LANGLEY GRANT
CR

IN-81
82672
P 211

1987 MID-YEAR REPORT

NASA Grant NAG 1-138

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

Principal Investigator
Roy H. Campbell

D. Laliberte
H. Render
R. Sum
W. Smith
R. Terwilliger

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.

(NASA-CR-181108) SAGA: A PROJECT TO AUTOMATE THE MANAGEMENT OF SOFTWARE PRODUCTION SYSTEMS Progress Report, Jan. - Jun. 1986 (Illinois Univ.) 211 p Avail: NTIS HC A14/MF A01 CSCL 05A G3/81

N87-27548

Unclas
0082672

1987 Mid–Year Report
NASA Grant NAG 1–138

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

Principal Investigator: R. H. Campbell.

Research Assistants:
D. Laliberte
H. Render
R. Sum
W. Smith
R. Terwilliger

University of Illinois at Urbana–Champaign
Department of Computer Science
1304 West Springfield Avenue
Urbana, Illinois 61801–2987
(217) 333–0215

ABSTRACT:

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.

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:

An Experimental Quality Software Development Environment

- Completion of the ENCOMPASS and PLEASE prototypes.
- PLEASE and ENCOMPASS use to develop small programs, including specification, prototyping, and mechanical verification.
- Use of PLEASE and ENCOMPASS with experimental automatic code generation techniques that produce implementation code from specifications.
- Completion of papers and a thesis describing ENCOMPASS, PLEASE and the results from our research.

Configuration Management

- A configuration management and version control model has been developed and refined.
- CLEMMA, an initial prototype automated configuration librarian to support configuration management in the SAGA environment has been completed.
- The Troll relational database manager has been integrated into the CLEMMA system to support information retrieval.
- The functionality of CLEMMA has been extended.

Project Management

- A prototypical project management model is being developed.
- A prototype implementation of *PROMAN*, an automated project manager, is being designed and built for the SAGA environment.
- A project task facility has been designed and is being coded.
- The TROLL data base support for management activities and states is complete.
- A primitive form compiler, FORMAN, has been built to support PROMAN. Many major user interfaces of the project management system use the completion of management forms as a paradigm for interaction.
- A prototype multiple screen library, WIN, has been built to support PROMAN. WIN allows forms to be displayed and completed in a screen–oriented mode on terminals.

Survey of Management Approaches

- Publication of the survey of software management techniques in AT&T. (The paper was reviewed by AT&T and is now used in their programmer training.)

Software Analysis Tools

- Completion of an experimental attribute graph grammar kernel intended to support continued investigation of appropriate methods to incorporate incremental semantic

analysis into SAGA tools including EPOS and other language–oriented editors.

- Completion of a specification analyzer, used to compile semantic specifications into tables to drive the attribute graph grammar kernel.
- Continuing use and modification of these prototypes to investigate usability of various semantic implementation models.

Language–Oriented Editor

- The GNU Emacs user interface for the EPOS editor has been improved.
- EPOS is being rewritten in C to be more efficient and to use YACC and LEX.
- GNU Emacs Lisp has been extended with specific objected–oriented features to support the development of additional EPOS tools.
- A programmers manual for EPOS GNU Emacs is being written.

User Interfaces Prototyping Tool

- KAOS, a prototype support library for user interface development, has been designed and partially implemented.

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

2. Overview

The SAGA (Software Automation, Generation and Administration) project has been active at the University of Illinois at Urbana–Champaign since the early eighties (see Appendix A). It is investigating both the formal and practical aspects of providing automated support for the full range of software engineering activities. Early efforts in SAGA were devoted to building software development tools such as Notesfiles, a distributed information base which operates on networks of heterogeneous machines; a source code control system for CDC machines; EPOS, a language–oriented editor based on an incremental LR(1) parser; and TED, a general purpose tree editor which is interfaced to a number of theorem provers and can serve as a proof management system.

ENCOMPASS is the first complete environment to be constructed by the SAGA project; it demonstrates the validity of many of the concepts on which SAGA is based. Specifically, the configuration control and project management systems used in ENCOMPASS serve as prototypes for more comprehensive systems which will be used in future SAGA environments.

In addition, ENCOMPASS demonstrates that environments can be built to integrate rapid prototyping with rigorous incremental development methodologies such as VDM. ENCOMPASS exploits the notion of an executable specification language as a vehicle for validating requirements and testing designs. The same specification language can be used in more formal verification methods including mechanical verification. By integrating executable specifications with an incremental refinement methodology,

ENCOMPASS introduces the possibility of reusable designs and more meaningful acceptance criteria for project milestones. Last, recent work has show the possibility of automating some of the decision making process involved in coding an implementation of a specification.

