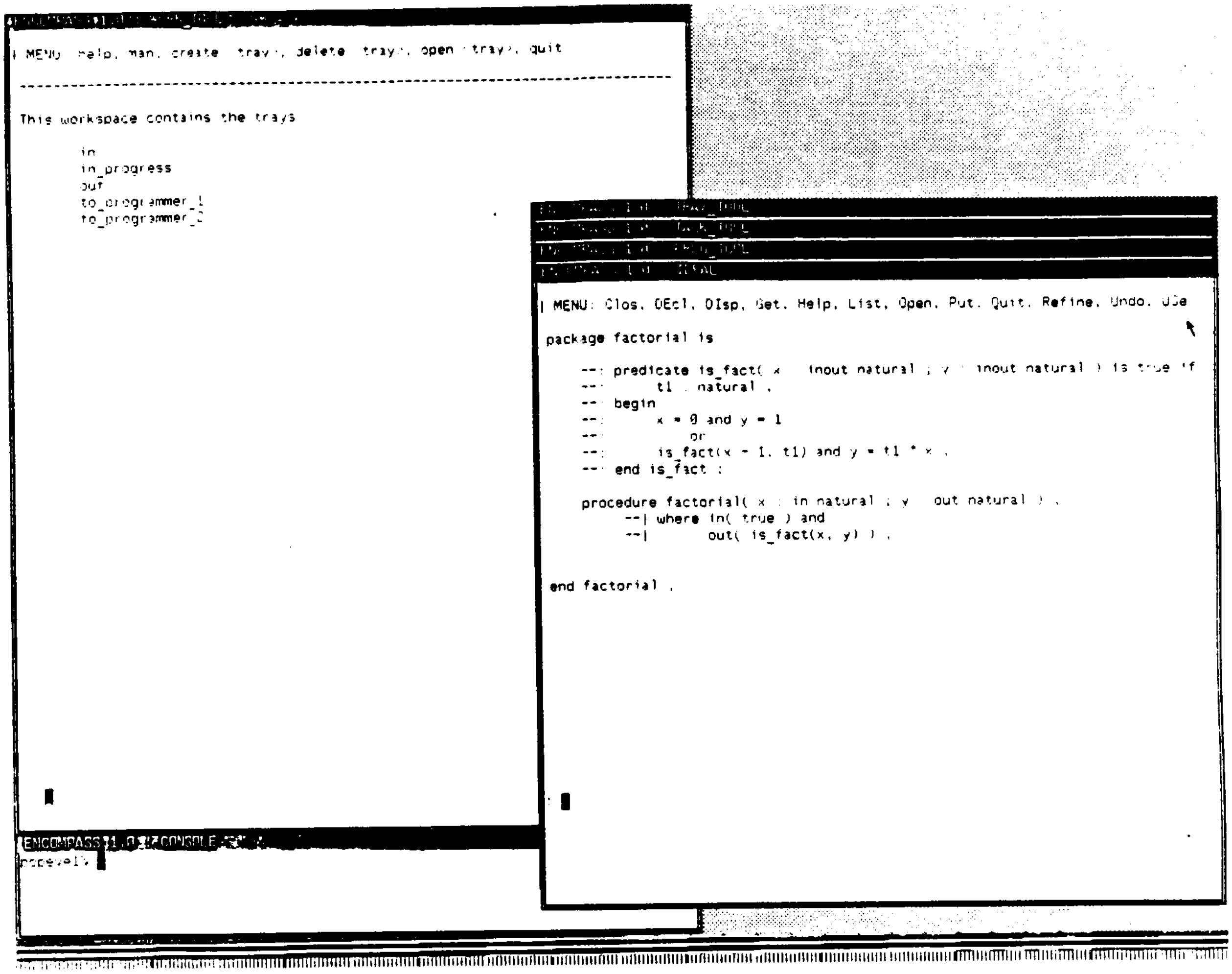
For our example, we will consider a programming team consisting of a leader and two programmers; there is a workspace for each member of the team. The team leader's workspace contains output trays to send assignments to the each of the programmers as well as an input tray in which he receives completed tasks. Each programmer's workspace contains an input tray in which he receives assignments from the leader and an output tray to facilitate the return of assignments to their originator. Assume that the team is assigned the task of developing a set of procedures to compute simple combinatoric quantities. The system is to be both validated by prototyping and formally verified. It will contain a procedure to calculate the factorial of a number as well as a procedure to compute the number of unique k-combinations of n items[2].

When the team leader receives the assingment by electronic mail, he creates a project library called *k_comb* in his in-progress tray. In the planning phase, the team leader consults with the customers and creates preliminary copies of two documents: the *system definition* and *project plan.* At this point, it is decided that the system will consist of two modules: one called *k_comb* and one called *factorial.* The team leader creates a *program object* containing two modules with these names; each module contains an empty symbol table and set of test cases. The team leader then *opens* the factorial module and uses ISLET to specify the procedure *factorial.*

Figure 4 shows the team leader's screen after completing the specification of *factorial.* The large window on the left of the screen gives the team leader access to his workspace, which contains the trays *in*, *in_progress*, *out*, *to_programmer_1*, and *to_programmer_2.* The small window on the left of the screen is to trap console messages that would disrupt the display. The windows on the right of the screen show the hierarchy of components through which the team leader accessed the *factorial* module. First the team leader opened the tray *in_progress* which contains the project library for the *k_comb* task; this created the window on the bottom of the stack which is labeled *TRAY_TOOL.* Next, he opened the project library, creating the window labeled *TASK_TOOL.* He then opened the program object to create the window labeled *PROG_TOOL*, and finally he invoked IDEAL on the factorial module to create the top window on the stack.

The top window shows the PLEASE specification of the *factorial* module. This specification defines a package *factorial*, which provides a procedure by the same name. In PLEASE, procedures are defined

[2] The number of k-combinations is equal to n!/(k!(n-k)!)



```
| MENU  help, man, create <tray>, delete <tray>, open <tray>, quit
--------------------------------------------------------------------------

This workspace contains the trays

        in
        in_progress
        out
        to_programmer_1
        to_programmer_2
```

```
ENCOMPASS 1.0 CONSOLE
toplevel%
```

```
| MENU: Clos, DEcl, DIsp, Get, Help, List, Open, Put, Quit, Refine, Undo, Use

package factorial is

    --: predicate is_fact( x : inout natural ; y : inout natural ) is true if
    --:       t1 : natural ;
    --: begin
    --:       x = 0 and y = 1
    --:          or
    --:       is_fact(x - 1, t1) and y = t1 * x ;
    --: end is_fact ;

    procedure factorial( x : in natural ; y : out natural ) ;
          --| where in( true ) and
          --|        out( is_fact(x, y) ) ;


end factorial ;
```

Figure 4. Team Leader's Screen After Specifying *factorial*

using pre- and post-conditions which are designated by *in(...)* and *out(...)* respectively. The pre-condition for a procedure specifies the conditions the input data must satisfy before procedure execution begins. The pre-condition for *factorial* is *true*; the type declarations for the parameters give all the requirements for the input. The post-condition for a procedure states the conditions the output data must satisfy after procedure execution has completed. The post-condition for *factorial* is *is_fact(x,y)*; the predicate *is_fact* must be true of the parameters to *factorial* after execution is complete.

The predicate *is_fact* is not pre-defined; it was developed by the team leader as *factorial* was specified. In PLEASE, a predicate syntactically resembles a procedure and may contain local type, variable, function or predicate definitions. At present, predicates are specified using Horn clauses: a subset of predicate logic which is also the basis for Prolog[22,26]. This simplifies translation from PLEASE to Prolog, but limits the expressive power of PLEASE. The predicate *is_fact* states that x factorial is equal to y if x equals zero and y equals one, or if x minus one factorial is equal to *t1* and y equals *t1* times x (in other words, *is_fact(x,y)* is true if $(x = 0 \wedge y = 1) \vee ((x\text{-}1)! = t1 \wedge y = t1 * x))$.

After *factorial* is specified, it is prototyped. From IDEAL, the team leader issues a command which automatically creates an executable prototype from the PLEASE specification. This prototype is compatible with the IDEAL test harness; the program produced reads x from input, calls *factorial*, and then writes y to output. From the test harness, input data can be edited, the prototype can be used to generate output, and the output can be manually checked for correctness. The team leader uses these tools to check that the factorial prototype performs correctly on simple test data. After *factorial* has been prototyped, the specification and prototyping processes are repeated for *k_comb*, which uses *factorial*.

After both modules are specified and prototyped, the validation phase begins. The prototype system is delivered to the customers for evaluation; it is subjected to a series of tests, and possibly installed for production use on a trail basis. The team leader consults with the customers to produce an updated set of documents, as well as a set of *acceptance tests*[37] which will be used to evaluate the final implementation. These tests are stored in a form compatible with the IDEAL test harness; the implementation can be run on pre-existing input and the results compared with those produced by the prototype. After the validation phase is complete, the refinement phase begins. The production of a verified implementation which passes the acceptance tests is the milestone for completion of this phase.

First, the implementation task is decomposed into sub-tasks that can be performed in parallel. It is decided that the implementation of *factorial* will be performed by the first programmer, while *k_comb* will be implemented by the second. The team leader creates two views of the project library; both provide access to all the documents produced in the development, but one provides access to *factorial* while the other provides access to *k_comb*. The team leader then transfers the first view to the tray labeled *to_programmer_1* in his workspace; this causes the view to appear in the first programmer's input tray. Similarly, the second view is sent to the second programmer.

Figure 5 shows the team leader's and programmer's workspaces after the transfers are complete. The team leader's workspace contains the project library, which contains two documents, the *system definition* and the *project plan*, as well as a program object containing the modules *factorial* and *k_comb*.

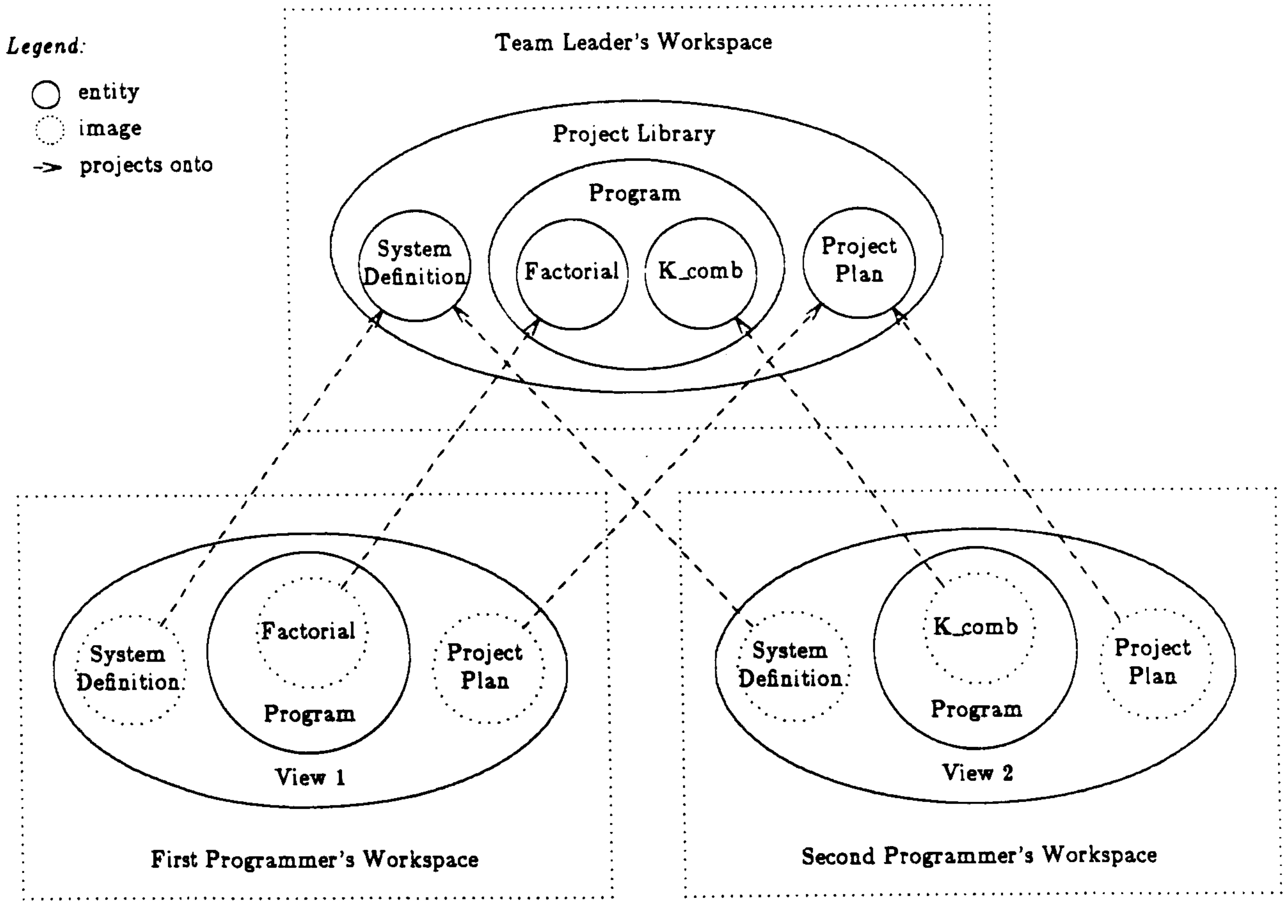


Figure 5. Different Views of the Project Library

The first programmer's workspace contains the first view, which contains an image of the *system definition*, the *project plan* and *factorial*; it does not provide access to *k_comb*. The view in the second programmer's workspace is similar, but gives access to *k_comb* and not *factorial*.

When the first programmer checks his input tray, he discovers the view of the project library; he can receive more information by electronic mail or in an auxiliary document. He then opens the view, the program object, and the factorial module. Using ISLET, the programmer then refines the specification of *factorial* into an implementation. As the refinement is performed, verification conditions are generated automatically. As the project plan calls for a formally verified implementation, the verification conditions are mechanically certified as the refinement is performed.

After the implementation is produced, the programmer uses the test harness to run the implementation on the acceptance tests produced in the validation phase. The milestone for completion of his assignment is the production of a formally verified implementation which passes the acceptance tests. When the milestone has been reached, the programmer transfers the view of the project library to his output tray; this causes the view to appear in the team leader's input tray. The second programmer follows a similar implement and verify, test, and transfer scenario with the *k_comb module*.

When the team leader discovers that both views are in his input tray, he knows the project should be complete. He checks to be sure that the milestone for the refinement phase has been reached; using tools in ENCOMPASS, he certifies that the implementations are formally verified and pass the acceptance tests. When the milestone has been verified, the project is delivered to the customers. At this point the project is complete, and can be transferred to a file tray for long term storage.

5. System Status

The SAGA project has been active at the University of Illinois at Urbana-Champaign for over five years. ENCOMPASS has been under development since the summer of 1984; a prototype implementation has been operational since the summer of 1986. This prototype includes simple implementations of the project management and configuration control systems, as well as IDEAL. It is written in a combination of C, Csh, Prolog and Ada. The subset of PLEASE currently implemented includes the *if*, *while*, and assignment statments, as well as procedure calls with *in*, *out* or *in out* parameters. The language now supports a small, fixed set of types including natural numbers, lists, booleans and characters. ENCOMPASS has been used to develop small programs, including the example given in this paper.

6. Summary

ENCOMPASS is an integrated environment being constructed by the SAGA project to support incremental software development in a manner similar to the Vienna Development Method. In ENCOMPASS, software is modeled as entities which may have relationships between them. These entities can be structured into complex hierarchies which may be seen through different views. The configuration management system stores and structures the components developed and used in a project, as well as providing a mechanism for controlling access. The project management system implements a milestone-based policy using the mechanism provided. In ENCOMPASS, software is first specified using a combination of natural language and PLEASE, a wide-spectrum, executable specification and design language. Components specified in PLEASE are then incrementally refined into components written in Ada; this process can be viewed as the construction of a proof in the Hoare calculus. Each refinement is verified before another is applied; therefore, the final components produced by the development satisfy the original specifications. PLEASE specifications may be used in formal proofs of correctness; they may also be transformed into executable prototypes which can be used in the validation and design processes. ENCOMPASS provides automated support for all aspects of software development using PLEASE. A prototype implementation of ENCOMPASS has been constructed at the University of Illinois at Urbana-Champaign. We believe the use of ENCOMPASS will enhance the software development process.

7. References

1. "Software Configuration Management", Standard 828-1983, IEEE Computer Society, Los Angeles, California, 1983.
2. *Special Issue on the Gandalf Environment.* **Journal of Systems and Software** (May, 1985) vol. 5, no. 2.
3. *Proceedings of the International Workshop on the Software Process and Software Environments.* **Software Engineering Notes** (August 1986) vol. 11, no. 4.
4. Agnarsson, Snorri and M. S. Krishnamoorthy. *Towards a Theory of Packages.* **Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments** (June, 1985) pp. 117-130.
5. Balzer, Robert. *A 15 Year Perspective on Automatic Programming.* **IEEE Transactions on Software Engineering** (November 1985) vol. SE-11, no. 11, pp. 1257-1268.

6. Balzer, Robert, Thomas E. Cheatham and Cordell Green. *Software Technology in the 1990's: Using a New Paradigm.* **IEEE Computer** (November 1983) vol. 16, no. 11, pp. 39–45.

7. Barstow, David R. *On Convergence Toward a Database of Program Transformations.* **ACM Transactions on Programming Languages and Systems** (January 1985) vol. 7, no. 1, pp. 1–9.

8. Bersoff, Edward H. *Elements of Software Configuration Management.* **IEEE Transactions on Software Engineering** (January 1984) vol. SE–10, no. 1, pp. 79–87.

9. Bersoff, E. H. *Software Configuration Management: A Tutorial.* **IEEE Computer** (January, 1979).

10. Beshers, George M. and Roy H. Campbell. *Maintained and Constructor Attributes.* **Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments** (June 1985) pp. 34–42.

11. Bjorner, D. and Cliff B. Jones. **Formal Specification and Software Development**. Prentice–Hall, Englewood Cliffs, N.J., 1982.

12. Bloomfield, Robin E. and Peter K. D. Froome. *The Application of Formal Methods to the Assessment of High Integrity Software.* **IEEE Transactions on Software Engineering** (September 1986) vol. SE–12, no. 9, pp. 988–993.

13. Blum, B. I. *The Life-Cycle - A Debate Over Alternative Models.* **Software Engineering Notes** (October 1982) vol. 7, pp. 18–20.

14. Britcher, Robert N. and James J. Craig. *Using Modern Design Practices to Upgrade Aging Software Systems.* **IEEE Software** (May 1986) vol. 3, no. 3, pp. 16–24.

15. Britcher, Robert N. and Allan R. Moore. *Increased Productivity Through the Use of Software Engineering in an Industrial Environment.* **Proceedings of the IEEE Computer Software and Applications Conference** (1981) pp. 199–205.

16. Buckle, J. K. **Software Configuration Management**. Scholium International Inc., Great Neck, N.Y., 1982.

17. Buxton, J. N. and V. Stenning. "Requirements for ADA Programming Support Environments, *Stoneman*", U.S. Dept. Defense, 1980.

18. Campbell, Roy H. and Robert B. Terwilliger,. *The SAGA Approach to Automated Project Management.* In: **International Workshop on Advanced Programming Environments**, Lynn R. Carter, ed. Springer–Verlag Lecture Notes in Computer Science, New York, 1986, pp. 145–159.

19. Campbell, Roy H. *SAGA: A Project to Automate the Management of Software Production Systems.* In: **Software Engineering Environments**, Ian Sommerville, ed. Peter Perigrinus Ltd, 1986.

20. Campbell, Roy H. and Peter A. Kirslis. *The SAGA Project: A System for Software Development.* **Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments** (April 1984) pp. 73–80.

21. Campbell, Roy H. and Paul G. Richards. *SAGA: A system to automate the management of software production.* **Proceedings of the National Computer Conference** (May 1981) pp. 231–234.

22. Chang, Chin–Liang and Richard Char–Tung Lee. **Symbolic Logic and Mechanical Theorem Proving**. Academic Press, New York, 1973.

23. Cheatham, Thomas E., Glenn H. Holloway and Judy A. Townley. *Program Refinement By Transformation.* **Proceedings of the 5th International Conference on Software Engineering** (1981) pp. 430–437.

24. Chen, Peter Pin–Shan. *The Entity-Relationship Model - Toward a Unified View of Data.* **ACM Transactions on Database Systems** (March 1976) vol. 1, no. 1, pp. 9–36.

25. ——. *ER - A Historical Perspective and Future Directions.* In: **The Entity–Relationship Approach to Software Engineering**, S. Jajodia C. G. Davis P. A. Ng and R. T. Yeh, ed. Elsevier Science, 1983, pp. 71–77.

26. Clocksin, W. F. and C. S. Mellish. **Programming in Prolog**. Springer–Verlag, New York, 1981.

27. Cottam, I. D. *The Rigorous Development of a System Version Control Program.* **IEEE Transactions on Software Engineering** (March 1984) vol. SE–10, no. 3, pp. 143–154.

28. Davis, Ruth E. *Runnable Specification as a Design Tool.* In: **Logic Programming**, K. L. Clark and S. A. Tarnlund, ed. Academic Press, London, 1982, pp. 141–149.

29. Davis, Carl G. and Charles R. Vick. *The Software Development System.* In: **Tutorial: Automated Tools for Software Engineering**, Edward Miller, ed. IEEE Computer Society, New York, 1979, pp. 138–153.

30. Defense, U. S. Dept. **Reference Manual for the ADA Programming Language** ANSI/MIL–STD–1815A–1983. Springer–Verlag, New York, 1983.

31. DeMillo, R. A., R. J. Lipton and A. J. Perlis. *Social Processes and Proofs of Theorems.* **Communications of the ACM** (May, 1979) vol. 22, no. 5, pp. 271–280.

32. Dijkstra, E. W. *Structured Programming.* In: **Software Engineering Principles**, J. N. Buxton and B. Randall, ed. NATO Science Committee, Brussels, Belgium, 1970.

33. ——. **A Discipline of Programming**. Prentice Hall, Englewood Cliffs, New Jersey, 1976.

34. Dolotta, T. A. and J. R. Mashey. *An Introduction to the Programmer's Workbench.* In: **Tutorial: Automated Tools for Software Engineering**, Edward Miller, ed. IEEE Computer Society, New York, 1979, pp. 154-158.

35. Estublier, J., S. Ghoul and S. Krakowiak. *Preliminary Experience with a Configuration Control System for Modular Programs.* **Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments** (April 1984) pp. 149-156.

36. Fagan, Michael E. *Advances in Software Inspections.* **IEEE Transactions on Software Engineering** (July 1986) vol. SE-12, no. 7, pp. 744-751.

37. Fairley, Richard. **Software Engineering Concepts**. McGraw-Hill, New York, 1985.

38. Gannon, John, Paul McMullin and Richard Hamlet. *Data-Abstraction Implementation, Specification, and Testing.* **ACM Transactions on Programming Languages and Systems** (July 1981) vol. 3, no. 3, pp. 211-223.

39. Gehani, Narain and Andrew D. McGettrick (eds.). **Software Specification Techniques**. Addison Wesley, Reading, Massachusetts, 1986.

40. Goguen, Joseph A. *Reusing and Interconnecting Software Components.* **Computer** (February 1986) vol. 19, no. 2, pp. 16-28.

41. Goguen, Joseph and Jose Meseguer. *Rapid Prototyping in the OBJ Exececutable Specification Laguage.* **Software Engineering Notes** (December 1982) vol. 7, no. 5, pp. 75-84.

42. Goguen, Joseph, James Thatcher and Eric Wagner. *An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types..* In: **Current Trends in Programming Methodology, IV**, Raymond Yeh, ed. Prentice-Hall, London, 1978, pp. 80-149.

43. Goldberg, A. **Smalltalk-80: The Interactive Programming Environment**. Addison-Wesley, Reading, MA, 1984.

44. Gries, David. **The Science of Programming**. Springer-Verlag, New York, 1981.

45. Gunther, R. **Management Methodology for Software Product Engineering**. Wiley Interscience, New York, 1978.

46. Guttag, J. V. and J. J. Horning. *The Algebraic Specification of Abstract Data Types.* **Acta Informatica** (1978) vol. 10, pp. 27-52.

47. Guttag, John V., James J. Horning and Jeannette M. Wing. *The Larch Family of Specification Languages.* **IEEE Software** (September 1985) vol. 2, no. 5, pp. 24-36.

48. Guttag, John V., Ellis Horowitz and David R. Musser. *Abstract Data Types and Software Validation.* **Communications of the ACM** (December 1978) vol. 21, no. 12, pp. 1048-1063.

49. Hammerslag, David H., Samuel N. Kamin and Roy H. Campbell. *Tree-Oriented Interactive Processing with an Application to Theorem-Proving.* **Proceedings of the Second ACM/IEEE Conference on Software Development Tools, Techniques, and Alternatives** (December, 1985).

50. Henderson, Peter. *Functional Programming, Formal Specification, and Rapid Prototyping.* **IEEE Transactions on Software Engineering** (February, 1986) vol. SE-12, no. 2, pp. 241-250.

51. Hoare, C. A. R. *An Axiomatic Basis for Computer Programming.* **Communications of the ACM** (October 1969) vol. 12, no. 10, pp. 576-580.

52. ——. *Proof of Correctness of Data Representations.* **Acta Informatica** (1972) vol. 1, pp. 271-281.

53. Horowitz, Ellis and Ronald C. Williamson. *SODOS: A Software Documentation Support Environment - Its Definition.* **IEEE Transactions on Software Engineering** (August 1986) vol. SE-12, no. 8, pp. 849-859.

54. Howden, William E. *Contemporary Software Development Environments.* **Communications of the ACM** (May 1982) vol. 25, no. 5, pp. 318-329.

55. Huff, Karen E. *A Database Model for Effective Configuration Management in the Programming Environment.* **Proceedings IEEE 5th International Conference on Software Engineering, San Diego, CA** (March 1981) pp. 54-61.

56. Jackson, M. **System Development**. Prentice-Hall, Englewood Cliffs, N.J., 1983.

57. Jones, Cliff B. *Constructing a Theory of a Data Structure as an Aid to Program Development.* **Acta Informatica** (1979) vol. 11, pp. 119-137.

58. ——. **Software Development: A Rigorous Approach**. Prentice-Hall International, Engelwood Cliffs, N.J., 1980.

59. ——. *Tentative Steps Toward a Development Method for Interfering Programs.* **ACM Transactions on Programming Languages and Systems** (October 1983) vol. 5, no. 4, pp. 596-619.

60. Kamin, Samuel. *Final Data Types and Their Specification.* **ACM Transactions on Programming Languages and Systems** (January 1983) vol. 5, no. 1, pp. 97-121.

61. Kamin, S. N., S. Jefferson and M. Archer. *The Role of Executable Specifications: The FASE System.* **Proceedings of the IEEE Symposium on Application and Assessment of Automated Tools for Software Development** (November 1983).

62. Kemmerer, Richard A. *Testing Formal Specifications to Detect Design Errors.* **IEEE Transactions on Software Engineering** (January 1985) vol. SE-11, no. 1, pp. 32-43.

63. Kirslis, Peter A., Robert B. Terwilliger and Roy H. Campbell. *The SAGA Approach to Large Program Development in an Integrated Modular Environment.* **Proceedings of the GTE Workshop on Software Engineering Environments for Programming-in-the-Large** (June 1985) pp. 44-53.

64. Kowalski, Robert. *Logic as a Computer Language.* In: **Logic Programming**, K. L. Clark and S. A. Tarnlund, ed. Academic Press, London, 1982, pp. 3-16.

65. Kruchten, Philippe, Edmond Schonberg and Jacob Schwartz. *Software Prototyping Using the SETL Programming Language.* **IEEE Software** (October 1984) vol. 1, no. 4, pp. 66-75.

66. Laff, Mark R. and Brent Hailpern. *SW 2 - An Object-based Programming Environment.* **Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments** (June, 1985) pp. 1-11.

67. Lampson, Butler W. and Eric E. Schmidt. *Organizing Software in a Distributed Environment.* **SIGPLAN Notices** (June 1983) vol. 18, no. 6, pp. 1-13.

68. ——. *Practical Use of a Polymorphic Applicative Language.* **Proceedings of the 10th ACM Symposium on Principles of Programming Languages** (January 1983) pp. 237-255.

69. Lauer, H. C. and E. H. Satterthwaite. *The Impact of Mesa on System Design.* **Proceedings of the 4th IEEE International Conference on Software Engineering** (September 1979) pp. 174-182.

70. Lehman, M. M., V. Stenning and W. M. Turski. *Another Look at Software Design Methodology.* **Software Engineering Notes** (April 1984) vol. 9, no. 2, pp. 38-53.

71. Lewis, Brian T. *Experience with a System for Controlling Software Versions in a Distributed Environment.* **Symposium on Application and Assessment of Automated Tools for Software Development** (November 1983) pp. 210-219.

72. Liskov, Barbara, Alan Snyder, Russll Atkinson and Craig Schaffert. *Abstraction Mechanisms in CLU.* **Communications of the ACM** (August 1977) vol. 20, no. 8, pp. 564-576.

73. Loeckx, Jacques and Kurt Sieber. **The Foundations of Program Verification**. John Wiley & Sons, New York, 1984.

74. Meyers, G. J. **The Art of Software Testing**. John Wiley & Sons, New York, 1979.

75. Mills, Harlan D. and Richard C. Linger. *Data Structured Programming: Program Design without Arrays and Pointers.* **IEEE Transactions on Software Engineering** (February 1986) vol. SE-12, no. 2, pp. 192-197.

76. Musser, David R. *Abstract Data Type Specification in the AFFIRM System.* **IEEE Transactions on Software Engineering** (January 1980) vol. SE-6, no. 1, pp. 24-32.

77. Neighbors, James M. *The Draco Approach to Constructing Software from Reusable Components.* **IEEE Transactions on Software Engineering** (September 1984) vol. SE-10, no. 5, pp. 564-574.

78. Ossher, Harold L. *A New Program Structuring Mechanism Based on Layered Graphs.* **Proceedings of the 11th ACM Symposium on the Principles of Programming Languages** (January 1984) pp. 11-22.

79. Osterweil, Leon J. *Toolpack - An Experimental Software Development Environment Research Project.* **IEEE Transactions on Software Engineering** (November 1983) vol. SE-9, no. 6, pp. 673-685.

80. Parent, Christine and Stefano Spaccapietra. *An Algebra for a General Entity-Relationship Model.* **IEEE Transactions on Software Engineering** (July 1985) vol. SE-11, no. 7, pp. 634-643.

81. Parnas, D. L. *On the Criteria To Be Used in Decomposing Systems into Modules.* **Communications of the ACM** (December 1972) vol. 15, no. 12, pp. 1053-1058.

82. ——. *The Use of Precise Specifications in the Development of Software.* **IFIP Congress Proceedings** (1977) pp. 861-867.

83. Partsch, H. and R. Steinbruggen. *Program Transformation Systems.* **Computing Surveys** (September 1983) vol. 15, no. 3, pp. 199-236.

84. Ramamoorthy, C. V., Vijay Garg and Atull Prakash. *Programming in the Large.* **IEEE Transactions on Software Engineering** (July 1986) vol. SE-12, no. 7, pp. 769-783.

85. Reps, Thomas and Bowen Alpern. *Interactive Proof Checking.* **Proceedings of the 11th ACM Symposium on the Principles of Programming Languages** (January 1984) pp. 36-45.

86. Richardson, Debra J. and Lori A. Clarke. *Partition Analysis: A Method Combining Testing and Verification.* **IEEE Transactions on Software Engineering** (December, 1985) vol. SE-11, no. 12, pp. 1477-1490.

87. Ross, Douglas T. *Structured Analysis (SA): A Language for Communicating Ideas.* **IEEE Transactions on Software Engineering** (January 1977) vol. SE-3, no. 1, pp. 16-34.

88. Shaw, R. C., P. N. Hudson and N. W. Davis. *Introduction of A Formal Technique into a Software Development Environment (Early Observations).* **Software Engineering** Notes (April 1984) vol. 9, no. 2, pp. 54-79.

89. Shigo, Osamu, Yoshio Wada, Yuichi Terashima, Kanji Iwamoto and Takashi Nishimura. *Configuration Control for Evolutional*

Software Products. **Proceedings of the 6th IEEE International Conference on Software Engineering** (September 1982) pp. 68–75.

90. Smith, Douglas R., Gordon B. Kotik and Stephen J. Westfold. *Research on Knowledge-Based Software Environments at Kestrel Institute.* **IEEE Transactions on Software Engineering** (November 1985) vol. SE-11, no. 11, pp. 1278–1295.

91. Smith, John M. and Diane C. P. Smith. *Database Abstractions: Aggregation.* **Communications of the** ACM (June, 1977) vol. 20, no. 6, pp. 405–413.

92. Smith, John Miles and Diane C. P. Smith. *Database Abstractions: Aggregation and Generalization.* ACM **Transactions on Database Systems** (June 1977) vol. 2, no. 2, pp. 105–133.

93. Standish, Thomas A. and Richard N. Taylor. *Arcturus: A Prototype Advanced ADA Programming Environment.* **Proceedings of the** ACM **SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments** (April 1984) pp. 57–64.

94. Sweet, Richard E. *The Mesa Programming Environment.* ACM **SIGPLAN 85 Symposium on Language Issues in Programming Environments** (June 1985) pp. 216–229.

95. Swinehart, Daniel C., Polle T. Zellweger and Robert B. Hagmann. *The Structure of Cedar.* ACM **SIGPLAN 85 Symposium on Language Issues in Programming Environments** (June 1985) pp. 230–244.

96. Teitelbaum, Tim and Thomas Reps. *The Cornell Program Synthesizer: A Syntax-Directed Programming Environment.* **Communications of the** ACM (September 1981) vol. 24, no. 9, pp. 563–573.

97. Teitelman, W. and L. Masinter. *The Interlisp Programming Environment.* **Computer** (April 1981) vol. 14, no. 4, pp. 25–33.

98. Terwilliger, Robert B. and Roy H. Campbell. *ENCOMPASS: a SAGA Based Environment for the Composition of Programs and Specifications.* **Proceedings of the 19th Hawaii International Conference on System Sciences** (January 1986) pp. 436–447.

99. ——. "PLEASE: Executable Specifications for Incremental Software Development", Report No. UIUCDCS-R-86-1295, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1986.

100. ——. *PLEASE: Predicate Logic based ExecutAble SpEcifications.* **Proceedings of the 1986** ACM **Computer Science Conference** (February, 1986) pp. 349–358.

101. Tichy, Walter F. *Software Development Control Based on Module Interconnection.* **Proceedings IEEE 4th International Conference on Software Engineering** (1979) pp. 29–41.

102. ——. *Design, Implementation, and Evaluation of a Revision Control System.* **Proceedings of the 6th IEEE International Conference on Software Engineering** (September 1982) pp. 58–67.

103. Tseng, Jine S., Boleslaw Szymanski, Yuan Shi and Noah S. Prywes. *Real-Time Software Life Cycle with the Model System.* **IEEE Transactions on Software Engineering** (February 1986) vol. SE-12, no. 2, pp. 358–373.

104. Warren, Sally, Bruce E. Martin and Charles Hoch. *Experience with A Module Package in Developing Production Quality PASCAL Programs.* **Proceedings of the 6th International Conference on Software Engineering** (September 1982) pp. 246–253.

105. Wegner, Peter. **Programming with Ada: an Introduction by Means of Graduated Examples.** Prentice-Hall, Englewood Cliffs, New Jersey, 1980.

106. Weinberg, Gerald M. and Daniel P. Freedman. *Reviews, Walkthroughs, and Inspections.* **IEEE Transactions on Software Engineering** (January 1984) vol. SE-10, no. 1, pp. 68–72.

107. Wirth, Niklaus. *Program Development by Stepwise Refinement.* **Communications of the** ACM (April 1971) vol. 14, no. 4, pp. 221–227.

108. Wolf, Alexander L., Lori A. Clarke and Jack C. Wileden. *Ada-Based Support for Programming-in-the-Large.* **IEEE Software** (March, 1985) vol. 2, no. 2, pp. 58–71.

109. Wulf, William A., Ralph L London and Mary Shaw. *An Introduction to the Construction and Verification of Alphard Programs.* **IEEE Transactions on Software Engineering** (December 1976) vol. SE-2, no. 4, pp. 253–265.

110. Yourdon, E. and L. L. Constantine. **Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.** Prentice-Hall, Englewood Cliffs, N.J., 1979.

111. Yuasa, Taiichi and Reiji Nakajima. *IOTA: A Modular Programming System.* **IEEE Transactions on Software Engineering** (February 1985) vol. SE-11, no. 2, pp. 179–187.

112. Zave, Pamela. *The Operational Versus the Conventional Approach to Software Development.* **Communications of the** ACM (February 1984) vol. 27, no. 2, pp. 104–118.

113. Zave, Pamela and William Schnell. *Salient Features of an Executable Specification Language and Its Environment.* **IEEE Transactions on Software Engineering** (February 1986) vol. SE-12, no. 2, pp. 312–325.

114. Zucker, Sandra. *Automating the Configuration Management Process.* **Proceedings SOFTFAIR, Arlington, Virginia** (July, 1983) pp. 164–172.

Appendix B

PLEASE: Executable Specifications for Incremental Software Development

Robert B. Terwilliger
Roy H. Campbell

Report No. UIUCDCS-R-86-1295
Department of Computer Science
1304 W. Springfield Ave.
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
217-333-4428

Preprint October 7, 1986

This research is supported by NASA Grant NAG 1-138

Appendix B

PLEASE: Executable Specifications for Incremental Software Development[*]

Robert B. Terwilliger
Roy H. Campbell

Department of Computer Science
University of Illinois at Urbana-Champaign
252 Digital Computer Laboratory
1304 West Springfield Avenue
Urbana, IL 61801
(217) 333-4428

[*]This research is supported by NASA grant NAG 1-138.

Abstract

PLEASE is an executable specification language which supports software development by incremental refinement. PLEASE is part of the ENCOMPASS environment which provides automated support for all aspects of the development process. Software components are first specified using a combination of conventional programming languages and predicate logic. These abstract components are then incrementally refined into components in an implementation language. Each refinement is verified before another is applied; therefore, the final components produced by the development satisfy the original specifications. PLEASE allows a procedure or function to be specified using pre- and post-conditions, a data type to have an invariant, and an implementation to be completely annotated. PLEASE specifications may be used in proofs of correctness; they may also be transformed into prototypes which use Prolog to "execute" pre- and post-conditions. We believe the early production of executable prototypes will enhance the development process.

1. Introduction

It is widely acknowledged that producing correct software is both difficult and expensive. To help remedy this situation, many methods for specifying[27,30,32,33,45,58,60] and verifying[34,38,39,43,54,74] software have been developed. The SAGA (Software Automation, Generation and Administration) project is investigating both the formal and practical aspects of providing automated support for the full range of software engineering activities[8,12-15,35,48,67-69]. PLEASE is a language being developed by the SAGA group to support the specification, prototyping, and incremental development of software components[69]. PLEASE is part of the ENCOMPASS environment which provides support for all aspects of the software development process[67,68]. In this paper we describe the development methodology for which PLEASE was created, give an example of development using the language, and describe the methods used to produce prototypes from PLEASE specifications.

Appendix B

A *life-cycle model* describes the sequence of distinct stages through which a software product passes during its lifetime[25]. There is no single, universally accepted model of the software life-cycle[3,5,11,76]. The stages of the life-cycle generate *software components*, such as code written in programming languages, test data or results, and many types of documentation. In many models, a *specification* of the system to be built is created early in the life-cycle; as components are produced they are *verified*[25] for correctness with respect to this specification. The specification is *validated*[25] when it is shown to satisfy the customers requirements.

Producing a valid specification is a difficult task. The users of the system may not really have defined what they want, and they may be unable to communicate their desires to the development team. If the specification is in a formal notation it may be an ineffective medium for communication with the customers, but natural language specifications are notoriously ambiguous and incomplete. *Prototyping* and the use of executable specification languages have been suggested as partial solutions to these problems[20,28,37,46,47,51,70,77]. Providing the customers with prototypes for experimentation and evaluation early in the development process should increase customer/developer communication and enhance the validation and design processes.

Even given a validated specification, it may be difficult to determine if an implementation is correct. Many techniques for verifying the correctness of implementations have been proposed. For example, *testing* can be used to check the operation of an implementation on a representative set of input data[26,56]. In a *technical review* process, the specification and implementation are inspected, discussed and compared by a group of knowledgeable personnel[24,72]. If the specification is in a suitable notation, formal methods can also be used to verify the correctness of an implementation[34,38,39,43,54,74]. Many feel that no one technique alone can ensure the production of correct software[22,23]; therefore, methods which combine a number of techniques have been proposed[64].

To help manage the complexity of software design and development, methods which combine standard representations, intellectual disciplines, and well defined techniques have been proposed[4,31,41,43,57,75]. For example, it has been suggested that *top-down development* can help control the complexity of program construction. By using *stepwise refinement* to create a concrete implementation from an abstract specification, we divide the decisions necessary into smaller, more comprehensible groups. Others have suggested that the development process be viewed as a sequence of *transformations* between different, but somehow equivalent, specifications[5,6,17,52,59,61].

The Vienna Development Method (VDM) supports the top-down development of software specified in a notation suitable for formal verification[9,10,19,42–44,65]. In this method, software is first written in a language combining elements from conventional programming languages and mathematics. A procedure or function may be specified using *pre-* and *post-conditions* written in predicate logic; similarly, a data type may have an *invariant*. These *abstract components* are then incrementally refined into components in an implementation language. The refinements are performed one at a time, and each is verified before another is applied; therefore, the final components produced by the development satisfy the original specification.

PLEASE is a wide-spectrum, executable specification language which supports a development method similar to VDM for software written in a *base language*; at present we are using Ada[1] as the base language. PLEASE extends the base language so that a procedure or function may be specified with pre- and post-conditions, a data type may have an invariant, and an implementation may be completely annotated. PLEASE specifications may be used in proofs of correctness; they also may be transformed into prototypes which use Prolog[18,50] to "execute" pre- and post-conditions, and may interact with other modules written in the base language. We believe that the early production of executable prototypes for experimentation and evaluation will enhance the software development process.

In section two of this paper, we describe the development methodology PLEASE was designed to support and in section three, we give an example of software development using PLEASE. First we present an example specification and describe how it may be used to derive an executable prototype. Then we show a refinement of this specification and discuss the process of verifying that the refined specification satisfies the original. In section four, we give an example of data type specification in PLEASE and in sec-

[1] Ada is a trademark of the US Government, Ada Joint Program Office.

tion five, we describe the current status of the system. In section six, we summarize the impact of using the PLEASE approach in software development.

2. Incremental Software Development

Figure 1 shows the life-cycle model PLEASE was designed to support; different perspectives are given in[13,67,68]. In our model, a *customer* requests that a system be constructed by the development team. In the *requirements definition phase*, the functions and properties of the software to be produced by the development are determined[25]. A *systems analyst* produces a *software requirement specification*[25], which precisely describes the attributes of the software to be produced. In our model, software requirements specifications include components specified in PLEASE. PLEASE specifications describe only the function of a component, not its performance, robustness or reliability. These other qualities are specified using natural language or other formalisms.

Although a software system may be shown to meet its specification, this does not necessarily imply that the system satisfies the customers' requirements. The *validation phase* attempts to show that any system which satisfies the specification will also satisfy the customers' requirements, that is, that the requirements specification is valid. If not, then the requirements specification should be corrected before the development proceeds any further. In this phase the systems analyst interacts with the users to produce the *system validation summary*[67], which describes the customers' evaluation of the software requirement specification.

To aid in the validation process, the PLEASE components in the specification may be transformed into executable prototypes which satisfy the specification. These prototypes may be used in interactions with the customers; they may be subjected to a series of tests, be delivered to the customers for experimentation and evaluation, or be installed for production use on a trial basis. The use of prototypes increases customer/developer communication and enhances the validation process. If it is found that the specification does not satisfy the customers, then it is revised, new prototypes are produced, and the validation process is reinitiated; this cycle is repeated until a validated specification is produced.

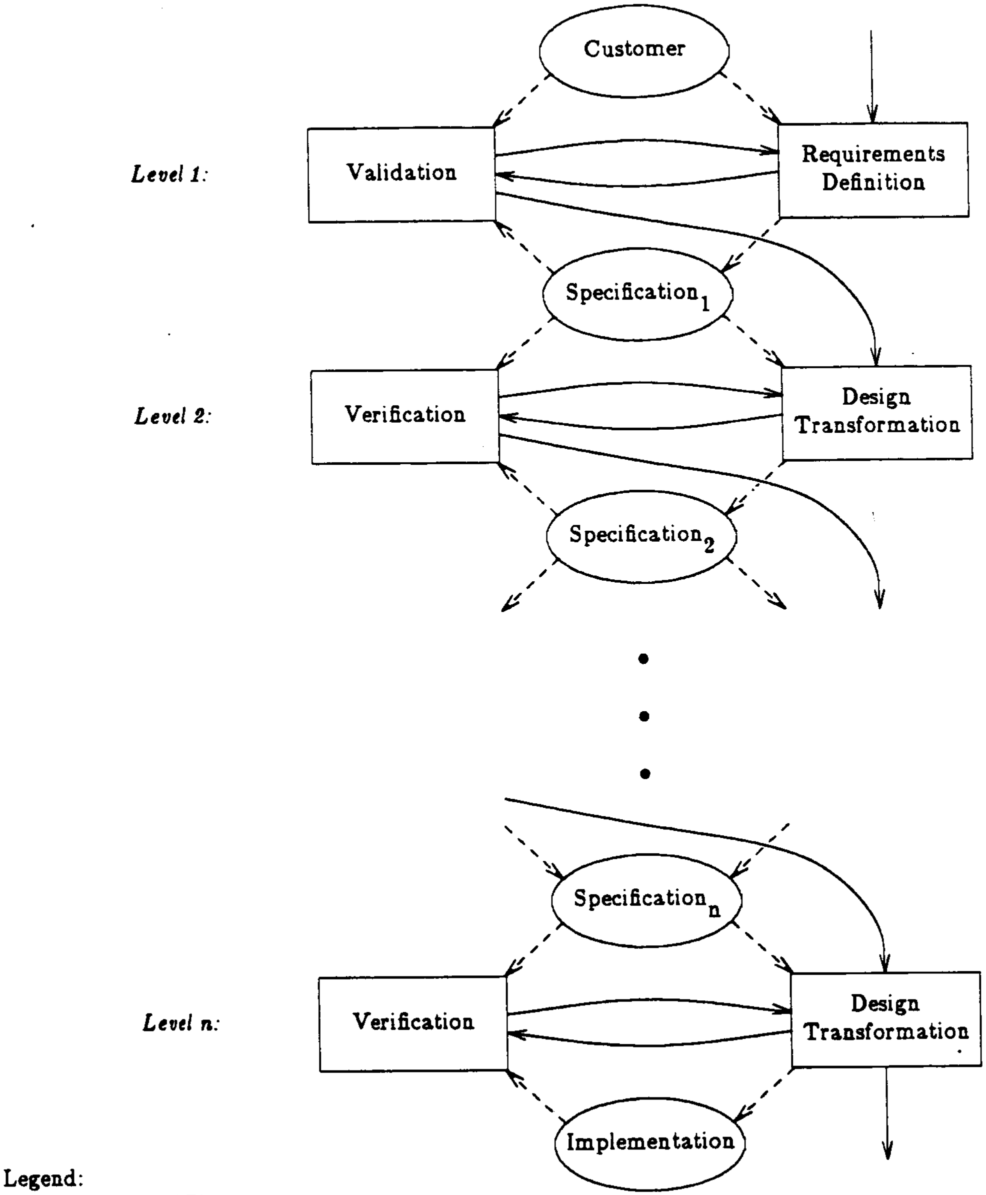


Figure 1. Program Development Model

In general, this process does not guarantee that the specification is valid. The fact that the prototype does satisfy the customers means only that at least one implementation which satisfies the specification is acceptable. For example, the post-condition for a procedure may hold true for an infinite number of values while the prototype will only return one. We say the specification of a component is *complete* if, for any input state, it is satisfied by only one output state. Although in some cases it is possible to require and verify that the specification of a component is complete, this is difficult in practice. We believe that while prototypes enhance the validation process, they do not replace communication with the customers and review of the specification.

When the validation phase is complete, the specification undergoes a refinement, or *design transformation*, in which more of the structure of the system is defined and implemented. This phase produces a *software design specification*[25], which provides a record of the design decisions made during the transformation. During the transformation, prototypes produced from PLEASE specifications may be used in experiments performed to guide the design process. The design transformation may produce annotated components in the base language as well as an updated requirements specification. Components which have been implemented need not be refined further, but components which are only specified will undergo further refinements until a complete implementation is produced.

Although a new specification has been created, its relationship to the original is unknown. Before further refinements are performed, a *verification phase* must show that any implementation which satisfies the lower level specification will also satisfy the upper level one. In our model, this is accomplished using a combination of testing, technical review, and formal verification. PLEASE specifications enhance the verification of system components using either testing or proof techniques. The specification of a component can be transformed into a prototype; this prototype may be used as a test oracle against which the implementation can be compared. Since the specification is formal, proof techniques may be used which range from a very detailed, completely formal proof using mechanical theorem proving, to a development "annotated" with unproven verification conditions. PLEASE provides a framework for the *rigorous*[43] development of programs. Although detailed mechanical proofs are not required at every step, the framework is present so that they can be constructed if necessary. Parts of a project may use detailed mechanical verification while other, less critical parts may be handled using less expensive techniques.

The life-cycle supported by PLEASE can be viewed as a sequence of transformations between different *specification levels*. On level one, the requirements definition phase transforms the customers desires into an initial, abstract specification. Also on level one, the correctness of this transformation is determined by the validation phase. On level two, the specification produced on level one undergoes a design transformation, the correctness of which is determined by a verification phase. All the remaining levels take the specification produced by the next higher level as input, and transform it into a more concrete form. The most concrete components are the annotated implementations, which are produced on the lowest level.

A somewhat more complex model might view the refinement process as a search through a space of possible implementations. A given specification can have a large number of correct implementations; these can be structured as a tree. In this tree, each interior node represents a specification and each leaf node represents a correct implementation. At any time, the development is located at a given node. A design decision chooses an arc which leads from a specification to a new specification or implementation. The goal of the refinement process is to search this tree for an acceptable implementation. An acceptable implementation would not only be correct, but would have performance and other characteristics which satisfy the users. In an actual refinement, some paths from a given specification will not lead to acceptable implementations; therefore, the refinement process may have to *backtrack* to find a solution. If an implementation is found inadequate, design decisions must be undone until the decision which caused the problem has been reversed. At this point a correct design decision can be made and, if possible, the rest of the development can be "replayed"[73].

In our model, each design transformation can be decomposed into a number of *atomic transformations*; if each atomic transformation is correct then so is the design transformation. Each design transformation is verified before another is applied; this allows errors in the specification and design processes to be detected and corrected sooner and at lower cost. However, a number of atomic transformations may be performed before any are verified; verifying each atomic transformation before the next is applied would be prohibitively expensive. Instead, the information necessary to verify each atomic transformation is recorded for use in the corresponding verification phase; at that time, they are verified using an appropriate method.

To clarify our model further and show how PLEASE specifications enhance the development process, we will consider an example of software development. We will follow the development through requirements definition, validation of the original specification, a single design transformation, and verification of the refinement.

3. An Example of Software Development

Assume that a customer needs a component which sorts a list of natural numbers. The component should take a possibly unsorted list as input and produce a sorted list which is a permutation of the original as output. A pre-existing component implementing lists of naturals is to be re-used. In the requirements definition phase, the customer discusses his needs with the systems analyst and a requirements specification is produced. Along with other documentation, this specification might contain a component specified in PLEASE.

3.1. Specifying a Component

Figure 2 shows the PLEASE specification of such a component; to increase readability and understandability, the syntax of PLEASE/Ada is similar to Anna[22]. In Ada, *packages* are used to group logically related components[21,71]. The specification uses the pre-defined package *NATURAL_LIST_PKG*, which uses the PLEASE type *list* to define the type *NATURAL_LIST* as *list of NATURAL*. In PLEASE, as in Lisp or Prolog, lists may have varying lengths and there is no explicit allocation or release of storage; however, the strong typing of Ada is retained and all the elements of a list must have the same type. In PLEASE, as in Prolog, the empty list is denoted by [], and a list literal is denoted by [*l*], where *l* is a comma separated list of elements. The functions *hd*, *tl*, and *cons* have their usual meanings and $L_1 \mid\mid L_2$ denotes the concatenation of the elements of L_1 and L_2.

Appendix B

```
with NATURAL_LIST_PKG , use NATURAL_LIST_PKG ,

package SORT_PKG is

     --  predicate PERM( L1, L2   in out NATURAL_LIST ) is true if
     --       FRONT, BACK   NATURAL_LIST ,
     --  begin
     --       L1 = [] and L2 = []
     --            or
     --       L1 = FRONT || cons(hd(L2),BACK) and
     --            PERM(FRONT || BACK, tl(L2))
     --  end ,

     --  predicate SORTED( L   in out NATURAL_LIST ) is true if
     --  begin
     --       L = []
     --           or
     --       tl(L) = []
     --           or
     --       hd(L) <= hd(tl(L)) and SORTED(tl(L))
     --  end ,

     procedure SORT( INPUT   in NATURAL_LIST , OUTPUT   out NATURAL_LIST ) ,
          --| where in( true ),
          --|       out( PERM(INPUT,OUTPUT) and SORTED(OUTPUT) ) ,

end SORT_PKG ,
```

Figure 2. Specification of *SORT_PKG*

The specification defines a package *SORT_PKG* which provides a procedure called *SORT*. The procedure takes two arguments: the first is a possibly unsorted input list, the second is a sorted list produced as output. The specification defines the predicates *PERM* (permutation) and *SORTED*, as well as giving pre– and post–conditions for the procedure. In PLEASE, the pre–condition for a procedure specifies the conditions that the input must meet before execution begins, while the post–condition specifies the conditions that the output must meet after execution has completed. In the specification, the state before execution begins is denoted by *in(...)*, while the state after execution has completed is denoted by *out(...)*. For example, the pre–condition for *SORT* is simply *true*; the type declarations for the parameters give all the requirements for the input. The post–condition for *SORT* states that the output is a permutation of the input and the output is sorted.

In PLEASE, a *predicate* syntactically resembles a procedure and may contain local type, variable, function or predicate definitions. For example, the predicate *PERM* states that two lists are permutations of each other if both of the lists are empty, or if the first element in the second list is in the first list and the remainder of the two lists are permutations of each other. At present, predicates are specified using Horn clauses: a subset of predicate logic which is also the basis for Prolog[16,18]. This approach allows a simple translation from predicate definitions into Prolog procedures; however, there are drawbacks. For example, in pure Horn clause programming there is no way to specify the falsehood of formulae; for example, the fact that *SORTED([2,1])* can never be true. The solution used in Prolog is the *closed world assumption*: if a fact is not provably true then it is assumed to be false. Unfortunately, the closed world assumption may cause inconsistencies for full first-order logic[62]; therefore, at present there is no way to specify negative information in PLEASE. Eventually, we plan to extend PLEASE to support a more powerful logic.

The specification contains no explicit I/O statements. Currently, all I/O is handled implicitly by the system; a program can be automatically generated which reads the *in* parameter to *SORT* from input, executes the procedure, and writes the *out* parameter to output. Although this approach limits PLEASE to the specification of programs with very simple I/O, it has several advantages: specifications without explicit I/O are smaller and simpler to write; omitting the sometimes messy, implementation specific details of I/O allows specifications to be more abstract; and the interaction of the specification, rapid prototyping and test harness capabilities of ENCOMPASS is greatly simplified.

After the requirements specification has been created, it must be validated. The systems analyst can discuss the specification with the customer and obtain test data and expected results for the system. The PLEASE specification can then be used to produce a prototype which satisfies the specification. If the prototype performs correctly on the test data it can be delivered to the customer for evaluation. If the prototype does not perform correctly then we know the specification is invalid.

3.2. Prototyping the Specification

The specification in Figure 2 can be automatically translated into a prototype written in a combination of Prolog and Ada. Figure 3 shows a simplified version of the Prolog code which is produced. The predicates *PERM* and *SORTED* and the pre– and post–conditions for *SORT* are translated into Prolog procedures, which are executed by an interpreter. When the *SORT* procedure is called, the *in* parameter is converted to the Prolog representation and the call is passed to the interpreter. When the Prolog procedure for *SORT* completes, the *out* parameter is converted to the Ada representation and the original call returns. Tools in the ENCOMPASS environment perform the translation and generate code to handle I/O

```
perm(L1,L2) ←
        eq(L1,[]),eq(L2,[]).
perm(L1,L2) ←
        eq(L1,Temp3),
        hd(L2,Temp1),
        cons(Temp1,Back,Temp2),
        append(Front,Temp2,Temp3),
        append(Front,Back,Temp4),
        tl(L2,Temp5),
        perm(Temp4,Temp5).

sorted(L) ←
        eq(L,[]).
sorted(L) ←
        tl(L,Temp1),
        eq(Temp1,[]).
sorted(L) ←
        hd(L,Temp1),
        tl(L,Temp2),
        hd(Temp2,Temp3),
        lseq(Temp1,Temp3),
        tl(L,Temp4),
        sorted(Temp4).

sort_pre(Input,Output) ← true.
sort_post(Input,Output) ← perm(Input,Output),sorted(Output).

sort(Input,Output) ←
        sort_pre(Input,Output),
        sort_post(Input,Output).
```

Figure 3. Prolog Code for *SORT* Procedure

and other implementation level details. The Prolog procedure for *SORT* simply "executes" the pre- and post-conditions.

The notion of execution is quite different for pre- and post-conditions. Executing a pre-condition involves checking that given data satisfies a logical expression. Executing a post-condition means finding data that satisfies a logical expression. For example, the post-condition for *sort* must find a value for the output list such that the input and output are permutations of each other and the output is sorted. To accomplish this, the Prolog procedure for the post-condition performs a naive sort of the input list. The Prolog procedure *perm* functions as a "generator" and the procedure *sorted* as a "selector". When the procedure for the post-condition is invoked, *perm* is called to generate a permutation of the input list and then *sorted* is called to determine if the permutation is sorted. If *sorted* fails, then execution backtracks and *perm* generates the next permutation to be evaluated. This continues until a sorted permutation is generated. The performance of this sorting algorithm is quite poor; however, it can be improved by transformation techniques applied to the logical formulae involved[36,40].

Although many implementations show significant deviations[66], a "pure" Prolog interpreter can be viewed as a resolution theorem prover for Horn clauses[16,18]. Using this model, the translation from PLEASE predicates to Prolog code is simply a sequence of transformations between equivalent formulae. The process consists of four steps. First the predicates are syntactically converted to the logical formulae they represent. Both the parameters to a predicate and its local variables represent universally quantified logical variables. For example, the predicate *PERM* in Figure 2 represents the logical formula:

$$\begin{array}{l} \forall\, L_1,L_2,\mathrm{Front},\mathrm{Back} \\ \quad (\ \mathrm{perm}(L_1,L_2) \leftarrow \\ \qquad L_1 = [\,] \wedge L_2 = [\,] \\ \qquad\qquad \vee \\ \qquad L_1 = \mathrm{append}(\mathrm{Front},\mathrm{cons}(\mathrm{hd}(L_2),\mathrm{Back})) \wedge \mathrm{perm}(\mathrm{append}(\mathrm{Front},\mathrm{Back}),\mathrm{tl}(L_2))\ ) \end{array}$$

Next, the terms on the right hand side of the implication are *unraveled* into conjunctions of relations. This is necessary because Prolog does not have a good notion of equality (for other solutions to this problem see[29,49]). We assume that for each function $f(\bar{x})$, there exists a relation $F(\bar{x},y)$ such that $f(\bar{x})=y$ iff $F(\bar{x},y)$. Axioms which characterize the relation $F(\bar{x},y)$ are part of the Prolog run-time library.

Appendix B

We unravel the formula $P(..f(\bar{x})..)$ into the equivalent formula $\exists t\ (F(\bar{x},t)$ and $P(..t..))$. The standard transformations to clause form are then used to convert the resultant formulae to Prolog procedures. To continue the previous example, the predicate *PERM* would produce the Prolog procedure:

```
perm(L1,L2) ←
        eq(L1,[]), eq(L2,[]).
perm(L1,L2) ←
        hd(L2,Temp1),
        cons(Temp1,Back,Temp2),
        append(Front,Temp2,Temp3),
        eq(L1,Temp3),
        append(Front,Back,Temp4),
        tl(L2,Temp5),
        perm(Temp4,Temp5).
```

The prototypes produced by this translation process are *partially correct*[54,55] with respect to the specifications. In other words, if a prototype terminates normally then the value returned will satisfy the post-condition. A prototype would be *totally correct*[54,55] if it was also guaranteed to terminate normally. The set of all logically valid formulae of predicate logic is not decidable[54,55]; therefore, in general it is not possible to extend our approach to total correctness. Furthermore, most Prolog implementations utilize an unbounded, depth-first search strategy which makes them *incomplete* as theorem-provers; although the Prolog procedures produced by our translation process have the proper logical properties, there is no guarantee that they will terminate.

In the last step of the translation process, a number of heuristic transformations are used on the Prolog procedures to increase the chances of termination. For example, the heuristic "move all equalities to the front of the clause" is applied to the procedure *perm* shown above to get the final Prolog procedure shown in Figure 3. To understand this heuristic, one must realize that the *eq* predicate always terminates. It can instantiate one of its arguments and thereby increase the amount of "information" available to subsequent procedures; this can increase the chances of termination. After the specification for *SORT_PKG* has been validated, it can be transformed into a more concrete form.

3.3. Refining the Specification

Assume that a decision is made to implement the sort procedure using the quicksort algorithm. As a first step, the original specification of *SORT_PKG* is refined so that *SORT* implements an abstraction of the quicksort algorithm. Figure 4 shows most of the refined specification. *SORT_PKG* contains three procedures which are called by *SORT*: *SELECT_ELMT*, *PARTITION*, and *COMBINE*. *SORT* has the same specification as before, but is now completely implemented. To sort the input list, *SELECT_ELMT* is called to select an element from the input list and then *PARTITION* is called to divide the list into two sublists, *LOW* and *HIGH*, so that all the members of *LOW* are less than the selected element and all the members of *HIGH* are greater. The lists *LOW* and *HIGH* are then sorted recursively and *COMBINE* is called to form a sorted permutation of the input from the sorted sub-lists.

The body of *SORT* is completely annotated; in other words, there is an assertion both before and after each executable statement. Each assertion states the conditions which must be satisfied whenever execution reaches that point in the procedure. The assertions plus the executable statements form a proof in the Hoare calculus[38,54,55]; this proof was incrementally created as the design transformation was performed. Each atomic transformation corresponds to a proof step; the transformation between Figure 2 and Figure 4 corresponds to a proof with a number of steps. Each transformation can be seen from either the *program view* or *proof view*. For example, Figure 5 shows the first step in the refinement of the *SORT* procedure from both the procedure and proof views. In the program view, an atomic transformation takes an incomplete program and produces a more concrete one; in the proof view, an atomic transformation adds another step to an incomplete proof tree. For more discussion on the relationship of proofs and programs see[7].

Although this refinement has narrowed the possible implementations to those using the quicksort algorithm, there are still many design decisions left unmade. The new specification may be refined into a *family* of quicksort programs; these programs might differ in many characteristics, but all would satisfy the specification. For example, the specification for *SELECT_ELMT* only requires that *ELMT* be a member of *LIST*; the algorithm used to select a particular element is not specified at this level of abstrac-

```
procedure SELECT_ELMT( LIST   in NATURAL_LIST . ELMT   out NATURAL )  is separate .
    --| where in( LIST /= [] ). out( member(ELMT.LIST) ) .

--  predicate IS_PART( LIST   in out NATURAL_LIST . ELMT   in out NATURAL .
--                     LOW. HIGH   in out NATURAL_LIST ) is true if
--  begin
--      PERM(LIST.LOW || [ELMT] || HIGH) and
--      LSEQALL(LOW.ELMT) and GREQALL(HIGH.ELMT)
--  end .

procedure PARTITION( LIST   in NATURAL_LIST . ELMT   in NATURAL .
                     LOW. HIGH   out,NATURAL_LIST ) is separate .
    --| where in( member(ELMT.LIST) ). out( IS_PART(LIST.ELMT.LOW.HIGH) ) .

procedure COMBINE( SORTED_L   in NATURAL_LIST . ELMT   in NATURAL .
                   SORTED_H   in NATURAL_LIST . LIST   out NATURAL_LIST ) is separate .
    --| where out( LIST = SORTED_L || [ELMT] || SORTED_H ) .

procedure SORT( INPUT   in NATURAL_LIST . OUTPUT   out NATURAL_LIST )  is

    LOW. HIGH. SORTED_L. SORTED_H   NATURAL_LIST . ELMT   NATURAL .

begin  -- SORT
    --| true .
    if INPUT = [] then
        --| true and INPUT = [] ;
        OUTPUT  = [] .
        --| PERM(INPUT.OUTPUT) and SORTED(OUTPUT) .
    else
        --| true and INPUT /= [] .
        SELECT_ELMT(INPUT.ELMT) .
        --| member(ELMT.INPUT) .
        PARTITION(INPUT.ELMT.LOW.HIGH) .
        --| IS_PART(INPUT.ELMT.LOW.HIGH) .
        SORT(LOW.SORTED_L) .
        --| IS_PART(INPUT.ELMT.LOW.HIGH) and PERM(LOW.SORTED_L) and SORTED(SORTED_L) .
        SORT(HIGH.SORTED_H) .
        --| IS_PART(INPUT.ELMT.LOW.HIGH) and PERM(LOW.SORTED_L) and
        --|     SORTED(SORTED_L) and PERM(HIGH.SORTED_H) and SORTED(SORTED_H) .
        COMBINE(SORTED_L.ELMT.SORTED_H.OUTPUT) ;
        --| PERM(INPUT.OUTPUT) and SORTED(OUTPUT) .
    end if ;
    --| PERM(INPUT.OUTPUT) and SORTED(OUTPUT) .
 end SORT ;
```

Figure 4. Refinement of Sort Specification

tion. Similarly, the specification for *PARTITION* only states that all the elements in *LOW* are less than or equal to *ELMT* and all the elements in *HIGH* are greater than or equal to *ELMT*; it says nothing about the algorithm used to produce these lists. As the specification is refined further these algorithms will be defined, thereby narrowing the acceptable implementations. However, before the new specification is refined further, it must be shown that any implementation which satisfies the new specification will also satisfy the original.

Program View

```
begin  -- SORT
    --| true ;
    <unknown_1>
    --| PERM(INPUT,OUTPUT)
    --|     and SORTED(OUTPUT) ;
end SORT ;
```

≡

Proof View

$$\{p\}\ S_1\ \{q\}$$

Where $p \equiv \text{true}$, $S_1 \equiv \text{unknown\_1}$,
$q \equiv \text{permutation(input,output)} \wedge \text{sorted(output)}$.

Refine unknown_1 into an *if-then-else* and generate appropriate assertions

Instantiate S_1 to an *if-then-else* and apply proof rule for conditional statements

```
begin  -- SORT
    --| true ;
    if INPUT = [] then
        --| true and INPUT = [] ;
        <unknown_2>
        --| PERM(INPUT,OUTPUT)
        --|     and SORTED(OUTPUT) ;
    else
        --| true and INPUT /= [] ;
        <unknown_3>
        --| PERM(INPUT,OUTPUT)
        --|     and SORTED(OUTPUT) ;
    end if ;
    --| PERM(INPUT,OUTPUT)
    --|     and SORTED(OUTPUT) ;
end SORT ;
```

≡

$$\frac{\{p \wedge e\}\ S_2\ \{q\},\ \{p \wedge \neg e\}\ S_3\ \{q\}}{\{p\}\ \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ S_2\ \textbf{\textit{else}}\ S_3\ \textbf{\textit{end if}}\ \{q\}}$$

Where $p \equiv \text{true}$,
$q \equiv \text{permutation(input,output)} \wedge \text{sorted(output)}$,
$e \equiv \text{input} = []$,
$S_i \equiv \text{unknown\_i}$.

Figure 5. Refinement as Proof Construction

3.4. Verifying the Refinement

A number of different methods may be used to show that the refined specification satisfies the original. In the most informal case, inspection of the original and refined specifications by a senior designer, or a peer review process might be used. A more rigorous approach might run prototypes produced from the original and refined specifications on the same test data and compare the results; this method gives significant assurance at low cost. However, in the words of E. W. Dijkstra, "Program testing can be used to show the presence of bugs, never to show their absence." In the most rigorous case, mathematical reasoning would be used.

In ENCOMPASS, the refinement process is viewed as the incremental construction of a proof in the Hoare calculus[38,54,55]; it is supported by ISLET[68], a language oriented editor similar to[63]. ISLET provides commands to add, delete and refine constructs; as the specification is transformed into an implementation (and the proof is constructed) the syntax and semantics are constantly checked. Many atomic transformations will generate verification conditions in the underlying first-order logic. These are algebraically simplified and then subjected to a number of simple proof tactics. If these fail, input is generated for TED, a proof management system that is interfaced to a number of theorem provers[35].

The use of general purpose theorem provers is quite expensive[1]; therefore, proofs using TED will usually not be performed during a design transformation. Simple methods are used to eliminate trivial verification conditions as they are generated; verification conditions which can not be eliminated by these methods are recorded by ENCOMPASS for use during the corresponding verification phase. For example, Figure 6 shows the verification conditions for the transformation from Figure 2 to Figure 4 which can not be proven by algebraic simplification and simple proof tactics alone; out of eleven refinements, only two generated non-trivial verification conditions. During the verification phase, these non-trivial formulae can be subjected to peer review, informal proof, or mechanical certification.

When all the atomic transformations have been verified, the design transformation is known to be correct. Once the design transformation has been verified, the new specification may be refined further and the process repeated until an implementation is produced. Although this example shows only the

```
INPUT = [] =>
      PERM(INPUT,[]) and SORTED([])

IS_PART(INPUT,ELMT,LOW,HIGH) and
      PERM(LOW,SORTED_L) and SORTED(SORTED_L) and
      PERM(HIGH,SORTED_H) and SORTED(SORTED_H) and
      LIST = SORTED_L || [ELMT] || SORTED_H =>
      PERM(INPUT,LIST) and SORTED(LIST)
```

Figure 6. Verification Conditions for Refinement

specification of a procedure, PLEASE may also be used to specify other classes of components, including data types.

4. Specifying Data Types

It has been proposed that the use of *abstract data types* can enhance software specification, validation and verification[30,32,45,53,58]. For example, Figure 7 shows the PLEASE specification of an Ada package defining the type *NATURAL_STACK* to provide a stack of natural numbers. In PLEASE, a data type has another type as its *representation*; for example, an object of type *NATURAL_STACK* is represented using an object of type *NATURAL_LIST*. As in VDM[43], a type has an *invariant* which restricts the set of legal representations; the invariant must be true of any values input to, or output from, functions on the type. For example, the type *NATURAL_STACK* has the invariant *true* meaning that all values of type *NATURAL_LIST* can be interpreted as values of *NATURAL_STACK*.

In PLEASE, the functions on a data type are specified with pre– and post–conditions in a manner similar to procedures. For example, the function *TOP* has *not(EMPTY)* as a pre–condition; the function is only defined on stacks with at least one element. The post–condition for *TOP* states that the value returned by the function is the head of the list given as an argument. The pre– post–conditions for a function are used to generate axioms which characterize its behavior. These axioms are used in both the Prolog prototypes produced from specifications and in the proof of theorems concerning the type.

Appendix B

```
with NATURAL_LIST_PKG ; use NATURAL_LIST_PKG ;

package NATURAL_STACK_PKG is

        type NATURAL_STACK is new NATURAL_LIST ;
          --| where S NATURAL_STACK => true ;

        function EMPTY_STACK return NATURAL_STACK ;
           --| where return S NATURAL_STACK => S = [] ;

        function EMPTY return BOOLEAN ;
           --| where return B BOOLEAN => B = (S = []) ;

        function PUSH( E   in NATURAL ; S   NATURAL_STACK ) return NATURAL_STACK ;
           --| return NS NATURAL_STACK => NS = cons(E,S) ;

        function POP( S   NATURAL_STACK ) return NATURAL_STACK ;
           --| where in( not(EMPTY) ),
           --|       return NS NATURAL_STACK => NS = tl(S) ;

        function TOP( S   NATURAL_STACK )  return NATURAL ;
           --| where in( not(EMPTY) ),
           --|       return E   NATURAL => E = hd(S) ;

 end NATURAL_STACK_PKG ;
```

Figure 7. *NATURAL_STACK* in Terms of *NATURAL_LIST*

NATURAL_STACK_PKG defines five functions on the type *NATURAL_STACK*. The function *EMPTY_STACK* returns an empty list to be interpreted as an empty stack, while the function *EMPTY* determines if any items are on a stack. The function *PUSH* takes a natural number and a stack as input, and returns a new stack which is equal to the old stack with the natural number on top. The function *POP* returns a stack with the top element removed, while the function *TOP* returns the element at the top of the stack. *NATURAL_STACK_PKG* can be used in other components to provide a stack of natural numbers; it can be used in parameter or variable declarations, as the basis for new type definitions, or in the specification of new software components.

5. System Status

The SAGA project has been active at the University of Illinois at Urbana-Champaign for over five years. The ENCOMPASS environment has been under development since the summer of 1984. A prototype implementation of ENCOMPASS has been operational since the summer of 1986; it is written in a combination of C, Csh, Prolog and Ada. This prototype includes the tools necessary to support software development using PLEASE: an initial version of ISLET, the language-oriented editor used to create PLEASE specifications and refine them into Ada implementations; software which automatically translates PLEASE specifications into Prolog procedures and generates the support code necessary to call these procedures from Ada; the run-time support routines and axiom sets for a number of pre-defined types; and interfaces to the ENCOMPASS test harness and TED. The subset of PLEASE currently implemented includes the *if*, *while*, and assignment statements, as well as procedure calls with *in*, *out* or *in out* parameters. The language now supports a small, fixed set of types including natural numbers, lists, booleans and characters. PLEASE and ENCOMPASS have been used to develop small programs, including specification, prototyping, and mechanical verification.

6. Summary

PLEASE is an executable specification language which supports program development by incremental refinement. PLEASE is part of the ENCOMPASS environment which provides automated support for all aspects of the software development process. Software components are first specified using a combination of conventional programming languages and predicate logic. These abstract components are then incrementally refined into components in an implementation language. Each refinement is verified before another is applied; therefore, the final components produced by the development satisfy the original specifications.

PLEASE specifications can be transformed into prototypes which use Prolog to "execute" pre- and post-conditions. We believe that the early production of executable prototypes for experimentation and evaluation will enhance the development process. Prototypes can increase the communication between customer and developer, thereby enhancing the validation process. Prototypes produced from PLEASE specifications can be used in experiments performed to guide the design process. Prototypes produced from a PLEASE specification and its refinement can be run on the same test data and the results compared; this method can give significant assurance that a refinement is correct at a low cost. PLEASE prototypes are based on existing Prolog technology, and their performance will improve as the speed of Prolog implementations increases (commercial Prolog compilers which produce native code compatible with conventional languages are already available[2]). As logic programming progresses, new versions of PLEASE can be built based on more powerful logics. We believe that the use of methods similar to those based on PLEASE specifications will enhance the design, development, validation and verification of software.

7. References

1. "Peer Review of a Formal Verification / Design Proof Methodology", NASA Conference Publication 2377, 1985.
2. "Quintus Prolog Users Guide and Reference Manual (Version 3)", Quintus Computer Systems, Palo Alto, California, 1985.
3. *Proceedings of the International Workshop on the Software Process and Software Environments.* **Software Engineering Notes** (August 1986) vol. 11, no. 4.
4. Balzer, Robert. *A 15 Year Perspective on Automatic Programming.* **IEEE Transactions on Software Engineering** (November 1985) vol. SE-11, no. 11, pp. 1257–1268.
5. Balzer, Robert, Thomas E. Cheatham and Cordell Green. *Software Technology in the 1990's: Using a New Paradigm.* **IEEE Computer** (November 1983) vol. 16, no. 11, pp. 39–45.
6. Barstow, David R. *On Convergence Toward a Database of Program Transformations.* **ACM Transactions on Programming Languages and Systems** (January 1985) vol. 7, no. 1, pp. 1–9.
7. Bates, Joseph L. and Robert L. Constable. *Proofs as Programs.* **ACM Transactions on Programming Languages and Systems** (January 1985) vol. 7, no. 1, pp. 113–136.
8. Beshers, George M. and Roy H. Campbell. *Maintained and Constructor Attributes.* **Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments** (June 1985) pp. 34–42.
9. Bjorner, D. and Cliff B. Jones. **Formal Specification and Software Development.** Prentice-Hall, Englewood Cliffs, N.J., 1982.
10. Bloomfield, Robin E. and Peter K. D. Froome. *The Application of Formal Methods to the Assessment of High Integrity Software.* **IEEE Transactions on Software Engineering** (September 1986) vol. SE-12, no. 9, pp. 988–993.
11. Blum, B. I. *The Life-Cycle - A Debate Over Alternative Models.* **Software Engineering Notes** (October 1982) vol. 7, pp. 18–20.
12. Campbell, Roy H. and Robert B. Terwilliger,. *The SAGA Approach to Automated Project Management.* In: **International Workshop on Advanced Programming Environments**, Lynn R. Carter, ed. Springer-Verlag Lecture Notes in Computer Science, New York, 1986, pp. 145–159.
13. Campbell, Roy H. *SAGA: A Project to Automate the Management of Software Production Systems.* In: **Software Engineering Environments**, Ian Sommerville, ed. Peter Perigrinus Ltd, 1986.
14. Campbell, Roy H. and Peter A. Kirslis. *The SAGA Project: A System for Software Development.* **Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments** (April 1984) pp. 73–80.
15. Campbell, Roy H. and Paul G. Richards. *SAGA: A system to automate the management of software production.* **Proceedings of the National Computer Conference** (May 1981) pp. 231–234.
16. Chang, Chin-Liang and Richard Char-Tung Lee. **Symbolic Logic and Mechanical Theorem Proving.** Academic Press, New York, 1973.
17. Cheatham, Thomas E., Glenn H. Holloway and Judy A. Townley. *Program Refinement By Transformation.* **Proceedings of the 5th International Conference on Software Engineering** (1981) pp. 430–437.

18. Clocksin, W. F. and C. S. Mellish. **Programming in Prolog**. Springer-Verlag, New York, 1981.

19. Cottam, I. D. *The Rigorous Development of a System Version Control Program.* **IEEE Transactions on Software Engineering** (March 1984) vol. SE-10, no. 3, pp. 143–154.

20. Davis, Ruth E. *Runnable Specification as a Design Tool.* In: **Logic Programming**, K. L. Clark and S. A. Tarnlund, ed. Academic Press, London, 1982, pp. 141–149.

21. Defense, U. S. Dept. **Reference Manual for the ADA Programming Language** ANSI/MIL-STD-1815A-1983. Springer-Verlag, New York, 1983.

22. DeMillo, R. A., R. J. Lipton and A. J. Perlis. *Social Processes and Proofs of Theorems.* **Communications of the ACM** (May, 1979) vol. 22, no. 5, pp. 271–280.

23. Dijkstra, E. W. *Structured Programming.* In: **Software Engineering Principles**, J. N. Buxton and B. Randall, ed. NATO Science Committee, Brussels, Belgium, 1970.

24. Fagan, Michael E. *Advances in Software Inspections.* **IEEE Transactions on Software Engineering** (July 1986) vol. SE-12, no. 7, pp. 744–751.

25. Fairley, Richard. **Software Engineering Concepts**. McGraw-Hill, New York, 1985.

26. Gannon, John, Paul McMullin and Richard Hamlet. *Data-Abstraction Implementation, Specification, and Testing.* **ACM Transactions on Programming Languages and Systems** (July 1981) vol. 3, no. 3, pp. 211–223.

27. Gehani, Narain and Andrew D. McGettrick (eds.). **Software Specification Techniques**. Addison Wesley, Reading, Massachusetts, 1986.

28. Goguen, Joseph and Jose Meseguer. *Rapid Prototyping in the OBJ Ezececutable Specification Laguage.* **Software Engineering Notes** (December 1982) vol. 7, no. 5, pp. 75–84.

29. Goguen, J. A. and J. Meseguer. *Equality, Types, Modules and (why not?) Generics for Logic Programming.* **Logic Programming** (1984) vol. 1, no. 2, pp. 179–210.

30. Goguen, Joseph, James Thatcher and Eric Wagner. *An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types..* In: **Current Trends in Programming Methodology, IV**, Raymond Yeh, ed. Prentice-Hall, London, 1978, pp. 80–149.

31. Gries, David. **The Science of Programming**. Springer-Verlag, New York, 1981.

32. Guttag, J. V. and J. J. Horning. *The Algebraic Specification of Abstract Data Types.* **Acta Informatica** (1978) vol. 10, pp. 27–52.

33. Guttag, John V., James J. Horning and Jeannette M. Wing. *The Larch Family of Specification Languages.* **IEEE Software** (September 1985) vol. 2, no. 5, pp. 24–36.

34. Guttag, John V., Ellis Horowitz and David R. Musser. *Abstract Data Types and Software Validation.* **Communications of the ACM** (December 1978) vol. 21, no. 12, pp. 1048–1063.

35. Hammerslag, David H., Samuel N. Kamin and Roy H. Campbell. *Tree-Oriented Interactive Processing with an Application to Theorem-Proving.* **Proceedings of the Second ACM/IEEE Conference on Software Development Tools, Techniques, and Alternatives** (December, 1985).

36. Hansson, Ake and Sten-Ake Tarnlund. *Program Transformation by Data Structure Mapping.* In: **Logic Programming**, K. L. Clark and S. A. Tarnlund, ed. Academic Press, London, 1982, pp. 117–122.

37. Henderson, Peter. *Functional Programming, Formal Specification, and Rapid Prototyping.* **IEEE Transactions on Software Engineering** (February, 1986) vol. SE-12, no. 2, pp. 241–250.

38. Hoare, C. A. R. *An Axiomatic Basis for Computer Programming.* **Communications of the ACM** (October 1969) vol. 12, no. 10, pp. 576–580.

39. ——. *Proof of Correctness of Data Representations.* **Acta Informatica** (1972) vol. 1, pp. 271–281.

40. Hogger, C. J. *Derivation of Logic Programs.* **Journal of the Association for Computing Machinery** (April 1981) vol. 28, no. 2, pp. 372–392.

41. Jackson, M. **System Development**. Prentice-Hall, Englewood Cliffs, N.J., 1983.

42. Jones, Cliff B. *Constructing a Theory of a Data Structure as an Aid to Program Development.* **Acta Informatica** (1979) vol. 11, pp. 119–137.

43. ——. **Software Development: A Rigorous Approach**. Prentice-Hall International, Engelwood Cliffs, N.J., 1980.

44. ——. *Tentative Steps Toward a Development Method for Interfering Programs.* **ACM Transactions on Programming Languages and Systems** (October 1983) vol. 5, no. 4, pp. 596–619.

45. Kamin, Samuel. *Final Data Types and Their Specification.* **ACM Transactions on Programming Languages and Systems** (January 1983) vol. 5, no. 1, pp. 97–121.

Appendix B

46. Kamin, S. N., S. Jefferson and M. Archer. *The Role of Executable Specifications: The FASE System.* **Proceedings of the IEEE Symposium on Application and Assessment of Automated Tools for Software Development** (November 1983).

47. Kemmerer, Richard A. *Testing Formal Specifications to Detect Design Errors.* **IEEE Transactions on Software Engineering** (January 1985) vol. SE-11, no. 1, pp. 32-43.

48. Kirslis, Peter A., Robert B. Terwilliger and Roy H. Campbell. *The SAGA Approach to Large Program Development in an Integrated Modular Environment.* **Proceedings of the GTE Workshop on Software Engineering Environments for Programming-in-the-Large** (June 1985) pp. 44-53.

49. Kornfeld, William A. *Equality for Prolog.* **Proceedings of the International Joint Conference on Artificial Intelligence** (1983).

50. Kowalski, Robert. *Logic as a Computer Language.* In: **Logic Programming,** K. L. Clark and S. A. Tarnlund, ed. Academic Press, London, 1982, pp. 3-16.

51. Kruchten, Philippe, Edmond Schonberg and Jacob Schwartz. *Software Prototyping Using the SETL Programming Language.* **IEEE Software** (October 1984) vol. 1, no. 4, pp. 66-75.

52. Lehman, M. M., V. Stenning and W. M. Turski. *Another Look at Software Design Methodology.* **Software Engineering Notes** (April 1984) vol. 9, no. 2, pp. 38-53.

53. Liskov, Barbara H. and Stephen N. Zilles. *Specification Techniques for Data Abstractions.* **IEEE Transactions on Software Engineering** (March 1975) vol. SE-1, no. 1, pp. 7-18.

54. Loeckx, Jacques and Kurt Sieber. **The Foundations of Program Verification**. John Wiley & Sons, New York, 1984.

55. Manna, Zohar. **Mathematical Theory of Computation**. McGraw-Hill, New York, 1974.

56. Meyers, G. J. **The Art of Software Testing**. John Wiley & Sons, New York, 1979.

57. Mills, Harlan D. and Richard C. Linger. *Data Structured Programming: Program Design without Arrays and Pointers.* **IEEE Transactions on Software Engineering** (February 1986) vol. SE-12, no. 2, pp. 192-197.

58. Musser, David R. *Abstract Data Type Specification in the AFFIRM System.* **IEEE Transactions on Software Engineering** (January 1980) vol. SE-6, no. 1, pp. 24-32.

59. Neighbors, James M. *The Draco Approach to Constructing Software from Reusable Components.* **IEEE Transactions on Software Engineering** (September 1984) vol. SE-10, no. 5, pp. 564-574.

60. Parnas, D. L. *The Use of Precise Specifications in the Development of Software.* **IFIP Congress Proceedings** (1977) pp. 861-867.

61. Partsch, H. and R. Steinbruggen. *Program Transformation Systems.* **Computing Surveys** (September 1983) vol. 15, no. 3, pp. 199-236.

62. Reiter, Raymond. *On Closed World Data Bases.* In: **Logic and Data Bases,** H. Gallaire and J. Minker, ed. Plenum Press, 1978.

63. Reps, Thomas and Bowen Alpern. *Interactive Proof Checking.* **Proceedings of the 11th ACM Symposium on the Principles of Programming Languages** (January 1984) pp. 36-45.

64. Richardson, Debra J. and Lori A. Clarke. *Partition Analysis: A Method Combining Testing and Verification.* **IEEE Transactions on Software Engineering** (December, 1985) vol. SE-11, no. 12, pp. 1477-1490.

65. Shaw, R. C., P. N. Hudson and N. W. Davis. *Introduction of A Formal Technique into a Software Development Environment (Early Observations).* **Software Engineering Notes** (April 1984) vol. 9, no. 2, pp. 54-79.

66. Stickel, Mark E. *A Prolog Technology Theorem Prover.* **Proceedings of the International Symposium on Logic Programming** (February 1984) pp. 211-217.

67. Terwilliger, Robert B. and Roy H. Campbell. *ENCOMPASS: a SAGA Based Environment for the Composition of Programs and Specifications.* **Proceedings of the 19th Hawaii International Conference on System Sciences** (January 1986) pp. 436-447.

68. ——. "ENCOMPASS: an Environment for the Incremental Development of Software", Report No. UIUCDCS-R-86-1296, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1986.

69. ——. *PLEASE: Predicate Logic based ExecutAble SpEcifications.* **Proceedings of the 1986 ACM Computer Science Conference** (February, 1986) pp. 349-358.

70. Tseng, Jine S., Boleslaw Szymanski, Yuan Shi and Noah S. Prywes. *Real-Time Software Life Cycle with the Model System.* **IEEE Transactions on Software Engineering** (February 1986) vol. SE-12, no. 2, pp. 358-373.

71. Wegner, Peter. **Programming with Ada: an Introduction by Means of Graduated Examples**. Prentice-Hall, Englewood Cliffs, New Jersey, 1980.

72. Weinberg, Gerald M. and Daniel P. Freedman. *Reviews, Walkthroughs, and Inspections.* **IEEE Transactions on Software Engineering** (January 1984) vol. SE-10, no. 1, pp. 68-72.

73. Wile, David S. *Program Developments: Formal Explanations of Implementations.* **Communications of the ACM** (November

1983) vol. 26, no. 11, pp. 902-911.

74. Wulf, William A., Ralph L London and Mary Shaw. *An Introduction to the Construction and Verification of Alphard Programs.* **IEEE Transactions on Software Engineering** (December 1976) vol. SE-2, no. 4, pp. 253-265.

75. Yourdon, E. and L. L. Constantine. **Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design**. Prentice-Hall, Englewood Cliffs, N.J., 1979.

76. Zave, Pamela. *The Operational Versus the Conventional Approach to Software Development.* **Communications of the ACM** (February 1984) vol. 27, no. 2, pp. 104-118.

77. Zave, Pamela and William Schnell. *Salient Features of an Executable Specification Language and Its Environment.* **IEEE Transactions on Software Engineering** (February 1986) vol. SE-12, no. 2, pp. 312-325.

SAGA Project Mid-Year Report 1986 Appendix J

Prolog Support Libraries for the Please Language

Philip Ray Roberts

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, Illinois

PROLOG SUPPORT LIBRARIES FOR THE PLEASE LANGUAGE

BY

PHILIP RAY ROBERTS

B.S., Oklahoma State University, 1984

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1986

Urbana, Illinois

Table of Contents

1. INTRODUCTION

The cost and difficulty of producing correct software are well-known problems in the computer industry. To help alleviate these problems, methods for specifying[8,14,15,20,22,23] and verifying[9,11,14,21,30] software have been developed. The SAGA (Software Automation, Generation, and Administration) project[1,2,4,10,18,19,27] is investigating the formal and practical aspects of providing automated support for a broad spectrum of software engineering activities. The PLEASE language[28] is being developed by the SAGA group to support the specification, prototyping, and rigorous development of software components. In this thesis, I describe a set of Prolog run-time support libraries for PLEASE.

Many programming methodologies have been proposed to help control the complexity of software design and development[12,14,29,31]. In *top-down development* methods, large programming problems are decomposed into a number of smaller, less complex problems. Top-down development methodologies have been defined[14,24] and implemented[26]. Using *stepwise refinement*[6], we start with an abstract *specification* of the problem and iteratively transform it into a real implementation; thus, the necessary development decisions are divided into smaller, more comprehensible groups. The specification is a precise statement of the function of the system. As the specification is incrementally refined, various software components, such as programs, test data, and various types of documentation, are generated. After each iteration, the components of the system are *verified* for correctness with respect to the specification.

Too often, systems are delivered which do not satisfy their users. A specification which accurately reflects the desires of the customer is difficult to produce. We say a specification is *validated* when it is shown to satisfy the customer's requirements. A formal specification may be difficult for the users to understand. It is easy to generate an informal specification, but it may be difficult to produce a system from a natural language description. *Prototyping*[7] and executable specification languages[16,17,22,32] may help alleviate these problems. Providing prototypes for the customers to use and evaluate early in the development process may increase communication between the customers and the developers. Once a valid specification is produced, a real system can be developed from it. Testing or formal verification techniques may be used to show that an implementation meets the requirements of the specification.

The Vienna Development Method[14,26] is an example of a top-down development methodology. The Vienna Development Method (VDM) contains methods for the formal verification of system components. In VDM, systems are specified in a language which combines elements of conventional programming languages and mathematics. *Pre-* and *post-conditions* written in predicate logic specify procedures. *Invariants* for user-defined data types are logical expressions which must be true both before and after the execution of any procedure which manipulates the data type. To enhance the expressive power of specifications, VDM adds the data types list, set, and map. These *abstract programs* are incrementally refined into programs coded in an implementation language. Each refinement is verified for correctness. Therefore, the final program produced by the development satisfies the original specification.

The PLEASE programming language is designed to support a methodology similar to the Vienna Development Method. In PLEASE, a procedure or function may be specified with pre- and post-conditions written in predicate logic, and a user-defined data type, called an *object*, may have an invariant. PLEASE specifications may be used in proofs of correctness. They may also be transformed into Prolog[5] prototypes. PLEASE specifications may interact with modules written in conventional languages.

In section 2 of this thesis, I describe the PLEASE programming language in more detail, giving an example PLEASE specification and its Prolog prototype. Section 3 contains a description of the run-time architecture of PLEASE. The representations of the PLEASE data types list, set, and map are described in section 4, along with of a description of the input/output support library and some miscellaneous functions useful in prototype development. In section 5, I summarise and draw some conclusions from the work done for this thesis.

2. THE PLEASE PROGRAMMING LANGUAGE

PLEASE is an extension to the programming language Path Pascal[3], which is an extension to standard Pascal[13]. In Path Pascal, an encapsulated data type, called an *object*, defines a block which follows the scope rules of standard Pascal. The object definition includes declarations of local variables which are only accessible by procedures and functions defined within the object. Entry procedures or functions, called *operations*, may be called from within the scope containing the object declaration. Objects provide a facility for defining *encapsulated data types*; the data within the object may only be accessed and manipulated outside the object definition through entry operations. In Path Pascal, an initialisation procedure (initially block) is called when an instance of the object is created. Path Pascal allows asynchronous execution of program structures called *processes*. Processes communicate through shared data structures within an object. Each object has a *path expression* specifying synchronisation constraints for the processes, functions, and procedures within the object.

In PLEASE, procedures may be specified with pre– and post–conditions written in predicate logic. Pre– and post–conditions[21] are logical expressions specifying conditions which must be true when a procedure is entered and exited. The pre–condition for a procedure specifies constraints on the input parameters and global variables which must be met when the procedure is entered. The post–condition must be true when the procedure is exited; it specifies the conditions the output must satisfy. Data–type invariants may be specified for objects. The data–type invariant is a logical expression which must be true both before and after an object is modified. In other words, the data–type invariant is part of the pre– and post–condition of every operation on an object.

Figure 1 shows a PLEASE specification of the abstract data type *stack of integers*. In the specification, a stack is defined as a Path Pascal object. The operations on a stack are *push*, *pop*, *empty*, and *top*. The path expression for the stack specifies that the operations may be performed in any order.

The stack in the example is represented with a list of integers. A list in PLEASE is similar to a list in LISP or Prolog. It is an ordered sequence of elements, all of which are of a uniform type. The list has no specified length and may grow and shrink according to the operations performed on it. The basic

```
type   stack = object

       path push, pop, top, empty end ;

       var s : list of integer ;

       invariant ;
             begin true end ;

       entry procedure push(elmt : integer) ;
             pre_condition ;
                   begin true end ;
             post_condition ;
                   begin s' = < element > || s end ;

       entry procedure pop ;
             pre_condition ;
                   begin true end ;
             post_condition ;
                   begin s' = tl(s) end ;

       entry function top : integer ;
             pre_condition ;
                   begin not(empty) end ;
             post_condition ;
                   begin s' = s and top' = hd(s) end ;

       entry function empty : boolean ;
             pre_condition ;
                   begin true end ;
             post_condition ;
                   begin
                         (empty' = true and s = empty_list) or
                         (empty' = false and s <> empty_list)
                   end ;

       initially ;
             pre_condition ;
                   begin true end ;
             post_condition ;
                   begin s' = empty_list end ;

end ; (* stack *)
```

Figure 1. Stack of Integers in Terms of *list of integer*

operations performed on a list are finding its head or tail, appending two lists to form a new list, and determining if a list is empty. When a stack is created, the *initially* block is executed and the list is initialised with an empty list. New elements are pushed on the stack by inserting them at the front of the list. Elements are popped from the stack by removing them from the front of the list.

In PLEASE, the notation *s'* is used to denote the value of the variable *s* after the procedure is executed. Lists are specified in PLEASE by enumerating their elements between the symbols "<" and ">". The notation "||" is used to specify the concatenation of two lists.

This PLEASE specification may be transformed by an expert prototyper into Prolog procedures which may then be executed. Prolog[5] is a programming language based on predicate logic. Prolog procedures are goals which may be satisfied by a state–space search. Prolog's *backtracking* mechanism automatically searches the state–space finding any or all possible solutions for Prolog goals. Therefore, Prolog procedures may be used both to check whether the inputs satisfy the pre–condition and to find the outputs which satisfy the post–condition.

Figure 2 gives the Prolog prototype created from the stack object specification. The pre– and post–conditions for each operation in the stack object have been transformed into Prolog procedures. Each operation is performed by executing the corresponding pre– and post–condition. Note, particularly, the Prolog procedure for the function *top*. The *top* function returns the element on the top of the stack. The pre–condition for the *top* function specifies that the stack must not be empty when *top* is entered. Executing the pre–condition involves checking for the condition when the function is called. The post–condition is executed just before the function returns. In the *top* procedure, the post–condition unifies the return value with the head of the list representing the stack. There are a number of ways a prototyping expert may code the pre– and post–conditions in Prolog. In this example, since the data type invariant for a stack is always true, the expert prototyper has not included it in the prototypes. Normally, the invariant would be checked in the procedure for each pre– and post–condition.

The stack object was specified with the PLEASE data type list. In addition to lists, PLEASE defines the data types set and map. A PLEASE set is an unordered collection of elements, all of which must be of the same type. PLEASE sets are not multisets. The basic operations on sets are determining if an element is a member of a set, finding the union or intersection of two sets, and determining if one set is a subset of

```
push_pre_condition :- true.
push_post_condition(S,int(Elmt),S_Prime) :-
               list_hd(S_Prime,int(Elmt)),
               list_tl(S_Prime,S).
push(S,int(Elmt),S_Prime) :-
               push_pre_condition,
               push_post_condition(S,int(Elmt),S_Prime).

pop_pre_condition :- true.
pop_post_condition(S,S_Prime) :-
               list_tl(S,S_Prime).
pop(S,S_Prime) :-
               pop_pre_condition,
               pop_post_condition(S,S_Prime).

top_pre_condition(S) :-
               empty(S,false).
top_post_condition(S,Top) :-
               list_hd(S,Top).
top(S,Top) :-
               top_pre_condition(S),
               top_post_condition(S,Top).

empty_pre_condition :- true.
empty_post_condition(S,true) :-
               list_empty(S).
empty_post_condition(S,false).
empty(Empty) :-
               empty_pre_condition,
               empty_post_condition(S,Empty).

initially_pre_condition :- true.
initially_post_condition(S_Prime) :-
               list_empty(S_Prime).
initially :-
               initially_pre_condition,
               initially_post_condition(S_Prime).
```

Figure 2. Prolog Prototype for Stack Object

another.

A map is similar to a relation in mathematics. It is a finite set of domain element – range element pairs. For each element in the domain there is at most one pair in the map. The pairs may be specified individually or by a domain set and a function which defines the corresponding elements in the range set.

Adhering to the strong type checking of Pascal, all elements of a set (including the domain and range sets of a map) must be of a uniform type.

PLEASE specifications contain pre– and post–conditions written in predicate logic and are useful in formal proofs of correctness. They are also easily transformed into executable Prolog prototypes. The data types list, set, and map are powerful tools for data abstraction and should be very useful in specifying systems. PLEASE specifications are incrementally refined into source modules coded in implementation languages such as C, Pascal, or Ada.

3. RUN-TIME ARCHITECTURE

As systems are refined from a specification to a real implementation, the modules specified in PLEASE will be expanded into routines coded in various implementation languages such as C, Pascal, and Ada. Therefore, there will be modules written in conventional languages and modules consisting of PLEASE prototypes written in Prolog. Since we do not have a Prolog compiler with an interface to standard implementation languages, we must be able to link object modules generated from conventional languages to Prolog procedures.

One way to do this is to provide a standard text interface from a conventional language to Prolog. The Prolog code is encoded as text in implementation language source modules and sent to a Prolog interpreter for execution. Parameters to the Prolog procedures are passed to the module containing the Prolog code, converted into text, and placed in the Prolog interface calls. The output parameters are converted from text into the calling language representation and returned. To execute Prolog procedures through a text interface from implementation language modules, the code for the Prolog procedures must be asserted

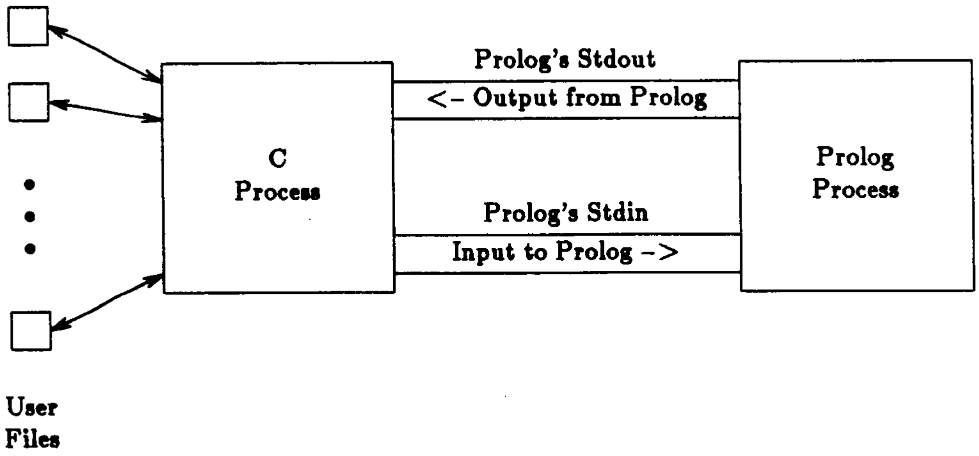


Figure 3. Interprocess Communication – Files Manipulated by C

in the Prolog data-base. Then procedure "calls" may be made by sending commands to Prolog to execute the code.

The PLEASE run-time architecture provides such an interface. The host process is an object program created from various source modules written in an implementation language. A separate process runs the UNSW Prolog interpreter[25]. Figure 3 illustrates how these processes communicate through Unix[1] pipes; the host process sends commands down a pipe to the Prolog interpreter which returns the results through another pipe.

Figure 4 illustrates how a C program "calls" Prolog. The *c_to_plg* library provides the standard text interface from C language modules to Prolog. The file "c_to_plg.h" is included to make all the necessary declarations and definitions for using the *c_to_plg* library. Prolog code is stored in a *P_BUF* and sent to the Prolog interpreter with *c_to_plg_call.* A *P_BUF* is a C string, up to 4K-bytes in length, and may be used in standard C string operations. Since these *P_BUF*s are C strings, they must be terminated with a "\0". Another *P_BUF* must be provided to receive the output generated by Prolog.

In this example, the C function *assert* adds all the procedure definitions for the stack object to the Prolog data base. Once these definitions have been added, they remain until the end of execution; therefore, the definitions only have to be asserted once. In the *test* function, two calls are being made to the procedures in the stack object. The first call pushes the integer "3" onto the stack. The second call uses the top function to see if it was pushed properly. The first parameter in *c_to_plg_call* is the input buffer. The second buffer receives Prolog's output. When this program is run, it outputs "X=int(3)".

When *c_to_plg_call* is invoked, the input string is sent through a pipe to Prolog's standard input. The first time *c_to_plg_call* is executed, it starts the Prolog process and sets up the necessary interprocess communication channels. Prolog executes the instructions received on its standard input and writes the output onto its standard output, which is piped back to the calling process. The *c_to_plg_call* function should return when Prolog has written all its output. Since the Prolog interpreter does not flush its output

[1] Unix is a trademark of AT&T Bell Laboratories

```
#include <stdio.h>
#include "c_to_plg.h"


assert()         /* assert definitions of stack object */
{
        P_BUF inbuf ;                   /* Prolog input buffer */
        P_BUF outbuf ;                  /* Prolog output buffer */

     sprintf(inbuf,"%s %s %s %s %s %s %s",
         "push_pre_condition :- true. " ,
         "push_post_condition(S,int(Elmt),S_Prime) :-" ,
         "  list_hd(S_Prime,int(Elmt)), " ,
         "  list_tl(S_Prime,S), " ,
         "push(S,int(Elmt),S_Prime) :- " ,
         "  push_pre_condition, " ,
         "  push_post_condition(S,int(Elmt),S_Prime). " ) ;
     c_to_plg_call(inbuf,outbuf) ;

        /* rest of the code for the stack would also be asserted */

}


test()       /* test push and top */
{
        P_BUF inbuf ;                   /* Prolog input buffer */
        P_BUF outbuf ;                  /* Prolog output buffer */

        sprintf(inbuf,"push(S,int(3),S_Prime)!") ;
        c_to_plg_call(inbuf,outbuf) ;

        sprintf(inbuf,"top(S,X)?");
        c_to_plg_call(inbuf,outbuf) ;
        printf("%s",outbuf) ;

}
```

Figure 4. Excerpt from C Program Testing Stack of Integers

pipe when it has finished writing, the calling routine must tell it when to do so. *C_to_plg_call* sends a flush command to Prolog after the user's input is sent down the pipe. When the user instructions have been executed, the flush command causes all output to be sent to the calling process by the operating system. *C_to_plg_call* assembles the output from Prolog into the the output buffer and returns when Prolog has completed the flush.

The run-time architecture places some restrictions on the way Prolog modules interact with the file system. Prolog's standard input and standard output are used for interprocess communication; therefore, they may not be used for file access. In Unix, each process gets a unique file descriptor for a file. Therefore, separate processes writing to the same file may overwrite one another's changes. All file processing may be done from either the implementation language modules or the Prolog modules, but because of the danger of processes overwriting one another's files, file processing should not be mixed between Prolog modules and implementation language modules. Figure 5 illustrates that all file processing is done from the Prolog modules. A library of Pascal-like file manipulation routines is provided for Prolog.

The *ptoplg* library provides a Prolog interface for Pascal and Path Pascal. Since standard Pascal does not support strings, the type *plgbuf* and operations on it are defined. Figure 6 shows how a Pascal program might execute Prolog code to test the stack specification. The file "ptoplg.h" contains the definitions necessary for using the *ptoplg* library. A *plgbuf* is a 4K-byte Pascal string. A *plgbuf* is

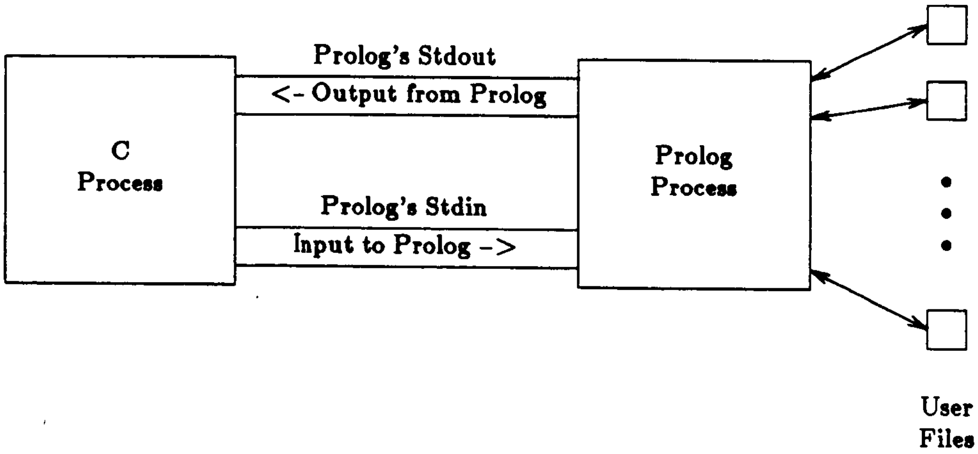


Figure 5. Interprocess Communication – Files Manipulated by Prolog

```
#include "ptoplg.h"

procedure test;
var i,o : plgbuf ;
begin

        plgbufinit(i) ;
        plgbufappend(i, 'push(S,int(3),S_Prime)! $') ;
        ptoplgcall(i,o) ;

        plgbufinit(i) ;
        plgbufappend(i, 'top(S,X)? $') ;
        ptoplgcall(i,o) ;
        plgbufwrite(o) ;

end;
```

Figure 6. Calling Prolog from Path Pascal

initialised and cleared with *plgbufinit.* Strings are appended to the existing contents of a *plgbuf* with *plgbufappend.* These strings must be terminated with a "$". *Ptoplgcall* works in the same way as *c_to_plg_call* and is, in fact, implemented with *c_to_plg_call. Plgbufwrite* prints the contents of a plgbuf on standard output. The output produced by this procedure is "X=int(3)".

In order to support the data types list, set, and map defined in the Vienna Development Method, standard Prolog representations of these types were defined and libraries of procedures were developed to support these representations. In addition, a set of file input/output procedures based on those provided by standard Pascal were defined to supplement the Prolog file input/output model. A library of miscellaneous procedures useful in debugging and making standard definitions was also developed. These libraries are loaded automatically when the Prolog interpreter is invoked.

4. PROLOG SUPPORT LIBRARIES

The Prolog support libraries use a standard set of data representations. In the libraries, instances of PLEASE data types are represented as Prolog terms of the form: "<data_type>(Value)". For example, the integer "3" would be represented "int(3)". The Prolog term is the most convenient way of representing structured information and it is useful to have the data type as the principal functor of the Prolog term that is representing an instance of a data type. This type information can be used as a selector in overloaded functions (such as a generic pretty-printing procedure). The type information is also needed for making the appropriate conversions of text output from Prolog into representations for other languages.

4.1. Prolog Representation of Lists, Arrays, Sets, and Maps

Since the list is the basic data structuring tool in Prolog, we represent all PLEASE data types in terms of the Prolog list. In Prolog, a list is an ordered sequence of elements. The first element in a list is called the head of the list. The tail of a list is the remainder of a list after its head is removed.

A PLEASE list is represented in Prolog as:

```
list( [Element,...,Element] )
```

where all *Elements* are instances of some PLEASE data type. The library of list routines includes procedures to create an empty list, find the head and tail of a list, append two lists, determine if two lists are equal, find the union or intersection of two lists, and various operations to index the elements of a list. All operations on arrays, sets, and maps are defined in terms of the list operations. Therefore, any increase in the efficiency of the list operations will improve the performance of the operations on other data types.

The array is the principal built-in data structure of Pascal. A single dimensioned array is provided as a data type in PLEASE. An instance of an array includes its lower and upper bounds as well as the items in the array:

```
array( int(Lowerbound), int(Upperbound), list( [Element,...,Element]) )
```

The elements of an array may be instances of any PLEASE data type. The library of array routines includes procedures for determining the size of an array, accessing the individual elements of an array, checking that two arrays are equal, and modifying the contents of an array.

A set is also implemented with a PLEASE list. The order of elements in a set is not preserved by the operations in the set library. There are two set representations.

```
set( list( [Element,...,Element] ) )
```

is an instance of an *enumerated* set. As with lists and arrays, the elements of the set may be instances of any PLEASE data type. All the operations in the set library manipulate instances of enumerated sets. There are operations to insert and remove elements from a set, find all the members of a set, take the union, intersection, or difference of two sets, create an empty set, and determine if two sets are equal. The sets {4,3} and {3,4} are equal and the equality routine will verify that two mathematically equivalent sets are equal, even if the elements are not enumerated in the same order. There is also a procedure for determining if one set is the subset of another.

For convenience, a second representation of sets, called a *concise representation* is provided. Many large sets are much too tedious to type in at a terminal (for example, imagine typing in the set of integers from one to a hundred, or a thousand). These sets may be defined with a "low" or "first" element, a "high" or "last" element, and two functions, one to generate the successor to an element of a set, and one to determine when two elements of the set are equal. For example,

```
setc( int(1), int(100), int_next(_,_), int_equal(_,_) )
```

together with

```
int_next( int(X), int(XPlus1) ) :-
  XPlus1 is X + 1.
int_equal( int(X), int(X) ).
```

is a full specification for the set of integers from 1 to 100. The next function and equal function must be asserted in the Prolog data base[2]. Note that the data type for an instance of a concise set is *setc*. Also, note that the full procedure head for the next and equal functions are used in the representation and that the variables in the procedure heads are specified with underbars. To convert this concise representation to a standard enumerated set, call the *set_transform* procedure in the library of set operations with the *setc* term as the first argument. The second argument should be a variable and will be unified with a completely instantiated enumerated set. Due to restrictions imposed by the Prolog interpreter currently in use, it is unwise to have sets with more than about 100 elements.

PLEASE maps also have two representations. The standard representation is a list of ordered pairs:

```
map( list( [ pair(Element,Element), ..., pair(Element,Element)] ) )
```

where each element is, again, an instance of any PLEASE data type. The first element in each pair is an element of the domain set and the second element of each pair is an element of the range set. All elements of the domain set should be of the same type and all elements of the range set should be of the same type. The elements in the pair do not have to be related in any way but the procedures in the map library assume that for any element in the domain set, there is only one corresponding element in the range set. There are procedures in the map library for finding the domain and range sets of a map, inserting a pair in the map, finding a range element given a domain element, and changing or removing a pair in the map. There is also a procedure to transform a set into a map when a function is provided to take domain elements and produce their corresponding range elements.

The concise representation for maps is very similar to the concise representations for sets. In addition to the definitions for the next function and the equality predicate, a function definition must be provided for the mapping of domain elements to range elements. For example,

[2] The successor and equality functions for integers are defined in the library of miscellaneous procedures (see Appendix A).

```
mapc( int(1), int(20), next_int(_,_), equal_int(_,_), square(_,_) )

square( int(X), int(XSquared) ) :-
    XSquared is X * X .
```

is the concise representation for the mapping from integers in the range one to twenty to their corresponding squares. The function *map_transform* in the map library takes a concise map as the first argument and returns the corresponding standard map in the second argument.

4.2. Procedure Classification

The procedures in the Prolog libraries may be classified as *functions*, *generators*, or *predicates*. A standard Prolog *function* returns one or more values when given one or more inputs. A *generator* takes one or more non-variable arguments and successively unifies the other argument(s) with all the possible values that satisfy the conditions. For example, *set_member*, a procedure in the set library, may be used as a generator. *Set_member*, when used as a generator, takes a set as the first argument and a variable as the second argument. Prolog will successively unify the second argument with each element of the set during backtracking. For example, the query

```
set_member( set(list([int(1),int(7),int(4),int(5)])), X ) ?
```

will yield the following output:

```
X=int(1)
X=int(7)
X=int(4)
X=int(5).
```

A procedure is used as a *predicate* when all arguments are *completely instantiated* (i.e. there are no variable terms in any argument). A predicate is, then, a logical expression that may be included in pre- and post-conditions. When the procedure is called as a predicate, it either succeeds or fails. Consider again *set_member*. If the first argument is a set and the second argument is an element, set_member will succeed only if the element is contained in the set. For example, the query

```
set_member( set(list([int(3),int(4),int(5)])), int(3) ) ?
```

will succeed, while the query

```
set_member( set(list([int(3),int(4),int(5)])), int(1) ) ?
```

will fail.

To document the use of a Prolog procedure, an annotation of its parameters is used. In the synopsis of the manual page entry for a library, each argument of a procedure is classified as an input parameter "+input", an output parameter "-output", a generated output "-generated", or a template output "-template". An input parameter is a completely instantiated Prolog term. An output parameter is a variable, and the procedure will unify it with a value which is a completely instantiated Prolog term. A generated output is a variable that will be unified with all possible values on backtracking (see the *set_member* example, above).

For example,

```
foo( +input, +input, -output)
foo( +input, -generated, -generated)
foo( +input, +input, +input)
```

tells us that the procedure "foo" can be used in any of three ways. First, if the first two arguments are inputs, the third argument will receive an output value. This is an example of using a procedure as a *function*. If only the first argument has a value, "foo" will generate values in the second and third arguments. "Foo" can also be used as a predicate; if all three arguments contain input values, "foo" will either succeed or fail.

In the future, we would like to investigate the use of *templates* as parameters. A template output is a variable that will be unified with a *partially instantiated* Prolog term. A good example of template output and its usefulness is the combined use of the *list_hd* and *list_tl* procedures from the list library:

```
list_hd(-template,+input)
list_tl(-template,+input).
```

If the second argument of the *list_hd* procedure is a completely instantiated Prolog term and the first argument is a variable, the first argument will be unified with a list template, a PLEASE list with a variable tail and the second argument of the *list_hd* procedure as the head. If the second argument of the *list_tl* procedure is a PLEASE list and the first argument is a variable, the first argument will become a PLEASE list with an uninstantiated head. We can use these together to create a new list.

For example, the query

```
list_hd( NewList, int(5) ),
list_tl( NewList, list( [int(6), int(7), int(8)] ) ) ?
```

will produce the output

```
NewList=list([int(5),int(6),int(7),int(8)])
```

If the second arguments to *list_hd* and *list_tl* are instantiated with the head and tail of a list, respectively, and the first argument of each procedure is the same Prolog variable, the variable will be unified with a new list.

4.3. File Input/Output Library

The Prolog file input/output interface is quite different from that provided in conventional languages. In order to provide a more conventional interface, a suite of Prolog procedures simulating the Pascal input/output model is provided for use in PLEASE prototypes. The fileio library provides procedures for opening, closing, reading, and writing files.

A reset operation opens a file for reading. The file is then read from the start. A rewrite operation opens a file for writing. The file is written from the start. There is no way to append output to the end of a file. The Prolog interpreter currently in use restricts the number of files open for input and/or output at any one time to 15.

The restriction that a reset or rewrite must be performed before reading or writing a file is enforced in the following manner. A reset on Filename causes a clause "open_read(Filename)" to be asserted in the Prolog data base. Whenever a rewrite is performed, "open_write(Filename)" is asserted. If the user attempts to read a file with no *open_read* clause or write a file with no *open_write* clause, an error message is printed, and the procedure returns an error. The cost of this error checking is one assertion for each open operation and one unification for each read or write operation.

The Prolog procedure *fileio_eval* allows the read (*fileio_read* and *fileio_readln*) and write (*fileio_write* and *fileio_writeln*) procedures to be called with a variable number of arguments. All calls to *fileio_read*, *fileio_readln*, *fileio_write*, and *fileio_writeln* must be included in *fileio_eval* as shown in the synopsis of the manual entry for fileio_lib.

Fileio_read and *fileio_readln* read Prolog terms from the specified file. Each argument will receive one Prolog term as a return value. If there are any terms remaining on a line after *fileio_readln* has unified its arguments, they will be ignored. If the end of file is reached, every remaining argument will be unified with the atom 'end_of_file'. If the file being read is not terminated with a newline, these procedures will hang. Remember that *fileio_read* and *fileio_readln* must be called within *fileio_eval.*

Various errors are detected by the fileio library at run-time. Each routine in fileio_lib has an error return code. When an error is detected, a message is printed on standard error and the error variable is unified with the name of the routine in which the error occurred. If no error occurs, the error code will be set to the Prolog atom 'false'.

4.4. Miscellaneous Tools

A useful debugging environment is also being developed for the PLEASE system. A set of procedures for manipulating global variables has been developed. These global procedures are extremely useful in debugging and may be used to implement a type of call-by-reference parameter passing. A global variable is a Prolog term with the global variable name (assigned by the *get_global* routine) as the principal functor and the value as the only argument in the term. For example, a global variable containing a single integer, 3, might be:

```
global0( int(3) ).
```

Values can be assigned to global variables and obtained from global variables using operations defined in the msc_lib. A procedure to dispose of a global variable is also provided.

Global variables are useful when debugging prototypes. Long instances of data types are tedious to type. It may be easier to assign an instance of a data type to a global variable and then dereference the global variable when its value is needed.

Global variables can also be used to implement call-by-reference in PLEASE prototypes. An argument to a procedure may be the name of a global variable. The variable may be dereferenced to obtain its contents. The new value may be assigned to the global variable before the procedure returns. This is also useful in reducing the traffic in procedure calls made from implementation language modules through the pipes. Instead of passing the entire value of a variable, the prototype procedure could be coded to operate on call-by-reference. Only the name of the variable is passed to the procedure. It is then dereferenced, modified, and stored back in the global variable.

5. SUMMARY AND CONCLUSIONS

PLEASE is a programming language which supports a methodology similar to the Vienna Development Method. PLEASE procedures may be specified with pre– and post–conditions written in predicate logic. User–defined abstract data types called objects may have data type invariants. PLEASE specifications may be transformed into executable prototypes written in Prolog. These prototypes are useful in helping the developers deliver a system that satisfies the customers desires. The specifications may also be used with formal verification techniques to show that an implementation meets the requirements of the specification.

The PLEASE data types list, set, and map are conveniently represented in Prolog. Libraries of standard operations on these data types have been developed. A run–time architecture has been developed which allows Prolog procedures to be executed from standard implementation language modules. A library of procedures which simulate the standard Pascal input/output model was defined in order to provide a more conventional i/o interface for PLEASE. A set of procedures to manipulate global variables was provided to facilitate debugging of prototypes. These procedures are also useful in implementing a form of call–by–reference in PLEASE prototypes.

The list is the principal data structuring device of Prolog. PLEASE lists were easily represented in terms of Prolog lists. PLEASE sets and maps were then defined in terms of PLEASE lists. The operations on sets and maps were implemented in terms of the operations on lists. There is a great deal of room for improving the efficiency of the library of list operations. Since the other operations were defined with the library of list operations, improvements in the efficiency of list operations will also improve the efficiency of operations on sets and maps.

At present, instances of PLEASE data types contain structural type information. However, the operations on the data type representations are not *type–safe*. For example, the function *list_empty* may be used to create a list with no elements in this situation. It is not possible to determine if the list is to be a list of integers, a list of characters, or a list of some compound data type. Schemes for run–time type checking were investigated, but we concluded that the overhead needed to provide this facility was too great. We also investigated the run–time checking of name compatibility between types. This idea proved to be extremely difficult.

The libraries developed for this thesis are a major step in the implementation of the PLEASE programming language. PLEASE should provide an interesting vehicle for the study of top–down development methodologies by the SAGA group. We feel these methodologies will enhance the software development process.

References

1. Beshers, George M. and Roy H. Campbell. *Maintained and Constructor Attributes.* **Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments** (June 1985) pp. 34–42.

2. Campbell, Roy H. and Peter A. Kirslis. *The SAGA Project: A System for Software Development.* **Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments** (April 1984) pp. 73–80.

3. Campbell, Roy H. and Robert B. Kolstad. *Path Expressions in Pascal.* **Proceedings of the Fourth International Conference on Software Engineering** (September 1979).

4. Campbell, Roy H. and Paul G. Richards. *SAGA: A system to automate the management of software production.* **Proceedings of the National Computer Conference** (May 1981) pp. 231–234.

5. Clocksin, W. F. and C. S. Mellish. **Programming in Prolog**. Springer-Verlag, New York, 1981.

6. Fairley, Richard. **Software Engineering Concepts**. McGraw-Hill, New York, 1985.

7. Goguen, Joseph and Jose Meseguer. *Rapid Prototyping in the OBJ Exececutable Specification Laguage.* **Software Engineering Notes** (December 1982) vol. 7, no. 5, pp. 75–84.

8. Guttag, J. V. and J. J. Horning. *The Algebraic Specification of Abstract Data Types.* **Acta Informatica** (1978) vol. 10, pp. 27–52.

9. Guttag, John V., Ellis Horowitz and David R. Musser. *Abstract Data Types and Software Validation.* **Communications of the ACM** (December 1978) vol. 21, no. 12, pp. 1048–1063.

10. Hammerslag, David H., Samuel N. Kamin and Roy H. Campbell. *Tree-Oriented Interactive Processing with an Application to Theorem-Proving.* **Proceedings of the Second ACM/IEEE Conference on Software Development Tools, Techniques, and Alternatives** (December, 1985).

11. Hoare, C. A. R. *Proof of Correctness of Data Representations.* **Acta Informatica** (1972) vol. 1, pp. 271–281.

12. Jackson, M. **System Development**. Prentice-Hall, Englewood Cliffs, N.J., 1983.

13. Jensen, Kathleen and Niklaus Wirth. **PASCAL User Manual and Report**. Springer-Verlag, New York, 1974.

14. Jones, Cliff B. **Software Development: A Rigorous Approach**. Prentice-Hall International, Engelwood Cliffs, N.J., 1980.

15. Kamin, Samuel. *Final Data Types and Their Specification.* **ACM Transactions on Programming Languages and Systems** (January 1983) vol. 5, no. 1, pp. 97–121.

16. Kamin, S. N., S. Jefferson and M. Archer. *The Role of Executable Specifications: The FASE System.* **Proceedings of the IEEE Symposium on Application and Assessment of Automated Tools for Software Development** (November 1983).

17. Kemmerer, Richard A. *Testing Formal Specifications to Detect Design Errors.* **IEEE Transactions on Software Engineering** (January 1985) vol. SE-11, no. 1, pp. 32–43.

18. Kirslis, Peter A., Robert B. Terwilliger and Roy H. Campbell. *The SAGA Approach to Large Program Development in an Integrated Modular Environment.* **Proceedings of the GTE Workshop on Software Engineering Environments for Programming-in-the-Large** (June 1985).

19. Lehman, M. M., V. Stenning and W. M. Turski. *Another Look at Software Design Methodology.* **Software Engineering Notes** (April 1984) vol. 9, no. 2, pp. 38–53.

20. Liskov, Barbara H. and Stephen N. Zilles. *Specification Techniques for Data Abstractions.* **IEEE Transactions on Software Engineering** (March 1975) vol. SE-1, no. 1, pp. 7–18.

21. Loeckx, Jacques and Kurt Sieber. **The Foundations of Program Verification**. John Wiley & Sons, New York, 1984.

22. Musser, David R. *Abstract Data Type Specification in the AFFIRM System.* **IEEE Transactions on Software Engineering** (January 1980) vol. SE-6, no. 1, pp. 24–32.

23. Parnas, D. L. *The Use of Precise Specifications in the Development of Software.* **IFIP Congress Proceedings** (1977) pp. 861-867.

24. Ross, Douglas T. *Structured Analysis (SA): A Language for Communicating Ideas.* **IEEE Transactions on Software Engineering** (January 1977) vol. SE-3, no. 1, pp. 16-34.

25. Sammut, C. A. and R. A. Sammut. *The Implementation of UNSW-Prolog.* **The Australian Computer Journal** (May 1983) vol. 15, no. 2, pp. 58-64.

26. Shaw, R. C., P. N. Hudson and N. W. Davis. *Introduction of A Formal Technique into a Software Development Environment (Early Observations).* **Software Engineering Notes** (April 1984) vol. 9, no. 2, pp. 54-79.

27. Terwilliger, Robert B. and Roy H. Campbell. *ENCOMPASS: a SAGA Based Environment for the Composition of Programs and Specifications.* **Submitted to 19th Hawaii International Conference on System Science** (January 1986).

28. ——. "PLEASE: Predicate Logic based Executable SpEcifications", 1986 ACM Computer Science Conference, Cincinnati, Ohio, 1986.

29. Wirth, Niklaus. *Program Development by Stepwise Refinement.* **Communications of the ACM** (April 1971) vol. 14, no. 4, pp. 221-227.

30. Wulf, William A., Ralph L London and Mary Shaw. *An Introduction to the Construction and Verification of Alphard Programs.* **IEEE Transactions on Software Engineering** (December 1976) vol. SE-2, no. 4, pp. 253-265.

31. Yourdon, E. and L. L. Constantine. **Structured Design**. Prentice-Hall, Englewood Cliffs, N.J., 1979.

32. Zave, Pamela. *An Overview of the PAISLey Project - 1984.* **Software Engineering Notes** (July 1984) vol. 9, no. 4, pp. 12-19.

Appendix A

The following pages are the UNIX Programmer's Manual

Entries for the programs written for this thesis.

NAME

array_lib - a Prolog library of array routines for PLEASE

SYNOPSIS

```
Array representation:
   array(int(Lowerbound),int(Upperbound),list([Element,...,Element]))

array_size(+input,+input)
array_size(+input,-output)
array_size(array(...),int(Length))

array_member(+input,+input)
array_member(+input,-generated)
array_member(Element,array(...))

array_equal(+input,+input)
array_equal(+input,-output)
array_equal(array(...),array(...))

array_index(+input,+input,+input)
array_index(+input,+input,-output)
array_index(array(...),int(Index),Element)

array_overwrite(+input,+input,+input,-output)
array_overwrite(array(...),int(Index),Element,array(...))
```

DESCRIPTION

Array_lib provides a Prolog library of predicates and functions to operate on arrays.

Array_lib is a library of array routines for the PLEASE system (see *please_intro*(1)). PLEASE is an executable specification language. It is an extension of Path Pascal and supports the Vienna Development Method. In the PLEASE system, programs are specified using pre– and post–conditions written in predicate logic. These pre– and post–conditions are transformed into Prolog and executed by the UNSW Prolog interpreter. Calls to array_lib functions may be included in these prototypes.

Array_lib is written in Prolog (see *Programming in Prolog* by Clocksin and Mellish). An array is represented as a Prolog term of the form:

```
array(int(LowerBound),int(Upperbound),list([Element,...,Element]))
```

where each Element is a Prolog term with data type information and a value. See *lib_intro*(3) for more information about the Prolog representation of PLEASE data types and the libraries of Prolog functions to operate on those data types.

The array library provides a predicate for determining if an element is present in a list. The array library provides functions to determine the size of an array, the contents of one of the positions of the array, and to overwrite the contents of one of the positions of the array.

Array_size takes an array as its first argument and returns an integer (int(Value)) whose value is the size of the array. If its second argument is instantiated to an integer, the function will act as a predicate to determine if the array is of the given size.

Array_member takes an array as its first argument and generates the members of the array in the second argument. If the second argument is not a variable, array_member is a predicate that succeeds if the element is in the array.

Array_equal is a predicate that determines if two arrays are equal. Two arrays are equal if corresponding elements are equal. If the second argument to array_equal is a variable, it will be unified with the array given in the first argument.

Array_index takes an array as its first argument and an integer index as its second argument and returns the element at that position in its third argument. If the third argument is not a variable, array_index is a predicate that succeeds if the element is at that position in the array.

Array_overwrite takes an array as its first argument, an integer index as its second argument, a new element as its third argument, and returns the first array with the new element substituted at position index in the third argument.

SEE ALSO

lib_intro(3), *please_intro*(1), *encompass_intro*(1), *Programming in Prolog* by Clocksin and Mellish

AUTHOR

Philip R. Roberts, Robert B. Terwilliger, Department of Computer Science, University of Illinois, 252 Digital Computer Laboratory, 1304 West Springfield Avenue, Urbana, IL 61801.

NAME

c_to_plg - functions to enable execution of Prolog commands from C

SYNOPSIS

```
#include "c_to_plg.h"

P_BUF inbuf, outbuf ;

void c_to_plg_call(inbuf, outbuf)
P_BUF *inbuf ;
P_BUF *outbuf ;

void c_to_plg_debug(debug)
int debug ;   /* constant ON or OFF */
```

DESCRIPTION

The c_to_plg functions provide a means for C programs to execute Prolog clauses.

The c_to_plg layer is one layer in the PLEASE system (see *please_intro*(1)). PLEASE is an executable specification language. It is an extension of Path Pascal and supports the Vienna Development Method. In the PLEASE system, programs are specified using pre– and post–conditions written in predicate logic. These pre– and post–conditions are transformed into Prolog and executed by the UNSW Prolog interpreter.

C_to_plg_call is a text interface from C to Prolog. C and Prolog communicate through strings. All strings must be terminated with a '\0'. C programs can send commands to Prolog to be executed by using c_to_plg_call. The command is placed in the inbuf. The results of the command executed by Prolog are returned in the outbuf. P_BUF contains a 4K–byte character string.

The c_to_plg_debug function turns debugging on or off for the c_to_plg layer. If debug is set to ON, a constant defined in the header file, the debugging is turned on for c_to_plg. If value is set to OFF, also a constant define in the header file, it is turned off.

FILES

${ENCOMPASS}/include/c_to_plg.h
${ENCOMPASS}/lib/c_to_plg.o

SEE ALSO

please_intro(1), *encompass_intro*(1)

AUTHOR

Philip R. Roberts, Robert B. Terwilliger, Department of Computer Science, University of Illinois, 252 Digital Computer Laboratory, 1304 West Springfield Avenue, Urbana, IL 61801.

NAME

fileio_lib – a library of Prolog routines to simulate the Pascal I/O interface in the PLEASE system

SYNOPSIS

```
fileio_reset(+input,-output)
fileio_reset(Filename,Error)

fileio_rewrite(+input,–output)
fileio_rewrite(Filename,Error)

fileio_eval(fileio_write(+input,+input,+input,...,+input),–output)
fileio_eval(fileio_write(Filename,Arg1,Arg2,...,ArgN),Error)

fileio_eval(fileio_writeln(+input,+input,+input,...,+input),–output)
fileio_eval(fileio_writeln(Filename,Arg1,Arg2,...,ArgN),Error)

fileio_eval(fileio_read(+input,–output,–output,...,–output),–output)
fileio_eval(fileio_read(Filename,Arg1,Arg2,...,ArgN),Error)

fileio_eval(fileio_readln(+input,–output,–output,...,–output),–output)
fileio_eval(fileio_readln(Filename,Arg1,Arg2,...,ArgN),Error)
```

DESCRIPTION

Fileio_lib provides a Prolog I/O library similar to that provided by Pascal.

Fileio_lib is a library of I/O routines for the PLEASE system (see *please_intro*(1)). PLEASE is an executable specification language. It is an extension of Path Pascal and supports the Vienna Development Method. In the PLEASE system, programs are specified using pre– and post–conditions written in predicate logic. These pre– and post–conditions are transformed into Prolog and executed by the UNSW Prolog interpreter. Calls to fileio_lib functions may be included in these prototypes.

Fileio_lib is written in Prolog (see *Programming in Prolog* by Clocksin and Mellish). It provides functions for opening, closing, reading, and writing files. The I/O routines are based on the Pascal I/O model.

In the fileio_lib, all parameters are input parameters except Arg1 through ArgN of fileio_read and Error of all functions. Output parameters must be Prolog variables (ie. first letter is capitalized). Filenames are UNIX filenames. All filenames and literal output should be enclosed in quotes.

Fileio_reset opens a file for reading. Fileio_reset must be called before a read can be performed on the file. Reading for the newly opened file begins at the start of the file.

Fileio_rewrite opens a file for writing. Fileio_rewrite must be called before a write can be performed on the file. If the file already exists, its contents will be cleared and writing will begin at the start of the file.

Fileio_eval allows the read (fileio_read and fileio_readln) and write (fileio_write and fileio_writeln) functions to be called with a variable number of arguments. All calls to fileio_read, fileio_readln, fileio_write, and fileio_writeln must be included in fileio_eval as shown in the SYNOPSIS.

Fileio_read and fileio_readln read Prolog terms from the specified file. Each argument will receive one Prolog term as a return value. If there are any terms remaining on a line after fileio_readln has unified its arguments, they will be ignored. If the end of file is reached, every remaining argument will be unified with the atom 'end_of_file'. If the file being read is not terminated with a newline, these function will hang. Remember that fileio_read and fileio_readln must be called within fileio_eval.

Fileio_write and fileio_writeln write their arguments to Filename. Fileio_writeln terminates its output with a newline whereas fileio_write does not. Remember that fileio_write and fileio_writeln must be called within fileio_eval.

Each routine in fileio_lib has an error return code. If no error occurs, this will be set to the Prolog atom 'false'. If an error occurs, the name of the routine where the error occurred will be returned.

DIAGNOSTICS

Various errors are detected by the fileio library at runtime. When an error is detected, a message is printed on standard error and the Error variable is unified with the name of the routine in which the error occurred.

SEE ALSO

please_intro(1), *Programming in Prolog* by Clocksin and Mellish

AUTHOR

Philip R. Roberts, Robert B. Terwilliger, Department of Computer Science, University of Illinois, 252 Digital Computer Laboratory, 1304 West Springfield Avenue, Urbana, IL 61801.

BUGS

There are some constraints on the I/O library because it is coded in Prolog. There can be at most 15 files open simultaneously. Filenames are UNIX filenames and must be enclosed in single quotes. There are some special restrictions on the fileio_read function. fileio_read reads Prolog terms. When the end of a file is reached, the value of every argument read after the end of file will be 'end_of_file'. If a file is not terminated by a new line, fileio_read will not detect end of file but will hang instead.

NAME

lib_intro – a set of libraries of Prolog functions to provide an I/O interface for PLEASE and to implement PLEASE data types

SYNOPSIS

List representation:

```
list([Element,...,Element])
```

Array representation:

```
array(int(Lowerbound),int(Upperbound),list([Element,...,Element]))
```

Standard set representation:

```
set(list([Element,...,Element]))
```

Concise set representation:

```
setc(LowElement,HighElement,NextFunction,EqualFunction)
NextFunction=FnName(_,_)
EqualFunction=FnName(_,_)
```

Standard map representation:

```
map(list([pair(DomainElement,RangeElement),...,
        pair(DomainElement,RangeElement)]))
```

Concise map representation:

```
mapc(LowElement,HighElement,NextFunction,EqualFunction,MapFunction)
NextFunction=FnName(_,_)
EqualFunction=FnName(_,_)
MapFunction=FnName(_,_)
```

Function description:

```
function_name( +input, -output, -generated)
function_name( argtype, argtype, argtype)
```

DESCRIPTION

Lib_intro describes a set of Prolog libraries of predicates and functions to define and operate on PLEASE data types; and to provide a Pascal-like I/O interface for PLEASE.

PLEASE is an executable specification language. It is an extension of Path Pascal and supports the Vienna Development Method. In the PLEASE system, programs are specified using pre– and post–conditions written in predicate logic. These pre– and post–conditions are transformed into Prolog and executed by the UNSW Prolog interpreter. Calls to library functions may be included in these prototypes.

List_lib (see *list_lib*(3)) is a library of list routines for the PLEASE system (see *please_intro*(1)). Array_lib (see *array_lib*(3)) is a library of array routines for the PLEASE system. Set_lib (see *set_lib*(3)) is a library of set routines for the PLEASE system. Map_lib (see *map_lib*(3)) is a library of map routines for the PLEASE system. Fileio_lib (see *fileio_lib*(3)) is a library of file input/output operations for the PLEASE system.

PLEASE data types are represented in Prolog as <type>(Value). For example, the integer "3" would be represented as int(3).

A list is represented as a Prolog term of the form:

```
list([Element,...,Element])
```

where each Element is a PLEASE data type, i.e. a Prolog term with type information and a value, <type>(Value).

For example, after the following PLEASE code fragment has been executed:

```
type integer_list = list of integer ;
variable i : integer_list ;
begin
 i := <1,2> ;
```

i would be represented as:

```
list([int(1),int(2)]).
```

An array is represented as a Prolog term of the form:

```
array(int(Upperbound),int(Lowerbound),list([Element,...,Element]))
```

where Upperbound is the highest index in the array and Lowerbound is the lowest. Each Element is a PLEASE data type, i.e. a Prolog term with type information and a value, <type>(Value). Arrays are single dimensioned, but an array of arrays could be constructed.

There are two representations for sets. A set can be described with the standard set notation or with the concise notation. All set operations are performed on sets in the standard notation. The concise notation is useful for describing large sets (i.e., the set of integers from 1 to 100). The set library has a function called *set_transform* that transforms a set in the concise notation to a standard set representation.

A standard set is represented as a Prolog term of the form:

```
set(list([Element,...,Element]))
```

where each Element is a PLEASE data type, i.e. a Prolog term with type information and a value, <type>(Value).

For example, after the following PLEASE code fragment has been executed:

```
type integer_set = set of integer ;
variable s : integer_set ;
begin
 s := set_union({1},{3,4}) ;
```

s might be represented as:

```
set(list([int(3),int(1),int(4)]).
```

In sets, the order of occurrence of elements is not important. The sets are not multisets; sets only contain one occurrence of each element.

To describe a set in the concise notation, the user must provide a low element, a high element, a function that produces the successor to a set element, and a function that determines when two elements are equal. For example, the following Prolog code describes the set of integers from 1 to 100:

```
next_int(int(X),int(XPlus1)) :- XPlus1 is X+1.
equal_int(int(X),int(X)).

Set=setc(int(1),int(100),next_int(_,_),equal_int(_,)).
```

Calling *set_transform* with Set as the first argument will produce the set of integers from 1 to 100 in the standard representation.

A map is represented as a Prolog term of the form:

```
map(list([pair(DomainElement,RangeElement),...,
        pair(DomainElement,RangeElement)])).
```

Each domain and range elements are PLEASE data type, i.e. a Prolog term with type information and a value, <type>(Value).

For example, after the following PLEASE code fragment has been executed:

```
type squares_map = map from integer to integer ;
    domain_set = set of integer ;
variable s : squares_map ;
       d : domain_set ;
function square( x : integer ) : integer
begin
  square := x*x
end ;
begin
  d := {1,2,3,4,5} ;
  s := map_construct(d,square) ;
```

s might be represented as:

```
map(list([pair(int(1),int(1)),pair(int(2),int(4)),
        pair(int(3),int(9)),pair(int(4),int(16)),
        pair(int(5),int(25))])).
```

To describe a map in the concise notation, the user must provide a low element for the domain set, a high element for the domain set, a function that produces the successor to a set element, a function that determines when two elements are equal, and a function that takes a domain element and returns the corresponding range element (the map function). The following Prolog code describes the mapping from the set of integers from 1 to 100 to their squares:

```
next_int(int(X),int(XPlus1)) :- XPlus1 is X+1.
equal_int(int(X),int(X)).
square_int(int(X),int(XSquared)) :- XSquared is X*X.

Map=mapc(int(1),int(100),next_int(_,_),equal_int(_,_),square_int(_,_).
```

Calling *map_transform* with Map as the first argument would generate the standard representation for the map from the set of integers from 1 to 100 to their squares. There is also a function *map_construct* that takes a set in the standard notation and the map function and generates the map with that set as the domain (see *map_lib*(3)).

Each function library has its own manual entry. Each function in the library is described briefly in the synopsis. The first few lines of the synopsis description contain information about how the arguments are to be used. For each argument, +input, -output, -template, or -generated, describes how the argument can be used. An argument that is marked "+input" should be a *completely instantiated* Prolog term. In other words, the term should have no variables or underbars. Arguments marked "-output" and "-generated" should be Prolog variables. A "-template" argument returns a partially instantiated Prolog term. The most useful instances of -template arguments are the list_hd and list_tl functions that can be used together to create a new list given a head and a tail (see *list_lib*(3)). In many functions, the arguments can be used in various combinations of input, output, and generated. Functions have varying numbers of arguments. For example,

```
foo( +input, +input, -output)
foo( +input, -generated, -generated)
foo( +input, +input, +input)
```

tells us that the function "foo" can be used in any of three ways. First, if the first two arguments are inputs, the third argument will receive an output value. If only the first argument has a value, "foo" will generate (see *generators*, below) values in the second and third arguments. "Foo" can also be used as a predicate. That is, if all three arguments contain input values, "foo" will either evaluate to true or false.

Some library functions can be used as *generators*. A generator takes one or more non-variable argument(s) and successively unifies the other argument(s) with all possible values that will satisfy the conditions. For example, set_member may be used as a generator. Set_member, when used as a generator, takes a set as the first argument and a variable as the second argument. The second argument will be successively unified with each element of the set during backtracking. For example:

```
set_member(set(list([int(1),int(7),int(4),int(5)])),X) ?

X=int(1)
X=int(7)
X=int(4)
X=int(5).
```

Some functions can also be used as *predicates*. A function is used as a predicate when none of the arguments are variables. When the function is called it either succeeds or fails. Consider again set_member. If the first argument is a set and the second argument is an element, set_member will succeed if the element is contained in the set. If the element is not contained in the set, set_member will fail. For example:

```
set_member(set(list([int(3),int(4),int(5)])),int(3)) ?
** yes.

set_member(set(list([])),int(3)) ?
** no.
```

SEE ALSO

please_intro(1), *encompass_intro*(1), *array_lib*(3), *list_lib*(3), *set_lib*(3), *map_lib*(3), *fileio_lib*(3), *prolog*(1), *Programming in Prolog* by Clocksin and Mellish

AUTHOR

Philip R. Roberts, Robert B. Terwilliger, Department of Computer Science, University of Illinois, 252 Digital Computer Laboratory, 1304 West Springfield Avenue, Urbana, IL 61801.

NAME

list_lib - a Prolog library of list routines for PLEASE

SYNOPSIS

```
List representation:
    list([Element,...,Element])

list_len(+input,+input)
list_len(+input,-output)
list_len(list([...]),int(Length))

list_equal(+input,+input)
list_equal(+input,-output)
list_equal(list([...]),list([...]))

list_member(+input,+input)
list_member(+input,-generated)
list_member(list([...]),Element)

list_hd(+input,+input)
list_hd(+input,-output)
list_hd(-template,+input)
list_hd(list([...]),Head)

list_tl(+input,+input)
list_tl(+input,-output)
list_tl(-template,+input)
list_tl(list([...]),Tail)

list_index(+input,+input,+input)
list_index(+input,+input,-output)
list_index(+input,-output,+input)
list_index(list([...]),int(Position),Element)

list_overwrite(+input,+input,+input,-output)
list_overwrite(list([...]),int(Position),Element,list([...]))

list_empty(+input)
list_empty(-output)
list_empty(list([...]))

list_append(+input,+input,-output)
list_append(list([...]),list([...]),list([...]))

list_intersect(+input,+input,-output)
list_intersect(list([...]),list([...]),list([...]))

list_difference(+input,+input,-output)
list_difference(list([...]),list([...]),list([...]))

list_union(+input,+input,-output)
list_union(list([...]),list([...]),list([...]))
```

DESCRIPTION

List_lib provides a Prolog library of predicates and functions to operate on lists.

List_lib is a library of list routines for the PLEASE system (see *please_intro*(1)). PLEASE is an executable specification language. It is an extension of Path Pascal and supports the Vienna Development Method. In the PLEASE system, programs are specified using pre– and post–conditions written in predicate logic. These pre– and post–conditions are transformed into Prolog and executed by the UNSW Prolog interpreter. Calls to list_lib functions may be included in these prototypes.

List_lib is written in Prolog (see *Programming in Prolog* by Clocksin and Mellish). A list is represented as a Prolog term of the form:

```
list([Element,...,Element])
```

where each Element is a Prolog term with data type information and a value (i.e., <type>(Value)). See *lib_intro*(3) for more information about the Prolog representation of PLEASE data types and the libraries of Prolog functions to operate on those data types.

The list library provides predicates for determining if an element is present in a list or if a list is empty. The list library provides functions to determine the length of a list, the head of a list (its first element), or the tail of a list (a list containing all the elements except the first). The list library also provides functions for appending two lists to form a new list; constructing a list containing elements that are in both of two lists; constructing a list containing the elements that are in one list, but not in another; and forming a list containing the elements of two lists but only one occurrence of each (i.e. merge two lists).

List_len takes a list as its first argument and returns an integer (int(Value)) whose value is the length of the list. If its second argument is instantiated to an integer, the function is a predicate that succeeds if the list is of the given length.

List_equal is a predicate if both arguments are instantiated. It succeeds if the two lists are equal. If the second argument is a variable, it will be unified with the first argument.

List_member takes a list as its first argument and generates the members of the list in its second argument. If its second argument is not a variable, list_member will act as a predicate, succeeding if the element is a member of the list.

List_hd takes a list as its first argument and returns the first element of the list as its second argument. If both arguments are instantiated, list_hd acts as a predicate which succeeds if the second argument is the head of the list (the first argument).

List_tl takes a list as its first argument and returns the tail of that list as its second argument (the tail of a list is the list with its first element removed). If both arguments are instantiated, list_tl acts as a predicate which is true if the second argument is the tail of the list.

List_hd and list_tl can be used together in the following fashion. Suppose we wanted to create a new list that had X as its head and Y as its tail. NewList is a template returned by each function. By giving the template the same name in each function, the Prolog unification operation fills in the empty slots to produce a completely instantiated list. We could do this with the following calls:

```
list_hd(NewList,X), list_tl(NewList,Y).
```

List_index is a predicate that succeeds if the Element given in the third argument is at the position given in the second argument of the list given as the first argument. If the third argument is a variable, it will be unified with the element of the list at the position given in the second argument. If the second argument is a variable, it will be unified with the position of the first element in the list equal to the third argument.

List_overwrite creates a new list by replacing the element of the list at Position (the second argument) with the element given as the third argument. This new list is returned as the fourth argument.

List_empty is true if its argument contains no elements. If its argument is a variable, list_empty will unify it with an empty list.

List_append creates a new list (its third argument) by appending two lists (its first two arguments).

List_intersection creates a new list (its third argument) which contains all the elements that are present in both of the lists passed in as its first two arguments. Each element will occur only once in the new list.

List_difference creates a new list (its third argument) which contains all the elements that are in its first argument and that are not in its second argument.

List_union creates a new list (its third argument) which contains all the elements in the first two arguments (lists). Each element occurs only once in the new list.

SEE ALSO

lib_intro(3), *please_intro*(1), *encompass_intro*(1), *Programming in Prolog* by Clocksin and Mellish

AUTHOR

Philip R. Roberts, Robert B. Terwilliger, Department of Computer Science, University of Illinois, 252 Digital Computer Laboratory, 1304 West Springfield Avenue, Urbana, IL 61801.

NAME

map_lib – a Prolog library of map routines for PLEASE

SYNOPSIS

Standard map representation:

```
map(list([pair(DomainElement,RangeElement),...,
     pair(DomainElement,RangeElement)]))
```

Concise map representation:

```
mapc(LowElement,HighElement,NextFunction,EqualFunction,MapFunction)
NextFunction=FnName(_,_)
EqualFunction=FnName(_,_)
MapFunction=FnName(_,_)
```

```
map_transform(+input,-output)
map_transform(mapc(...),map(...))

map_construct(+input,+input,-output)
map_construct(set(...),MapFunction,map(...))
MapFunction=FnName(_,_)

map_empty(+input)
map_empty(-output)
map_empty(map(...))

map_domain(+input,-output)
map_domain(map(...),set(...))

map_range(+input,-output)
map_range(map(...),set(...))

map_overwrite(+input,+input,-output)
map_overwrite(map(...),pair(DomainElement,RangeElement),map(...))

map_apply(+input,+input,+input)
map_apply(+input,+input,-output)
map_apply(map(...),DomainElement,RangeElement)
```

DESCRIPTION

Map_lib provides a Prolog library of predicates and functions to operate on maps.

Map_lib is a library of map routines for the PLEASE system (see *please_intro*(1)). PLEASE is an executable specification language. It is an extension of Path Pascal and supports the Vienna Development Method. In the PLEASE system, programs are specified using pre– and post–conditions written in predicate logic. These pre– and post–conditions are transformed into Prolog and executed by the UNSW Prolog interpreter. Calls to map_lib functions may be included in these prototypes.

Map_lib is written in Prolog (see *Programming in Prolog* by Clocksin and Mellish). There are two representations for maps, a standard representation and a concise representation. All map operations are performed on the standard representation of a map. The standard representation of a map contains a list that enumerates the pairs of elements of the map. The concise representation of a map includes the low element in the domain set, the high element in the domain set, the head of a clause that will produce the "next" element of the domain set, the head of a clause that will determine if two elements of the domain set are "equal", and the head of a clause that will return the range element that corresponds to the domain element. The concise representation provides a means for giving a short description of a large map (too large to enumerate). See *lib_intro*(3) for a description of PLEASE data types and general information about operations on those data types.

The map library provides a predicate for determining if a map is empty. The map library provides functions for finding the domain or range of a map, overwriting a pair in a map, or finding the range element that corresponds to the domain element of a map. All of these operations work on standard map representations. There is a function that converts a concise representation into a standard representation.

Map_transform takes a concise map representation as its first argument and returns the corresponding standard map representation as its second argument. It is important to remember that if a concise map representation is given, the user MUST provide functions definitions for the next function, the equal function, and the mapping itself.

Map_construct takes a standard set (the domain set, see *set_lib*(3)) as its first argument, the head of a function that describes the map as its second argument, and returns a standard map as the third arguement. The standard map is constructed by applying the function to each element of the standard set.

Map_empty succeeds if its argument (a standard map representation) is empty. If its argument is a variable, it will be instantiated to an empty map.

Map_domain takes a standard map as its first argument and returns a standard set that is the domain of the map. Map_range takes a standard map as its first argument and returns a standard set that is the range of the map.

Map_overwrite takes a standard map as its first argument, a mapping pair as its second argument, and returns a new map. If the domain element exists in the map, the range element will be replaced by the new range element in the mapping pair. If the domain element does not exist in the map, the mapping pair is inserted. If the range element is "_", the mapping pair for the domain element is removed from the map.

Map_apply takes a standard map as its first argument and a domain element as its second argument and returns the range element that corresponds to the domain element as its third argument. If the third argument is not a variable, map_apply is a predicate that succeeds if the third argument is the range element that corresponds to the domain element.

SEE ALSO

lib_intro(3), *please_intro*(1), *encompass_intro*(1), *Programming in Prolog* by Clocksin and Mellish

AUTHOR

Philip R. Roberts, Robert B. Terwilliger, Department of Computer Science, University of Illinois, 252 Digital Computer Laboratory, 1304 West Springfield Avenue, Urbana, IL 61801.

NAME

msc_lib – a library of miscellaneous routines for PLEASE

SYNOPSIS

```
% Global variable manipulation

get_global(–output)
get_global(Name)

allocate_global(+input)
allocate_global(Name)

assign_global(+input,+input)
assign_global(Name,Value)

value_global(+input,–output)
value_global(Name,Value)

remove_global(+input)
remove_global(Name)

% Useful operations on integers

int_equal(+input,+input)
int_equal(int(X),int(X))

int_next(+input,–output)
int_next(int(X),int(Y))

int_prev(+input,–output)
int_prev(int(X),int(Y))
```

DESCRIPTION

Msc_lib provides a Prolog library of predicates and functions to perform various miscellaneous operations.

Msc_lib is a library of miscellaneous routines for the PLEASE system (see *please_intro*(1)). PLEASE is an executable specification language. It is an extension of Path Pascal and supports the Vienna Development Method. In the PLEASE system, programs are specified using pre– and post–conditions written in predicate logic. These pre– and post–conditions are transformed into Prolog and executed by the UNSW Prolog interpreter. Calls to msc_lib functions may be included in these prototypes.

Msc_lib is written in Prolog (see *Programming in Prolog* by Clocksin and Mellish). One group of miscellaneous functions are used to allocate, manipulate, and deallocate global variables.

Global variables are very useful in prototyping PLEASE specifications. The representation of lists and sets can sometimes be very long and tedious to type. Global variables provide an easy way to manipulate these large representations. Use get_global(Name) to get a global name. Name will be unified with the name of the global variable (global0, global1,...). To allocate a global with a name of your own choosing, use allocate_global(Name). Use assign_global to assign a value to a global variable. Suppose you wanted to assign the term "function_names(push,pop,create,destroy)" to a global variable. First type "get_global(Name)?" to allocate a global variable. Then type "assign_global(Name,function_names(push,pop,create,destroy)" where Name is the global name returned by the call to get_global. Use value_global(Name,Value) to find the current value of a global variable. Use remove_global(Name) to deallocate a global variable.

This library also contains a set of useful operations on integers. Int_equal succeeds if the two integer arguments are equal. Int_next returns the successor of the integer given as the first argument. Int_prev returns the predecessor of the integer given as the first argument.

SEE ALSO

please_intro(1), *encompass_intro*(1), *Programming in Prolog* by Clocksin and Mellish

AUTHOR

Philip R. Roberts, Robert B. Terwilliger, Department of Computer Science, University of Illinois, 252 Digital Computer Laboratory, 1304 West Springfield Avenue, Urbana, IL 61801.

NAME

ptoplg – functions to enable execution of Prolog commands from Pascal or Path Pascal

SYNOPSIS

```
#include "ptoplg.h"

var inbuf, outbuf : plgbuf ;
   debug : integer ;

plgbufinit(inbuf) ;

plgbufappend(inbuf,'...   $') ;

ptoplgcall(inbuf, outbuf) ;

plgbufwrite(plgbuf) ;

ptoplgdebug(debug) ; /* debug is constant ON or OFF */
```

DESCRIPTION

The ptoplg functions provide a means for Pascal or Path Pascal programs to execute Prolog clauses.

The ptoplg layer is one layer in the PLEASE system (see *please_intro*(1)). PLEASE is an executable specification language. It is an extension of Path Pascal and supports the Vienna Development Method. In the PLEASE system, programs are specified using pre– and post–conditions written in predicate logic. These pre– and post–conditions are transformed into Prolog and executed by the UNSW Prolog interpreter.

Ptoplg is a text interface between Pascal and Prolog. Pascal programs can communicate to Prolog through plgbufs (Prolog buffers). An empty buffer can be created by declaring it as a plgbuf and then calling plgbufinit with it as the single argument. Strings can be appended to the end of the buf with plgbufappend. It is important to note that the strings passed to plgbufappend must end with a '$'. To clear a buffer, call plgbufinit with the desired buffer as the argument. Plgbufwrite writes the contents of the buffer on the standard output. Prolog commands can be constructed in these buffers using the plgbufinit and plgbufappend commands and then sent to Prolog using ptoplgcall. The command is placed in the inbuf. The results of the command executed by Prolog are returned in the outbuf. The plgbufs are 4K–bytes in sise.

The ptoplgdebug function turns debugging on or off for the ptoplg layer. If debug is set to ON, a constant defined in the header file, the debugging is turned on. If debug is set to OFF, also a contant defined in the header file, it is turned off.

FILES

${ENCOMPASS}/include/ptoplg.h
${ENCOMPASS}/lib/ptoplg.o

SEE ALSO

please_intro(1), *encompass_intro*(1), *plc_intro*(1)

AUTHOR

Philip R. Roberts, Robert B. Terwilliger, Department of Computer Science, University of Illinois, 252 Digital Computer Laboratory, 1304 West Springfield Avenue, Urbana, IL 61801.

NAME

set_lib – a Prolog library of set routines for PLEASE

SYNOPSIS

Standard set representation:

```
set(list([Element,...,Element]))
```

Concise set representation:

```
setc(LowElement,HighElement,NextFunction,EqualFunction)
NextFunction=FnName(_,_)
EqualFunction=FnName(_,_)
```

```
set_transform(+input,–output)
set_transform(setc(...),set(...))

set_member(+input,+input)
set_member(+input,–generated)
set_member(set(...),Element)

set_empty(+input)
set_empty(set(...))

set_union(+input,+input,–output)
set_union(set(...),set(...),set(...))

set_intersection(+input,+input,–output)
set_intersection(set(...),set(...),set(...))

set_difference(+input,+input,–output)
set_difference(set(...),set(...),set(...))

set_subset(+input,+input)
set_subset(set(...),set(...))

set_equal(+input,+input)
set_equal(set(...),set(...))

set_insert_element(+input,+input,–output)
set_insert_element(Element,set(...),set(...))

set_remove_element(+input,+input,–output)
set_remove_element(Element,set(...),set(...))
```

DESCRIPTION

Set_lib provides a Prolog library of predicates and functions to operate on sets.

Set_lib is a library of set routines for the PLEASE system (see *please_intro*(1)). PLEASE is an executable specification language. It is an extension of Path Pascal and supports the Vienna Development Method. In the PLEASE system, programs are specified using pre– and post–conditions written in predicate logic. These pre– and post–conditions are transformed into Prolog and executed by the UNSW Prolog interpreter. Calls to set_lib functions may be included in these prototypes.

Set_lib is written in Prolog (see *Programming in Prolog* by Clocksin and Mellish). There are two representations for sets, a standard representation and a concise representation. All set operations are performed on the standard representation of a set. The standard representation of a set contains a list that enumerates the elements of the set. The concise representation of a set includes the low element in the set, the high element in the set, the head of a clause that will produce the "next" element of the set, and the head of a clause that will determine if two elements of the set are "equal". The concise representation provides a means for giving a short description of a large set (too large to enumerate). See *lib_intro*(3) for a description of PLEASE data types and general information about operations on those data types.

The set library provides predicates for determining if an element is present in a set, if a set is empty, if one set is a subset of another, or if two sets are equal (are made up of the same elements). The set library provides functions for finding the union or intersection of two sets, finding the difference of two sets (the set difference A–B is the set of all elements in A that are not contained in B), inserting an element in a set, or removing an element from a set. All of these operations work on standard set representations. There is a function that converts a concise representation into a standard representation.

Set_transform takes a concise set representation as its first argument and returns the corresponding standard set representation as its second argument. It is important to remember that if a concise set representation is given, the user MUST provide function definitions for the next function and the equal function.

Set_member determines if its second argument (an element) is a member of its first argument (a standard set representation). If the second argument is a variable, set_member will work as a generator to successively generate the members of the set during backtracking.

Set_empty determines if its argument (a standard set representation) is empty.

Set_union takes two sets (standard set representations) as its first two arguments and returns the union of those two sets. Set_intersection returns the intersection of the first two sets.

Set_difference finds the difference of its first two arguments. The set difference A–B is all the elements of set A that are not in set B (A does not have to be a superset of B). Set_difference(A,B,C) will produce C=A–B.

Set_subset determines if its second argument is a subset of the first argument. Again, both arguments must be standard set representations.

Set_equal determines if its two arguments are equal. Both arguments must be standard set representations.

Set_insert_element inserts the first argument (an element) into the second argument (a standard set representation). The third argument is this new set. If the element is already present, the new set is the same as the old set.

Set_remove_element removes the first argument (an element) from the second argument (a standard set representation). The third argument is this new set. If the element was not present, the new set is the same as the old set.

SEE ALSO

lib_intro(3), *please_intro*(1), *encompass_intro*(1), *Programming in Prolog* by Clocksin and Mellish

AUTHOR

Philip R. Roberts, Robert B. Terwilliger, Department of Computer Science, University of Illinois, 252 Digital Computer Laboratory, 1304 West Springfield Avenue, Urbana, IL 61801.

Organizing Differences for More Effective Use by Programmers

Carol S. Beckman-Davies

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois

Organizing Differences for More Effective Use by Programmers

by

Carol S. Beckman-Davies

Preliminary Examination Statement

May 20, 1986

Programmers can use differences between versions of a program for a variety of purposes. Some people have acknowledged this usefulness, but few have done anything to help the programmer view differences more efficiently. Many researchers recognize the usefulness of tools which allow the programmer to refer to and manipulate programs in terms of their structure, lexical, syntactic, and semantic. The plethora of syntax-directed and language-oriented editors and environments surrounding these editors testifies to this recognition. No attention has been given to extending the ability to the viewing and manipulation of differences.

My thesis is that an interactive difference viewing system, which includes the ability to organize differences based on the lexical and syntactic structure of the program, can help a programmer use differences between versions of a program.

1. Why View Differences

A programmer, working in either development or maintenance, may want to view differences between versions of a program. During program development, several situations may prompt a programmer to look at the differences between versions of a program. If several programmers are working on a project, a programmer who makes a change to shared code could see the changes that he or she has made by looking at the differences between the version with the changes and the main version. In this way, the programmer can easily check changes to see if they look complete before inflicting them on the rest of the group. Checking whether the changes will affect someone else also should be easier.

A programmer might also want to see the differences between his or her own version of a file and another programmer's version. Each could have a version of the file if they plan to merge the versions later. Or each might have made changes to separate copies inadvertently. In either case, the versions must be merged. The programmers can check fairly easily whether the changes are compatible [Heckel, 1978].

A programmer working on maintaining a program has many reasons to look at differences between versions of a program. One of the most common is probably the need to find a new bug. While modifying a program, a programmer may accidently cause an error. Seeing the differences between the working version and the nonworking version can help pinpoint the cause more quickly [Heckel, 1978].

A project might have more than one programmer working on its maintenance. A large portion of maintenance is understanding what the program and the procedures that must change do and what ramifications a change might have. If a maintainer is planning a modification and has looked at the program before, but since then someone else has modified the program, differences could help the maintainer understand the program again. If the programmer remembers what the program did before the other changes, looking at the differences between the version with which he or she last worked and the current version could update his or her understanding of the program more quickly than looking at the entire program again.

Viewing differences can also help a programmer see how something was done in the past. This could be useful in two situations. Suppose the way some feature was implemented was changed. After several modifications to other things, it became clear that the new method was inadequate. It would then be necessary to go back to the old method or to try to incorporate some features of the old method into the new. Simply going back to a version which used the old method is not possible since other changes have been made. The programmers could look at the differences between the last version using the old method and first version using the new or the last version using the old method and the current version. These differences would show the differences between the two methods. (The latter would include unrelated differences, but might be necessary if the new method has changed since its inception.) Seeing these differences might also be helpful for a programmer who had another program to modify. If this other program uses either the old or new method of the program that has been changed, and the method must be changed in the other program, viewing the differences for the first program could be instructive.

Related to the second use of the differences mentioned in the previous paragraph, seeing differences of one program may help in customizing another. Suppose one program already has several versions for different machines or operating systems. A second program has been written for one of these systems but needs versions for others. The differences between versions for different systems for the first program will show a programmer the types of things in the second program that might need to change and how they might need to change.

One final reason for a maintainer to look at differences between versions is to find a quick fix for a bug in a version in the field. A version being used by a customer may have a bug which has been corrected in a version under development. The customer may need to have the bug fixed before the new version is released. The developers cannot simply give the customer the version under development since it may not be complete or giving the customer a new version might be against the company's policies. Looking at the differences between the customer's version and the developer's version may let a programmer find a fix for the bug without duplicating the effort that already went into fixing the bug in the development version.

Some reasons for viewing differences apply to both development and maintenance. At either stage, a programmer may have several changes to make to a version of a program. The programmer may elect to make the changes in stages. Each change or set of changes can be made and tested individually. After making some changes, the programmer may not remember which changes are complete, which partially complete, and which not started. The differences between the version the programmer began modifying and the version he or she has changed give an easy way to check the changes [Heckel, 1978].

After finishing a set of changes, the programmer can use the differences between the old version and the new one to check that all the changes are documented. The programmer can check that comments in the code document the changes, as well as seeing if existing comments have changed to reflect the new situation. The differences are also useful in looking at all the changes so that a record of what has changed may be kept, as part of a version control system [Thompson, 1980].

In either development or maintenance, going back to an older version may be necessary because of an incorrect change. However, more changes than just the incorrect one may have been made. Seeing the differences between the current version and the one that does not have the incorrect change will show the programmer what other changes will be lost by going back to the old version.

Finally, if a programmer wants to see a history of a program, he or she may want more detail than a summary of the changes made between each version, but not the text of all the versions. The differences between versions is a compromise in the amount of detail and may provide what the programmer wants without providing much excess information [Tichy, 1982].

2. Features for a Difference Viewing System

A system for viewing differences between programs should have many features. It should be interactive. The user should be shown one difference at a time and be allowed to skip backward and forward in the set of differences shown.

The exact difference between the two pieces shown should be highlighted in some way. It is very frustrating to the user to be shown a long line from each version that look very similar and not to be shown what makes them different. The user is forced to scan the text to determine the change himself or herself, a task which the computer could do easily, much more quickly, and with fewer mistakes. If the difference is flagged for some reason which is not visible, for example, blanks on the end of one line but not the other, the user will waste quite a bit of time trying to determine that no significant difference exists.

The user should be able to select parts of the program for which differences should be displayed. He or she may be interested in changes to only certain sections of the program. The viewing system should not force the user to look at differences which he or she does not want.

The user should be able to select the amount of context shown around a difference. Varying amounts of context may be needed for the user to identify where the change is.

The difference viewing system should present differences which are divided into logical sections. The changes to two statements, for example, should be shown as two differences regardless of the relative positions of the two statements. Changes to declarations and executable statements should be shown separately. When several changes are thrown together the user must sort out which parts of the differences shown belong to which logical section.

The system could determine context based on the logical sections of the program. This makes more sense than using line boundaries. However, the system must not issue a large amount of context. The user will not want a page of context, so context based on logical units must be tempered by the amount of output it would generate.

The difference viewing system should be able to summarize differences. The user may only need to know which procedures have changed, for instance. If none of the procedures in which the user is interested have changed, he or she need not look at all the differences. In other situations, knowing which procedures have changed may be enough to remind the user of the changes.

Summaries at various levels should be available. The information that the variable declarations changed could tell the user that the changes are or are not relevant to what he or she needs to know.

The user should be able to select the level of the summary from a set of levels. He or she may be looking for changes in variable declarations, or may know which procedures changed but want to see what statements have changed. The user may want to skip summaries and just see the text that changed.

The choices of summaries should be interactive. The user should be able to get a summary of which procedures, for example, have changed, then ask for more detail, that is, a summary at a lower level, for some of the procedures. Which summaries have changes shown in more detail should be selectable. The system should not force the user to see more detail for all the procedures. The ability to ask for more detail for particular differences should be possible until the text of the differences is displayed.

The system should allow the user to simply ask for more detail without specifying a level. The system should select a reasonable level of detail to present to the user. If the change is such that several levels will present nearly the same information, the system should use the lowest of these levels. The user should not be shown several levels which do not appreciably increase the information provided.

In order for the summaries to be useful, each construct to appear in summaries should have a name and a scheme by which an identifying name for a specific instance of the structure can be found. The name of the kind of construct is apparent. These would include procedure, variable declarations, assignment statement, while statement, and expression. Clearly each instance must get an identifying name. If three procedures change, having a system print "a procedure changed" three times is not very useful. Finding an identifying name for procedures is easy; but the system also needs a scheme for naming assignment statements, variable declarations, while statements, and other constructs. Some of the possible information the system could use as names includes: for a statement, the kind of statement augmented with some distinguishing feature, for example, assignment statement plus the variable whose value changes, for a while, if, or repeat, the kind of statement and the condition, for a case statement, case and the case expression, for a declaration, the name of the object declared. If versions for a letter, divided into paragraphs, sentences, and words are compared at the paragraph level, the text of the first sentence or the first words of the first sentence might identify the paragraph. In a list of objects with no particular distinguishing feature, the position, or number, within the list of the object which changed could be used.

These names should also be used for labeling differences so that the user can tell where the change is located. The change could be labeled by the procedure in which it is contained or by labels for all the summary levels which contain it, or some subset of this. The list of all the labels would be more informative, but could get too large to be displayed practically.

Another desirable feature for a system that compares programs is the ability to ignore formatting information. For virtually all reasons that a programmer wants to see differences between program versions, the formatting is irrelevant. With pretty printing programs and editors that format programs, having different formats for versions becomes more likely. The difference system should not produce differences which will never be important.

The system should be able to produce the differences quickly. A faster system will encourage more use.

To improve speed, the system should take advantage of existing, available information. An example of such information is the differences stored in a version control system. If this information will speed up the difference system, it should be used.

The difference system should be able to find differences between any two versions of a program. Differences involving the most recent versions will probably be needed most frequently, so having these combinations favored could improve efficiency, but all combinations should be possible.

For all the options which the system has, the user should not have to specify which option to use. The system should have reasonable defaults for all options. This will save time for the user.

In addition, the difference system should incorporate principles of good interface design. Also, it should be able to use screen capabilities of terminals when possible.

In sum, a difference viewing system should be interactive, highlight the exact difference, allow the user to restrict which parts of the program have differences shown, allow selection of the amount of context to show, divide differences into logical sections, use these logical sections to determine context when practical, summarize differences at various levels, allowing the user to select the level if he or she chooses and to interactively elect to see more detail for some differences, identify the location of differences with labels from the summaries, be able to ignore formatting changes in finding differences, be fairly fast, use available information from other sources, and work with any version from a version control system.

A difference viewing system should also be integrated with other tools. The interactions between the difference system and the other tools will help both.

The difference system should be integrated with an editor. This should allow the user to easily see differences between the version being edited and other versions. The user will then be able to see what changes he or she has made.

The difference system should provide the editor with an undo command based on the differences. A difference–based undo allows the user to view differences and select which to undo. (The user could be allowed to select differences to undo after having viewed all the differences, or be allowed to select differences to undo as they are displayed.) Undoing a difference consists of deleting the text that is in the new version and replacing it with the text in the old version. The changes that can be undone are limited by what versions for the file exist.

The undo should take advantage of the difference system dividing and summarizing differences. Dividing differences lets the user choose a smaller unit to undo. If changes were not divided, the user would not be able to undo one change without undoing all the others. Dividing differences makes a difference–based undo more responsive.

Summarizing differences also makes a difference–based undo more convenient. If all the changes in a procedure need to be undone, the user can get a summary of changes at the procedure level and ask for that difference, which could contain many textual differences, to be undone. The difference-based undo operating on the summary level allows the user to restore one procedure, say, to a previous state without having to request the undoing of each difference individually.

The user should be able to take the version he or she is editing, choose some differences to undo, and easily create another version based on the current version with the selected differences undone. This would help a programmer in debugging. If a bug has appeared, the programmer would be able to selectively eliminate changes in a temporary version without disturbing the current version. He or she could then test the temporary version. If the bug was still present, the programmer could go back to the undisturbed version and try undoing some other changes until the one(s) that are the source of the bug are found.

The editor with which the difference system is integrated should be a structural, e.g., syntax-directed or language-oriented, editor. This kind of editor will have the program represented in some tree form, such as an abstract syntax tree or parse tree. This would make dividing the differences into logical units or summarizing the differences easier for the difference system. For these tasks, if the program were not already represented in a tree form, the difference system would have to get it into such a form itself. Having the tree structure kept makes the difference system faster and more flexible.

Having a structural editor also allows the difference system to get a little extra information. The editor can fairly readily record which parts of the program have changed. This can help the difference system identify changes more quickly.

Another tool with which a difference system should be integrated is a version control system. As mentioned before, the difference system should be able to compare any two versions. Also, the difference system should use information available in the version control system. The version control system will store multiple versions by storing differences between versions. If the user asks to see differences for versions for which the difference is stored in the version control system, the difference system should use this to locate the differences. Further, if a sequence of differences between the versions exists, the difference system can combine these to locate differences in the two versions. Using the information in the version control system will make locating differences faster.

The difference system can also help the version control system in merging two versions. The difficulty with merging comes when a conflict arises--two versions insert different text in the same place, or one version deletes text around the location at which another version inserts, for example. The difference system can help in several ways.

Use of the divisions of the differences and summary levels can help when two versions both have code inserted at the same location. The differences can be marked to indicate what kind of section contained them. If the two sections to be inserted came from different kinds of sections, this could order the sections. For example, suppose one version had declarations inserted at the end of the declaration section and another had statements inserted at the beginning of the executable statements. To a merging program which considers the program as text, this would look like two insertions at the same spot. But if the differences were marked with which kind of section they were, a merging program could find the kind of section on the left and right of the point of insertion and place the sections by the same kind of section as that from which they came.

Dividing the differences into logical sections would help if each version had inserted both declarations and statements in the same spot. The new declarations and new statements, though contiguous, would be divided into separate differences. Thus in the merged version both new declaration sections could be placed together, before the new statements.

If the difference system is integrated with a structural editor, differences can be done on the tokens. Having this eliminates some conflicts. Changes which were made to the same line in two versions and which are separated by a token will not conflict. This situation could arise commonly when elements are added to a list, such as a list of variables being declared or the definition of an enumerated type.

When conflicts arise, the user must look at the problem area and edit the merged version so that it is correct. This should be interactive in a manner similar to difference viewing. The interactive system could take the user from one conflict to another. Allowing the user to easily see other parts of the program so that he or she can see the results of merging that did not cause a conflict but may bear on how to resolve one is important. The conflicts should be labeled by their locations. Getting summaries of where conflicts are located might also be useful. With several versions being merged, the person attempting to resolve conflicts may not know enough to resolve them all. The user could select the conflicts in the procedures which he or she changed and let other people resolve others.

Having conflicts resolved with the aid of a special tool allows commands specific to merging to be included. The specific commands would depend upon how conflicts are represented. Some possibilities include leaving the code as it is, selecting the text of one version or the other, or asking for the text from one version followed by that from the other.

Another tool with which a difference system could be integrated is a program slicer. This will make the difference system more useful, but not the slicer. A program slicer takes a point in the program and a set of variables and finds all the statements which affect the values of those variables at that point. In essence the result is a program which would give as results the values of the set of variables at that point.

With a program slicer integrated with the difference system, the user should be able to ask for only differences that affect the value of selected variables at a certain point. This might reduce the amount of text that the user would need to see.

The difference system can also be integrated with any system which does incremental analysis which can be batched. Some possible tools that are amenable to incremental analysis and whose results are not needed immediately after each change include an incremental recompilation system, a tool which performs consistency checks between the source code and its documentation or specification, a test case generator, a test coverage analyzer (perhaps with data flow analysis), and software management systems. Use of one tool should not interfere with the use of any other tool. A new tool could be added to this system easily.

3. Previous Work

3.1. Uses of Differences

Differences between strings have many uses. They are used extensively in biology and speech recognition. The first use in computer science, as indicated by Sankoff and Kruskal [Sankoff and Kruskal, 1983] and Hall and Dowling [Hall and Dowling, 1980], was in spelling correction. The problem is to find a correct spelling of a misspelled word. The solution is to find, out of the set of possible correct words, either one that is the closest or one close enough to the incorrect word.

Several methods are based on abbreviating the words. Blair [Blair, 1960] devises an abbreviation by eliminating letters based on their positions in the word and the letters' frequency of occurrence. Words are matched based on their abbreviations. If no match is found, the system gives up. If more than one match is found, larger abbreviations are used until only one match exists.

Davidson [Davidson, 1962] also uses abbreviations to retrieve names in an airline reservation system. His system takes the first letter of the surname, the first three characters remaining after eliminating all vowels, *hs*, *ws*, and *ys* and removing one occurrence of any letters doubled after this. The last letter included is the first initial. Names are retrieved solely from the abbreviation. Additional information, such as the person's phone number is used, if available, to eliminate multiple retrievals. If this is not possible, the operator receives all the matching records and selects the correct one.

Davidson's system does not rely on always finding a match. If no record exactly matches the abbreviation, the records which best match the abbreviation are retrieved. How good the match is is determined by listing the character positions that match in both abbreviations and finding the length of the longest increasing subsequence. This is also the length of the longest common subsequence of the two abbreviations.

In general, Blair's and Davidson's methods are applicable only to spelling. They offer no help in comparing words or other strings for any other purpose.

Faulk [Faulk, 1964] defines three measures of similarity between strings. Each is a number between zero and one, with a larger number indicating more similarity. The three numbers indicate the extent to which the strings share common elements, the common elements are in the same order, and the common elements are in the same positions. These measures help choose the best match out of a list, and can suggest how similar two strings are, but are not helpful in showing the differences.

Damerau's [Damerau, 1964] method attempts to correct words with one typing error: a substitution of one character for another, insertion or deletion of one character, or transposition of two (adjacent) characters. His method is specific to checking if a word could be derived from the given word by one of these errors. It also includes a few steps to decrease the number of words in the vocabulary which must be tested.

Alberga [Alberga, 1967] took several spelling correction methods and a set of misspellings from spelling exams to see which method did the best job. The results of this study are not interesting for finding differences, but the paper does give an interesting summary of various spelling correction methods.

Morgan [Morgan, 1970] is interested in correcting spelling and typing errors to decrease the number of runs a user must make to get job control and programs correct. His method uses semantic information to narrow the search for possibilities. Then Damerau's method is applied to find a correct word from the list of possibilities. The semantic information that Morgan uses includes what items are in the follow set and which identifiers in the symbol table are of the correct type.

Another area in which differences are used is in correction of syntax errors. Several methods use a cost function to help determine which correction to make. The cost, in essence, is based on the edit operations needed to transform the input to one of the possible corrections. Anderson, et al. [Anderson, et al., 1983], Graham and Rhodes [Graham and Rhodes, 1975], and Mickunas and Modry [Mickunas and Modry, 1978] all use the costs of inserting and deleting symbols to find the cost of a correction. These methods do not use the techniques for getting the minimum edit distance, but the ideas are similar.

Tai [Tai, 1978] actually uses one of the methods of minimizing edit costs with insertions, deletions, replacements, and transpositions allowed. After finding possible corrections, the method to find the edit cost is applied to find the correction which is closest to the input text.

Perhaps the most widely recognized use of differences in computer science is in storing multiple versions of a file. If someone wants to save several versions of a file, the versions will usually have more in common than different. Instead of saving the entire text of all versions, which would usually consist of a large amount of common material, one version can be saved in its entirety along with enough information to produce the other versions from this one. If versions are kept in this way, a set of tools should store the information necessary to retrieve versions and the user should be able to specify the version desired and have it retrieved automatically. As long as tools exist to keep track of versions, they usually perform other functions, such as keeping logs of what changes have been made and providing exclusive use of a version to a user.

Despite the savings in space that can be achieved by using differences some systems which save versions save the complete text of each version that is kept. The Distributed Programming Assistant [Ramsay, 1983] keeps all versions of programs and also all the supporting files that are ever produced. The Project Automated Librarian [Prager, 1983] stores entire copies of versions, but saves only a set number.

Other systems store multiple versions and save the common parts only once but do not use differences. These systems keep all the versions in one file and have control information so that the appropriate lines are used for the desired version. One system [Stanaway, et al., 1979] uses conditional assembly to get the correct statements for the desired version. Another [Hague and Ford, 1976] keeps the file with control information and has a tool to extract the version needed.

Cargill [Cargill, 1980] has developed a system that uses a hierarchical directory structure to store versions. The system was developed to store the programs for an operating system intended to run on different machines. Each machine has some functions which must be customized. The system is set up with a directory for each function. In it is the source for the common function. Any machine that needs something else has a subdirectory with the files it needs. Some space is saved since common files are stored once, but anything in common between the versions for specific machines will be duplicated.

Many systems use differences saved as edit scripts to save multiple versions. A good example of this is SCCS, the Source Code Control System [Rochkind, 1975]. It saves the original version. Each additional version is saved by storing the difference between it and the version before it.

Several systems have been patterned after SCCS. Two of these are the systems developed by Pedersen and Buckle [Pedersen and Buckle, 1978] and Bauer and Birchall [Bauer and Birchall, 1978]. Pedersen and Buckle's system allows a tree structure of versions. Bauer and Birchall performs many management functions as well as merging differences in object files when possible.

Another system which uses differences to save multiple versions is RCS, the Revision Control System [Tichy, 1982]. RCS allows a tree structure of versions. Instead of storing the oldest version and differences to generate the more recent version (forward deltas), RCS stores the most recent version and differences to generate the older versions (backward deltas). This allows the newer versions, which presumably will be accessed more frequently, to be generated more quickly.

Another version control system [Kaiser and Habermann, 1983] concentrates on specification and management issues, rather than space considerations. What method it uses for storing versions is not stated.

A fourth use of differences in computer science is in updating text which is already at the receiving site. Differences can be used to update programs, manuals, and display screens. When a site has a version of a program or data set and needs a new version, the differences will usually be shorter and can be transmitted more quickly.

Screen oriented programs also use differences to attempt to reduce the amount of characters transmitted to update the screen display. Gosling [Gosling, 1981] describes an algorithm and a heuristic for updating the display of a screen editor, if terminal has certain abilities.

Some attention has been given to providing differences that can be viewed. Suppliers of operating systems often provide a general utility for finding differences between text files. UNIX [UNIX User's Manual, 1984] and VMS [Digital Equipment Corp., 1985] are some examples of operating systems which provide such a tool.

A tool under development that helps display differences between versions of programs is an editor that edits multiple versions of a program [Kruskal, 1984]. The user of the editor specifies which versions to edit. Any changes made apply to all the versions being edited, or a subset of those if the user so specifies. Parts of the text that differ among the versions being edited are highlighted. The editor has a restore command that lets the user put text from an older version into the versions being edited.

3.2. Finding Differences

The use of general difference finding algorithms seems to have developed in biology before developing in computer science. The first mention in the computer science literature seems to be in 1974 in two separate papers [Sellers, 1974] [Wagner and Fischer, 1974]. Sellers presents an algorithm that takes O(m^2n) time and space, where m and n are the lengths of the strings being compared. This algorithm finds the smallest number of changes (deletion of a character from either string or replacement of one character with another) needed to convert both strings to the same string. Wagner and Fischer present an algorithm to find the number of insertions, deletions, and replacements of single elements needed to convert one string into the other. Their algorithm uses O(mn) time and space.

Lowrance and Wagner [Lowrance and Wagner, 1975] give an algorithm for an extension to Wagner and Fischer's problem. They allow swapping two adjacent elements or two elements that would be adjacent after all the deletions are performed but before any insertions are done. This algorithm also uses O(mn) time and space.

All the algorithms mentioned so far are based on a dynamic programming approach to the problem. The solution is found for substrings of the two strings. One element is added to one substring and the solution for the new substrings is found based on the solution for the smaller substrings. The substrings used are prefixes (or suffixes) of the two strings. The solution is found for each pairing of substrings. So each entry in an $m \times n$ (or $(m + 1) \times (n + 1)$ if zero length prefixes are included) matrix is found. Masek and Paterson [Masek and Paterson, 1980] attempt to find the solution more quickly by precomputing all possible differences between costs in the matrix for submatrices, then combining the appropriate precomputed values for the particular strings. This produces an algorithm that executes in time of O($mn/\log n$), but which can only be used in problems with a finite alphabet.

Heckel [Heckel, 1978] proposes a method which is not based on dynamic programming. Heckel describes his method in terms of files and lines in the files. The algorithm enters each line into a symbol table and records information, such as the number of occurrences of the line in each file, about it. If a line in the symbol table has exactly one occurrence in each file, the occurrences are considered the same. Lines which are identical and are adjacent to lines considered the same are considered the same. Any other lines are considered to be inserted or deleted. This method finds lines that have moved as well. Its weakness is in relying on having many lines with exactly one occurrence in each file to get a good match.

Tichy [Tichy, 1984] has developed a method for finding block moves. This method includes any element in both strings in a block move. This minimizes the number of elements inserted. Then the number of moves to generate the rest of the string is minimized. By using a suffix tree for the string, the algorithm can run in time and space of O(m + n). The advantage of Tichy's method is that it attempts to reduce the amount of space the editing commands take. Presumably an insert command, which must include the text to insert, takes more space than a move command. A disadvantage is that the original string will not be accessed sequentially, and so, unless it can be accessed randomly, rebuilding the new string will normally require multiple passes through the original.

A problem closely related to the one of finding an edit script to convert one string into another is that of finding the longest common subsequence of two strings. The solution to the longest common subsequence problem can be used to produce an edit script by inserting elements in the new string but not the common subsequence and deleting elements in the original string but not in the common subsequence. Likewise, any method that finds edit scripts with insertions and deletions can be used to find the longest common subsequence. Methods that include replacement and transposition can also be used by setting the cost of a replacement or transposition above the cost of an insertion and deletion together so that inserting and deleting will always be preferred.

Hirschberg [Hirschberg, 1975] takes Fischer and Wagner's algorithm and notes that the values of the ith row depend only on the (i − 1)th row. Thus the length of the longest common subsequence can be found using O(m + n) space. Finding the sequence itself is more difficult but can also be done using a linear amount of space.

Hunt and Szymanski [Hunt and Szymanski, 1977] developed an algorithm that works well when the strings match in few places. The method keeps a list for each position in one string of matching locations in the other string. It takes O((r + n)log$n$) time and O(r + n) space, where r is the number of pairs of matching positions.

Hirschberg [Hirschberg, 1977] developed two other algorithms. One works well when the length of the longest common subsequence is short and the other works well when it is long. If p is the length of the longest common subsequence, the first runs in $O(pn + n\log n)$ time and the second in $O(p(m + 1 - p)\log n)$ time.

Nakatsu, et al. [Nakatsu, et al., 1982] have another algorithm that works well for strings with a long common subsequence. Their algorithm compares $(m - p)n$ elements of the strings and computes $(p + 1)(m - p + 1)$ elements of a two dimensional array, where again p is the length of the longest common subsequence.

Finally, Hsu and Du [Hsu and Du, 1984] presented some improvements of two known algorithms. Where Hirschberg's algorithm uses a linear search, theirs uses a binary search. They also recommend a faster merging algorithm for part of Hunt and Szymanski's algorithm.

Several people have worked on bounds on the complexity of the longest common subsequence and string editing problem. Assuming the only type of comparisons allowed tell whether two elements in the strings are equal or not equal, Aho, Hirschberg, and Ullman [Aho, Hirschberg, and Ullman, 1976] developed three lower bounds on the number of comparisons needed to solve the longest common subsequence problem. If s is the size of the alphabet and both strings are of length n then the lower bounds are $s/2(n + s/2)$ if $s \leq n$, $3/4ns$ if $n \leq s \leq 4/3n$, and n^2 if $4/3n \leq s$. If no comparisons between elements in the same string are allowed, the lower bound is n^2 if $s \geq 3$.

Wagner [Wagner, 1975] looked at the extended string editing problem, that is producing an editing sequence of insertions, deletions, replacements and transpositions that will convert one string into the other. He let some of the operation costs be infinite and showed that some of these problems are NP–complete.

Wong and Chandra [Wong and Chandra, 1976] used the same comparison model that Aho, Hirschberg, and Ullman used. Also, they assumed an arbitrarily large alphabet. With these assumptions, the problem of developing an edit sequence with insertions, deletions, and replacements has a lower bound on the number of comparisons of O(mn).

Hirschberg [Hirschberg, 1978] looked at the longest common subsequence problem again. If comparisons between string elements can return a result of less than, equal, or greater than, a lower bound on the number of comparisons needed is $n\log n$ where n is the length of both strings.

Attention has also been given to the problem of comparing trees. Selkow [Selkow, 1977] developed an algorithm patterned after Sankoff's [Sankoff, 1972] and Wagner and Fischer's. It allows changing the label of a node and insertion and deletion of leaf nodes. This is not to say that only nodes that are leaves in the original tree may be inserted or deleted, but rather, at the time that a node is inserted or deleted, it must be a leaf. So to delete an interior node, all its descendents must be deleted. The algorithm takes O(mn) time and space, where m and n are the number of nodes in the original and new trees.

Tai [Tai, 1979] developed a less restrictive algorithm. It allows interior nodes to be inserted or deleted. When an interior node is deleted, its children are attached to its parent in the deleted node's position. If an interior node is inserted, it may take some of its parent's children as its own, in such a way that deletion is the inverse of insertion. This algorithm operates in O(mnh^2i^2) time, where h and i are the heights of the original and new trees.

Wilhelm [Wilhelm, 1981] was interested in finding a mapping between tree nodes that would map all nodes in the original tree with a node in the new tree with the same label to some node in the new tree and preserve the most parent–child links in the tree. The algorithm is designed for trees in which all nodes in the original tree have unique labels and all nodes with the same label have the same number of children. Wilhelm gives an analysis of the time the algorithm would take for two types of trees, a complete tree and a degenerate tree, both having all interior nodes with r children. If h is the height of the original tree and n is the number of occurrences of the nodes in the original tree in the new tree, the time for the complete tree is O($n(nr)^h$) and the time for the degenerate tree is O($n^{h+1}(r-1)$).

Tichy [Tichy, 1985] has developed an unpublished algorithm for finding differences between trees. The algorithm assumes that each node of the same type has the same number of children. The trees are linearized by taking the preorder traversal. Then an algorithm to find the differences between strings is applied.

3.3. Problems with Existing Work

Although some researchers have recognized the usefulness of differences between versions of programs to programmers, little emphasis has been given to differences. Work with differences has dealt mainly with their use in storing versions so that less space is required than would be if versions were stored in their entirety. For programmers wishing to view differences between program versions, little support beyond the rudimentary tools can be found.

Version control systems, though they have the versions that would be compared and sometimes use differences to store these versions, for the most part do not provide facilities for programmers to see differences between versions. Some exceptions, such as RCS and SCCS, exist. These both provide a command which will show the differences between two versions. The commands check out the desired versions and use a UNIX diff command to compare them. It seems that other version control systems do not attempt to help programmers see differences.

When differences are provided to the programmer, they show what sections have had changes made to them, without regard to whether several unrelated changes have been made to the section. Most difference tools also cannot distinguish between an actual change in the program and a change in the formating. Current algorithms really can do no better than this. With a program represented as text, the algorithms have no basis for deciding anything beyond which sections have changed. With many tools and environments treating program as trees—abstract syntax trees or parse trees—doing a better job should be possible. The four existing tree comparison algorithms do not seem up to the task.

Wilhelm's algorithm is clearly inappropriate. The requirement that all nodes in the original tree have unique labels would not be met.

Selkow's algorithm is not general enough. For both abstract syntax trees and parse trees, interior nodes can be deleted and inserted without all the descendents being deleted. An example of this is changing a repeat statement into a while statement. The statement block, which could be large, would not change.

Tichy's algorithm is designed more to store versions of trees compactly than to find differences to display. Changes made to several adjacent subtrees would all be one difference to this algorithm. This is the same problem the string comparison algorithms have. Also, the restriction to trees in which all nodes with the same label have the same number of children would generally be a problem. The grammar for a tool using a parse tree might have multiple rules with differing numbers of elements for one nonterminal. Abstract syntax trees and parse trees using regular right part grammars also would have nodes with the same label and differing numbers of children. A good example of this is lists of objects.

Tai's algorithm has the most promise. It is general, so that it will produce differences for parse trees. However, it may not produce the required information. Because changes can be adjacent, changes that should be divided may still appear as one difference. Alternately, changes might be reported at a lower level than the person viewing the differences would want. This algorithm is also too general. Changes made to a parse tree are more restricted than deleting or inserting any node. It should be possible to devise an algorithm specific to the type of changes that occur for parse trees and that would be faster.

4. Experience

The SAGA editor has a simple system which generates differences between versions of a program. The user begins by telling the system to use the version he or she is currently editing as the base version. All differences will then be shown relative to this base version until the user sets another version to be the base.

As the user edits the program, the editor records where changes are made by setting a field in the terminal nodes. The modified fields are set in nodes which are inserted and in nodes whose neighbors are deleted. The difference system uses the modified fields to locate the changed sections of the program. The system saves the information about the differences and reuses it if the user asks to see the differences again before he or she makes additional changes to the program.

The display of the differences is screen-oriented. The differences are displayed one at a time. If the text does not fit on one screen, the user may scroll it up or down. The screen is divided between the part of the difference that shows what the program currently has (the new part) and what it had when the base was set (the old part). These parts can be scrolled independently or together. The system also highlights the tokens which have changed (as opposed to the context which the user requested around the change).

The difference system also includes the potential for an undo command. The user can select a difference to undo. The system will produce a script which will delete the text in the new part of the difference and reinsert the text that had been in the program (the old part of the difference) for the editor to execute.

5. Proposal

Many of the features of a good difference system would be straight-forward to implement. Either similar features exist in other types of tools or methods for gathering and using the necessary information are clear.

Other features are not as easy. I want to concentrate on two of these: dividing differences into logical sections and summarizing differences. Three problems related to these are determining the conditions that the sets of nonterminals for dividing differences and for summarizing differences must meet, devising a scheme for storing the methods to find names for the summarized sections, and determining criteria for deciding at what level summaries of differences should be made.

Many programming environments now include program editors which keep the parse tree or abstract syntax tree for the program. With a tree representation available, it should be possible to use the structure of the trees to divide contiguous sections which have changed and which would normally be shown as one section, into several, more reasonable, sections. This problem can be divided into four parts. These are limiting the subtrees that must be compared, eliminating some subtrees of those subtrees from consideration, finding the differences, and displaying the differences.

Since it seems that tree comparison schemes general enough to use for changes in parse trees are expensive, using one to compare entire trees is impractical. Given the nature of parse trees, a change in the tree will always include a change in the leaves. Also, for parse trees, looking at just the leaves is meaningful. Thus for parse trees it is possible to find the differences in the leaves, considered as strings of tokens, and to use this information to find subtrees which contain changes.

The costs of the tree comparisons depend on the number of nodes in the trees or the heights of the trees. The subtrees compared should be as small as possible, while being large enough to produce useful information. What subtrees are compared can be based on the string difference between the terminals of the tree. For each different section, the subtrees in the new and old trees that correspond to the change in the terminals can be found. Since the idea is to present differences in logical sections, treating each changed section of the terminal lists separately seems reasonable.

Several methods can be used to find subtrees for a changed section of the terminal lists. A simple approach would be to find the smallest subtree which contains all the terminals that have changed, and to do this in both the new and old trees. A problem with this approach is that the subtree will not necessarily contain all the changes in the tree structure caused by the change in the terminal list. For example, with an LR(1) parser, the extent of the effect of the change to the tree to the left of the change in the terminals is limited, but the effect to the right is not. To inform the user of all the ramifications of the change, a subtree which contains all the changed terminals as well as all parts of the tree that changed because of them should be included in the subtree.

One way to accomplish this would be to find the subtree based on an incremental parsing algorithm. The incremental parsing algorithm can find a subtree that contains all the changes to the tree caused by a change in the terminal list. For the Ghezzi and Mandrioli algorithm [Ghezzi and Mandrioli, 1980], the subtrees in both the new and old trees can be found since the nonterminal at which the algorithm stops matches in the trees. This has the added advantage of finding two subtrees which have the same root to compare. This is not essential, but is assumed by some tree comparison algorithms, for example Tai's.

The subtrees found by an incremental parsing algorithm have another advantage. Changes which are related but which are not contiguous will be grouped together into one subtree. Changing a *while* to a *repeat*, for example, requires changes that can be widely separated, but the subtree containing all the changes associated with changing *while* to *repeat* and deleting the condition will include the change to insert *until* and a condition at the end of the loop. Grouping related changes would present differences more reasonably. A problem arising from this is how to recognize unrelated changes that also appear in the subtrees and how to deal with them. If the statement of a *while* that changed to a *repeat* also changed, the changes to the statement would be included in the subtrees for the *while*, but would not be related. Another problem arises if a subtree chosen for one change includes a previous change which has already been grouped and matched. Some way to deal with this would have to be developed. The subtrees to compare obtained from an incremental parsing algorithm have some very nice properties, but also have a potential problem in finding useful information. The subtrees' roots may be a nonterminal that is meaningless to the user. A question is whether this matters.

The grammar used by the LR parser will contain nonterminals which exist solely to make the language easier to parse. A good example of this is nonterminals and production rules added to produce an unambiguous grammar. The user will not care about seeing differences based on all the nonterminals of the grammar. Even if all the nonterminals represented unique entities, the user would not want to see differences based on all of them. That would provide differences on too fine of a scale. Thus the information shown to the user should be based on some subset of the nonterminals of the grammar in which the user will be interested.

If having the roots of the subtrees be "interesting" nonterminals is important, such subtrees could be obtained in several ways. One possibility is to find the smallest subtree which contains all the changed section of the terminals and whose root is an interesting nonterminal, and to do this in both the new and old trees. This would no longer guarantee that all parts of the trees affected by the change in terminals were included in the subtrees. However, since the purpose of the comparison is to display changes to the user in a logical fashion, and not to record changes to the parse tree per se, this may not matter. The advantage of grouping related changes into one subtree would be obtained with this method as well. All the changes to an entity in which the user is interested would be in the subtrees. As with finding the subtrees based on an incremental parsing algorithm, this method could find a subtree which would include a change before this one which has already been grouped and matched or a change between two related changes.

One advantage selecting the subtrees based on an incremental parsing algorithm has over selecting based on interesting nonterminals is that the roots of the subtrees will be the same. This is not essential even for Tai's algorithm, since artificial matching roots can be added to the subtrees. However, it may be desirable. If so, in both the new and old trees, the smallest subtree which contains the changed terminal section and whose root is an interesting nonterminal which matches the root of the subtree from the other tree could be chosen. This poses some problems. Let N_N be the root of the smallest subtree containing all the changed terminals in the new tree, and N_O be defined similarly for the old tree. Let n_N be the number of ancestors of N_N which are interesting nonterminals, and n_O be defined likewise for N_O. Then in the worst case finding matching ancestors would take time of $O(n_N n_O)$. Another problem is choosing between multiple matches. If N_N matches N_O, M_N matches M_O, M_N is an ancestor of N_N, and M_O is an ancestor of N_O, the choice is clear. But if instead N_N is an ancestor of M_N, the choice is not obvious. Some criteria for choosing would have to be developed.

Another way to get roots for the new and old subtrees so that they are interesting nonterminals would be to combine finding subtrees using an incremental parsing algorithm and finding nodes that are interesting nonterminals. The incremental parsing algorithm could be used to find the new and old subtrees which contain all the changes to the parse tree caused by the changed section of the terminal list. Then in both the new and old trees, the first ancestor of the root of this subtree which is an interesting nonterminal could be found and taken as the root of the subtree for comparison. This combines most of the advantages of the two methods.

One advantage not achieved by combining the methods is that of obtaining subtrees for comparison which have the same root. As when subtrees were found based solely on interesting nonterminals, this could be remedied by finding in both trees, the smallest subtree which contains the subtree found based on the incremental parsing algorithm and whose root is the same interesting nonterminal as that for the subtree in the other tree. This of course has the same problems as before.

The discussion so far has mentioned potential advantages of various methods, but no disadvantages. Aside from not possessing all the advantages of another method, the only area in which disadvantages arise seems to be the size of the subtrees obtained. The purpose of restricting the subtrees for comparison is to decrease amount of space and time required. Since some of these techniques for finding the subtrees get larger subtrees, they do not accomplish the major goal as well as other techniques. Trying to get interesting nonterminals that match or an interesting nonterminal whose subtree contains the subtree based on an incremental parsing algorithm will necessarily find larger subtrees than some of the other methods.

Many methods can be employed to limit the subtrees that must be compared. Which is best depends on several factors. First, various methods can be employed in subsequent steps. One method for limiting the subtrees might work best for one method of finding the differences, while another might work best for another method of finding the differences. Another factor might be the particular grammar used for the parse tree. If a situation in which one method performs better than another never or rarely arises with a particular grammar, the one producing the smaller trees for comparison would be better. A third factor is the set of nonterminals that make up the interesting nonterminals. Some requirements must be imposed on the set. What these requirements are will affect the outcome of the limiting process. Further, given a set of requirements, different sets satisfying the requirements may cause one method to perform better than another. Finally the expectations of the user of the system will affect the choices. A user willing to accept occasional odd results from a faster system will prefer a different method than a user who demands perfection no matter what the cost. Comparing these methods to see which result in the smallest trees, the fastest comparisons, the fewest odd results, and the most information will be interesting.

The next step in dividing differences into logical sections is to eliminate subtrees from the subtrees found in the first step. For clarity, call the subtrees that are selected for comparison the trees, so that subtrees will refer to the subtrees of these trees that are to be eliminated from consideration in finding differences.

Several reasons for eliminating subtrees exist. One reason is similar to the reason for limiting subtrees—the time and space required to find the differences should be less since the trees to compare will have fewer nodes. Another benefit of eliminating subtrees is better results from the tree comparison. If some section of terminals is unchanged and the trees above them match, the sections should probably match, so they should be reported in that way. Depending on the tree comparison algorithm and the tree structure, these sections may or may not be reported as changed. A final advantage of eliminating subtrees is that it might make more of the tree comparison algorithms applicable. For example, Selkow's algorithm could be used but it will report sections that have not changed as changed (this will be explained when the third step, comparing the trees, is examined). This would probably make the algorithm unusable unless eliminating subtrees can match enough unchanged subtrees so that Selkow's algorithm reports few spurious changes.

Several methods could be employed to eliminate subtrees. One could start from the leaves and go up as long as the trees were the same. This would start from unchanged sections of the terminal list. These could be in the trees to compare because of the grouping of related changes. An example of this is the statement of the *while* if a *while* is changed to a *repeat*. Starting from the terminals, the trees above could be compared until the trees are different. Parts of the unchanged sections on the left and right edges may be dropped from the subtree as parts of the tree on the left and right do not match or include parts of the changed section of the terminal lists. The subtrees obtained should be the largest subtrees in the trees that contain only unchanged terminals as leaves and which are the same. These matching subtrees could be found for each section of unchanged terminals included in the trees.

Another possibility is to find a series of trees that match rather than just one for each unchanged section of terminals. This would presumably remove more of the tree from the part that must be compared, which should make the comparison faster. The larger number of sections that are already matched with which the tree comparison would have to deal might mitigate this. The subtrees that are matched might be small. This would tend to make bookkeeping more expensive for little benefit. It would also tend to make reporting the differences to the user more complex. The user would see many small matches in a difference, which could serve to obscure the real change.

As with limiting the trees to be compared, a question with eliminating subtrees is whether the root of the subtree should be a nonterminal that is interesting to the user. The reporting of the change would probably be more meaningful if the report mentions the unchanged sections. The subtrees matched would be the largest subtrees which contain only unchanged terminals, which match, and whose roots are interesting nonterminals, or a series of such subtrees. The matching subtrees would be smaller if an interesting nonterminal must be the root. For a series of matched subtrees requiring the roots to be interesting nonterminals might be better. Depending upon the set of interesting nonterminals, the size of the subtrees which are matched would be reasonable. The worry about too much overhead for too little benefit and a confusing display for the user could disappear.

Another possibility for eliminating matching subtrees is based on incremental parsing. Ghezzi and Mandrioli's incremental LR(0) parser contains a section which will reuse parts of the tree to the right of the change. The subtrees that are reused would contain, at least in part, trees that are the same. One problem with the trees that are reused is that parts of the tree can contain changed terminals. This part of the reused subtree could be avoided by taking the largest subtree that is reused but does not contain any changed terminals. A series of such subtrees could also be found with this method.

One question with finding the trees for comparison which does not arise with subtrees for elimination is whether the roots of the subtrees should match. Because the entire subtrees match, this does not happen.

An issue to examine for eliminating subtrees is how it is affected by the method used in the previous step and how it affects subsequent steps. It should not be affected by what are selected as roots of the trees to compare, that is, by what method is used to find the trees. Using similar methods in both steps might produce better, more internally consistent results. For example, if the roots for the trees to compare are selected to be interesting nonterminals, the roots of the subtrees to eliminate could be selected to be interesting nonterminals.

The method used to find the subtrees to eliminate will affect the subsequent steps. If subtrees are eliminated, then the methods finding the differences must handle subtrees that have already been matched. If a series of matched subtrees are eliminated, the comparisons must account for that. The comparison method used must change depending upon what is done in the elimination step. Another affect of the elimination step could be to make the results of the tree comparison better.

Eliminating subtrees could also affect how the differences are displayed. If subtrees are eliminated, either the text of the eliminated subtrees or a short representation of the matched subtrees can be displayed. Also to be decided is whether subtrees that are eliminated from the comparison should be treated differently from parts of the trees that the comparison says are the same.

Eliminating subtrees can be done in several ways. I want to try the methods for this in combination with the methods for selecting trees for comparison to see which produce the best results. Another possibility to compare is not eliminating subtrees at all. This will help show whether effort on that is really useful.

The third step in dividing the differences into logical sections is the actual comparison of the trees. Many possibilities exist for this step. Now examine how three of the existing algorithms might perform in grouping and separating differences if elimination of subtrees is not done, then at what implications the elimination of subtrees has for these algorithms, what other possibilities for comparing the parse trees exist, and, finally, what forms of trees might be useful for the tree comparisons.

The results the tree comparison algorithms produce should be able to group related differences and separate unrelated differences. Another problem with which the algorithms would have to deal is a changed section that contains parts that need to be grouped and parts that need to be separated. This might present more difficulties for the algorithms and should be considered in evaluating which is best. It does not seem to add enough to the simple analysis here to be considered now.

One of the existing algorithms is that developed by Selkow. This algorithm allows insertions and deletions only at leaves. For grouping changes to the same structure, this algorithm will work well. If an *if* is changed to a *while*, for example, the algorithm will report that the entire subtree constituting the *if* must be deleted and the entire subtree for the *while* inserted. This is a result of the requirement that only leaves may change. The tree will have a node that indicates an *if*. This will have to change to one that indicates a *while*. For that node to change, it must become a leaf, so all its children must be deleted, the node changed, then all the children of the new node inserted. If a few more internal nodes that are present to make the grammar more amenable to parsing also change, they will also need to be inserted or deleted, but the whole change from an *if* to a *while* would still be one group of nodes to delete and one group to insert. Thus the related changes would be grouped together. One problem with this is that the fact that the condition and statement have not changed is not detected. The user would have to scan the text, which could be a considerable amount, to determine whether anything had changed beside the *if* to *while*.

Selkow's algorithm may or may not separate unrelated changes, depending upon the tree structure. If the levels of nodes in changed structures do not change, then Selkow's algorithm will match those nodes. Then the changes to the contiguous structures can come out as separate changes to the tree. However, if the levels of nodes common to both the new and old trees change, the algorithm will say to delete and insert everything. The separate changes would come out looking like one change.

The second tree comparison algorithm is Tai's. It allows insertions and deletions anywhere in the tree. For grouping differences this would not work well. In the *if* to *while* example, Tai's algorithm would report three changes: deleting *if* and inserting *while*, deleting *then* and inserting *do*, and deleting and inserting the nonterminals that indicate an *if* and a *while*. These would all be reported as separate changes, so Tai's algorithm does not help to group related changes. It would report the condition and statement as unchanged, which is an advantage over Selkow's algorithm.

Tai's algorithm should perform better at separating differences. Since it can match nodes at any level, it would not be affected in the way Selkow's algorithm is by changes that affect the level of unchanged nodes. If two changes are contiguous but unrelated, the unchanged sections should match, which will put changes to the separate structures into separate differences. If the changes are to some unit which is meaningless to the user, more needs to be done to translate the changes into a change the user would understand.

The third tree comparison algorithm that could be used is Tichy's. This algorithm assumes that all nodes with the same label have the same number of children. It then puts the trees into a linear representation, such as the preorder traversal, and uses a string comparison algorithm on the linear representation. If this algorithm were appropriate for parse trees, its results would be similar to those from Tai's algorithm. Nodes would match no matter what the level. Changes would not be grouped together. They would be separated, but not necessarily into units that that user would understand.

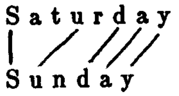
A question is whether the assumption that all nodes with the same label have the same number of children is appropriate for parse trees. The most useful labels for parse tree nodes are the terminals and nonterminals that they represent. Nodes that represent the same nonterminal can have different numbers of children because a nonterminal can be on the left hand side of many production rules, which could have right hand sides of various lengths. A possible solution could be to use the rule number as the label of the node rather than the nonterminal, if this information is available in addition to or instead of the nonterminal. It is likely that the user would be interested in changes in nonterminals, not rule numbers. This problem might be alleviated by using rule numbers for the initial comparison then doing further matching on nonterminals. Using rule numbers would not help at all if the parse trees were from a regular right part grammar.

A solution similar to using rule numbers would be to make the unit of comparison a nonterminal and number of children, rather than just the nonterminal or rule number. This would also need further comparisons. Unlike using rule numbers, it would be applicable to regular right part grammars.

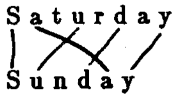
Another possible solution is to include a special mark element after all rightmost children, then treat these elements just like the string comparison elements that are terminals and nonterminals. This could lead to some strange matches. As long as these happened rarely, the strange matches could be tolerable.

A possible way to avoid strange matches would be to restrict matches that cross mark elements and not to treat the mark elements as normal string elements. The algorithm would have to change to handle this. A problem could be that this requirement would be as restrictive as Selkow's algorithm or more so. It could require nodes to be at the same level and be in the same numerical position in the list of children to match.

If subtrees are eliminated, then the tree comparison algorithms would have to account for this. At this point the concept of a trace is useful. For string comparisons, a trace matches the elements that are unchanged. The trace would be a set of ordered pairs giving the positions in the strings that are matched. For strings *Saturday* and *Sunday*, a trace would be



In a trace, none of the lines can cross, that is, if (i, j) and (m, n) are in the trace, then $i < m$ iff $j < n$. Thus



would not be a trace. Traces for trees can also be defined. For trees, if the nodes are listed in preorder, no lines would cross.

The tree comparison algorithms restrict themselves to differences that produce a trace. If this is to be true when parts of the trees are matched before the comparison is done, then the tree comparison algorithms must be changed. This might be just another dynamic programming problem. For Tichy's algorithm, which uses a string comparison, this is no problem. If only one pair of subtrees is matched, the strings would just be divided into two pairs of strings to compare. If more than one pair is matched, the string is divided into more pairs that are compared without regard to other pairs. This is not as simple a problem for the other two tree comparison algorithms.

Another possibility is to simply eliminate the matched subtrees and not worry about lines in the trace that cross. This might help locate parts of the tree that moved, but only in a limited way. It would also make reporting of the changes to the user more complicated. This solution for handling matched subtrees would be interesting to compare with other methods.

Matching subtrees could help the tree comparisons be more informative. It would help Selkow's algorithm with grouping differences, as mentioned before. It might help Tai's and Tichy's algorithms as well. By removing subtrees that might separate related changes, these changes may become one difference to these algorithms. This is not necessarily true, however. Depending upon how trees are selected, say the root must be an interesting nonterminal, or only one pair of subtrees is matched so unchanged parts outside this subtree are left to the tree comparison algorithm, unchanged parts of the tree could be left separating the related changes. Then Tai's and Tichy's algorithms would still report the related changes separately.

Eliminating subtrees might hinder separating unrelated differences in much the same way that it would help group related differences. If all the matching nodes between two unrelated changes are eliminated, the changes could become one difference again.

Besides the tree comparison algorithms and modifications to those, several other methods for comparing trees are possible. These include developing a tree comparison algorithm for parse trees, rather than trees in general, using just the string comparison on the terminals in conjunction with the location of interesting nonterminals, using a string comparison on strings of interesting nonterminals, and not doing anything except possibly eliminating matching subtrees.

A tree comparison algorithm for parse trees might be better than algorithms that are for any tree. Selkow's and Tichy's algorithms have some problems because their assumptions are not applicable to parse trees. Tai's algorithm uses much time and space. An algorithm designed for changes to parse trees might perform better. However, it might have the same problems as Tai's algorithm in grouping and separating differences. Also, such an algorithm might be too dependent on the grammar to be useful in general.

Basing the differences on the string differences of the tokens and the tree structure, without any type of tree comparison, might produce reasonable results. The tree comparison algorithms do not seem to produce the results needed to group and separate differences. Once the tree comparison is done, further processing is needed to produce something which will group and separate differences and be meaningful to the user. A reasonable question is whether the added knowledge of what tree structure changed provides information that would help with this and that the string difference of the tokens would not give. If it does give some useful information, unless the resulting displays for the user are significantly better than those produced from the string difference and tree structure alone, the tree comparisons may not be worth the added expense.

Another possibility which uses more information from the tree than comparing the strings is to get strings of interesting nonterminals based on the string difference of the terminals and use a string comparison algorithm on that. A possible method for getting a list of interesting nonterminals for a changed section of terminals is to get the root of the smallest subtree that contains the first changed terminal and whose root is an interesting nonterminal. Eliminate the terminals in that subtree and find the interesting nonterminal for the reduced list of terminals. Continue until no more changed terminals for this section remain. Get the string of interesting nonterminals for both the new and old trees. Apply a string comparison algorithm on these. Basing the cost of changing one interesting nonterminal into another on the terminals in the trees rooted at the interesting nonterminals or the trees' structure may be worthwhile. This is similar to the algorithm for finding differences between screen displays which Gosling presents [Gosling, 1981].

One other possibility is to find the trees to compare, eliminate whatever subtrees should be eliminated, and call that the difference. This would work well for grouping differences, but not at all for separating them. It would be interesting to compare this to other methods.

In addition to the method to use to compare the parse trees and generate the differences, a consideration is what form of the trees to compare. Some possibilities are to treat lists of items in the grammar differently from other tree structures, use only interesting nonterminals for the comparison, or use abstract syntax trees.

Many languages have lists of elements, such as lists of statements or lists of declarations. They will usually be represented in the grammar by rules like

```
item ::= ...
list-of-items ::= ε ¦ item list-of-items
```

or

list–of–items ::= item ¦ item list–of–items

or

list–of–items ::= ϵ ¦ list–of–items item

or

list–of–items ::= item ¦ list–of–items item

depending upon whether the list can be empty and whether the production rule is left– or right–recursive. The trees produced by these rules will be narrow and tall. If an item changes, the trees chosen as containing the change for the comparison will contain all the items before or after the one that was changed, inserted, or deleted, depending upon whether the grammar is left– or right–recursive. This list could be quite long. Depending upon how well subtree elimination worked, the trees to be compared could be quite large. If several items in the list were changed, inserted, or deleted, the entire list or a large part of it would need to be compared. The question is whether it is better to recognize that the elements that the comparison must deal with is a list of items and then compare them as lists or to ignore the special nature of the trees and compare them as trees. Using a comparison algorithm for strings would allow something like the method Gosling suggests for comparing display screens, that is, using a comparison to find the cost of converting an element of one list to an element of the other. Treating tree structures that represent lists as lists rather than trees could produce a better comparison or produce a comparison more efficiently.

Another possibility for comparing the parse trees is to involve only the interesting nonterminals in the comparison. Since the user will want to see differences only in terms of the interesting nonterminals, comparing the tree in these terms seems reasonable. For the comparison, all the nodes in the trees except the interesting nonterminals and the terminals could be ignored. This new form of the parse tree could have all the terminals and interesting nonterminals treated as the children of the closest ancestor that is an interesting nonterminal. Looking only at interesting nonterminals could have three advantages. Because the uninteresting nonterminals are not involved in the comparison, the tree comparisons would be handling fewer nodes and would be faster. Since only interesting nonterminals are considered, further processing after the comparison to get the differences into terms meaningful to the user should not be necessary. Finally, the tree comparison algorithms might have fewer problems grouping and separating differences.

A final form of the tree to consider using for the comparisons is abstract syntax trees. With abstract syntax trees, presumably the set of interesting nonterminals would not be necessary. The tree should not contain nonterminals that exist to simplify parsing. It could of course be possible that the user would still not be interested in all the nonterminals used in the abstract syntax tree. A possible example is an item that would be a small amount of text. If a subscript on an array reference changed, the user might prefer having the array reference reported as changed, rather than just the subscript. A problem with abstract syntax trees is that changes in the tree structure would not necessarily have a corresponding change in the leaves of the tree. An example is that of changing an *if* to a *while*. Both would have children of expression and statement. These would not change. It is of course possible to include parts of the syntax in the leaves of the tree, but it would seem that a true abstract syntax tree would not contain these. If it did not, it would not be possible to locate the areas where the tree structure had changed by comparing the leaves. The only way to find the differences in the trees would be to compare the trees in their entirety. This would take quite a bit of time. Involving only the terminals and interesting nonterminals in the comparisons probably has the advantages of comparing abstract syntax trees without the disadvantages.

The fourth and final step in dividing the differences into logical units is to display the results to the user. This has not received much thought yet. A few points have been mentioned in the discussion of the other steps. Some of the issues to decide are how to display changes which are physically distant but logically related, how to display the unchanged sections between related changes, and whether to treat sections that are matched in step two specially. No doubt more issues will arise as the other steps develop and what type of information can be obtained from the comparison step is seen.

Dividing differences into logical sections can be done in many ways. Many combinations of methods for various steps are possible. Finding how the methods behave and which produce good results will be interesting.

The second feature I want to develop is summarizing differences. The summaries would also be based on the parse tree structure. Algorithms developed for dividing differences into logical sections might serve as a good base for summarizing differences into logical sections. I want to investigate these possibilities. If that proves fruitless, I will develop a method for summarizing differences independent from the methods for dividing differences.

For both dividing and summarizing differences, a subset of the grammar's nonterminals must be chosen. If all the nonterminals were used for dividing, differences would be divided too finely, which would be more confusing than helpful. For summarizing, seeing summaries for each nonterminal would be too time consuming. It would also not be worthwhile for the person viewing the differences since the nonterminals would include nonterminals whose purpose was to simplify parsing. To function well, the subsets will probably have to satisfy certain conditions.

For dividing differences, it might be that all that is necessary is a set of nonterminals which can generate all the terminals. The requirements for summarizing will be more complex. The summaries should be on different levels. This probably means that each level will have its own set of nonterminals. These sets will have to meet certain requirements, each set individually and in relation to the sets of the levels above and below.

Some characteristics of these sets of nonterminals seem desirable. The nonterminals in any one level should be able to generate all the terminals in the language. In this way, any change in the higher level tree can be reported by reporting on changes in the lower level trees. Getting nonterminals that satisfy this restriction will not always be possible. For example, some grammars have terminals which serve as punctuation to separate lists as children of possible higher level nonterminals. These have no intermediate level which could generate them. As another example, a nonterminal needed for a set to generate the terminals may contain nothing of interest or only terminals which rarely change. For example, one grammar includes a nonterminal begin_symbol which goes to the token begin. Certainly, some difficulties can be overcome by manipulating the grammar; however, this cannot fix all difficulties and should not be a requirement to use the difference system. Obtaining sets which individually can generate all the terminals will not always be possible, but sets which come close should be used.

Since finding sets which can all generate all the terminals is not always possible, some way to deal with changes in the larger tree not in any of the interesting subtrees must be developed. If the difference system has told the user that a statement was inserted, reporting the insertion of a semicolon is not very informative. If the only change in a larger subtree is in part of the terminals not generated by the next level, some change must be reported. Otherwise the user could be misled and also come to distrust a system which reports a difference at one level but reports no difference when more detail is requested.

Some method for dealing with text which is not in the parse tree must be devised. An example of such text is comments in programs. Many decisions must be made: how the system decides when a comment is in a subtree (is it in a subtree only if terminals on both sides of it are in the subtree?), if the only thing that changed in a subtree was a comment, whether the subtree should be reported as changed, and whether a different message should be used to report that the tree did not change, but something attached to it did. Whatever strategy is chosen, it should be general enough that the reporting makes sense for any language which might be edited with a syntax-directed editor and for which a parse tree can be built. The strategy must also make sense for text besides comments which might be attached to a parse tree without being part of it.

Another factor to consider in choosing nonterminals for dividing and summarizing differences is the relationship of the set of nonterminals for dividing differences and the sets for summarizing differences to each other. A relationship may not be necessary, but it might make more sense to the user if some relationship existed. In looking at conditions the sets should satisfy, I will also see what relationships might profitably exist between them.

Another problem to be solved for summarizing differences is devising a scheme to store the many methods for finding names in the trees. Some way to name the segment that has changed must exist so that the user will have some idea in which part of the program the change is. This was explained in more detail previously. I have not looked at this extensively, other than to identify some of the kinds of things that would be reasonable names and which should fit into such a scheme.

The third problem is to find a way to decide what level of differences to display in a summary when the user does not specify a level. The problem seems to be to determine what relevant information is available and how it can be used. Some information that might be useful is the tree structure, the number of levels above and below a level, and the amount of text that a display at a certain level would generate. Another question is whether the decision mechanism could be parameterized so that the user could have some control.

These are the five problems that I want to solve: dividing differences, summarizing differences, choosing nonterminals for dividing and summarizing, storing schemes to find names for summarized sections, and deciding what level of detail to display. I want to design methods that will solve these problems.

6. References

UNIX User's Manual: Reference Guide. Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California, 1984.

VAX VMS DCL Dictionary. Digital Equipment Corp., Maynard, Massachusetts, 1985.

A. V. Aho, D. S. Hirschberg and J. D. Ullman. *Bounds on the Complexity of the Longest Common Subsequence Problem.* **Journal of the Association for Computing Machinery**, Vol. 23, No. 1, pp. 1–12, January 1976.

Cyril N. Alberga. *String Similarity and Misspellings.* **Communications of the Association for Computing Machinery**, Vol. 10, pp. 302–313, May 1967.

S. O. Anderson, R. C. Backhouse, E. H. Bugge and C. P. Stirling. *An Assessment of Locally Least-Cost Error Recovery.* **The Computer Journal**, Vol. 26, No. 1, pp. 15–24, February 1983.

H. A. Bauer and R. H. Birchall. *Managing Large Scale Software Development with an Automated Change Control System.* **COMPSAC78, Proceedings of the IEEE Computer Society's Second International Computer Software and Applications Conference**, pp. 13–17, November 13–16, 1978.

Charles R. Blair. *A Program for Correcting Spelling Errors.* **Information and Control**, Vol. 3, pp. 60–67, March 1960.

T. A. Cargill. *Management of the Source Text of a Portable Operating System.* **COMPSAC80, Proceedings of the IEEE Computer Society's Fourth International Computer Software and Applications Conference**, pp. 764–768, October 27–31, 1980.

Fred J. Damerau. *A Technique for Computer Detection and Correction of Spelling Errors.* **Communications of the Association for Computing Machinery**, Vol. 7, pp. 171–176, March

1964.

Leon Davidson. *Retrieval of Misspelled Names in an Airlines Passenger Record System.* Communications of the Association for Computing Machinery, Vol. 5, pp. 169–171, March 1962.

Ramon D. Faulk. *An Inductive Approach to Language Translation.* Communications of the Association for Computing Machinery, Vol. 7, pp. 647–653, November 1964.

C. Ghezzi and D. Mandrioli. *Augmenting Parsers to Support Incrementality.* Journal of the Association for Computing Machinery, Vol. 27, No. 3, pp. 564–579, July 1980.

James Gosling. *A Redisplay Algorithm.* SIGPLAN Notices, Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, Vol. 16, No. 6, pp. 123–129, June 1981.

Susan L. Graham and Steven P. Rhodes. *Practical Syntactic Error Recovery.* Communications of the Association for Computing Machinery, Vol. 18, No. 11, pp. 639–650, November 1975.

Stephen J. Hague and Brian Ford. *Portability--Prediction and Correction.* Software---Practice and Experience, Vol. 6, No. 1, pp. 61–69, January–March 1976.

Patrick A. V. Hall and Geoff R. Dowling. *Approximate String Matching.* Computing Surveys, Vol. 12, No. 4, pp. 381–402, December 1980.

Paul Heckel. *A Technique for Isolating Differences Between Files.* Communications of the Association for Computing Machinery, Vol. 21, No. 4, pp. 264–268, April 1978.

D. S. Hirschberg. *A Linear Space Algorithm for Computing Maximal Common Subsequences.* Communications of the Association for Computing Machinery, Vol. 18, No. 6, pp. 341–343, June 1975.

Daniel S. Hirschberg. *Algorithms for the Longest Common Subsequence Problem.* Journal of the Association for Computing Machinery, Vol. 24, No. 4, pp. 664–675, October 1977.

D. S. Hirschberg. *An Information-Theoretic Lower Bound for the Longest Common Subsequence Problem.* Information Processing Letters, Vol. 7, No. 1, pp. 40–41, January 1978.

W. J. Hsu and M. W. Du. *New Algorithms for the LCS Problem.* Journal of Computer and System Sciences, Vol. 29, pp. 133–152, 1984.

James W. Hunt and Thomas G. Szymanski. *A Fast Algorithm for Computing Longest Common Subsequences.* Communications of the Association for Computing Machinery, Vol. 20, No. 5, pp. 350–353, May 1977.

Gail E. Kaiser and A. Nico Habermann. *An Environment for System Version Control.* COMPCON83, pp. 415–420, 1983.

Vincent Kruskal. *Managing Multi-Version Programs with an Editor.* IBM Journal of Research and Development, Vol. 28, No. 1, pp. 74–81, January 1984.

Roy Lowrance and Robert A. Wagner. *An Extension of the String-to-String Correction Problem.* Journal of the Association for Computing Machinery, Vol. 22, No. 2, pp. 177–183, April 1975.

William J. Masek and Michael S. Paterson. *A Faster Algorithm Computing String Edit Distances.* **Journal of Computer and System Sciences**, Vol. 20, pp. 18–31, 1980.

M. Dennis Mickunas and John A. Modry. *Automatic Error Recovery for LR Parsers.* **Communications of the Association for Computing Machinery**, Vol. 21, No. 6, pp. 459–465, June 1978.

Howard L. Morgan. *Spelling Correction in Systems Programs.* **Communications of the Association for Computing Machinery**, Vol. 13, pp. 90–94, February 1970.

Narao Nakatsu, Yahiko Kambayashi and Shuzo Yajima. *A Longest Common Subsequence Algorithm Suitable for Similar Text Strings.* **Acta Informatica**, Vol. 18, pp. 171–199, 1982.

Jan T. Pedersen and John K. Buckle. *Kongsberg's Road to an Industrial Software Methodology.* **IEEE Transactions on Software Engineering**, Vol. SE–4, No. 4, pp. 263–269, July 1978.

J. M. Prager. *The Project Automated Librarian.* **IBM Systems Journal**, Vol. 22, No. 3, pp. 214–228, 1983.

Allan Ramsay. *A Distributed Programming Assistant.* **Software—Practice and Experience**, Vol. 13, No. 11, pp. 983–992, November 1983.

Marc J. Rochkind. *The Source Code Control System.* **IEEE Transactions on Software Engineering**, Vol. SE–1, No. 4, pp. 364–370, December 1975.

David Sankoff. *Matching Sequences under Deletion/Insertion Constraints.* **Proceedings of the National Academy of Sciences, USA**, Vol. 69, No. 1, pp. 4–6, January 1972.

David Sankoff and Joseph B. Kruskal (eds.). **Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.** Addison–Wesley Publishing Company, Inc., Reading, Massachusetts, 1983.

Stanley M. Selkow. *The Tree–to–Tree Editing Problem.* **Information Processing Letters**, Vol. 6, No. 6, pp. 184–186, December 1977.

Peter H. Sellers. *An Algorithm for the Distance Between Two Finite Sequences.* **Journal of Combinatorial Theory (A)**, Vol. 16, pp. 253–258, 1974.

J. J. Stanaway Jr., J. J. Victor and R. J. Welsch. *Traffic Service Position System No 1: Software Development Tools.* **The Bell System Technical Journal**, Vol. 58, No. 6, pp. 1307–1333, July-August 1979.

Kuo–Chung Tai. *Syntactic Error Correction in Programming Languages.* **IEEE Transactions on Software Engineering**, Vol. SE–4, No. 5, pp. 414–425, September 1978.

Kuo–Chung Tai. *The Tree–to–Tree Correction Problem.* **Journal of the Associaton for Computing Machinery**, Vol. 26, No. 3, pp. 422–433, July 1979.

T. J. Thompson. *Designer's Workbench: Providing a Production Environment.* **The Bell System Technical Journal**, Vol. 59, No. 9, pp. 1811–1825, November 1980.

Walter F. Tichy. *Design, Implementation, and Evaluation of a Revision Control System.* **Proceedings of the Sixth International Conference on Software Engineering**, pp. 58–67, September 13-16, 1982.

Walter F. Tichy. *The String-to-String Correction Problem with Block Moves.* **ACM Transactions on Computer Systems**, Vol. 2, No. 4, pp. 309-321, November 1984.

Walter F. Tichy. "Personal communication", 1985.

Robert A. Wagner. *On the Complexity of the Extended String-to-String Correction Problem.* **Proceedings of the Seventh Annual ACM Symposium on the Theory of Computing**, pp. 218-223, 1975.

Robert A. Wagner and Michael J. Fischer. *The String-to-String Correction Problem.* **Journal of the Association for Computing Machinery**, Vol. 21, No. 1, pp. 168-173, January 1974.

Reinhard Wilhelm. *A Modified Tree-to-Tree Correction Problem.* **Information Processing Letters**, Vol. 12, No. 3, pp. 127-132, June 13, 1981.

C. K. Wong and Ashok K. Chandra. *Bounds for the String Editing Problem.* **Journal of the Association for Computing Machinery**, Vol. 23, No. 1, pp. 13-16, January 1976.

GNU Emacs Uniform User Interface for the SAGA Software Development Environment

Dan LaLiberte

W. Smith

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, Illinois

GNU Emacs uniform user interface for the SAGA Software Development Environment

Dan LaLiberte
W. Smith

University of Illinois
Department of Computer Science
1304 W. Springfield Ave.
Urbana, IL 61801-2987.
217-333-0215

Abstract:

The GNU Emacs editor has been incorporated into the SAGA Software Development Environment as a uniform user interface. The extensibility and interprocess communication features of GNU Emacs are used to integrate several separate SAGA utilities including an incremental parser, an incremental semantics processor, and a configuration management system.

1. Introduction.

The amount of time users spend in an editor is a large percentage of their on-line time. While not discussing the issue of interactive vs. batch oriented development methodolgies, this paper is concerned with maximizing the effectiveness of the human-computer interface in a software development environment.

Our motivation was the need to integrate several different utilities with a common user interface. A software development environment consists of a number of utilities more or less tied together by some user interface. In the standard UNIX system, for example, the user interface is commonly the shell. The different utilities are activated by calling them explicitly from the shell, or implicitly from a script.

1.1. SAGA Software Development Environment: A software development environment includes editors, compilers, linkers, loaders, debuggers. In development are verification systems and configuration management systems.

In the remainder of this paper, we describe the approach adopted to provide an improved human interface to EPOS using the GNU Emacs editor. The editor provides many typical features found in full-screen editors, is interfaced to raster display devices as well as terminals, is programmable, and can be used with several different windowing system packages including the MIT X-Windows system. Finally, the GNU Emacs editor provides a general interface which may be used with many other SAGA tools. Figure 1 illustrates the relationship between GNU Emacs and several SAGA tools. Figure 2 shows several features of the GNU Emacs environment which will be discussed in the following sections.

2. GNU Emacs

2.1. Standard Character-level Editing: GNU Emacs provides standard character-level editing with a full screen, multi-window, tiled display [1]. All the typical character manipulation commands are available as well as cursor movement, screen paging, global search and replace, etc. - things that a programmer would expect in an editor. Character-level editing is what programmers are used to, but the reason for using GNU Emacs stems from its extensibility more than its familiarity.

Like other editors in the Emacs family, GNU Emacs allows the user to extend the initial command set by using a LISP-like language to write functions which may then be bound to key sequences. GNU Emacs LISP is a fairly complete LISP extended to include primatives for editing in a multi-buffer, multi-window context.

2.2. Language Specific Modes

Each buffer may have several modes associated with it which correspond to buffer-specific commands, variables, etc., appropriate for editing the text in the buffer. Several language modes are typically provided in the Emacs library of LISP programs. A language mode may be automatically associated with a buffer based on the name of the file being edited.

2.3. Holophrasting, Tags, etc. Several language-independent functions useful to program development are provided with GNU Emacs. A global holophrasting feature allows the user to select the indent level beyond which text is not displayed for a specific buffer. A general Tags facility allows the user to maintain a database of tags which are associations between names and references to locations within several text files.

2.4. Command Completion Templates: User defineable macros and abbreviations are supported by GNU Emacs. A general completion function allows the user to build customized tools for completion of initial character sequences. We have used this capability to provide a language specific template system. The user enters the initial characters of a symbol followed by a completion command (key press). If the initial characters match one of the symbols in the completion list, they are replaced by the full symbol name or by an associated template.

2.5. Help Facilities: GNU Emacs provides extensive on-line documentation of all the editing commands. A user defined command may make use of the same documentation facility by including a documentation string in the command definition. Nevertheless, considering the large number of commands available to the user, it is sometimes difficult to quickly find the appropriate command. We have written a hierarchical menu interface to most of the Emacs commands in the style of Lotus 1 2 3. That is, a prefix command opens up a single line help menu; several lines of menu items are possible in the case of large menus. The first letter of each menu item is a key command which opens up a lower-level menu, etc., down to a real command. When a real command is found, the documentation string for the command is available; the associated key sequence for the command, if any, may be reported; or the command may be executed immediatly.

2.6. Incremental Parser The first subprocess which has been installed under the GNU Emacs front end is an incremental parser. The SAGA research group had previously created an incremental parser with its own screen-oriented editor called EPOS. The user interface for EPOS is difficult to use and the large program was difficult to maintain. We decided that the extensible GNU Emacs editor could be a powerful front end for the incremental parser as well as for other SAGA projects.

With GNU Emacs as the front end, the user is allowed to modify any text in the character representation of a program. As changes are made, the corresponding terminal tokens in the parse tree representation of the program must be updated and the tokens reparsed. The reparsing algorithm, described in [2], minimizes the extent of the reparse by maintaining the parsing stack state at the time each token is parsed. In the worst case, the whole text stream must be reparsed, but usually only a small neighborhood around the change requires a reparse.

A modification of GNU Emacs (as distinct from a LISP extension) was required to support the incremental parser. The modification made use of the Emacs Undo capability which allows the user to undo previous changes as far back as it can remember. As changes are made to the text buffer, they are passed to a LISP function with identification of the kind of change. The LISP function collects contiguous changes until a non-contiguous change is made. At that point, it sends the contiguous change to the incremental parser. A reparse is performed automatically with every new contiguous change or an explicit reparse may be requested by the user.

An example of one contiguous change is given in Figure 3. A contiguous change consists of a deletion and an insertion at the same point. As each new change is made, it is either incorporated into the contiguous change if it overlaps or abuts the current contiguous change; otherwise the change is the beginning of a new contiguous change. Three kinds of changes are possible: insertion, deletion, and replacement. Both the beginning and end points of a change may each fall in one of three regions relative to the contiguous change: before, within, and after.

3. Adapting the Parser to the New Interface

To use the EPOS incremental parser with GNU Emacs as a front end, a new simplified command language was developed that allows a front end to give commands such as move the cursor, delete text, or insert text. Theoretically, this command language could be used by a human, and in fact it was so used for testing purposes, but for any significant program this would be impractical. However, this modularity means that the parser could be used with another front end editor without modification.

To be useful with a real text editor, the parser must be able to handle any text a user may enter. The original EPOS editor only allowed spaces before tokens, and consequently trailing blanks on a text line where not permitted. In addition, tabs could not be used at all. As the parser was adapted to the GNU Emacs front end, this unacceptable limitation was removed by changing the internal representation of the tokens in the parse tree. Another limitation of the old EPOS was the restriction of comments to a single line only. Now, each line of a multi-line comment is a separate token.

In the process of extracting the parser and making modifications to it, a number of previously unknown bugs were discovered and fixed. The changes made were made with the intention of supporting language independence. The only language specific parts of the parser which remain are in the lexical analyzer.

3.1. Multiple Syntax Errors: One of the advantages of the SAGA incremental parser is that any number of syntax errors may be present in the parse tree concurrently. This is accomplished by maintaining the erroneous, unparsable tokens under a "marked" non-terminal. This marked text will be reparsed if it is affected by a future change.

Often while editing a program, the programmer will find it most efficient to leave the text in a syntactically erroneous state. An example is illustrated in Figure 4. To enclose several statements in a Repeat loop, the initial "repeat" must be inserted leaving a syntax error later in the program usually at the point where an "until" is expected.

3.2. Text vs. Template Editing: An alternative to text editing with an incremental parser is template editing. A template editor may restrict the kinds of modifications of a program text to syntactically correct transformations, or it may reparse the whole text, or reparse at the expression level for convenience. For the above example, the task of enclosing several statements in a Repeat loop involves first cutting all the statements, second inserting the "repeat ... until" template, and finally pasting the statements into the Repeat loop.

A text editor with an incremental parser provides the most flexibility by allowing arbitrary text modifications while supporting templates if desired. We have implimented a simple template system keyed on an initial substring of the template text. The user enters the first few unique letters of a template followed by a "completion" key. If the letters match a template, it is expanded in place of the letters. If the letters do not match a template, an error message is given, but if the letters match more than one template, a help list of the possible matches may be displayed for the user in a separate window.

3.3. Parse Tree Commands: Since a parse tree representation of a user's program is being maintained, the user may wish to make use of it for more than error checking. Typical commands which must interact with the parse tree include token and subtree selection, forward and reverse motion by token or subtree, and subtree transformations. Such user-level commands are "translated" by a LISP program into messages to the parser. The parser responds with messages which may indicate the appropriate relative character motion, region selection or a replacement string.

We have developed a package of transformation routines to speed the conversion of while loops to repeat loops, case statements to nested if statements, etc. Logical consistency is maintained across the transformations by negating and reducing logical conditions or duplicating statements, as required. The transformation routines run as an additional subprocess and are given access to the parse tree. The output of the transformation routine may be simply displayed in an alternate window or may be inserted as a replacement string.

4. Incremental Semantics Processor

An important component of the SAGA environment is semantics processing. An incremental semantics evaluator is being developed which will run as another subprocess under the GNU Emacs front end. Changes to the parse tree and commands which interact with semantic information will be communicated to the semantics evaluator which maintains its own semantic-level representation of the program. The semantics evaluator may also return commands or text to the editor.

As an example, the transformation of a while loop to a repeat loop described earlier is more appropriately handled by a semantics-level routine. Specification of the type of transformation and the subtree to be transformed is first sent to the semantics routine; the transformation is applied; the new text representation is returned to the editor to replace the original text; and the replacement action is sent to the incremental parser for reparsing.

The semantics component of the SAGA environment will play in important role as an attribute evaluator. In addition to incremental compilation, an attribute evaluation system may be used for program verification, incremental refinement, and project management. But for all of these, a unified user interface is required as well, and the extensible GNU Emacs is suitable.

5. Conclusion

We have explored the practicality of using an extensible text editor as the front end for a number of aids for program development. GNU Emacs has proven to be worthy of this task in providing the generality of a powerful text editor and the flexibility required for communication with independently running subprocesses.

6. References

1. Stallman, Richard, "GNU Emacs Manual," Third Edition, Emacs Version 17, December 1985. 2. Kirslis, Peter, "The SAGA Editor: A Language-Oriented Editor Based on Incremental LR(1) Parser," Doctoral Thesis, December 1985, University of Illinois, Urbana-Champaign.

GNU Emacs SAGA Environment

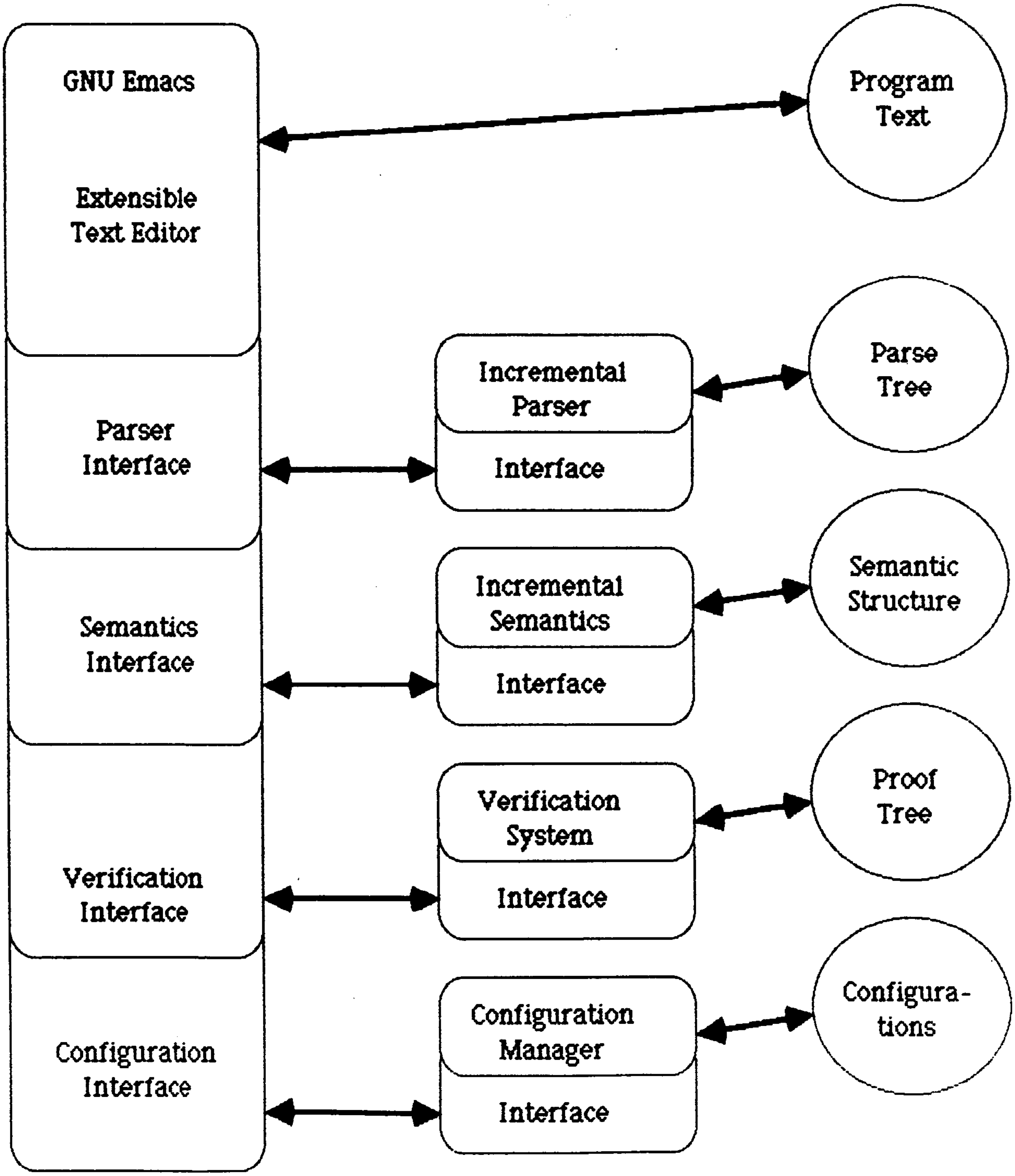
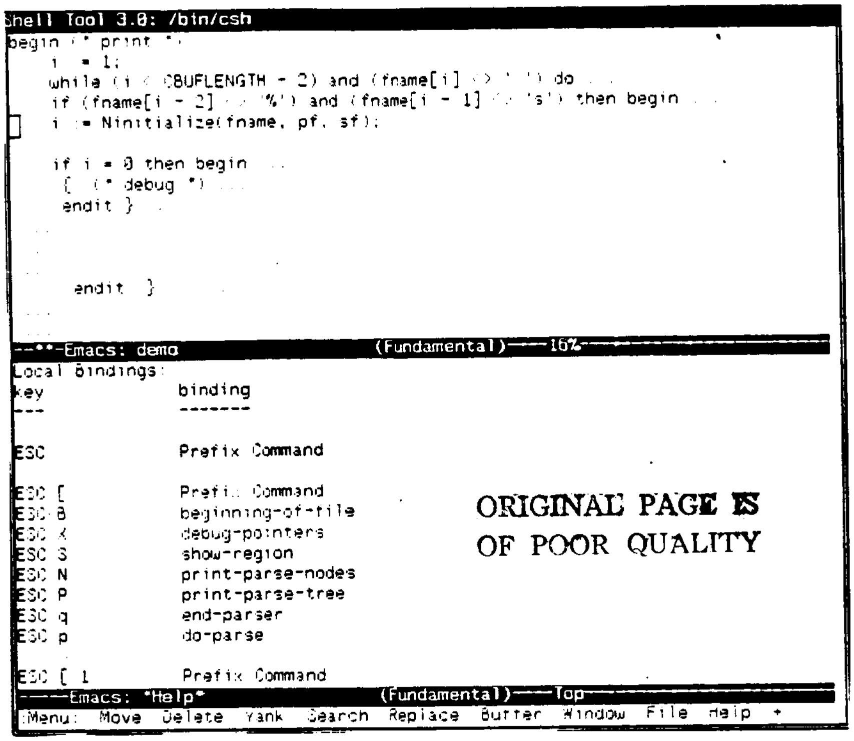
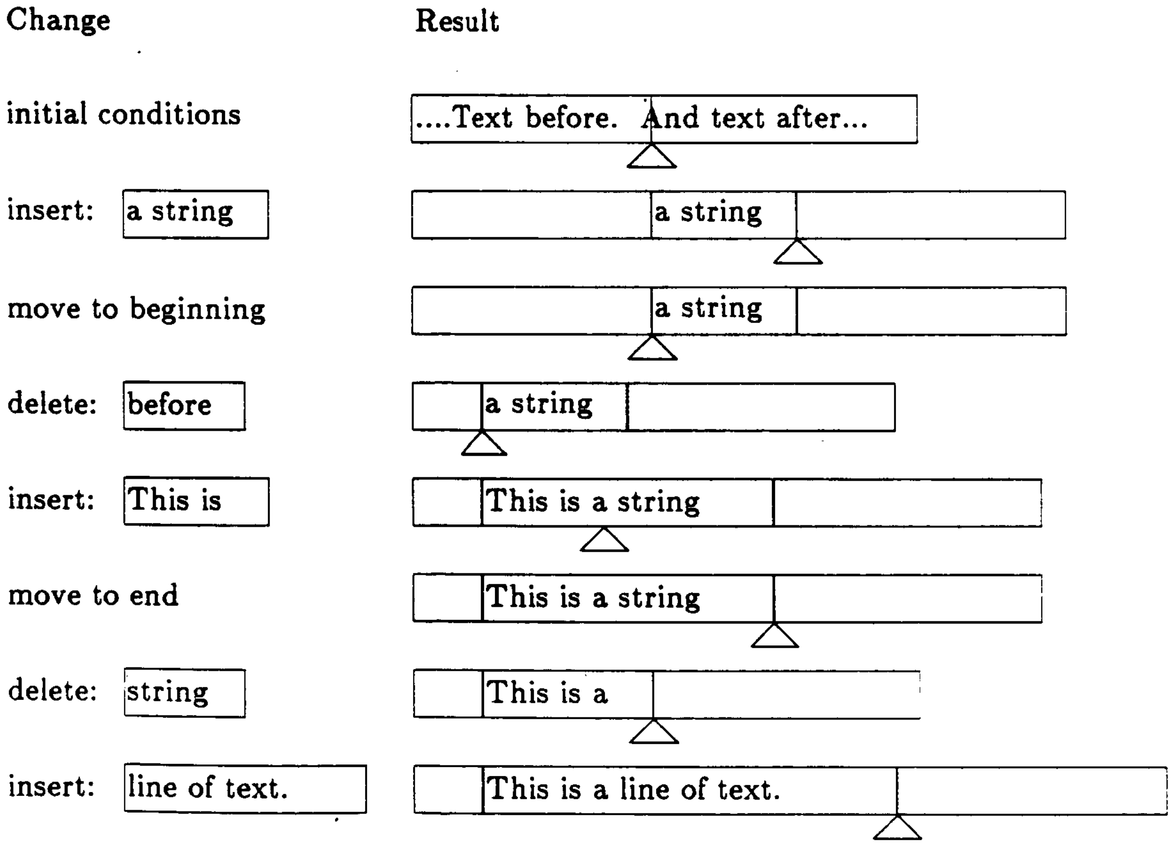


Figure 1



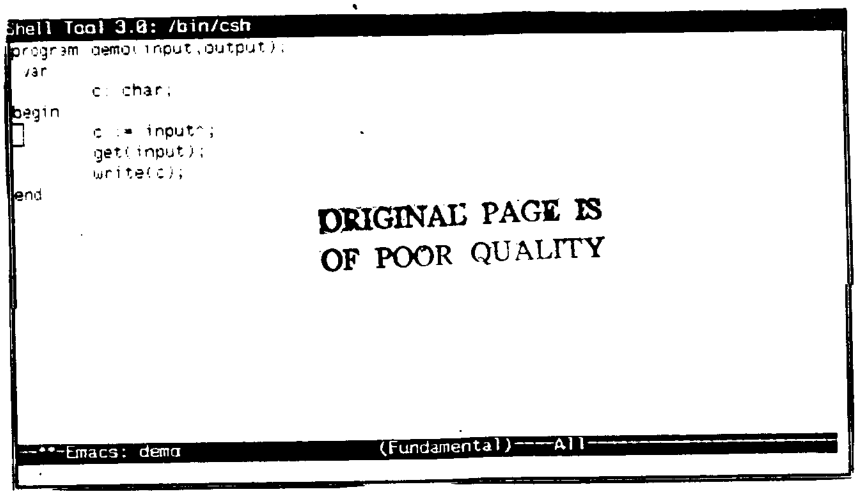
```
Shell Tool 3.0: /bin/csh
begin (* print *)
    i  = 1;
    while (i < CBUFLENGTH - 2) and (fname[i] <> ' ') do ...
    if (fname[i - 2] <> '%') and (fname[i - 1] <> 's') then begin ...
    i := Ninitialize(fname, pf, sf);

    if i = 0 then begin ...
     { (* debug *) ...
     endit } .
 ...

 ...
      endit  }
 ...
 ...
---**-Emacs: demo                    (Fundamental)----16%----------------
Local bindings:
key             binding
---             -------

ESC             Prefix Command

ESC [           Prefix Command
ESC B           beginning-of-file
ESC X           debug-pointers
ESC S           show-region
ESC N           print-parse-nodes
ESC P           print-parse-tree
ESC q           end-parser
ESC p           do-parse

ESC [ 1         Prefix Command
-----Emacs: *Help*                   (Fundamental)----Top----------------
:Menu:  Move  Delete  Yank  Search  Replace  Buffer  Window  File  Help  +
```

Figure 2. GNU Emacs Holophrasting, some Key Bindings, and a Menu Line.

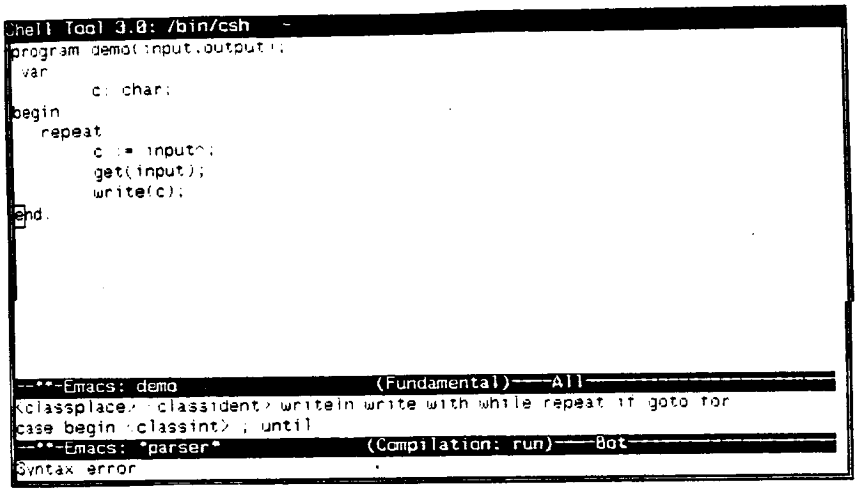


Change	Result
initial conditions	....Text before. And text after...
insert: a string	a string
move to beginning	a string
delete: before	a string
insert: This is	This is a string
move to end	This is a string
delete: string	This is a
insert: line of text.	This is a line of text.

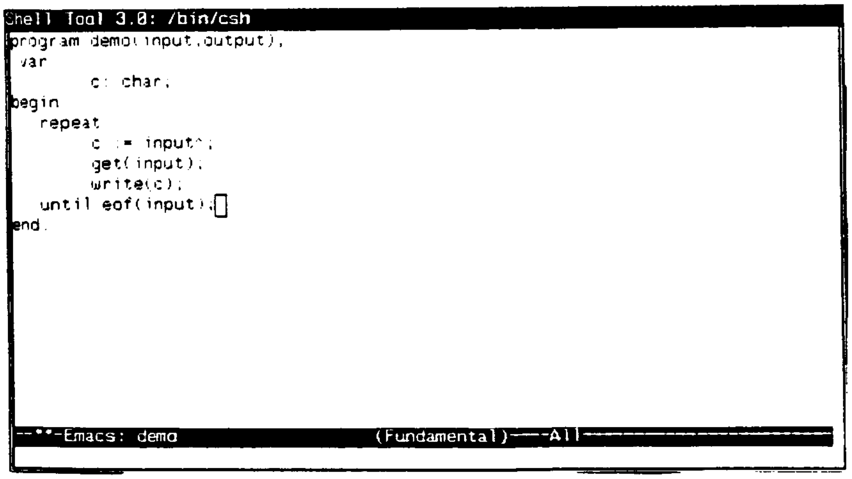
Figure 3. Example of one contiguous change



```
Shell Tool 3.0: /bin/csh
program demo(input,output);
 var
        c: char;
begin
        c := input^;
        get(input);
        write(c);
end

--**-Emacs: demo                    (Fundamental)----All-----------
```



```
Shell Tool 3.0: /bin/csh
program demo(input,output);
 var
        c: char;
begin
    repeat
        c := input^;
        get(input);
        write(c);
end.

--**-Emacs: demo                    (Fundamental)----All-----------
<classplace> <classident> writeln write with while repeat if goto for
case begin <classint> ; until
--**-Emacs: *parser*                (Compilation: run)----Bot-------
Syntax error
```



```
Shell Tool 3.0: /bin/csh
program demo(input,output);
 var
        c: char;
begin
    repeat
        c := input^;
        get(input);
        write(c);
    until eof(input);
end.

--**-Emacs: demo                    (Fundamental)----All-----------
```

Figure 4. Enclosing several statements within a Repeat loop. The second figure shows the "follow set" of legal symbols before the cursor.

SAGA Project Mid-Year Report 1986 Appendix M

CLEMMA: An Automated Configuration Librarian

Hal S. Render

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, Illinois

Clemma: An Automated Configuration Librarian

Hal S. Render

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Abstract

Control of system configurations has long been a problem. The SAGA project is investigating several such problems in the area of software development. **Clemma**, a prototype system for managing configurations on several levels, is presented with a discussion of the details of the system's guiding principles.

1. Introduction

A growing problem in the development and maintenance of software projects is the organization, manipulation and storage of the large number of components involved. A single medium-sized software system, with 10 to 50 thousand lines of code, may be composed of several dozen separate computer files. Requirement specifications, design documents, project plans, user manuals, source code, test data—all may be stored on-line and all must be maintained throughout the lifetime of a project. This requires the ability to track, identify and control all changes made to a system's files. As the size and complexity of systems grows, the difficulty of performing these operations also grows.

Another, more recent, problem is the distribution of a system's component files. Modern software development theory promotes *modularity*, the grouping of system components into logically-related clusters [ref?]. This technique has several recognized benefits, both to the software and to the engineers involved in its production. Unfortunately, the separation of the components of a system increases the difficulty of treating a large system as a single entity, or even as a limited number of modules. In addition, most means of grouping software system components into modules are still relatively unsophisticated, and seem to have little support in many development environments. What is needed is a way of being able to refer to and manipulate the components of a system on several different levels, from that of a single file to that of a module to that of a system. Current efforts at solving this problem are widespread, though few have gained any widespread use in "the real world"[12].

Traditionally, the task of keeping records on all material produced during a software project and taking responsibility for change control is the duty of a *project librarian* [ref?]. This entity (sometimes a single person) is responsible for tracking all of the components developed, identifying the state of each, and ensuring that a particular component is releasable for use by other project members or users. This function is crucial to a project, as it is often the principal interface between management and staff for gauging and controlling the progress of a project.

With the push to automate various functions of the software development life–cycle, a means of tracking the state of a large system automatically was inevitable, and several efforts are notable. Early efforts resulted in systems called *project support libraries*, which essentially automated some of the work of a human project librarian. More recently, the entire area of identifying, tracking, and controlling changes to systems has been classed as *software configuration management.* The effort to apply CM techniques to software development has resulted in SCM, and with varying degrees of success. The principal problems arise in the youth of software engineering as field of endeavour. Not enough is known yet about constructing software systems in a reliable fashion to easily enable one to automate its management. For this reason, SCM is still an area open to experimentation.

Since the SAGA project (Software Automation, Generation, and Administration) is concerned with automating the prduction of software systems, it was inevitable that we should investigate the issue of configuration management for such systems. Our efforts to date demonstrate the need for an automated means of handling the components of large, hierarchical software systems, on several levels of abstraction. A prototype *configration librarian*, **Clemma**, is our attempt to provide a means of investigating the problem of configuration management in large software systems.

2. Background

Before Clemma is discussed in more detail, a few of the terms relevant to a discussion of software configuration management should be defined. A *configuration*, for our purposes, is a "snapshot" of the components of a system, describing their states and their interrelationships at a specific point in time. Each of the individual elements of a configuration is called a *configuration item* (CI). The effort to deal with the problems of controlling the development and evolution of configurations is called *configuration management* (CM). Specifically, Bersoff defines this as the discipline of identifying the configuration of a system at discrete points in time for the purpose of systematically controlling changes to the configuration and maintaining the integrity and traceability of the configuration throughout the system life cycle [Bersoff, 84]. Software configuration management is the application of CM techniques to projects composed principally of software.

3. Clemma

Clemma is a prototype of a configuration control system. The system is modeled on a project library concept, and as such most of the operations of the system are analogous to conventional library operations. But the requirements of a project librarian are different from those of a conventional librarian, so in some places there are operations which are wholly new to the idea of a library.

An important aspect of Clemma which should be mentioned is the fundamental configuration item in the system: a file. Many of the efforts currently underway to provide configuration control for software systems currently use such logical entities as subprograms and shared data structures as the basic controlled items. In analysing the problems we wished to address, we found that the isolation and recognition of logical entities within files vastly complicated the management issue, particularly in an environment which will be multilingual and which, hopefully, will be used to model different development methodologies. Current systems for dealing with independently produced project components seem to impose strict constraints on the developer, so that any item produced conforms to format standards which allows the system to identify the configuration items. Such systems thus pay a price in flexibility for this sophistication.

However, files are relatively easy to handle. They are easily recognizable, have discernable attributes, and can be manipulated easily in most operating systems. Rather than limit the applicability of Clemma to software developed to rigid structural guidelines, we have opted to place some of the burden on the user by allowing him/her to program as s/he wants. This does require the user to inform Clemma of some of the logical attributes of a configuration item manually, as they would be difficult to determine automatically. The task of performing this manual characterization of items is being investigated, so it is possible in the future that an even finer granularity of items will be possible. For now, having the file as the basic element of a configuration is sufficient.

3.1. System Architecture

As mentioned, Clemma is a configuration *librarian.* Configuration items are stored in *libraries*, and nearly all of the available operations are analogous to those of a conventional library. A Clemma library has four main parts: a *repository*, containing copies of all of the configuration items in the library; a *catalog*, which is a database holding all of the information on the items in the repository; a temporary storage pool for the read-only copies of checked-out items, call the *user area*; and a table of the current users of the library items, called the *usage list.* The purpose of each of these structures will be detailed as the operations provided by Clemma are described below.

3.2. Operations

As a configuration librarian, Clemma has several functions.

- **Create** a configuration library. This operation causes all of the library data structures to be created and initialized. The creator of the library also establishes *directors* for the library—individuals who have total control over the creation and deletion of library elements and all other library capabilities. A **delete** operation exists to undo all of the actions of **create**.

- **Catalog** a configuration item. This creates an entry for an item in the catalog. Information about the item is collected and stored in the database, and an empty version chain is initialized for the item in the repository. In addition, *manager* permissions are established for this item by the individual who catalogs it. If a user has manager permission for an item, s/he then has total control over that item. (Directors subsume all of the powers of a manager.) The **uncatalog** operation peforms the inverse of the **catalog** operation.

- **Install** a version of a configuration item. A copy of a cataloged item is attached to the version chain for that item in the library repository. Additional information about the particular version being installed is collected and stored in the catalog. Note that only an item manager or someone who has been granted permission by a manager may install a version of an item. Managers may create a list of allowable users to restrict access to an item. **Remove** is the operation which removes installed versions from a library.

- **Checkout** a version of a CI for **read-only** use. This gives the user performing the operation access to a read-only copy of a library item. The copy resides in the *user area*, and is shared by all individuals who have checked the item out for read-only use. If the access to the item is restricted, then a manager of the item must give an individual permission to checkout the item. When an item is checked out of the library, an entry is made in the usage list recording this fact.

- **Checkout** a version of a CI for **modification**. This gives the user a writable copy of the CI. Permission to check an item out for modification must be granted by a director or one of the item's managers. An entry is made in the usage list when the item is checked out.

- **Return** a version of a CI. This operation is used for returning a checked-out copy of an item to the library. The user's access to the item or local copy is removed and the user's name is deleted from the usage list for that item. This **does not**, however, put any revised items in the repository—the **install** operation must be used for that purpose.

- **Collect** individual configuration items into a single item. This operation is used for the creation of *collections*, which are formatted lists of configuration items. These require some explanation. When a software system is created, it is often broken up into modules for reasons well known to structured-programming enthusiasts. In a configuration, one often wants to treat not only the individual files in the configuration as CIs, but also the modules into which the system is divided. To do this in Clemma, all of the useful files of a module are first cataloged and installed as CIs. When the files are cataloged, they are each assigned a *call number*, which uniquely identifies a particular CI to the library. The collect command takes the list of CIs comprising a module, and creates a specially formatted list of their call numbers and stores this in a file. This file can then be cataloged and installed as its own (albeit special) CI. The type of a CI (file or collection) is stored as an attribute of the CI in the catalog.

- **Assign** attribute values to a configuration item. This operation is used to store attribute values for CIs in the catalog.

- **Compare** the differences between versions of a configuration item. This prints out a listing of the changes made from one version to another of a specified CI.

- **Identify** items from the library. For a given item, it is often necessary to provide a history of the item and its development. The identify operation prints a formatted listing of all the information pertinent to a configuration item or a particular version of a configuration item. This information allows reasoned decisions to be made about the item.

- **Retrieve** items from the library using attribute-matching. The retrieve operation yields the call numbers of all the CIs in the library who have attribute values matching those given to the operation. This allows indexing of items, and is a great aid to promoting re-use of software components.

All of the operations implemented thus far in Clemma have been chosen for their accordance with the library model and for their applicability to the problems involved in software configuration management. But, perhaps their prime value is as a means of investigating the types of operations which would naturally be required by someone trying to perform configuration control on a developing software system.

4. Implementation Issues

The current implementation of Clemma is based on the capabilities provided by the Unix™ operating system, and some of the terminology used is specific to that system.

The implementation of the four principle data stuctures comprising the library is fairly straightforward. A library is given a home directory when it is created, and subdirectories are set up for the repository, the user area, the catalog and the usage list. This helps to provide some encapsulation for the underlying implementation of each. The repository will hold copies of all the CIs in the library, which could possibly number in the thousands. For this reason, we are investigating various means of file organization, compression, and archival so that an efficient means of dealing with such a large number of files may be ascertained. Robustness is also a strong concern, as any system such as this must ensure its users that their components, when installed in a library, are as secure or even more secure than they would be when left in the users own directories. Various protection schemes are under scrutiny which may provide this security.

The catalog of a library is probably the next most important data structure. It is used to provide the central storage and indexing facility for all of the attributes of the library items. As this function is primarily that of a database, the Troll/USE DBMS is being used in the current implementation of Clemma. Troll provides a powerful, flexible, robust interface to the catalog, and seems to be a tool which will have a great deal of applicability in the future of Clemma and other SAGA tools.

The usage list is primarily an indexing tool, and so may also be implemented in Troll. Because of a somewhat simpler nature, however, other types of structure are being looked at as a method of implementation. If the inherent slowness of a DBMS can be avoided while still providing the necessary function and robustness, then it is obvious that such efforts are necessary.

The last structure of a library, the user area, is simplest. This is a directory of read-only copies of checked-out items. The user gets a link to one of the copies when s/he does a check-out on that item, and the link is removed when the item is returned. Since all of the copies are owned by the director of the library, there is no chance for accidental deletion of the item by the user. This scheme provides a simple means of controlling the sharing of such items by several users.

5. Conclusion

Clemma is an attempt to provide a simple, flexible means of constructing and maintaining configurations of small- to medium-sized software systems. The basic premise is the treatment of the components of a system as attributed objects, and the use of a library model for the storage, indexing, and sharing of these objects in a configuration of a system. The Unix™ file system used as the implementation medium, and the Troll/USE DBMS is used to provide for the storage and indexing of the attributes of the stored items. We believe that Clemma is a useful tool and one that will provide many important insights into the problems involved in software development.

6. Bibliography

1. Bersoff, Edward H. *Elements of Software Configuration Management.* **IEEE Transactions on Software Engineering** (January 1984) vol. SE–10,1, pp. 79–87.

2. Schwartz, David P. *Summary of the 4th IEEE SCM WG Meeting.* **ACM SIGSOFT Software Engineering Notes** (January 1985) vol. 10, no. 1, pp. 69–73.

3. Shigo, Osamu, Yoshio Wada, Yuichi Terashima, Kanji Iwamoto and Takashi Nishimura. *Configuration Control for Evolutional Software Products.* **Proceedings of the 6th ACM/IEEE International Conference on Software Engineering** (September 1982) pp. 68–75.

4. Lampson, Butler W. and Eric E. Schmidt. *Organizing Software in a Distributed Environment.* **ACM SIGPLAN Notices** (June 1983) vol. 10, no. 6, pp. 1–13.

5. Cristofer, Eugene, T. A. Wendt and B. C. Wonsiewicz. *Source Control + Tools = Stable Systems.* **Proceedings of COMPSAC 80** (1980) pp. 527–532.

6. Tichy, Walter F. *Design, Implementation, and Evaluation of a Revision Control System.* **Proceedings of the IEEE/ACM 6th International Conference on Software** (September 1982) pp. 58–67.

7. Feldman, Stuart I. *Make -- A Program for Maintaining Computer Programs.* **Software--Practice and Experience** (1979) vol. 9, pp. 255-265.

8. Tichy, Walter F. "Smart Recompilation", Conference Record of the 12th ACM Symposium on the Principles of Programming Languages, New Orleans, LA, 1985, pp. 1-21.

9. ----. *Software Development Control Based on Module Interconnection.* **Proceedings of the IEEE 4th International Conference on Software Engineering** (1979) pp. 29-41.

10. Kaiser, Gail E. and A. Nico Habermann. *An Environment for System Version Control.* **Proceedings of CompCon 83** (Spring 1983) pp. 415-420.

12. Bazelmans, Rudy. *Evolution of Configuration Management.* **ACM SIGSOFT Software Engineering Notes** (October, 1985) vol. 10, no. 8, pp. 37-46.

14. Estublier, J., S. Ghoul and S. Krakowiak. "Preliminary Experience with a Configuration Control System for Modular Programs", Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, PA, 1984, pp. 149-156.

16. Lockemann, Peter C. *Analysis of Version and Configuration Control in a Software Engineering Environment.* In: **Entity-Relationship Approach to Software Engineering**, S. Jalodia C.G. Davis P.A. Ng, ed. Elsevier Science Publishers B.V. (North-Holland), 1983, pp. 701-713.

17. Kirslis, Peter A., Terwilliger, Robert B. and Roy H. Campbell. "The SAGA Project: Large Program Development in an Integrated Modular Environment", Proceedings of the GTE/ACM Software Environments Workshop, Harwichport, MA, 1985.

18. Ossher, Harold L. "Grids: A New Program Structuring Mechanism Based on Layered Graphs", 11th ACM Conference on the Principles of Programming Languages, 1983, pp. 11-22.

A Preliminary Proposal for a Software Engineering Management Tool

Robert N. Sum, Jr.

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, Illinois

A Preliminary Proposal for a Software Engineering Management Tool

Robert N. Sum, Jr.

University of Illinois at Urbana-Champaign
Department of Computer Science
Urbana, IL 61801

1. Preface

This paper is a statement of ideas that are currently being investigated. We believe that many of them will be useful in a software management tool. We would appreciate comments, criticisms, and references to similar work.

2. Introduction

We wish to automate much of the management and tracking of the products involved in the lifetime of a software system. To do this we need a model of the tasks involved and a means to implement the model. We present a consumer producer model that is based on a production cycle that occurs in what we view to be similar situations in the "real world," e.g. the construction industry [Spector and Gifford, 86]. For the implementation, only speculation about characteristics of the tool is now offered. We will close with how this management tool relates to some of the ideas in the literature and the SAGA project.

3. Model

We base our model on a management by objectives approach where a producer satisfies the need of a consumer. A consumer has a need for a product, either goods or services, which he must request from someone other than himself. Therefore, he procures a producer to provide the product.

This model (see Figure 1) is a simplification of what we perceive to be the process in the "real world." Often a product requirements are given to many producers who submit proposals for a product that satisfies the consumers requirements. The consumer then chooses the producer with the best proposal and works with him to create a specification for the product. The producer then creates a product to meet this specification. In some sense, the specification is implemented under timing constraints and other acceptance criteria. After the consumer has received and accepted the product, the production cycle ends. We call this production cycle a *task*. Finally, we note that if a producer is not able to satisfy the specification sometime during the course of the production, then the consumer and the producer may agree to some revision of the specification so that it may be satisfied.

In this model, a specification specifies the consumer, producer, resources supplied by the consumer, product(s) to be delivered by the producer (including progress reports), start and end dates, delivery dates, and acceptance criteria for product(s). The specification does not specify how the producer will fulfill the specification. A specification may request such a large product that the producer in turn becomes the consumer several (sub)products. These (sub)products should be invisible to the original consumer.

4. Characteristics

In this section we describe some of the characteristics of the proposed management system. We include some basics about the tasks used in the system and the requirements made of the system.

Consumer	Producer
Product Requirements	
	Product Proposal
{create/revise specifications	
	modify/accept specifications}$^+$
authorize task	begin implementation
.	.
.	.
.	.
{sometimes some problems	
	resulting in specification revision}
.	.
.	.
.	.
	Product is delivered
Product is accepted	

Figure 1. Consumer Producer Model.

4.1. Tasks

The *task* is the basic entity in the system. It will be a highly structured document in two parts. The first part is the specification. The second part is the implementation. Both parts will be machine interpretable. This will be accomplished with programming language techniques, although the tasks and their relationships form a database. For example, we expect the specification and implementation parts to have a relationship similar to the definition and implementation modules in Modula-2, while keeping track of the state of tasks and the relationships between them is best done using database methods.

The task definition will be visible to both consumer and producer. It contains consumer identification, producer identification, start date, end date, delivery dates, resources supplied by the consumer, products to be delivered, and their acceptance criteria.

The task implementation is the part of the task that is private to the producer. It includes the definitions of (sub)tasks and possibly other actions that the producer must perform. It is expected that these (sub)tasks and actions may be related in a manner similar to the events in PERT charts. Simple PERT charts are not sufficient, however, because we need to be able to "execute" them. In particular, we may wish to use looping constructs that "trigger" on resources or inputs supplied by the consumer. For example, in a change control board we would like all user change requests to follow the same procedure during a maintenance task.

4.2. System Requirements

We want the management system to accept task specifications, execute task implementations, "notify" consumers of producer failures for certain criteria, and automatically generate certain types of reports (given some description by the consumer within the task specification). However, we currently assume that managers will use separate tools for such things as cost estimation and that data from these tools would be entered into tasks by hand during their development. (A project history may be maintained by the system so that these tools and future projects could draw on the experience of present and past projects.)

We expect the management system to be able to run if only task specifications are available. In other words, a consumer and a producer may authorize a task before the producer completes or even starts the task's implementation. The implementation must, however, be completed before the system needs to execute it, i.e. before the start date of the task.

The early stages of requirements may be done as informal development of the task specification by the consumer and the (prospective) producer(s). Authorization of the task would be at the time that it is submitted to the management system (e.g. compiled and loaded).

If we are to allow for task revision as noted in our model, then we need a very flexible system, to say the least. This may be handled in part by a version control mechanism. We hope to avoid full-fledged object oriented systems like Smalltalk because of their complexity and difficulty with efficient implementation. We do notice that a blend of programming language techniques (e.g. task contents) and database techniques (e.g. report generation) will be required.

Finally, we would like to have a friendly user interface. It may be possible to do task specifications with form fillers or structured document editors (e.g. [Kimura, 86]). For the implementations, we would prefer a graphical interface as it would make the PERT qualities more apparent. We note, however, that our goal is to build a management system, not a slick user interface.

5. Conclusion

We have presented a consumer producer model to be the basis for a software management system. We now mention some relationships of the software management system (tool) to the literature and the SAGA project.

5.1. Literature Relationships

The STARS Project [Martin, 83] defined various task areas in software engineering that it would work on. One of these is the project management task area [Lubbes, 83]. In [Lubbes, 83], Lubbes presents a table of functional capabilities for software management. Figure 2 shows those capabilities that we believe this proposed system will support in some part.

We also believe that the consumer producer model can support various management structures. In [Daly, 79], Daly compares and contrasts the three main management structures: project, functional, and matrix. Even though we have not yet worked detailed examples, we believe that the consumer producer model is sufficiently flexible to capture the dependencies in each structure, including those in which one person is responsible to more than one manager.

Some similar ideas appear in the BRICS system [Howes, 84] which was done manually initially. An automated version was (is) under development. Among these ideas is the ability to model work breakdown structures. Tasks and sub-tasks should be able to model work breakdown structures nicely.

We are still searching the literature for information about such management systems. We also expect that there are some corporate systems without publication exposure.

5.2. SAGA Relationships

The management system will be integrated with other SAGA tools. The most important of these tools are the configuration management and electronic communications (see Figure 2). Resource specifications for access to system documents, libraries, and workspaces may be specified in tasks. During execution, tasks will call upon the configuration management to supply resources. Communication of tasks and and notification of status changes may be done using mail, notesfiles, or trays.

In [Campbell and Terwilliger, 86], there is an example using a change control board in the SAGA ENCOMPASS environment. Figures 3 and 4 show some kinds of forms which task specifications could be based on. We have used some BNF-like notation in the figures to indicate the use of standard forms for tasks, i.e. the system may be able to support different "types" of tasks. At this time, we are still investigating semantics for the relationships of the data in a task. These semantics will depend on the database aspects of the management system.

Capability	Support
Database Management	queries and reports about tasks, possibly to other tools
Telecommunications	notification via mail, notesfiles [Essick, 84], and/or trays [Campbell and Terwilliger, 86]
Interactive Work Planning	creation/revision of tasks
Schedule Generation	tasks, especially details inside implementations
Mgt. Information Reporting	automatic reporting in task specification, automatic notification of task failures
Configuration Management	interface to configuration manager for resource allocation and work spaces

Figure 2. Management Tool Capabilities.

```
                         User Change Request <request_id>

Originator: <person>       Receiver: <person>
At: <address>              At: <address>
Phone: <phone>             Phone: <phone>
{Net: <net_address>}       {Net: <net_address>}

Received: <date>
Accepted: <date>
Closed: <date>

Product: <id>
Product Number: <product_number>
Version: <version_id>

Related Products: {
                           Product: <id>
                           Product Number: <product_number>
                           Version: <version_id>
                           }

Request Type: <Error|Modification|Enhancement>
Severity: <severity_level>

Current Behavior: <text>

Requested Behavior: <text>

                               Resolutions

[Temporary:
                           <date>
                           <Restriction|Workaround|Patch|Simulation>
                           <text>]

Permanent:
                           <date>
                           <Update|Release>
                           <text>
```

Figure 3. User Change Request.

Program Modification Request <request_id>

Requested by: <person>	Analyst: <person>
At: <address>	At: <address>
Phone: <phone>	Phone: <phone>
{Net: <net_address>}	{Net: <net_address>}

Received: <date>
Accepted: <date>
Completed: <date>

Associated UCR: <pointer_to_user_change_request>

Resources: {<access_to_other_services>}

Findings: <text>

Recommendation: <Accept|Reject>

Associated PMP: <pointer_to_program_modification_plans>

Figure 4. Program Modification Request.

6. References

[Campbell and Terwilliger, 86]
Campbell, Roy H. and Robert B. Terwilliger. *The SAGA Approach to Automated Project Management.* In: International Workshop on Advanced Programming Environments, Lynn R. Carter, ed. Springer-Verlag Lecture Notes in Computer Science, New York, 1986.

[Daly, 79]
Daly, Edmund B. *Organizing for Successful Software Development*, Datamation, December 1979.

[Essick, 84]
Essick, Raymond B., IV. *Notesfiles: A Unix Communication Tool*, M.S. Thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1984.

[Howes, 84]
Howes, Norman R. *Managing Software Development Projects for Maximum Productivity*, IEEE Transactions on Software Engineering, SE-10 (1), January 1984.

[Lubbes, 83]
Lubbes, H. O. *The Project Management Task Area*, IEEE Computer, 16 (11), November 1983.

[Kimura, 86]
Kimura, Gary D. *A Structure Editor for Abstract Document Objects*, IEEE Transactions on Software Engineering, SE-12 (3), March 1986.

[Martin, 83]
Martin, Edith W. *Strategy for a DoD Software Initiative*, IEEE Computer, 16 (3), March 1983.

[Spector and Gifford, 86]
Spector, Alfred and David Gifford. *A Computer Science Perspective of Bridge Design*, Communications of the ACM, 29 (4), April 1986.

A Summary of the Software Development Cycle of AT&T in Middletown, NJ

Robert N. Sum, Jr.

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, Illinois

A Summary of the Software Development Cycle of AT&T in Middletown, NJ

Robert N. Sum, Jr.

Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield Ave.
Urbana, IL 61801

1. Introduction

This paper is a summary of the software development organization and practices used in the System 75 and related projects at AT&T Information Systems in Middletown, NJ. It is the result of a one week observation by Robert Sum (the author) of the University of Illinois at Urbana-Champaign, Urbana, IL. The purpose of the week's observation was to gather information about current software engineering practices for input to research projects in software engineering at the University.

The next section of this paper presents an overview of the AT&T life cycle and the AT&T mangement structure. Subsequent sections discuss various processes in the life cycle from the viewpoints of the people and meetings the author attended. Often the content of these discussions will be derived primarily from a meeting with one particular person. During these discussions, some specifics about tools will be presented, including things that work, things that do not work, and suggestions for things people would like to have. The paper closes with a summary, acknowledgements, and references.

2. AT&T Organization

In this section, the life cycle and personnel structure that AT&T uses in software development are described. It should be noted that the author observed several projects at different stages of development. Even though all were based on the same philosophy and common ancestry, there were some differences. This paper is a synthesis of these projects' development. Hopefully, it reflects their philosophy in its most recent form without evolutionary differences causing problems.

2.1. Life Cycle

The AT&T software life cycle is essentially a classical "waterfall" model. Figure 1 outlines this life cycle showing the processes (ellipses) involved and their products (boxes). The processes used are: product definition, requirements definition, architecture definition, feature definition, high level design, code and test, integration and quality development, system test, and qualification. Most products are documents until the last half of the life cycle where code is produced. The documents and code produced include: technical prospectus, requirements, architecture, feature specifications, external design, development specifications, development code, system code, field code, and released code. The major divergence from the classical "waterfall" model in the AT&T model is separate independent development of the system's

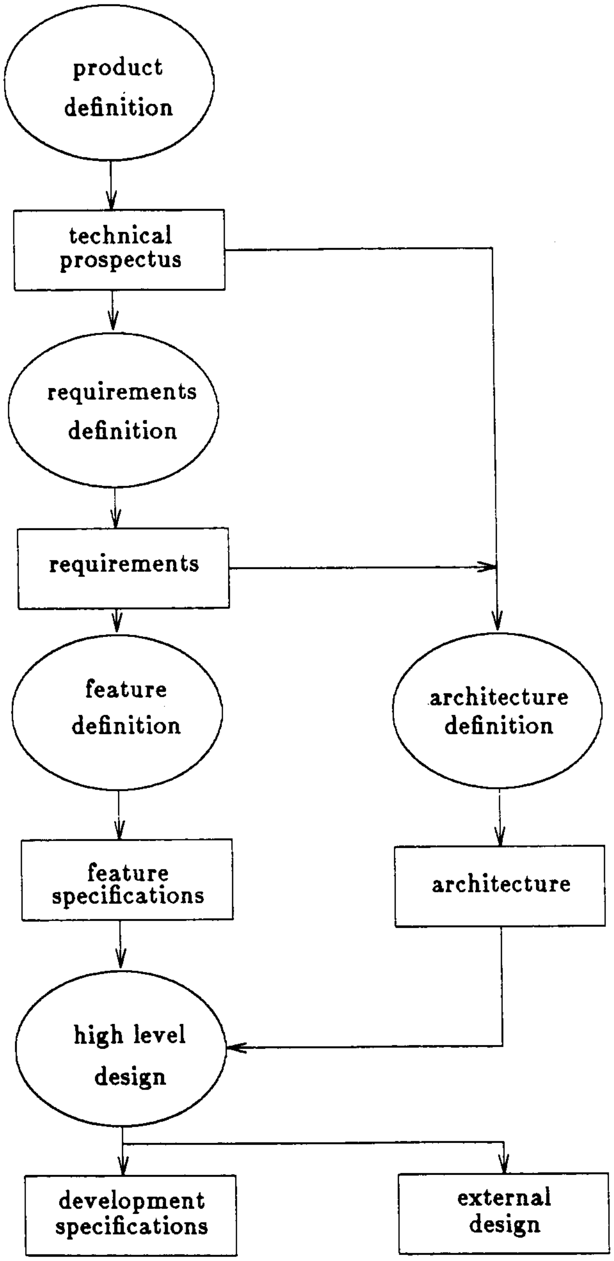


Figure 1. AT&T Life Cycle

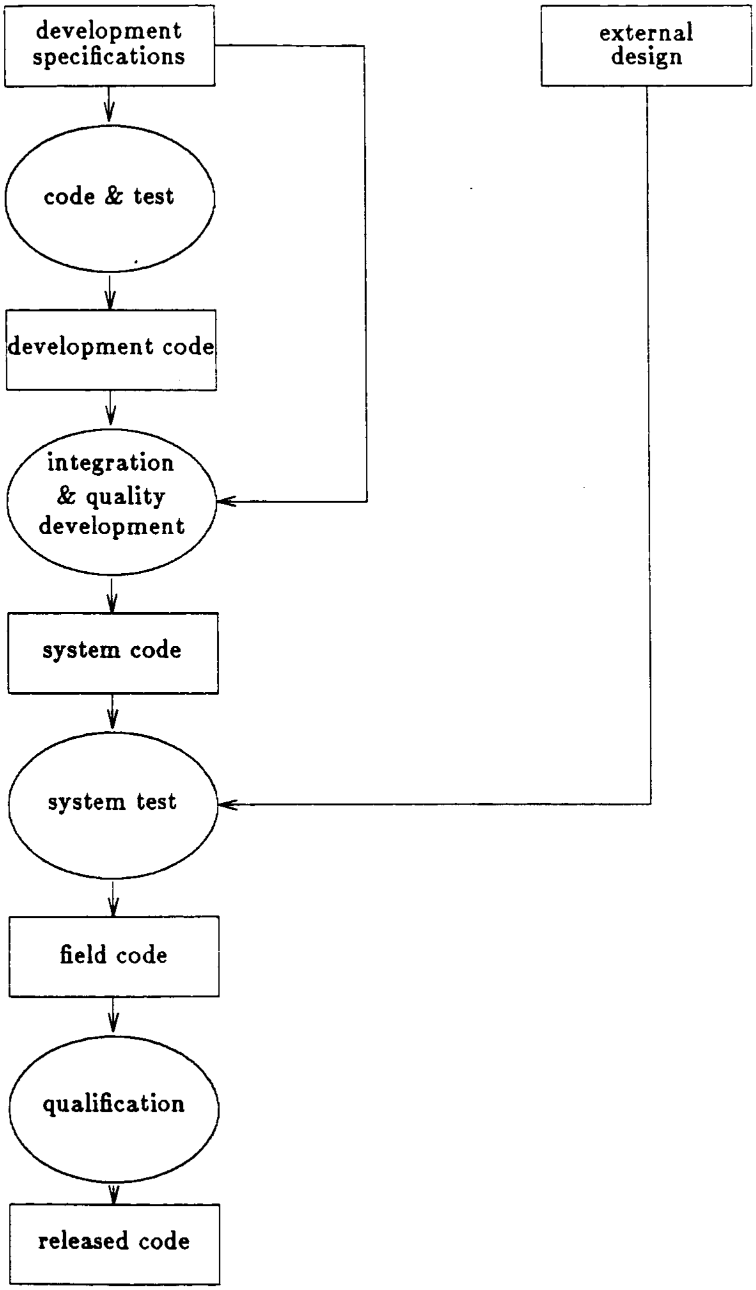


Figure 1. AT&T Life Cycle (cont'd)

architecture and the system's features. This separation is also visible in the documents produced during development where one side (left) is concerned with the external behavior of the system and the other is internal construction. (In relating Figure 2 to Figure 1, the process specifications and process decomposition specifications are parts of the development specifications. Also, dashed boxes refer to code while solid boxes are documents.)

One should note that Figure 1 describes the primary development cycle and that several other smaller cycles run in parallel and interact with it. These other smaller cycles include system test development and project management. System test uses a development cycle that is very similar to the primary development cycle. It includes system test plans and various design and implementation steps. The

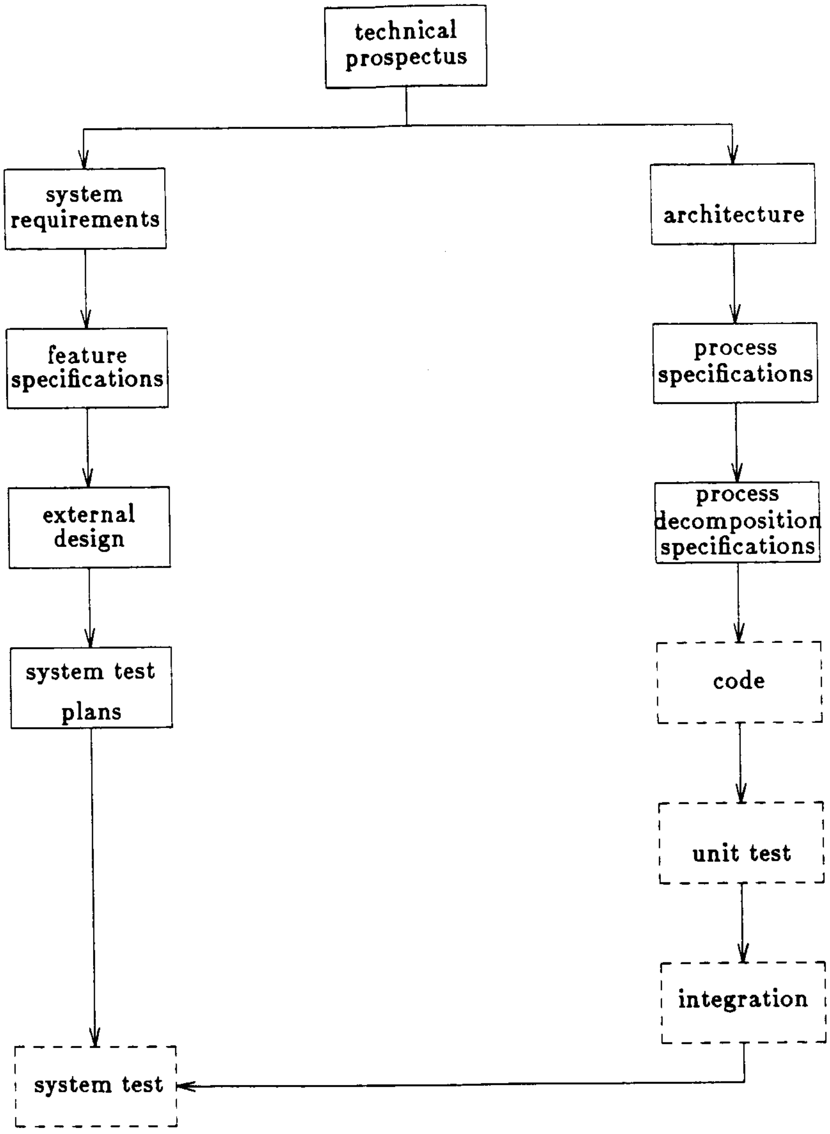


Figure 2. Document Hierarchy

project management cycle is based on sampling of the primary development cycle to monitor its progress and ensure its integrity.

2.2. Personnel Structure

The management of the development process is done with many specialized groups. These include system engineering, project management, project coordination, software design, software tools, development, integration, quality development, system test, and field support. While most groups have input to several of the processes in the life cycle, many of the groups control a particular process. For example, the product definition process is controlled by systems engineering in that they produce the technical prospectus, but product management and software design provide an equal if not greater amount of input to the technical prospectus. Also, the technical prospectus is reviewed by system test, field support, and other groups to alleviate any difficulties that they may find early in the project's life time. Figure 3 lists most of the relationships between development groups and development processes.

AT&T's personnel structure is a project oriented structure with some leanings toward a matrix structure to gain some of the management advantages. For instance, the developers are all devoted to a particular project, but a member of a project coordination group may have several projects to coordinate. It is also possible for one person to do more than one job such as project management and heading a development team. It is often the case that tasks such a project management are distributed in a functional manner. This has become even more prevalent with the newest project that is being developed concurrently at sites separated geographically and computationally. The basic structure of the personnel and project that the author observed is depicted in Figure 4.

Development Groups and Processes

Development Group	Processes		Contact Person
	Controls	Reviews	
Systems Engineering	Product Definition Requirements Definition	Feature Specifications	(none)
Project Management	(Whole Project!)	Technical Prospectus	L. Beaumont V. Sherman
Architecture	Architecture Definition High Level Design	Technical Prospectus Requirements	M. Johnson
Development	Feature Definition Code & Test	Technical Prospectus Requirements Development Specifications	S. McKechnie P. Gerhardt
Integration & Quality Development	Integration Quality Development	Technical Prospectus Development Specifications	P. Matteo R. Wrigley
System Test	System Test	Technical Prospectus External Design	S. Siverstein
Field Support	Qualification	Technical Prospectus	C. Allison
Tools			G. Yates T. Pederson

Figure 3. Development Groups

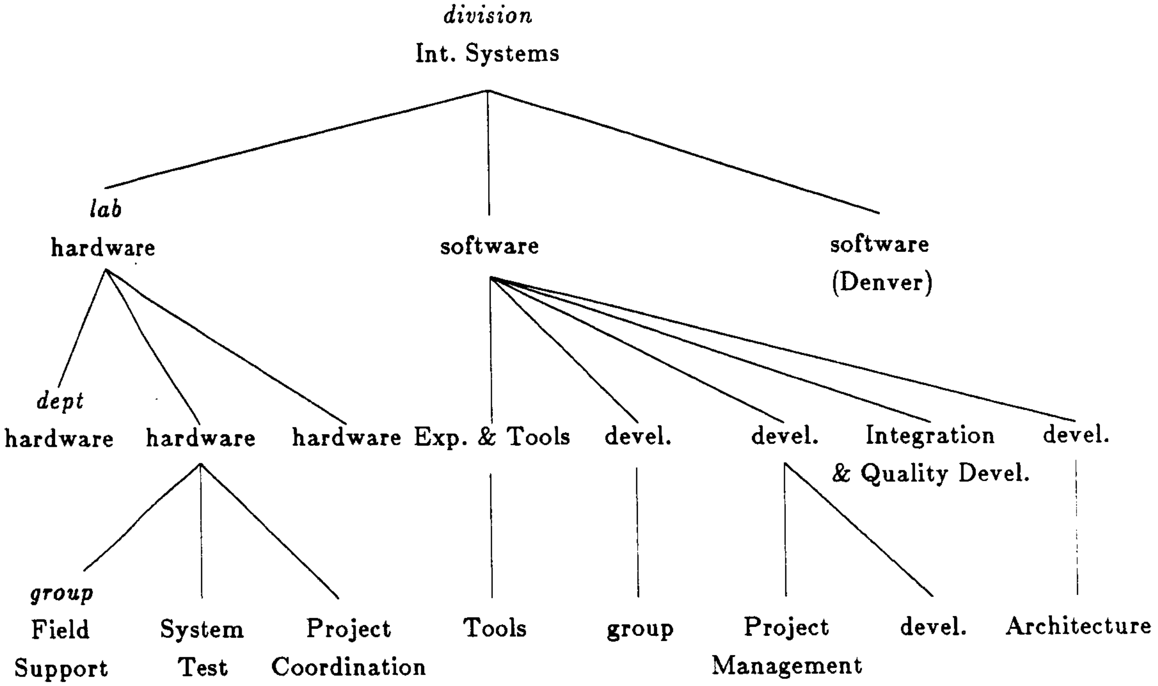


Figure 4. Personnel Structure

3. Systems Engineering

Systems Engineering is the liaison between development and marketing. Initially, SE receives from marketing the perceived needs of the customer. Then, SE receives from development information about technical capabilities that customer may want. It is possible for development to propose a project and then have SE check with marketing to see if it is marketable, but this is not common primarily because of its high failure rate in producing marketable products. Often, SE acts as a mediator between marketing's perception of the customers' needs and Development's desire to create a system with all of the latest and greatest technology. After doing its own analysis, SE decides whether to start initiate the development of a new product.

To initiate a new product, SE begins the product definition process of the main development cycle (noted previously in Figure 1). SE brings together the Project Management, Architecture, and Development people with the goal of producing the Technical Prospectus. It is in production of the TP that SE often finds its most trying times as a mediator. The Technical Prospectus describes the purpose of the product, its environment, and its features. This document is informal and has a varying level of detail about the items it describes. It may contain only one half page per feature and still be 200 pages long.

After the TP has been completed, SE works together with Architecture and Development during the Requirements Definition process to produce a formal Requirements document that clearly states the features to be provided by the product and the resources required to develop and maintain the product.

The last major interaction that SE has with a project is the review of the feature specifications which are done by the Development group. At this time SE makes sure that the features described earlier in the

Requirements are specified correctly so that upon implementation the product will meet the customer needs.

Final Note: This information about SE is derived from various meetings and conversations as the author did not have the opportunity to meet with someone from Systems Engineering.

4. Project Management and Coordination

In this section the author includes both Project Management and Project Coordination because of their close relationship in managing and monitoring a project. Project Mangement concerns the overall organization of the project concentrating on the resources (people, machines, etc.) needed to develop and maintain a project. Project Coordination concerns the organization of the project in time by tracking deliverables and their delivery dates to ensure that the project stays on schedule. Finally, we spend some time discussing the main meetings used to monitor product development.

4.1. Project Management

In general, text book (formal) management methods exist but they have problems when being applied to actual developments. Many of these problems arise from the fact that most projects have a history, i.e. very few projects actually start totally from scratch. History related problems include uncertainties in re-used code, incremental feature development, retrofitting new features into old products, and merging technologies. Other problems result from the inexactness with which certain resources (most notably people) can be measured and predicted. Resource problems include variations in personnel experience, personnel productivity, and personal preferences which can require a lot of political effort to solve.

Project Management is most visible during the early processes of the development cycle. It has direct input during Product Definition and prepares the Project Plan in association with the Architecture and Requirements. The Project Plan includes schedules of varying detail (including staffing), a brief product description (at most one paragraph per feature), resource descriptions (including re-used software, hardware, computer support), development method outlines, and a list of open (unresolved) issues. After this early activity, PM is always present in the background dealing with unresolved issues ensuring the project's progress toward completion in a timely and efficient manner.

4.2. Project Coordination

Most of the monitoring of the project's progress is done by the Project Coordination group. Generally, each project has one project coordinator. This project coordinator is charged with the task of ensuring that deliverables promised by one group to another will be delivered by the time that they are needed. The coordinator schedules all project milestones and acts as a negotiator for all the development groups. She oversees or writes project plans, tracks all milestones and deliverables, and aids project audits.

Project Coordination starts during Product Definition and continues actively throughout the project's life time. The primary mechanism for deciding project coordination issues are planning and status meetings. Planning meetings plan the future and status meetings make sure the present agrees with the plan. These meetings are often held at different levels of detail for different managerial positions. Meetings for developers and supervisors (first level managers) are down to the individual deliverable that is being produced whereas meetings for department heads and directors look at larger time scales like project phases.

To create the project's schedules, the project coordinator often starts with a marketing date and then must make development fit a schedule designed to be done by that date. Although there are some automated tools to support some aspects of project coordination and tracking, many (most) of the work is still done by hand. (More details about tools will follow later.) Some of the basic problems are:

1. People do not reliably inform project coordinators of a task's completion or delays,
2. People try to prevent lateness by moving dates without dependency information,
3. Current tools do not communicate with each other,

4. Tools do not provide a way to selectively view time slices,
5. It is awkward to talk about partial completeness of tasks,
6. The project coordinator must have basic knowledge of all development processes and groups on a global scale.

AT&T has tried individual coordinators for parts of projects or for types of tasks, but has had more success with the global coordinator.

4.3. Meetings

There are three basic meetings used for management, scheduling, and tracking of project development. They are planning, status, and integration meetings.

4.3.1. Planning

The planning meetings decide the project's goals, deliverables, schedules, and task assignments. These meetings are held either weekly or bi-weekly.

The agenda of planning meetings work on the high level view of the project and how the project should be organized. A planning meeting for a project just beginning (almost planning plans) may include:

1. software development planning,
2. hardware development planning,
3. integration planning,
4. software responsibilities,
5. summary of meeting, and
6. open discussion of things for the next meeting.

Most of the items above are discussed with concern what are reasonable milestones and deliverables for each milestone. During the open discussion, special tasks and problems are brought up so that they may be investigated before the next meeting and discussed then if necessary. A list of important tasks and problems is kept. An example of special tasks might be: how to define certain deliverables or how to define terms like "quality" with respect to some system performance.

4.3.2. Status

The status meetings review the current state of a project and compare it the completed tasks with those that should be completed according to the project's schedule. One should note that planning meetings do not run just one or two weeks ahead of status meetings, but that they run substantially ahead of status meetings (several months).

The agenda of status meetings changes depending upon the age of a project. In the case of a new project, some of the following may be discussed:

1. problems with coordination of development start up at multiple sites,
2. additions and deletions to parts of the project (this has repercussions in planning meetings),
3. problems with tools, equipment, and other resources needed for development,
4. contents and completion of documentation plan for the project,
5. changes to items being worked on since the last meeting,
6. discussion on how to track software development progress, and
7. methods including verification methods to be implemented.

On the other hand, a somewhat older project with several releases in the field may discuss:

1. what fixes have been put into various releases,
2. importance of new bugs and how soon they need to be fixed,

3. problems with fixes that have not yet been completed,
4. dates for system testing of new versions.

Also in older projects, the status meeting handles most of the short term scheduling and planning work.

4.3.3. Integration

Integration meetings concern the building of new releases (versions, revisions) of a product and the problems encountered in the process. Some of these might be code changes that conflict some other code or the discovery of an inadequacy in an integration tool. Problems are resolved to ensure the integrity of the product.

A typical integration meeting might include the following on its agenda:

1. tool problems,
2. laboratory hardware (new and old),
3. modification requests (MRs) i.e. bug fixes and product enhancements,
4. development workspaces,
5. special items such as introducing new tools, and
6. integration procedures.

Another meeting closely related to the integration meeting is the MR review meeting. The MR review meeting reviews all pending MRs on the project and whether their status should be changed. A list of the most critical bugs is maintained so that fixing them receives the most attention. The MR review meeting is composed of specialists from each part of the project to expertise in all areas in deciding the nature of the MRs. Several members are from the Quality Development group as well. It is after an MR has been fixed that it appears in the integration meeting.

5. Architecture

The systems Architecture is developed from the Technical Prospectus and the Requirements by the Architecture group. The Architecture is a very high level design document that specifies how the system will be implemented to support the features in the Technical Prospectus and the Requirements. In Architecture, as in other processes, history and experience are the major tools. This section briefly describes the Architecture Definition process and the creation of a work breakdown structure.

5.1. Architecture Definition

This description of Architecture Definition is based on the Architecture Definition used for a family AT&T switching systems. Almost every system has a predecessor which provides an immediate basis for the new system's architecture. In the event that the new system is "just" a new release or an extension of an existing system, then the new system may re-use both hardware and software. Even when a totally new system is designed, it often replaces another system which has some related functionality.

For the first switch in this family, there were other related systems that experience was drawn from to create the Technical Prospectus. Three areas visible in the first switch's Technical Prospectus that can be seen in all of the switches are:

1. interactive development – the telephone user and the functionality that he sees,
2. system administration – the customer's person in charge of programming and maintenance,
3. system maintenance – installation, use without user intervention, reliability, audits, and self-diagnostics.

Consideration of the support required by each of these areas lead to a layered architecture with a kernel operating system (see Figure 5). Each layer in the product provides primitives for the layer above. Interfaces between each area were defined in terms of the primitives that each layer in the architecture provided. With each new system in the family, adding new features is done by looking in the existing architecture and finding where the appropriate primitives are provided or where they should be added. General opinion at AT&T is that this architecture was quite well designed as it has supported a successful family of switches without becoming unbearably complex. The same architecture has been used for different hardware (including microprocessors with different hardware architectures).

5.2. Work Breakdown Structures

Work breakdown structures are derived from the Technical Prospectus and the Requirements. In general, large areas are mapped onto company structures such as departments or groups (for example, system test). Individual features are eventually mapped to individual developers.

Initially, Systems Engineering sends the Requirements to various supervisors that will be involved in the project. SE meets with the supervisors to determine answers to:

1. How many people are needed and are there enough people?
2. Is the project technically feasible?

To determine the answers, the supervisors rely primarily on personal experience. They do a quick-and-dirty high level design (not generally committed to paper) to determine a basis for staffing. This quick-and-dirty method has steps like:

1. Choose a large section of the project
2. categorize its features
3. fit features with the Architecture and people.

In fitting the people to the features, supervisors use a ranking of employees by ability. Sometimes a supervisor maintains a concrete list, but more often this is a mental list. This way supervisors try to allocate

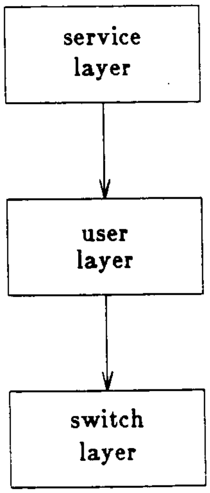


Figure 5. Switch Architecture

the most difficult jobs to the best people. After this rough design and people fitting, the whether the project is technically feasible and whether there is enough people should be answered.

Next, Project Management and Project Coordination determine dependencies and track progress using planning and status meetings. Estimates of work and milestones are developed in a forum where people make collective decisions on work estimates. PM and PC notify supervisors of late items and coming events. If a milestone is missed, then either functionality is dropped or the delivery date is slipped. This decision is usually made by Project Management or Systems Engineering.

5.3. Some Psychology

There is a lot of psychology and politics involved in determining system architectures and work breakdown structures. For instance, if development wants to build a project, then they tend to underestimate the cost of the project. On the other hand, if they are not interested in building it, then they overestimate the cost. Planning is the mechanism by which Systems Engineering and development reach a compromise. If one lets either side win over the other then the project usually results in failure. If development is interested but SE has no market, then there is a wasted effort. On the other hand, if SE has a market and development is uninterested, then development does a poor job because they do not care about what they are working on. Understanding peoples wants and needs is the biggest problem of project development.

One example of this is in the Architecture of the switches mentioned earlier. The Architecture is based on primitives and layers. But, it could have also been done vertically by feature. Why was one chosen over the other? One hopes that technical decisions such as how well the architecture supports current and proposed extensions prevail. Often, however, some approach has a strong "political" spokesperson that convincingly argues his approach. And, there is the ever present history mechanism. Looking back over several generations of switches, it is seen that "new" architectures appear only about every 15 years.

6. Development

Development's two major products are the Feature Specifications and the Development Code. The Feature Specifications are the exact specification of how each feature will act including error conditions. The Development Code is the code expected to be released that must pass the various stages of integration and testing. This section looks at the developer's position in the life cycle in terms of both new development and old development (fixing bugs).

6.1. New

The developer is basically at the bottom of the development structure looking up to see the entire project. In the beginning of a project, it has been beneficial to have other groups such as System Test review the various specification and design documents. Near the end of the development, it has been beneficial to have an "in house" system to use as a test. (This system is not in a testing lab, but rather the phone system used in day to day work.) Often, however, the developer can not get a big picture of the system he is working on. In other words, it can be very difficult to understand interactions and dependencies between his part of the project and the rest of the project. A system that allows the developer to see these dependencies would be very welcome.

The environment that the developer works in is currently a collection of tools built on top of other tools to force them to work together. For instance, the developer uses a combination of tools from the Local Administrative Tool Kit (LATK) and the Object Generation System (OGS) to create workspaces in which to work. OGS is in turn built on top of MESA which is built on top of SCCS. While most of these things work pretty well, there are some failings especially in maintaining dependencies between modules that cause problems with building the system. In addition, there is very little connection between development code, the Modification Request (MR) system, and the Project Document (PD) system. This lack of connection results in much time spent tracking down the correct person to talk to when a problem occurs. Or, in the case of a bug report (MR), there is no automatic suggestion to other releases that this bug may also be present. This last problem is most important when multiple versions of a system are being maintained, possibly during beta-testing or controlled introduction.

Other problems that occur while a developer is coding a new system include:

1. Large amounts of code make it difficult to find definitions and other things,
2. Debugging a large (real time) system can be very difficult,
3. Edit-build-test-debug cycle can take a very long time - especially when using multiple machines,
4. communication - finding the right people and information quickly and easily.

Some (partial) solutions to these problems include:

1. vi with tags - a version of vi that uses tags generated by the compiler and other utilities that allows vi to be used as a browser. A few key strokes finds the definition of a define, variable, or function and puts it in a separate window.
2. A complicated debugging simulator that runs on the development machines rather than the target machines makes debugging much easier.
   There are also some tools that index the error messages so that one can find where and error message was generated. But these are not fully developed.
3. One reason for the length of the edit-build-test-debug cycle is building with cross-compiling and down loading. Reliable high speed communications can alleviate some of the tedium and frustration.
4. For communication, enhancements to mail and a bulletin board system have worked well. An electronic calendar system to help schedule appointments would be helpful. Also possibly, a voice storage phone system (i.e. a centralized phone answering machine) would be helpful.

6.2. Old

Old refers to maintaining the development of many releases simultaneously including bug fixes. One of the biggest problems here is that of a bug fix being needed in releases both older and newer than the most current one. Currently, Modification Requests are for one person on one release. If he realizes that the problem may be more wide spread, then he may be able to search out the person responsible in a similar area on the other release(s) and notify him. Often this is not possible. The rest of this section details this problem and some solution ideas.

In general, bugs are handled in using the MR mechanism in the following manner:

1. an MR is received from customer, developer, or whoever,
2. it is assigned to someone to investigate (and plan a fix),
3. it is fixed,
4. the fix is sent to integration for inclusion in the next system build,
5. System Test then verifies by testing that the problem is fixed.

This process depends on some tools in LATK linking MR, MESA and other tools together. At AT&T the Integration group overseas and reloves the integration problems with bugs and keeps track of what bugs are fixed in which versions and releases. This seems to work and a few tasks such as some of the MR status changes are done by the LATK tools. But, each Integrator is concerned only with a specific release and does not necessarily know about the others. A lot of time is spent taking care of these bug reports and even more is consumed by migrating them to other versions and releases.

The process of migrating a bug fix is called a "bugout." The major problem areas with bugouts are:

1. different releases may have different structure,
2. how much testing is necessary to ensure fix,
3. which changes be broadcast to other releases or versions.

Figure 6 shows a couple releases each with a few versions. It also includes examples of bugout paths and feature porting which can often happen as well.

The problem of different releases having different structure refers to some software architecture changes. These changes are small in that they may not affect the system architecture, but source code diffs will become unusable because the context of the diffs is preserved across releases (or versions). Figure 7 shows how a single process may be broken into smaller ones or even recombined into one again in later versions. In some straight forward cases diffs are used resulting in a saving of human labor, but most often some data structure or internal function has changed forcing the re-invention of the fix.

When testing a fix, there is the question of how much testing is necessary. This question arises a second time when a fix is ported. If the fix was tested and verified in another release, then it probably does not need as much testing when it is migrated, or does it? In practice, it is possible to eliminate some testing, but interactions of the fix must be carefully examined first.

Another facet of different releases having different structure is the difference in features between two releases. In this case a table (e.g. Figure 8) of features and releases may be helpful in suggesting what releases need to have a bugout, especially when the bugout is localized to a specific feature. Perhaps a field could also be added to the MR to indicate what features and what versions a particular bug would apply

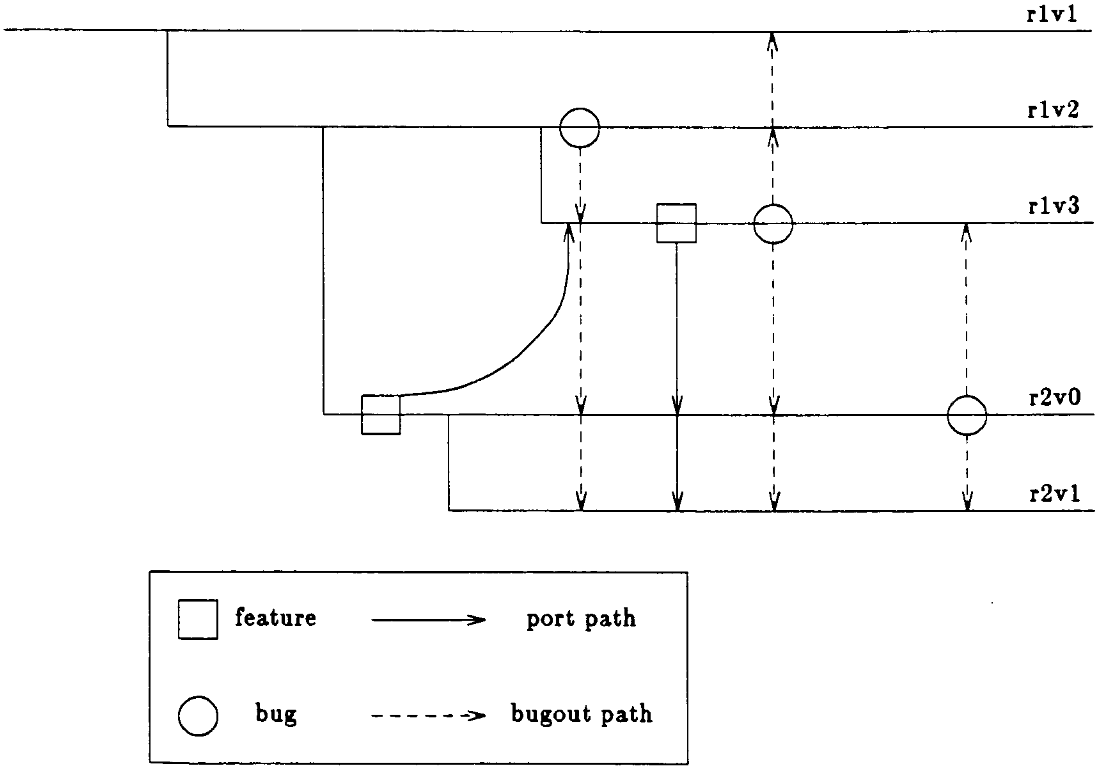


Figure 6. Bugout Paths

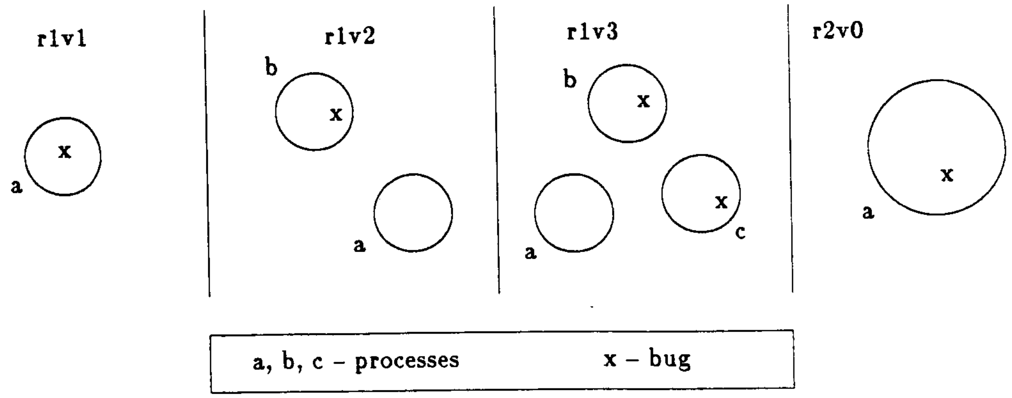


Figure 7. Architectural Changes

Features and Versions					
Feature	r1v1	r1v2	r1v3	r2v0	r2v1
H/M	n	n	y	y	y
ACD	n	n	y	y	y
SW	y	y	y	y	y
ADM	y	y	y	y	y

Figure 8. Bugout Table

to. Then, MRs could be sent automatically to the appropriate Integrators. How much information the system can send and track is difficult to determine. Fixes and enhancements can require a lot of code and other work, even if it is expressible just as diffs to documents and code.

Currently, AT&T people individually generate diffs for versions that are very closely related. For releases that are not closely related, they rely on the fixer finding out who to contact on the other release(s) and forwarding the information about the fix to the other release(s).

7. Tools

The Tools group interacts with most other development groups to provide support for them. Most of the Tools group's work is spent in automating various parts of the development including control of source code, keeping and tracking bug reports, and control of project documents. The Tools group makes very few decisions concerning products, but does figure very heavily in the resources necessary to create the product. Most of the work of the tools group is early in a product's life time, but tools do evolve to better meet the needs of developers during the product life time. This section is included in the paper here because most of the software tools developed to date interact with Development and Integration. Some categories of tools are those for source code control, modification requests, project documents, and project management.

7.1. Source Code Control

The primary source code control mechanism is the Object Generation System (OGS). OGS provides the ability to generate multiple versions of a software product from a single source hierarchy. It adds functionality on top of MESA (Management Environment for Software Administration) which is a facility for maintaining hierarchical structures of SCCS (Source Code Control System) files.

OGS uses a hierarchical directory structure in the UNIX file system for project management. The top level is the project (PJ) level. Successively lower levels are the system, process, and book level. Each project has a base project area for source code and each revision of the project can have its own base object area. This allows many revisions to be kept simultaneously while all share the same source code area. OGS utilities manage the various object code generation details by using MESA to get the appropriate source code and "make" to create the system object code. The MESA system maintains a hierarchy of source code files and their dependencies so that make files can be automatically constructed. The handling of dependencies is done in part by inspecting files and also by giving the utilities knowledge about file types by using special file suffixes. (Actual construction of the make files is done by OGS.) SCCS maintains multiple revisions of individual files using a forward delta storage scheme.

A developer typically follows these steps when using OGS:

1. setup – create a workspace of parallel source directories,
2. sget – get the individual source files to be changed including locks to prevent multiple concurrent changes,
3. edit the source and possibly build local copy to test changes,
4. usubmit – submit the workspace to the Integrator for integration. This last step requests and MR number if there is one, informs the Integrator that this workspace has been submitted, "hides" the workspace by changing the ownership and permissions of its contents, and creates a special shell script that can be used to unify it with the rest of the system.

The commands above are actually Local Administrative Tool Kit (LATK) coatings of the raw OGS commands. This was done in order to help link the OGS system in with other systems such as the MR system for Modification Requests. It also alleviates the individual developer from needing to know the details of more than one code management system.

AT&T has several variations on its source code management. This is partly because each project tends to modify the "standard" tools to fit their needs. For example, the MESA system has capabilities not yet supported by OGS. These include Independent MESA and CASSI. Independent MESA is a simple mechanism by which development may proceed on multiple machines simultaneously. It also includes having more than one user working on the same source file. However, in the latter case the system stores the multiple copies and requires human intervention to produce a single copy that is a merge of the multiple copies. CASSI is an attempt to link the MR system (described below) and the source code systems more closely. Some parts of AT&T other than Middletown have been experimenting with it, but there have been some problems with it that have prevented its widespread use.

7.2. Modification Requests

The Modification Request system is essentially a large database system for tracking and storage of bug reports and enhancement requests. It is used throughout AT&T as the Change Management Tracking System (CMTS). The PCS/MR system at Middletown is an enhancement primarily for user friendliness including automatic notification via electronic mail of certain status changes that occur to MRs.

MRs include the expected data about origin, products, severity, and current status. The status of an MR can be:

a) ui – under investigation, i.e. brand new,

b) pa – pending approval, i.e. a fix has been found but not approved by the MR review board,
c) bf – being fixed, i.e. someone is actually trying to fix it,
d) dup – duplicate of another MR,
e) def – deferred, i.e. not important enough to be considered at this time,
f) nc – no change, i.e. not a problem or not to be implemented after some consideration,
g) c – complete, i.e. fixed or added.

Of the status possibilities above, complete, no change, deferred, and duplicate are considered to be resolved. Deferred MRs can be "unresolved" at a later time should it be decided that they should be implemented.

An MR may also acquire "child" MRs. These are used to note various resolutions and to note that other items (e.g. documents) may need to be updated as a result of this MR. In the latter, child MRs have the same effect as parent MRs and require their own status and resolution changes.

7.3. Project Documents

The Project Document (PD) system is a library of project documents. It maintains control over a project documentation in a manner similar to using SCCS. A project document is given a mnemonic identifier that includes its producer, its type, and its sequence number. As a document is revised the system assigns release numbers to it so that people know whether they have the lastest version or not. Documents may also have different status codes depending on their state of completion. These status codes include draft, preliminary, changed, final, and obsolete. The PD system allows all of a project's documents to be baselined and have MRs written against them. It also makes it easier to distribute project documents to project members. In fact, a project notebook including project documents describing project procedures for documents, MRs, reviews, coding standards, and other things is given to every project member.

7.4. Management Tools

AT&T has a few management tools, but they are help with only a small part of the management tasks. These tools include the Milestone Schedule and Tracking System (MSTS) and a program called Timeline that runs on an AT&T pc. A hand done procedure (tool) called Priority Feedback System (PFS) is sometimes used by managers to help with work assignments and monitoring.

Currently, MSTS allows schedules to be kept on–line on the development computers. Milestones consist of the contractor, the producer, the consumer, original due date, current due date, and previous due date. MSTS does not have a visual representation other than tables and it does not have a good mechanism for setting up and maintaining dependencies. Also, MSTS is used across entire projects and does not have a mechanism to view subset of interrelated milestones. It is very much just a database or record keeping system. MSTS is used by the Project Coordinator and can hold milestones for the entire project.

Timeline runs on personal computers and has dependencies, and some critical path and cost analysis capabilities. It still has some trouble handling everything for a large project, but has proven useful for individual managers to keep track of their groups.

For milestones, one would like a system with the PERT/CPM abilities of Timeline, the scope of MSTS, selective viewing of dependencies, and automatic notifications of approaching milestones to the people producing deliverables and to the Project Coordinator. L. Beaumont has started some experimenting with a relational database to see if most of these capabilities can be developed using various entities to represent documents, milestones, dependencies, and people.

To help manage people and work assignments, some managers use the Priority Feedback System. This is not an automated tool, but it could be automated, at least in part. It consists of a form in which a worker and his manager order the worker's tasks by priority. The worker and his manager then meet every so often to review the tasks and set personal milestones for the worker. This allows the worker's progress to be tracked in a more quantifiable way. It also keeps the manager more informed about the individual's work load and to whom new tasks might be assigned. This tool is sometimes helps with work breakdown structures.

8. Integration & Quality Development

Integration is the process of forming one system from many developers' code. Quality development includes testing the integrated system in a "white box" manner to maintain system stability. AT&T has found it advantageous to have split these two tasks.

8.1. Integration

The Integrator is an individual assigned to a project to do the system Integration. His responsibilities include collection of workspaces, resolution of integration conflicts, maintenance of each integration's MR list, creation of the new system, and integration testing of the new system.

The Integrator begins by collecting workspaces which have been submitted by the developers. Collecting workspaces includes checking to see that all the developers have submitted their work to the system and nagging those that have not. After all the workspaces have been acquired, the Integrator can add the source to the system.

When updating the system source, the Integrator must resolve any conflicts. For instance, it is possible that more than one MR required one source file to change in separate places. If the changes are not obviously independent, then the Integrator must have the developers agree on some merging procedure. Another possibility is special cases that the integration tools can not handle automatically, thereby requiring the Integrator's intervention. He must also make sure that the status of the MRs are updated and that the list of MRs included in a particular system is updated.

After the system source is updated, the new system is made. Most of the time, this runs without any trouble. Occasionally, however, some source code dependencies may be missed and the Integrator will have to fix them by hand. In order to minimize this, a complete rebuilding from scratch of a system is done from time to time (possibly once per week).

After the object code has been produced, the Integrator downloads it into the target machines and runs some simple tests. The download program is also reasonably clever in that it will only download newly built code. The tests that the Integrator runs are fairly straight forward. They just check the current system starts up and appears to run correctly.

The Integrator uses tools from OGS and LATK. In the process described above, the Integrator must set up several environment variables that describe the software base, the target hardware, and the system type (e.g. development or field). The are some tools to help with setting these parameters and examining what OGS will do with them. The Integrator uses the command "review" to help keep track of the MRs in each revision of the system. "Runinstall" uses as input the developers output from "usubmit" to install the workspaces using MESA. Finally, new object code is produced using "runbld" which uses dependency information to rebuild those parts of the system that need it.

Overall, the system works. Most dependencies are handled correctly. There is only one collection of source code with separate object code areas for each revision. Also, the tools have used electronic mail very effectively as a means of communicating error conditions to the Integrator. However, the Integrator has some problems:

1. The system only knows about predefined set of file suffixes so not all files are handled automatically,
2. Tools are so loosely coupled that there is too much room for mistakes in parameters and order of execution,
3. Aside from closing the dependency holes in 1, one would also like dependencies to be more exacting (i.e. there are still some times when the system recompiles more than it needs to).

8.2. Quality Development

It was recognized that there were some problems that fell through the cracks between development and system test concerning the quality and stability of the system. Quality Development is designed to seal those cracks at the point between integration and system test.

8.2.1. Motivation

The major motivation for QD is that customers get extremely upset by bugs appearing where they originally did not exist. The appearance of bugs in places where they previously did not exist make the system look like it is deteriorating rather than improving. The desire to have the system always improve results in some friction between marketing and development. The marketing view being that if the customer does not see a bug, then the system is not broken and should not be fixed.

To achieve stability, it needs to be built into the development process. This is not easy to do because of the following problems:

1. it is difficult to determine and control the stability of a system,
2. it is unknown how to prove when a system is stable because stability is not well defined,
3. stability is very closely tied to reliability but the relationship is not clear,
4. designing stability in make the front of development process slower with a hoped for speed up at the end (e.g. later practical demonstrations disturbs management),
5. The tradeoff between make things work and maintaining the status quo is not always clear.

Some of these problems are touched upon in the next section, but QD is very new and therefore not as clearly described or defined.

8.2.2. Implementation

QD tests the system in a "white box" manner. It purposefully attacks weaknesses in the system to improve the robustness (quality) of the system. QD stresses internal interfaces and support features that are not directly accessible to the customer. This means that QD uses the Development Specifications rather than the External Design that System Test uses. In some cases this is similar to the "classiscal view" of integration testing.

QD is very prominent in the MR review process mediating concerns and disputes about system stability. MRs are reviewed by people from many areas to best determine their importance and impact, but QD has the final say on which MRs are included in a system.

Although it occasionally appears that QD has its fingers in everything from development through delivery, its primary function is to maintain system stability. System stability is ensuring that a system does not change too rapidly and that it does not have news bugs appearing where there were no bugs previously. (In other words, QD makes sure that the fixes do not break other things.) In order to ensure stability, one often builds and maintains separate releases for individual or small groups of customers. This can put a large amount of stress on the configuration management system, the MR system, and the Integrator.

During a product's lifetime, the emphasis on stability changes. In the beginning, almost any fix is accepted because the functionality of the system is still being completed. In the middle, only bug fixes that are believed or can be proven not to break anything else are accepted. Finally, near the product's end, only those fixes guaranteed not to break anything else are accepted. Unfortunately, much of this guarantee is still based on the intuition of Development and QD. Occasionally, System Test will find problems.

9. System Test

System Test is the group that acts as the user's advocate to ensure that the system the developers produce meets it specification. System Test tests the system in a "black box" manner to keep the user's view. System Test involves both hardware and software.

System Test begins with the development of the project's Technical Prospectus and External Design by reviewing them for completeness and testability. System Test tries to keep Development honest by simulating the user so that these documents are not ambiguous. Once the TP is finished, System Test begins its development process with the creation of the System Test Plan (STP). The STP includes detailed tests of each feature under normal and exceptional conditions over a variety of system configurations. The STP also includes a description of the tools, peripherals, and simulations to be used to test the system.

Some stress between System Test and Development results from a couple inherent problems. One problem is that System Test provides negative feedback (error reports) to Development that is not always appreciated. In order to reduce the stress that occurs, System Test interacts with Development to make sure that they are doing things correctly. One such interaction is Development's review of the System Test Plan to ensure its accuracy and completeness. Another problem occurs if Development slips its schedule. In this case, System Test is often put under pressure to finish in less than the scheduled amount of time so that the product's delivery date does not slip.

Often as System Test develops its tests, a pre-release is received from Development that allows ST to test their testing tools and generally see how the system works. This is especially important in the set up of new hardware. It should also be noted that System Test is a large project and has a lot of things to manage. Therefore, System Test has everything under some form of version control including both hardware and software. This enables System Test to more effectively determine the location of the bug by component and release. Sometimes System Test also gets advice from Field Support people (customer engineers) to improve its model of the system's expected users.

Ideally, when it is time for the new system to under go system test, System Test receives a final release of the software. In practice, however, this is almost impossible. Instead System Test receives a "smear" of releases. There are several problems with this including:

1. Not all tests can be run on time due to missing functionality (i.e. development is behind schedule),
2. Many (most) tests are run several (many) times on each revised release in the smear to verify that fixes have not broken things that previously worked.

One way that AT&T has helped solve the regression testing and user simulation problems in the observed products was by developing an automated testing system called GAMUT [LaSS] that includes both hardware and software. The GAMUT system uses computers and special hardware to automatically and reproducibly execute tests that simultaneously simulate many users of a telephone switch.

During System Test, MRs are used to reports all bugs found. In fact, there is a lot of MR "cycling" where Mrs are filed by System Test, reponded to by Development, and returned to System Test for re-testing. In some cases MRs acquire "child" MRs which are used to indicate problems in related sub-systems or areas.

When a system passes System Test, not only is the tested code passed on, but a Factory Release Document is also produced. It contains information about:

1. the MRs fixed and what they affected,
2. special procedures or workaround patches,
3. special installation instructions, if needed,
4. compatibility with previous hardware and software, and
5. controlled introduction history.

These items describe the major changes that the Field Support and manufacturing people need to know about when producing and servicing the new product.

10. Field Support

Field Support is the section of the development process responsible for installation and maintenance of a system. Field support has three substages called controlled introduction, scheduled availability, and general availability. Field support interacts with Development, System Test, Manufacturing, and Marketing. This section will concentrate on the Controlled Introduction stage because it is most closely tied to system development and the other (last) two stages are primarily expansion of production levels to accommodate full scale marketing.

10.1. Controlled Introduction

Controlled Introduction is like a beta test in the "classical" model of the software life cycle and maps into the Qualification part of the AT&T life cycle in Figure 1. Controlled Introduction includes:

1. standard product for customer environments,
2. final technical evaluation of a new product,
3. evaluation of product preformance,
4. customer acceptance criteria,
5. identify necessary improvements, and
6. validate product support processes.

To accomplish these tasks, CI has several subprocesses: requirements and schedules, customer selection, implementation planning, customer briefings, product order and delivery, field support, and customer evaluation.

Requirements and schedules are done in conjunction with Development, Project Mangement, Marketing, and Manufacturing. It is here that the number and kinds of customer sites is chosen and the schedules for testing various product feature is determined.

Customer selection is done based on the test requirements, customer needs and cooperation (want friendly customers), and geographical location. Usually, controlled introduction is performed close to the development area because it may be necessary to get development to examine and fix problems.

Implementation planning is part of the global project management work of developing CI milestones, project/customer commitments, and making sure that both are met.

Customer briefings are used to form a partnership with the customer so that CI can be as pleasant and experience as possible for the customer. These briefings include discussions of the customers detailed needs and schedules for when and how the system can be installed into the customers environment.

Product order and delivery is all the manufacturing processes necessary to build the customer's system. This includes order processing, system customization, and quality control tests at the manufacturing site. During CI, the various processes used for manufacturing are tested and reviewed.

Field support is complete customer installation including all components, wiring, administration, and installation tests. It also includes problem identification, problem resolution, and customer traffic analysis and system analyses.

Finally, *customer evaluation* surveys and interviews the customers about the systems features, operation, performance, and documentation including user and administrator opinions. Statistical analysis of the responsed is used to understand the strengths and weaknesses of the system and its support.

10.2. Scheduled and General Availability

Scheduled availability is an intermediate time period when Project Management, Manufacturing, and National Product Scheduling carefully monitor and expand production facilities to handle the expected market load.

General availability is the point at which the product is available simply by ordering it and having it delivered in an expected time interval. The product, its documentation and support processes are complete and most services are provided following a standard procedure.

10.3. Tool Comments

Field Support currently has very few tools. There are tools for factory orders and some manufacturing. There are also tools for processing some of the system performance statistics. But, there are few tools for the scheduling and coordination of Field Support. Some of the difficulties are:

1. the large number of people that Field Support depends upon,
2. these people are scattered all over the U.S. and possibly the world,
3. there are so many changes that they occur almost continuously.

In other words, some large distributed system would be necessary and it is not easy to define exactly how is should interact with the Field Support people.

11. Summary

This paper has been a summary of the AT&T software development life cycle. It is seen that basically a "waterfall" model is used. Each process in the life cycle has specific inputs and outputs, usually documents or code. The life cycle differs from the waterfall model in some respects by having separation between specification of customer features and internal architecture. Many groups have influence over more than one process in the development of a process to provide checks and balances which promote a more coherent and easy to understand product structure.

We have seen the that the organizational structure of AT&T is primarily by project, except for some management and marketing type functions. It appears to work fairly well in keeping the technical expertise focused on one entity while allowing management and marketing enough dissociation to make fair decisions about the direction and coordination of a project.

While touring the life cycle, we have discussed the products of each process and how they are produced. We have also looked at some of the tools in use to help automate these processes. Even though we only saw brief overviews, some strengths and weaknesses were described for most of these tools. Also, some suggestions for capabilities of future tools were mentioned.

This observation has provided invaluable experience for research into automation of software development from the development of program fragments through global tracking and high level management. It has given the author much to think about.

12. Acknowledgements

The author thanks all those with whom he spoke at AT&T for their time and cooperation. He disrupted many of their plans on short notice. Extra thanks go to G. Yates for making the arrangements (and scheduling those disruptions) at AT&T. The observation was overseen by R. Jantz, who is the AT&T contact person for the University of Illinois, and Professors R. Campbell and J. Liu, who are the principal investigators of the AT&T sponsored research projects that made this summary possible.

13. References

Some of the material for this paper is also included in **AT&T Technical Journal**, Vol. 64 No. 1 Part 2, January 1985. Below are listed specific articles from this issue.

[KePW] Kennedy, T. S., D. A. Pezzutti, and T. L. Wang. *Project Development Environment.*

[LaSS] Lake, C. J., J.J. Shanley, and S. M. Silverstein. *GAMUT: A Message Utility System for Automatic Testing.*

[McFM] McFarland, M. A. and J. A. Miller. *Introduction Activities and Results.*

[PeRS] Pedersen, T. J., J.E. Ritacco, and J. A. Santillo. *Software Development Tools.*

Project Scheduling Software

User's Manual

Laurie Emmons

Tammy Foster

Amy VanVoorhis

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois

CS 327 Software Engineering Project

10/10 3

PROJECT SCHEDULING SOFTWARE

USER'S MANUAL

TEAM 30

LAURIE EMMONS
TAMMY FOSTER
AMY VAN VOORHIS

TABLE OF CONTENTS

Section

SECTION I

PRODUCT OVERVIEW AND RATIONALE

INTRODUCTION

Product Overview and Rationale

Our project planning software automates the COCOMO cost estimation model. The software will also provide a project phase and milestone analyzer which outputs estimates of dates for project phases and due dates for appropriate milestones.

The use of this project planning software will reduce the errors which are made when these cost estimates are calculated manually. Also, the automation of the planning techniques will encourage project managers to use the techniques.

SECTION II

USING THE PROJECT SCHEDULING SOFTWARE

LOGGING_ON

The user should begin by starting up the S9000. The machine ould already be on and all the user needs to do is turn the brightness control knob to the right if it isn't already turned up.

The user first must log on to the machine. This is done by typing in the login the user has been assigned. Presumably, this is the same as that which has been given to the students of CS327 which is "cs327". The user should respond to the login prompt with the following:

```
cs327 <cr>    cr = carriage return
```

The user should wait for the system prompt which has the following format before typing anything else:

```
cs327<xx>#:
```

where xx is the command line number.

The permissions for the present directory should also be set to read/write/execute for all users. This is done by first moving to the directory above the present one:

```
cd .
```

Then the command

```
chmod 777 <directory name>
```

is issued. Finally, the user needs to move back to this directory using the command

```
cd <directory name>
```

LOADING PROJECT SCHEDULING SOFTWARE

Before going on the user needs to load the project scheduling software. The following steps should be followed:

1) Insert the disk containing the project scheduling software into disk drive. The disk is inserted with the label facing left, the edge with the write protect notch going in first. The notch will be in the lower half.

2) Then the following command should be entered:

```
tar xvfn /dev/rfdA ps
```

where A is the number of the disk drive in which the disk containing the project scheduling software was inserted. The drive on the left is 0, the other is drive 1.

Once the program is loaded the user need only type:

```
ps
```

to enter the project scheduling software. At this point the software will take control and the user will be prompted for any other input.

The user can stop the software from running at any time by hitting <ctrl>-c. However, hitting <ctrl>-c while the software is saving the project values may cause errors in the output file. After exiting the software the user should save any work done on their disk by following procedures listed under Saving Work and Logging Out.

NORMAL RUN

A normal run consists of:

1) Logging On

2) Entering Data

3) Reviewing Results

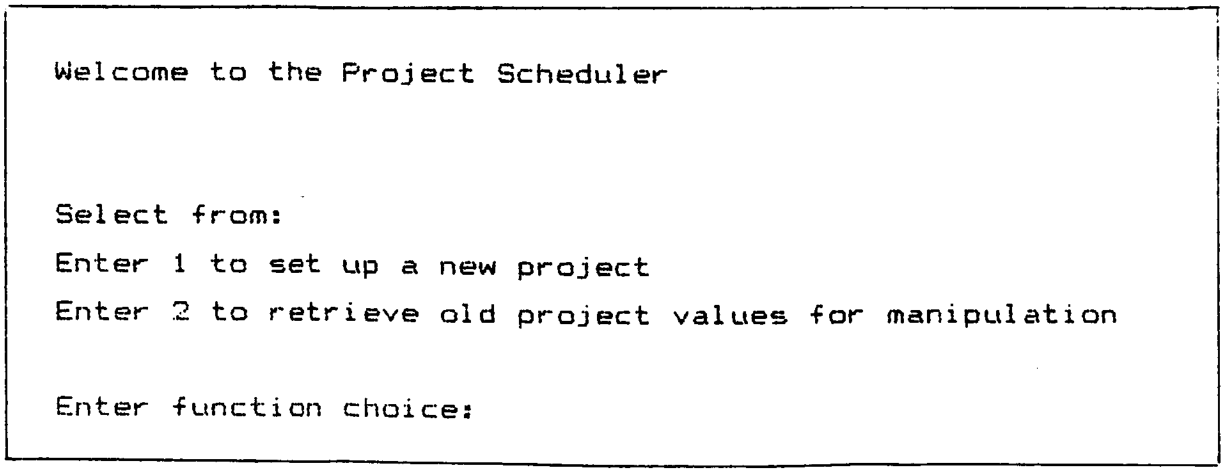
4) Saving Work and Logging Out

LOGGING ON

Logging on has been previously described in Section I. The user should follow steps up to and including the invoking of the software.

ENTERING DATA

The first item the user is prompted for is the function which the user wants to perform. Figure 1 illustrates the menu. For a normal run the user will choose option 1 -- to set up a new project. At any time the user can hit the backspace key when responding to a prompt.

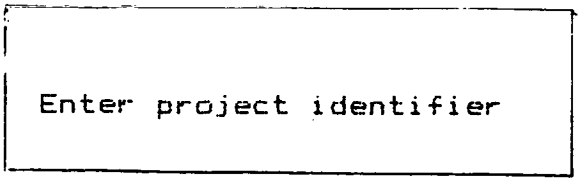


```
Welcome to the Project Scheduler



Select from:
Enter 1 to set up a new project
Enter 2 to retrieve old project values for manipulation

Enter function choice:
```

Figure 1

Also, each response to a prompt should be followed by a carriage return. NOTE: The system will not continue until the carriage return is typed.

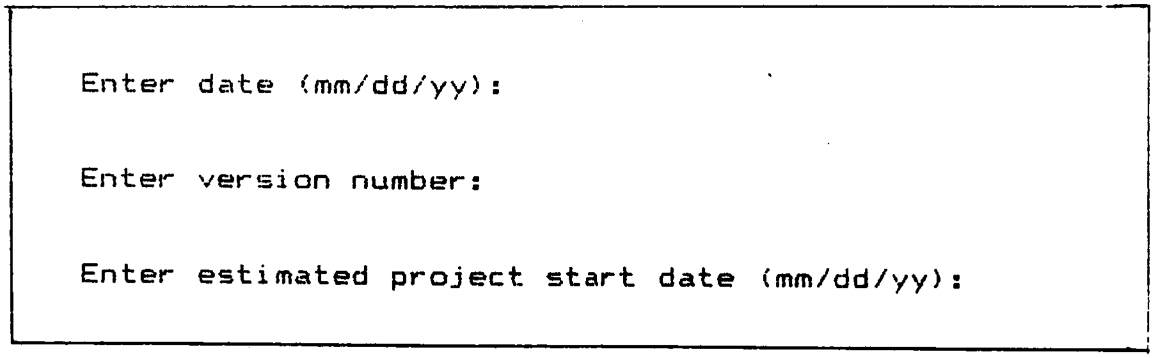
Once this option is chosen, the user will first be prompted for a oject identifier as shown in figure 2. The project identifer may consist of any combination of characters up to a total of 9.



```
Enter project identifier
```

Figure 2

Next the user will be prompted for the present date. The user's response should be in the form of mm/dd/yy. NOTE: The system will not continue until two slashes and a return have been entered. On the same screen, the user will next be prompted for the version number -- which can be either of type real or integer. Finally, on the same screen (figure 3), the user will be prompted for the project start date. The response again should be in the form of mm/dd/yy. Also, the software will not continue until two slashes and a carriage turn have been typed in.

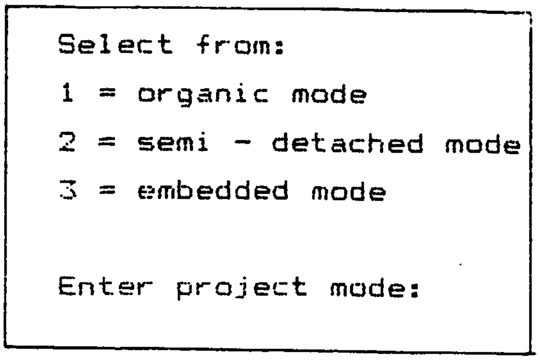


```
Enter date (mm/dd/yy):

Enter version number:

Enter estimated project start date (mm/dd/yy):
```

Figure 3

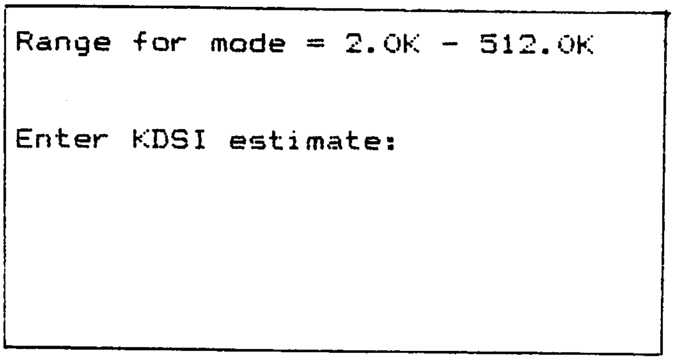
The system will then prompt for the project mode (figure 4). A 1, 2 or 3 is entered depending on the mode of the project which is either organic, semi-detached or embedded, respectively.

Next, the system prompts for the estimate of KDSI. The range in which the response must fall is also presented (figure 5). If a value is entered which does not fall within the range specified, the system will prompt again until a valid value is entered. NOTE: The user may enter a number without a fractional part.



```
Select from:
1 = organic mode
2 = semi - detached mode
3 = embedded mode

Enter project mode:
```

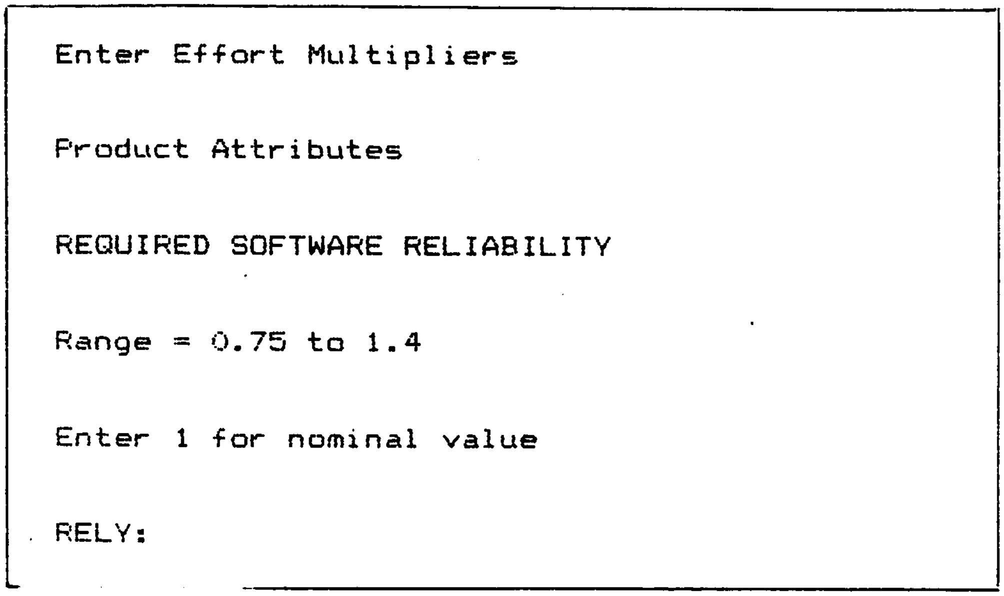
Figure 4



```
Range for mode = 2.0K - 512.0K

Enter KDSI estimate:
```

Figure 5

The last input items which must be entered are the effort multipliers. Each multiplier will be prompted for separately. An example of the format of the screen for all multipliers is shown in figure 6. An example of each screen is shown in Section VIII.

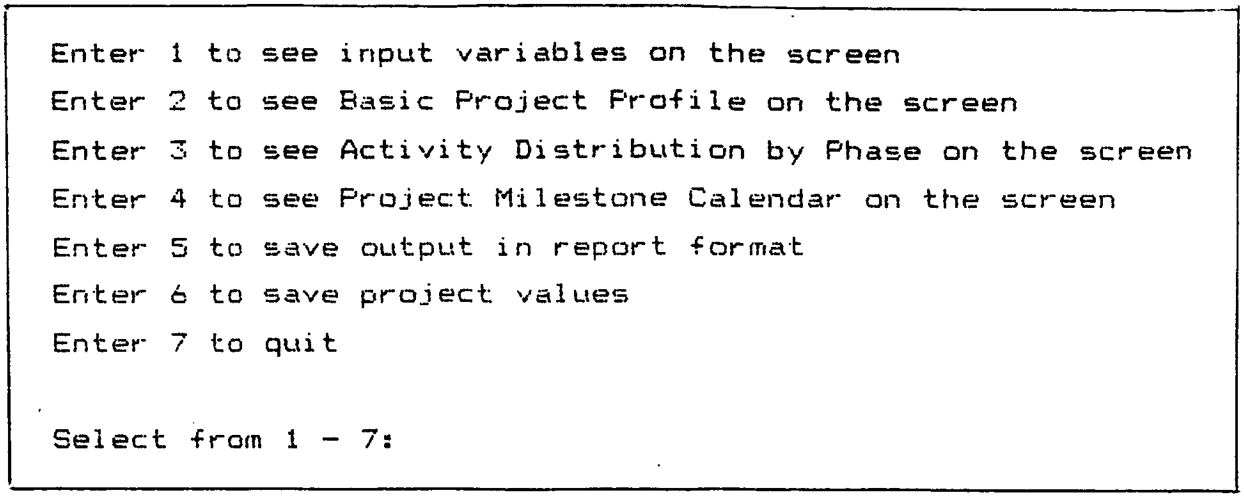


```
Enter Effort Multipliers

Product Attributes

REQUIRED SOFTWARE RELIABILITY

Range = 0.75 to 1.4

Enter 1 for nominal value

RELY:
```

Figure 6

If an incorrect value is entered, the system will print an error message and prompt again. NOTE: If an incorrect value is entered three times, the user will be exited from the software to the operating system and all values previously typed in will be lost.

After the last effort multiplier has been entered, the system will automatically begin calculating results and when finished the review results menu (figure 7) will appear on the screen.



```
Enter 1 to see input variables on the screen
Enter 2 to see Basic Project Profile on the screen
Enter 3 to see Activity Distribution by Phase on the screen
Enter 4 to see Project Milestone Calendar on the screen
Enter 5 to save output in report format
Enter 6 to save project values
Enter 7 to quit

Select from 1 - 7:
```

Figure 7

REVIEWING_RESULTS

The review results menu as shown in figure 7 allows the user to review the results calculated by the project scheduler, the user need only enter the menu choice he desires.

Options 1 - 4 allow the user to review the reports described in ction IV on the screen. The user need only type in the desired option followed by a return. When the user is done examining the report he has chosen, he types a carriage return to return to the review results menu. Options 5, 6 and 7 are described in the next section - Saving Work and Logging Out.

SAVING_WORK_AND_LOGGING_OUT

Option 5 of the review results menu should be chosen if the user wants to print the reports on paper or save them on his disk. This option stores the reports so that when the user leaves the project scheduling software, he can use the lpr command to print the output file. The following steps should be taken to do this:

GETTING PRINTOUT OF REPORTS

1) Choose option 5 from the review results menu.

2) Enter filename. The system will prompt for the filename and user must respond to continue.

3) Review results menu will appear on the screen again and option 7 should be chosen, if the user is ready to end this session.

4) At system level type

```
lpr fname
```

where fname is the filename previously entered within the project scheduler.

Option 6 should be chosen from the review results menu, if the user wants to save the input values for later review and/or use for running the project scheduler at a later time. Once option 6 is chosen, the system will prompt for the filename. After entering the filename the review results menu will appear again on the screen.

If the files created by the project scheduler are to be saved for future use the user must enter:

```
tar uvfn /dev/rfdA   fname  ...
```

where A is the drive which the disk to write to resides and fname is the file to be saved on disk.

To logout the user need only type

```
logout
```

NOTE: Be sure to save work on disk before typing logout or all work will be lost.

ERROR_MESSAGES

The only real error messages the user will get is while entering input. The following is the type of input and what an error would mean that occurred while entering this type of input.

INPUT	ERROR_MEANING
Main function menu choice	Choice not equal to 1 or 2
Project mode	Mode entered not equal to 1, 2 or 3
KDSI estimate	Estimate entered not within range specified
Effort multipliers	Value entered not within range specified

If for some reason another type of error occurs, the user should pe <ctrl>-c. This will take the user back to the operating system and the user can type

```
ps
```

to begin the project scheduler again. Possible causes for error:

1) File containing old project values is incomplete.

2) An invalid response was made to a prompt.

RESPONSE_FORMATS

The following is a list of the input for the project scheduler d the expected format of the user's response:

INPUT	EXPECTED FORMAT
Main function menu choice	Integer 1 or 2
Project identifier	Character string of length < 9
Date and start date	General form = mm/dd/yy Integer followed by a slash followed by another integer, slash and a final integer
Version number	Real or integer number
Project mode	Integer 1, 2 or 3
KDSI estimate	Integer or real vàlue
Effort multipliers	Integer or real value
Filename	Character string of length < 19
Review results menu choice	Integer 1, 2, 3, 4, 5, 6 or 7

Notes about responding to prompts:

1) Backspace may be used when entering response.

2) A carriage return must follow every response.

3) When consecutive carriage returns are entered, no values will be given to these variables, so eventually the program will crash.

4) If more than one character is typed for the mode or a menu choice, the program will crash.

5) If invalid data is entered such as a character for an integer, it is converted to its integer equivalent.

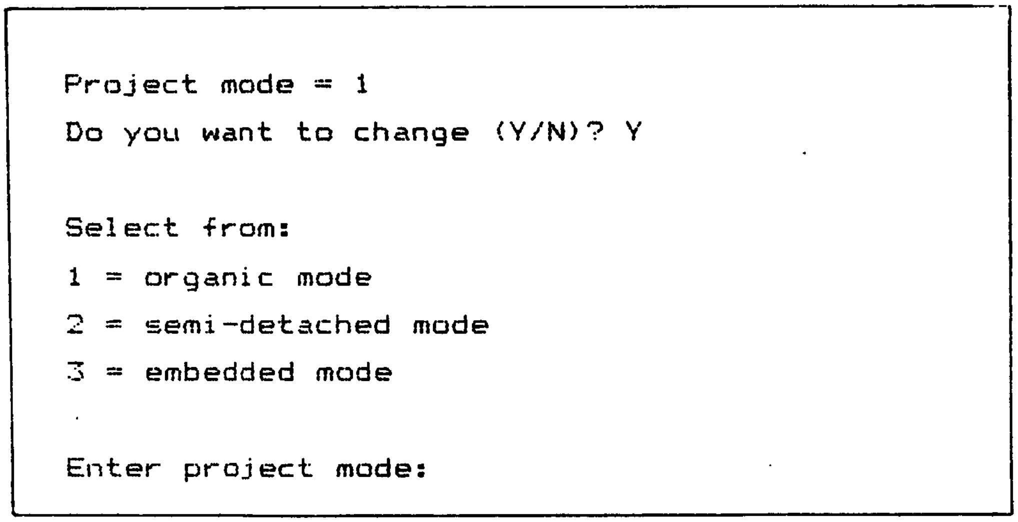
REVIEWING PROJECT DATA

Reviewing project data is very similar to a normal run except for two things:

1) The user is prompted for the filename which the project values are stored.

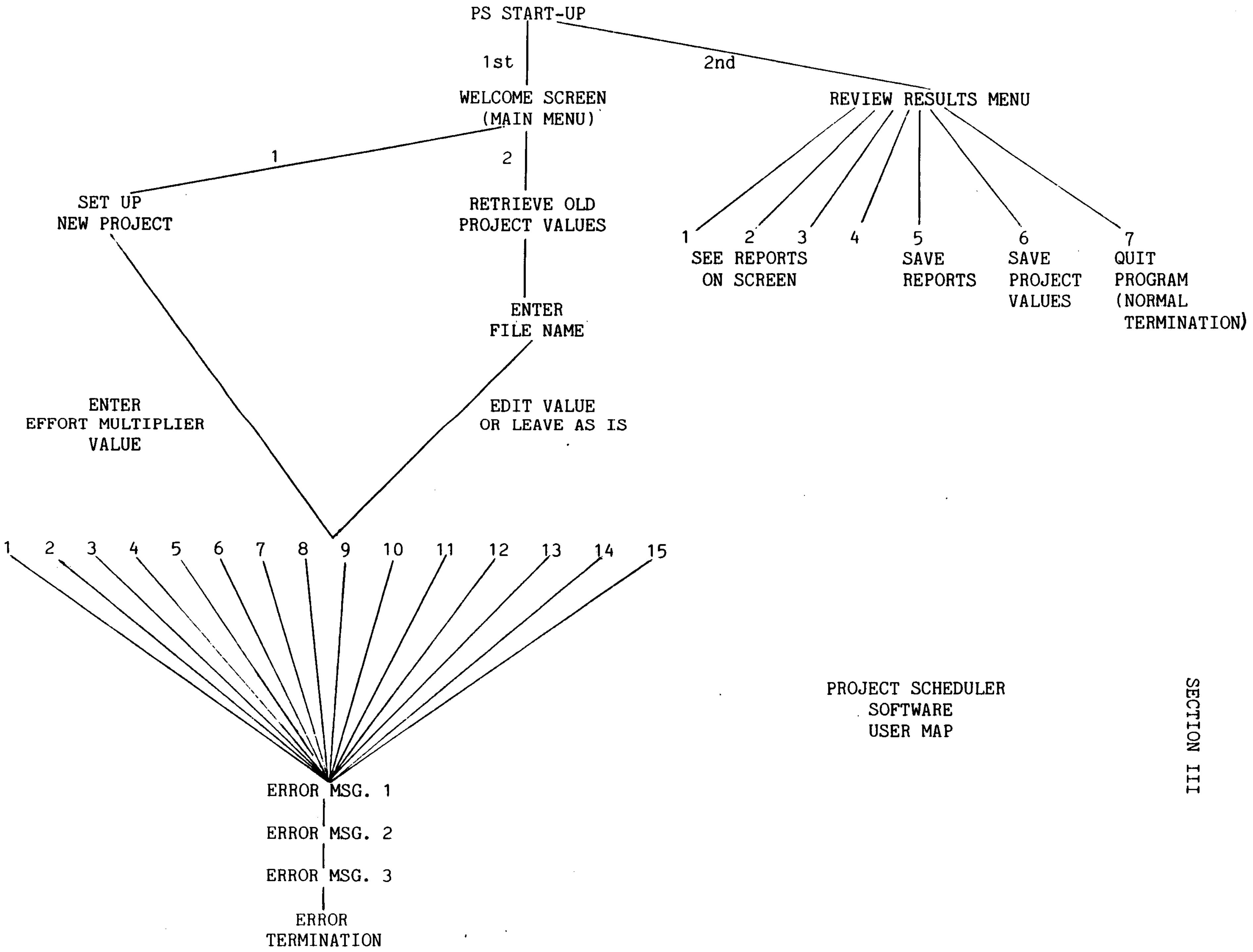
2) The user is prompted as to whether he wants change the listed value.

Figure 8 shows an example of what the screen looks like when the option to change is made for the mode. NOTE: To change, the user must enter a capital Y at the prompt. Anything else typed in is considered a no.



```
Project mode = 1
Do you want to change (Y/N)? Y

Select from:
1 = organic mode
2 = semi-detached mode
3 = embedded mode

Enter project mode:
```

Figure 8



PROJECT SCHEDULER
SOFTWARE
USER MAP

SECTION III

SECTION IV

REPORT FORMATS

Report #1

DESCRIPTIVE PROJECT VALUES

PROJECT IDENTIFIER: DATE

VERSION NUMBER:

ESTIMATED PROJECT STARTING DATE:

ESTIMATED PROJECT LENGTH (WEEKS):

PROJECT MODE:

KDSI ESTIMATE:

Product Attributes	Computer Attributes	Personnel Attributes	Project Attributes
RELY:	TIME:	ACAP:	MODP:
DATA:	STOR:	AEXP:	TOOL:
CPLX:	VIRT:	PCAP:	SCED:
	TURN:	VEXP:	
		LEXP:	

MM:

TDEV:

PRODUCTIVITY:

PROJECT AVERAGE FSP:

Report #2

BASIC PROJECT PROFILE

```
                QUANTITY                    MODE =
-------------------------------------------------------------------------
Total effort (MM)
    Plans and requirements
  Product design
  Programming
    Detailed design
    Code and unit test
  Integration and test
Total schedule (months)
    Plans and requirements
  Product design
  Programming
  Integration and test
Average personnel (FSP)
    Plans and requirements
  Product design
  Programming
  Integration and test
Productivity (DSI/MM)
  Code and unit test only (DSI/MM)
```

Report #3

ACTIVITY DISTRIBUTION BY PHASE

	Phase							
Activity	Plans and Requirements		Product Design		Programming		Integration and Test	
	Percent	FSP	Percent	FSP	Percent	FSP	Percent	FSP
Requirements Analysis								
Product Design								
Programming								
Test Planning								
Verification and Validation								
Project Office								
CM/QA								
Manuals								
Total								

Report #4

PROJECT MILESTONE CALENDAR

PROJECT IDENTIFIER:
VERSION NUMBER:
DATE:

ESTIMATED PROJECT STARTING DATE:
ESTIMATED PROJECT LENGTH (WEEKS):

REVIEW	WORK PRODUCTS REVIEWED	WEEK #
PRODUCT FEASIBILITY REVIEW	SYSTEM DEFINITION	
	PROJECT PLAN	
SOFTWARE REQUIREMENTS REVIEW	SOFTWARE REQUIREMENTS SPECIFICATION	
	PRELIMINARY USER'S MANUAL	
	PRELIMINARY VERIFICATION PLAN	
PRELIMINARY DESIGN REVIEW	ARCHITECTURAL DESIGN DOCUMENT	
CRITICAL DESIGN REVIEW	DETAILED DESIGN SPECIFICATION	
	USER'S MANUAL	
	SOFTWARE VERIFICATION PLAN	
SOURCE CODE REVIEW	WALKTHROUGHS & INSPECTIONS OF SOURCE CODE	
ACCEPTANCE TEST REVIEW	ACCEPTANCE TEST PLAN	
PRODUCT RELEASE REVIEW	ALL OF ABOVE DOCUMENTS	
PROJECT POST-MORTEM	PROJECT LEGACY	

SECTION V

TECHNICAL SPECIFICATIONS

Packaging Specifications

The Project Scheduling Software consists of four files of source code.

compute.c - This serves as the main section of code. It co-ordinates the calling of all of the other functions. It also contains the code to compute the MM, TDEV, Productivity, Project length, Project Average FSP, FSP distribution, and Milestones.

tablemanip.c - This file of code co-ordinates the accessing of the reference table values and computes the Effort, Schedule, and Activity distribution output values.

in.c - This file contains the input and output code. The input code co-ordinates the interactive input and fills the globals input program values. It is also in this code that all reading from old project files or writing to files (for storage of project values or output reports) is managed. The output portion of this code co-ordinates the formatting of the output reports.

defs.h - This file contains the declarations for all of the global variables for the software.

The Project Scheduling Software is compiled into compute.o, tablemanip.o, and in.o using the following command:

```
cc -o ps compute.c tablemanip.c in.c -lm
```

The software is then executed by typing: ps

PROJECT SCHEDULING SOFTWARE - COMPUTATIONAL SPECIFICATIONS

C-7

Calculations used in the Project Scheduling software (PSS):

The PSS implements the intermediate COCOMO model as described in Software Engineering Economics by Barry Boehm (1981). The following is a description of the equations and procedures used.

1) Intermediate COCOMO Nominal Effort Estimating Equations:

Development Mode	Nominal Effort Equation
Organic	(MM)nom = 3.2(KDSI)**1.05
Semi-detached	(MM)nom = 3.0(KDSI)**1.12
Embedded	(MM)nom = 2.8(KDSI)**1.20

2) MM = MM(nom) * (product of the 15 effort multipliers) [rounded]

3) Basic and Intermediate COCOMO Schedule Estimating Equations:

Development Mode	Schedule Equation
Organic	TDEV = 2.5(MM)**0.38 [rounded]
Semi-detached	TDEV = 2.5(MM)**0.35 [rounded]
Embedded	TDEV = 2.5(MM)**0.32 [rounded]

4) Productivity = (KDSI * 1000) / MM [rounded]

5) Project Average FSP = MM / TDEV

6) Estimated project length =(time for the planning and requirements phase) + (tdev * 4.33) [conversion of tdev to weeks]

7) Milestones: The week number assignments for each of the project phases are calculated by: 1) converting to weeks (value * 4.33) the corresponding values for the allotment of schedule time for the phase in the schedule distribution and 2) creating a time line for the project by adding each allotment of weeks to the sum of the allotments of weeks for the previous phases. The work products are assigned using the table of Reviews and Milestones in the Phased Life-cycle Model from Fairley (1985) Software Engineering Concepts (page 42). This table is presented in the appendix.

8) Calculation of the Distribution Outputs:

8a) Effort Distribution: Each row in the Effort Distribution output is calculated by multiplying MM and the appropriate percentage value from the Effort Distribution Table. The appropriate percentage value is found by identifying the correct reference table using the mode and KDSI combination, and then selecting the percentage value in the row that corresponds to the activity value currently being calculated. If KDSI is not an exact reference table column value, interpolation is performed to determine the percentage value to be used.

8b) Schedule Distribution: Each row in the Schedule Distribution output is calculated by multiplying TDEV and the appropriate percentage value from the Schedule Distribution Table. The appropriate percentage value is found by identifying the correct reference table using the mode and KDSI combination, and then selecting the percentage value in the row that corresponds to the activity value currently being calculated. If KDSI is not an exact reference table column value, interpolation is performed to determine the percentage value to be used.

8c) Average Personnel (FSP) Distribution: Each row in the Average Personnel (FSP) Distribution is calculated by dividing the corresponding MM value from the Effort Distribution by the corresponding TDEV from the Schedule Distribution.

8d) Activity Distribution by Phase: The Activity Distributions are calculated using the Average Personnel (FSP) Distribution. Each of the four values in the Average Personnel (FSP) Distribution is expanded to show the breakdown in terms of percentage of personnel for that phase on each of the eight project activities that occur in some percentage throughout the whole project. Each Average Personnel (FSP) Distribution value is multiplied by the appropriate percentage value from the Project Activity Distribution by Phase Tables. The appropriate percentage value is found by identifying the correct reference table using the mode and KDSI combination, and then selecting the percentage value in the row that corresponds to the activity value currently being calculated. If KDSI is not an exact reference table column value, interpolation is performed to determine the percentage value to be used.

SECTION VI

SAMPLE RUN

```
                         DESCRIPTIVE PROJECT VALUES

 PROJECT IDENTIFIER: test1                                   DATE:  5/13/86
 VERSION NUMBER: 1
 ESTIMATED PROJECT STARTING DATE: 10/24/86
 ESTIMATED PROJECT LENGTH (WEEKS): 110
 PROJECT MODE: 3
 KDSI ESTIMATE:  80.0

      Product          Computer         Personnel       Project
      Attributes       Attributes       Attributes      Attributes
      RELY: 1.00       TIME: 1.00       ACAP: 1.00      MODP: 1.00
      DATA: 1.00       STOR: 1.00       AEXP: 1.00      TOOL: 1.00
      CPLX: 1.00       VIRT: 1.00       PCAP: 1.00      SCED: 1.00
                       TURN: 1.00       VEXP: 1.00
                                        LEXP: 1.00

 MM:538.0
 TDEV:  19.0
 PRODUCTIVITY: 149
 PROJECT AVERAGE FSP: 28.32
```

```
                        BASIC PROJECT PROFILE

              QUANTITY                      MODE = Embedded
------------------------------------------------------------------------------
Total effort (MM)                          538.00
    Plans and requirements                    8.00%        43.04
  Product design                           18.00%        96.84
  Programming                              52.50%       282.45
    Detailed design                          25.50%       137.19
    Code and unit test                       27.00%       145.26
  Integration and test                     29.50%       158.71
Total schedule (months)                    19.00
    Plans and requirements                   34.00%          6.46
  Product design                           35.00%         6.65
  Programming                              38.00%         7.22
  Integration and test                     27.00%         5.13
Average personnel (FSP)                    28.32
    Plans and requirements                   23.53%          6.66
  Product design                           51.43%        14.56
  Programming                             138.16%        39.12
  Integration and test                    109.26%        30.94
Productivity (DSI/MM)                        149
  Code and unit test only (DSI/MM)           551
```

ACTIVITY DISTRIBUTION BY PHASE

Activity	Plans and Requirements		Product Design		Programming		Integration and Test	
	Percent	FSP	Percent	FSP	Percent	FSP	Percent	FSP
Requirements Analysis	45.00	3.00	10.00	1.46	3.00	1.17	2.00	0.62
Product Design	14.50	0.97	42.00	6.12	6.00	2.35	4.00	1.24
Programming	7.00	0.47	12.50	1.82	55.00	21.52	42.00	12.99
Test Planning	4.50	0.30	6.50	0.95	6.50	2.54	4.00	1.24
Verification and Validation	8.50	0.57	8.50	1.24	10.50	4.11	24.00	7.43
Project Office	11.00	0.73	10.00	1.46	6.50	2.54	7.50	2.32
CM/QA	4.00	0.27	3.00	0.44	7.00	2.74	9.00	2.78
Manuals	5.50	0.37	7.50	1.09	5.50	2.15	7.50	2.32
Total	100	6.66	100	14.56	100	39.12	100	30.94

PROJECT MILESTONE CALENDAR

PROJECT IDENTIFIER: test1
VERSION NUMBER: 1
DATE: 5/13/86

ESTIMATED PROJECT STARTING DATE: 10/24/86
ESTIMATED PROJECT LENGTH (WEEKS): 110

REVIEW	WORK PRODUCTS REVIEWED	WEEK #
PRODUCT FEASIBILITY REVIEW	SYSTEM DEFINITION	14
	PROJECT PLAN	
SOFTWARE REQUIREMENTS REVIEW	SOFTWARE REQUIREMENTS SPECIFICATION	28
	PRELIMINARY USER'S MANUAL	
	PRELIMINARY VERIFICATION PLAN	
PRELIMINARY DESIGN REVIEW	ARCHITECTURAL DESIGN DOCUMENT	42
CRITICAL DESIGN REVIEW	DETAILED DESIGN SPECIFICATION	57
	USER'S MANUAL	
	SOFTWARE VERIFICATION PLAN	
SOURCE CODE REVIEW	WALKTHROUGHS & INSPECTIONS OF SOURCE CODE	73
ACCEPTANCE TEST REVIEW	ACCEPTANCE TEST PLAN	88
PRODUCT RELEASE REVIEW	ALL OF ABOVE DOCUMENTS	110
PROJECT POST-MORTEM	PROJECT LEGACY	110

SECTION VII

GLOSSARY

GLOSSARY

activity distribution by phase - break-down of project average personnel (PAFSP) required to perform each of 8 activities during each phase of the project

average personnel (FSP) distribution - break-down of project average personnel (PAFSP) into number of personnel required to complete each phase of the project

cd - unix command to change directory

chmod - unix command to change read/write/execute permissions

cocomo - constructive cost model of software development cost estimation

descriptive project values - one of the output reports produced by PSS. Shows all the values input by the user, and the basic cocomo values calculated by the program: mm, tdev, productivity, and project average FSP

effort distribution - break-down of total effort (mm) into effort required to complete each phase of the project

effort multiplier - value within a given range which represents the project's rating in terms of one of 15 software cost drivers, such as complexity or programmer ability

embedded mode - software which interacts directly with the hardware; corresponds to systems programming

error message - a message printed on the screen by the PSS program indicating that the user has entered an incorrect value

fsp - full-time software personnel

kdsi - thousand delivered source instructions; the anticipated size of the project in terms of source code instructions

load - the process of copying the PSS program into the computer's main memory in preparation for running the program

log on - the process of identifying yourself to the computer in order to be admitted to the operating system

login - a password which identifies the user and is used to gain access to the operating system; the PSS user should use the login cs327

logout - the command used to exit from the unix operating system after a PSS session, after all work has been saved on disk using tar uvfn command

lpr fname - the unix system command used to obtain a paper copy of the PSS program output; substitute the name of the file in which you saved the output for fname milestone - a significant event such as the completion of a phase or an important review in the life cycle of the software project; expressed as the week number of the project by PSS

mm - the total amount of effort, in man-months, required to complete a software project of a given mode, size and description

mode - the general category to which the user's project belongs: 1 = organic (applications programs); 2 = semi-detached (utility programs); 3 = embedded (systems programs)

organic mode - programs which use an environment provided by a language compiler; corresponds to applications programming

pafsp (project average fsp) - the average number of full-time software personnel needed to staff a project of a given mode, size and description

prod(uctivity) - the number of delivered source instructions per man-month for a project of given mode, size and description

project identifier - any combination of 9 or fewer characters which the user chooses and enters to identify the project which he/she wishes to schedule

ps - the command used to invoke the PSS program to start it running

S9000 - the IBM computer system on which the Project Scheduler Software runs

schedule distribution - break-down of elapsed time (tdev) into time required to complete each phase of the project

semi-detached mode - programs which provide processing environments and sophisticated use of the operating system; corresponds to utility programming

software cost driver - one of 15 factors which strongly influence the cost in terms of time and money to complete a software development project (see Appendix, Software Cost Driver Rating Reference Table I and II)

tar uvfn /dev/rfdA fname(s) - unix system command to save any files created when running the PSS program; substitute the correct drive number (0 or 1) for A, and the name(s) of your file(s) for fname(s)

tar xvfn /dev/rfdA ps - the unix command used to tell the operating system to load the PSS program in preparation for running the program; substitute the correct drive number (0 or 1) for A in the command

tdev - elapsed time, in months, required to complete a software project of a given mode, size and description

unix - the operating system running on the IBM S9000

version number - a real or integer number chosen and entered by the user to identify which version of a project he/she wishes to schedule

SECTION VIII

APPENDIX

INPUT SCREENS

```
Welcome to the Project Scheduler



Select from
Enter 1 to set up a new project
Enter 2 to retrieve old project values for manipulation


Enter function choice:




Enter project identifier:



Enter date (mm/dd/yy): ?



Enter version number:



Enter estimated project start date (mm/dd/yy):



Select from:
1 = organic mode
2 = semi-detached mode
3 = embedded mode

Enter project mode
```

```
Range for mode = [illegible] K - [illegible] K

Enter KDSI estimate.




Enter Effort Multipliers

Product Attributes

REQUIRED SOFTWARE RELIABILITY

Range = 0.750000 to 1.400000

Enter 1 for nominal value

RELY:




Enter Effort Multipliers

Product Attributes

DATA BASE SIZE

Range = 0.940000 to 1.160000

Enter 1 for nominal value

DATA:




Enter Effort Multipliers

Product Attributes

PRODUCT COMPLEXITY

Range = 0.700000 to 1.650000

Enter 1 for nominal value

CPLX:
```

ORIGINAL PAGE IS
OF POOR QUALITY

ORIGINAL PAGE IS
OF POOR QUALITY

```
Enter Effort Multipliers

Computer Attributes

EXECUTION TIME CONSTRAINT

Range = 1.000000 to 1.660000

Enter 1 for nominal value

TIME.




Enter Effort Multipliers

Computer Attributes

MAIN STORAGE CONSTRAINT

Range = 1.000000 to 1.560000

Enter 1 for nominal value

STOR.




Enter Effort Multipliers

Computer Attributes

VIRTUAL MACHINE VOLATILITY

Range = 0.870000 to 1.300000

Enter 1 for nominal value

VIRT.
```

```
Enter Effort Multipliers
Computer Attributes

COMPUTER TURNAROUND TIME
Range = 0.870000 to 1.150000
Enter 1 for nominal value
TURN.



Enter Effort Multipliers
Personnel Attributes

ANALYST CAPABILITY
Range = 0.710000 to 1.460000
Enter 1 for nominal value
ACAP.



Enter Effort Multipliers
Personnel Attributes

APPLICATIONS EXPERIENCE
Range = 0.820000 to 1.290000
Enter 1 for nominal value
AEXP.



Enter Effort Multipliers
Personnel Attributes

PROGRAMMER CAPABILITY
Range = 0.700000 to 1.420000
Enter 1 for nominal value
Enter Effort Multipliers

Personnel Attributes

VIRTUAL MACHINE EXPERIENCE

Range = 0.900000 to 1.210000

Enter 1 for nominal value

VEXP.




Enter Effort Multipliers

Personnel Attributes

PROGRAMMING LANGUAGE EXPERIENCE

Range = 0.950000 to 1.140000

Enter 1 for nominal value

LEXP.




Enter Effort Multipliers

Project Attributes

USE OF MODERN PROGRAMMING PRACTICES

Range = 0.820000 to 1.240000

Enter 1 for nominal value

MODP




Enter Effort Multipliers

Project Attributes

USE OF SOFTWARE TOOLS

Range = 0.830000 to 1.240000

Enter 1 for nominal value
```

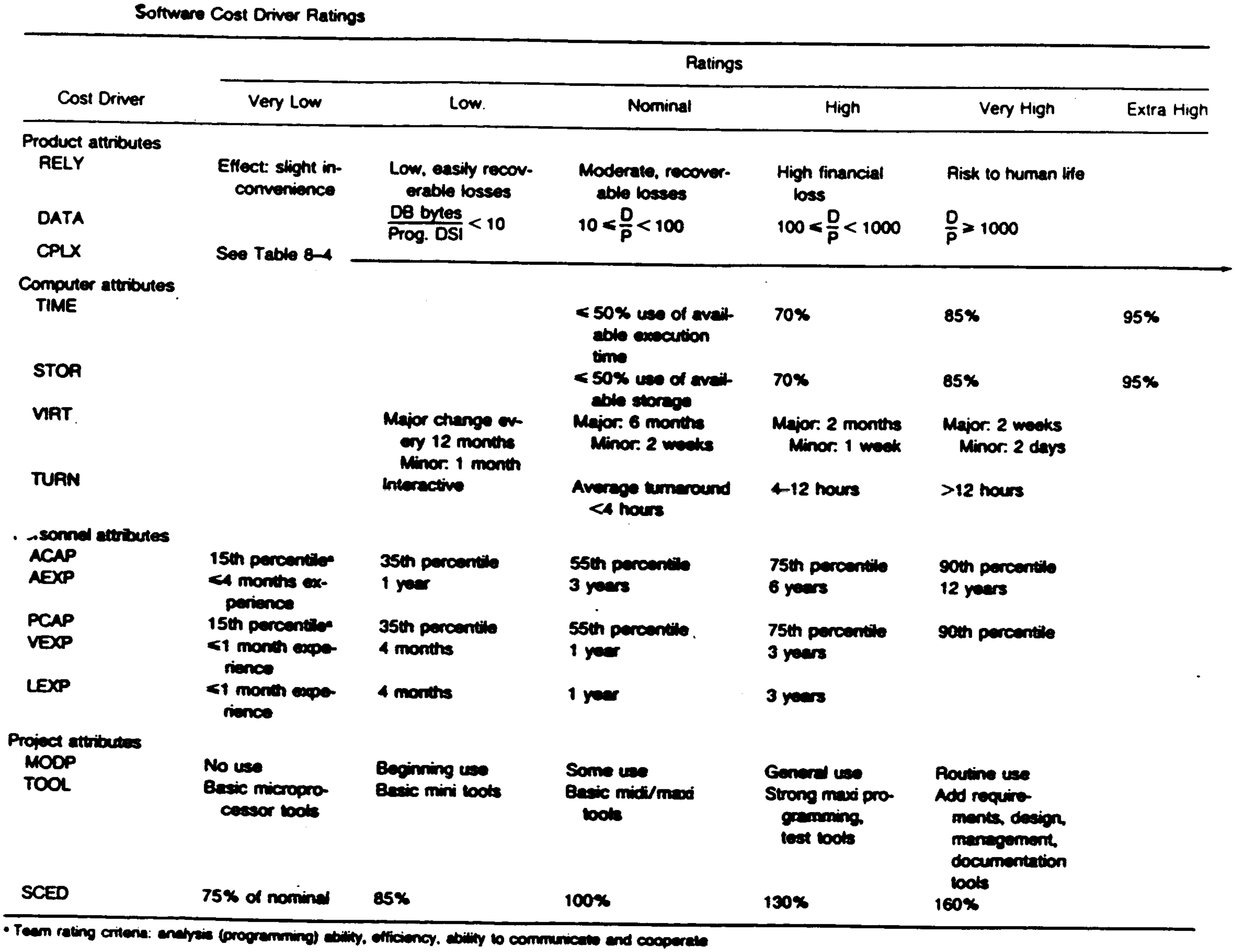
ORIGINAL PAGE IS
OF POOR QUALITY

```
Enter Effort Multipliers

Project Attributes

REQUIRED DEVELOPMENT SCHEDULE

Range = 1.100000 to 1.230000

Enter 1 for nominal value

SCED:
```

Distinguishing Features of Software Development Modes

	Mode		
Feature	Organic = 1	Semidetached = 2	Embedded = 3
Organizational understanding of product objectives	Thorough	Considerable	General
Experience in working with related software systems	Extensive	Considerable	Moderate
Need for software conformance with pre-established requirements	Basic	Considerable	Full
Need for software conformance with external interface specifications	Basic	Considerable	Full
Concurrent development of associated new hardware and operational procedures	Some	Moderate	Extensive
Need for innovative data processing architectures, algorithms	Minimal	Some	Considerable
Premium on early completion	Low	Medium	High
Product size range	<50 KDSI	<300 KDSI	All sizes
Examples	Batch data reduction Scientific models Business models Familiar OS, compiler Simple inventory, production control	Most transaction processing systems New OS, DBMS Ambitious inventory, production control Simple command-control	Large, complex transaction processing systems Ambitious, very large OS Avionics Ambitious command-control

Description of the mode project value. (Boehm, 1981)

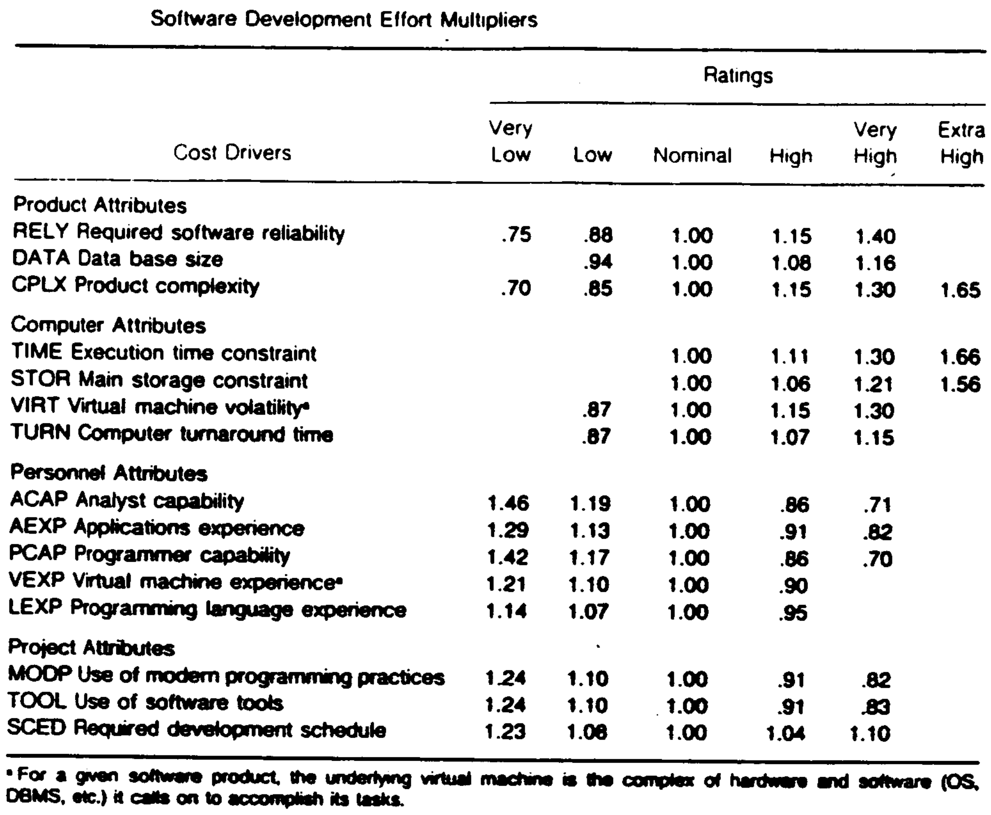


Software Cost Driver Ratings

Cost Driver	Very Low	Low	Nominal	High	Very High	Extra High
	Ratings					
Product attributes						
RELY	Effect: slight inconvenience	Low, easily recoverable losses	Moderate, recoverable losses	High financial loss	Risk to human life	
DATA		$\frac{\text{DB bytes}}{\text{Prog. DSI}} < 10$	$10 \le \frac{D}{P} < 100$	$100 \le \frac{D}{P} < 1000$	$\frac{D}{P} \ge 1000$	
CPLX	See Table 8–4					
Computer attributes						
TIME			≤ 50% use of available execution time	70%	85%	95%
STOR			≤ 50% use of available storage	70%	85%	95%
VIRT		Major change every 12 months Minor: 1 month	Major: 6 months Minor: 2 weeks	Major: 2 months Minor: 1 week	Major: 2 weeks Minor: 2 days	
TURN		Interactive	Average turnaround <4 hours	4–12 hours	>12 hours	
sonnel attributes						
ACAP	15th percentile*	35th percentile	55th percentile	75th percentile	90th percentile	
AEXP	≤4 months experience	1 year	3 years	6 years	12 years	
PCAP	15th percentile*	35th percentile	55th percentile	75th percentile	90th percentile	
VEXP	≤1 month experience	4 months	1 year	3 years		
LEXP	≤1 month experience	4 months	1 year	3 years		
Project attributes						
MODP	No use	Beginning use	Some use	General use	Routine use	
TOOL	Basic microprocessor tools	Basic mini tools	Basic midi/maxi tools	Strong maxi programming, test tools	Add requirements, design, management, documentation tools	
SCED	75% of nominal	85%	100%	130%	160%	

* Team rating criteria: analysis (programming) ability, efficiency, ability to communicate and cooperate

Software Cost Driver Rating Reference Table I.
(Boehm, 1981)



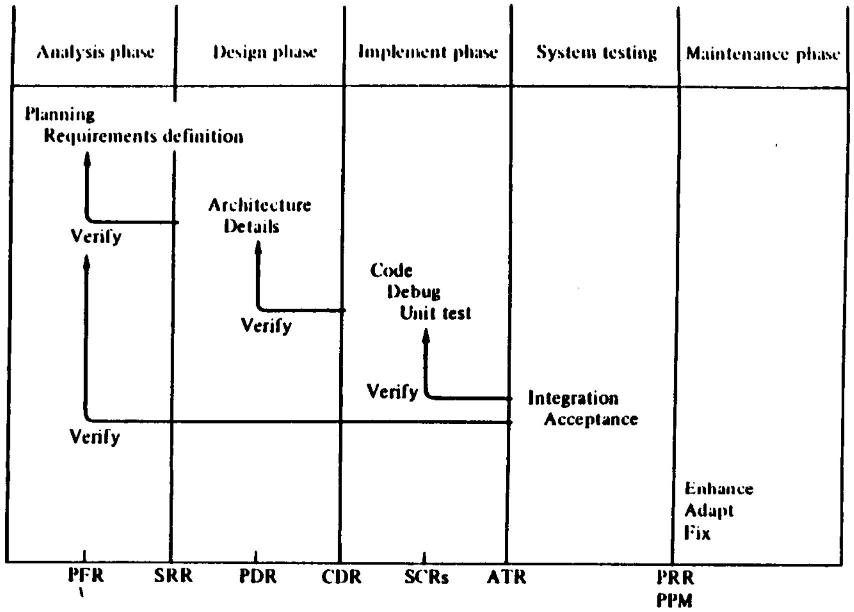
Software Development Effort Multipliers

	Ratings					
Cost Drivers	Very Low	Low	Nominal	High	Very High	Extra High
Product Attributes						
RELY Required software reliability	.75	.88	1.00	1.15	1.40	
DATA Data base size		.94	1.00	1.08	1.16	
CPLX Product complexity	.70	.85	1.00	1.15	1.30	1.65
Computer Attributes						
TIME Execution time constraint			1.00	1.11	1.30	1.66
STOR Main storage constraint			1.00	1.06	1.21	1.56
VIRT Virtual machine volatility•		.87	1.00	1.15	1.30	
TURN Computer turnaround time		.87	1.00	1.07	1.15	
Personnel Attributes						
ACAP Analyst capability	1.46	1.19	1.00	.86	.71	
AEXP Applications experience	1.29	1.13	1.00	.91	.82	
PCAP Programmer capability	1.42	1.17	1.00	.86	.70	
VEXP Virtual machine experience•	1.21	1.10	1.00	.90		
LEXP Programming language experience	1.14	1.07	1.00	.95		
Project Attributes						
MODP Use of modern programming practices	1.24	1.10	1.00	.91	.82	
TOOL Use of software tools	1.24	1.10	1.00	.91	.83	
SCED Required development schedule	1.23	1.08	1.00	1.04	1.10	

• For a given software product, the underlying virtual machine is the complex of hardware and software (OS, DBMS, etc.) it calls on to accomplish its tasks.

Software Cost Driver Rating Reference Table II.
(Boehm, 1981)

ORIGINAL PAGE IS
OF POOR QUALITY



	REVIEW	*WORK PRODUCTS REVIEWED*
PFR:	Product Feasibility Review	System Definition Project Plan
SRR:	Software Requirements Review	Software Requirement Specification preliminary User's Manual preliminary Verification Plan
PDR:	Preliminary Design Review	Architectural Design Document
CDR:	Critical Design Review	Detailed Design Specification User's Manual Software Verification Plan
SCR:	Source Code Review	Walkthroughs & Inspections of the Source Code
ATR:	Acceptance Test Review	Acceptance Test Plan
PRR:	Product Release Review	All of the Above.
PPM:	Project Post-Mortem	Project Legacy

Reviews and milestones in the phased life-cycle model.

Reference data used to assign milestones and workproducts.
(Fairley, 1985)

Phase Distribution of Effort and Schedule: All Modes

		Size				
	Effort distribution					
Mode	Phase	Small 2 KDSI	Inter- mediate 8 KDSI	Medium 32 KDSI	Large 128 KDSI	Very Large 512 KDSI
Organic	Plans and requirements (%)	6	6	6	6	
	Product design	16	16	16	16	
	Programming	68	65	62	59	
	Detailed design	26	25	24	23	
	Code and unit test	42	40	38	36	
	Integration and test	16	19	22	25	
Semidetached	Plans and requirements (%)	7	7	7	7	7
	Product design	17	17	17	17	17
	Programming	64	61	58	55	52
	Detailed design	27	26	25	24	23
	Code and unit test	37	35	33	31	29
	Integration and test	19	22	25	28	31
Embedded	Plans and requirements (%)	8	8	8	8	8
	Product design	18	18	18	18	18
	Programming	60	57	54	51	48
	Detailed design	28	27	26	25	24
	Code and unit test	32	30	28	26	24
	Integration and test	22	25	28	31	34
	Schedule distribution	2 KDSI	8 KDSI	32 KDSI	128 KDSI	512 KDSI
Organic	Plans and requirements (%)	10	11	12	13	
	Product design	19	19	19	19	
	Programming	63	59	55	51	
	Integration and test	18	22	26	30	
Semidetached	Plans and requirements (%)	16	18	20	22	24
	Product design	24	25	26	27	28
	Programming	56	52	48	44	40
	Integration and test	20	23	26	29	32
Embedded	Plans and requirements (%)	24	28	32	36	40
	Product design	30	32	34	36	38
	Programming	48	44	40	36	32
	Integration and test	22	24	26	28	30

Reference tables used to calculate the Effort and Schedule distributions

(Boehm, 1981)

Project Activity Distribution by Phase: Organic Mode

Phase	Plans and Requirements	Product Design	Programming	Integration and Test
Product Size	S I M L	S I M L	S I M L	S I M L
Overall Phase Percentage	6	16	68 65 62 59	16 19 22 25
Activity percentage				
Requirements analysis	46	15	5	3
Product design	20	40	10	6
Programming	3	14	58	34
Test planning	3	5	4	2
Verification and validation	6	6	6	34
Project office	15	11	6	7
CM/QA	2	2	6	7
Manuals	5	7	5	7

Project Activity Distribution by Phase: Semidetached Mode

Phase	Plans and Requirements					Product Design					Programming					Integration and Test				
Product Size	S	I	M	L	VL	S	I	M	L	VL	S	I	M	L	VL	S	I	M	L	VL
Overall Phase Percentage	7	7	7	7	7	17	17	17	17	17	64	61	58	55	52	19	22	25	28	31
Activity percentage																				
Requirements analysis	48	47	46	45	44	12.5	12.5	12.5	12.5	12.5	4	4	4	4	4	2.5	2.5	2.5	2.5	2.5
Product design	16	16.5	17	17.5	18	41	41	41	41	41	8	8	8	8	8	5	5	5	5	5
Programming	2.5	3.5	4.5	5.5	6.5	12	12.5	13	13.5	14	56.5	56.5	56.5	56.5	56.5	33	35	37	39	41
Test planning	2.5	3	3.5	4	4.5	4.5	5	5.5	6	6.5	4	4.5	5	5.5	6	2.5	2.5	3	3	3.5
Verification and validation	6	6.5	7	7.5	8	6	6.5	7	7.5	8	7	7.5	8	8.5	9	32	31	29.5	28.5	27
Project office	15.5	14.5	13.5	12.5	11.5	13	12	11	10	9	7.5	7	6.5	6	5.5	8.5	8	7.5	7	6.5
CM/QA	3.5	3	3	3	2.5	3	2.5	2.5	2.5	2	7	6.5	6.5	6.5	6	8.5	8	8	8	7.5
Manuals	6	6	5.5	5	5	8	8	7.5	7	7	6	6	5.5	5	5	8	8	7.5	7	7

Project Activity Distribution by Phase: Embedded Mode

Phase	Plans and Requirements					Product Design					Programming					Integration and Test				
Product Size	S	I	M	L	VL	S	I	M	L	VL	S	I	M	L	VL	S	I	M	L	VL
Overall Phase Percentage	8	8	8	8	8	18	18	18	18	18	60	57	54	51	48	22	25	28	31	34
Activity percentage																				
Requirements analysis	50	48	46	44	42	10	10	10	10	10	3	3	3	3	3	2	2	2	2	2
Product design	12	13	14	15	16	42	42	42	42	42	6	6	6	6	6	4	4	4	4	4
Programming	2	4	6	8	10	10	11	12	13	14	55	55	55	55	55	32	36	40	44	48
Test planning	2	3	4	5	6	4	5	6	7	8	4	5	6	7	8	3	3	4	4	5
Verification and validation	6	7	8	9	10	6	7	8	9	10	8	9	10	11	12	30	28	25	23	20
Project office	16	14	12	10	8	15	13	11	9	7	9	8	7	6	5	10	9	8	7	6
CM/QA	5	4	4	4	3	4	3	3	3	2	8	7	7	7	6	10	9	9	9	8
Manuals	7	7	6	5	5	9	9	8	7	7	7	7	6	5	5	9	9	8	7	7

Reference tables used to calculate the activity distributions.

(Boehm, 1981)

SECTION IX

INDEX

INDEX