Guided by the results from ENCOMPASS, the SAGA Project is already developing Clemma, a configuration librarian, and PROMAN, a second generation project management system. These two new systems provide a more general purpose and more powerful implementation of the ideas in ENCOMPASS. Both systems use a data base and can support the management of large software projects. In addition, both systems improve upon the functionality available in the ENCOMPASS system. In future work, these systems will greatly enhance the capabilities of the next complete experimental environment, the successor to ENCOMPASS. Appendix C contains a paper reviewing the current status of the project.

3. The ENCOMPASS Prototype Software Development Environment

The ENCOMPASS system was built to study the design issues involved in constructing a quality software development environment. The system formed the major part of Bob Terwilliger's Ph.D. thesis. A draft of the thesis is contained in Appendix B but the summary from the thesis is reproduced in this section to provide an overview for the work.

In the thesis will be found a proposal to automate the rigorous development of a software system using existing methods and tools. Although the system is primitive and many of the methods and tools inadequate, the approach provides a foundation upon which a quality software production facility could be built. The approach successfully combines many themes from current research including prototyping, program testing and verification, incremental development, and automatic code generation.

The prototype includes PLEASE, an Ada–based, wide–spectrum, executable specification and design language; IDEAL, an environment for programming–in–the–small using PLEASE; and ENCOMPASS, a simple environment for programming–in–the–large. Together, these form an integrated system to support incremental software development in a manner similar to VDM. In ENCOMPASS, software is specified using a combination of natural language and PLEASE. In PLEASE, software can be specified using Horn clauses: a subset of first–order, predicate logic. In ENCOMPASS, PLEASE specifications can be incrementally refined into Ada implementations. Each step is verified before the next is applied; therefore, errors can be detected and corrected sooner and at lower cost. In ENCOMPASS, a refinement can be verified using peer review, testing, or proof techniques.

Executable prototypes can be automatically constructed from PLEASE specifications by translating pre– and post–conditions into Prolog procedures. PLEASE prototypes are based on existing Prolog technology, and their performance will improve as the speed of Prolog implementations increases. As logic programming progresses, new versions of PLEASE can be built based on more powerful logics.

PLEASE prototypes can enhance the validation, design, and verification processes. During the validation phase, 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. PLEASE prototypes can also be used to verify the correctness of refinements; most simply, the prototype produced from a PLEASE specification can be used as a test oracle against which implementations are compared. In a more complex case, the prototypes produced from the original and refined specifications can be run on the same data and the results compared. PLEASE specifications also enhance the verification of system components using proof techniques; for the purpose of formal verification, the refinement process can be viewed as the construction of a proof in the Hoare calculus.

IDEAL is an environment concerned with the specification, prototyping, implementation and verification of single modules. IDEAL provides facilities to create PLEASE specifications, construct prototypes from these specifications, validate the specifications using the prototypes produced, refine the validated specifications into Ada implementations, and verify the correctness of the refinement process. IDEAL is an environment for the *rigorous* 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. Proofs may range from a very detailed, completely formal proof using mechanical theorem proving, to a development "annotated" with unproven verification conditions. Parts of a project may use detailed mechanical verification while other, less critical parts may be handled using less expensive techniques. Our experience so far leads us to believe that the complete, mechanical verification of large programs will be prohibitively expensive; however, inexpensive methods can eliminate a large percentage of the verification conditions generated during a development. By eliminating these "trivial" verification conditions, the total number is reduced so that the verification conditions remaining can be more carefully considered by the development personnel.

ENCOMPASS provides facilities to store, track, manipulate and control all the objects used in the software development process: documents, specifications, source code, proofs, test data, and load modules are all supported. ENCOMPASS also provides mechanisms to support the interactions among developers; the system allows the creation, decomposition, distribution, monitoring and completion of tasks. In ENCOMPASS, the configuration management system structures the software components developed by a project, while the project management system uses facilities provided by the configuration management system to control both access to data and interactions between developers. ENCOMPASS is based on a traditional life–cycle, modified to support the use of executable specifications and VDM. In ENCOMPASS, some modules of a system may be developed using PLEASE and IDEAL, while others are developed using conventional techniques. This allows the practical power of Ada and the formal power of PLEASE to be combined in a single project. ENCOMPASS can be extended with a central repository to support software reuse; we have also constructed an automated change control system based on ENCOMPASS.

ENCOMPASS Status

The ENCOMPASS environment has been under development since 1984. A prototype implementation has been operational since 1986; it is written in a combination of C, Csh (the Berkeley 4.3 UNIX™ command interpreter), Prolog and Ada. The prototype implementation of IDEAL 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. PLEASE, IDEAL and ENCOMPASS have been used to develop a number of programs, including specification, prototyping, and mechanical verification. At present, all the programs developed have been less than one hundred lines in length, but some have included more than one module, allowing demonstrations of the ENCOMPASS configuration control and project management systems.

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. The current implementation of PLEASE is based on the UNSW Prolog interpreter and the Verdix Ada Development System; it runs under Berkeley Unix on a Sun 2/170. The Prolog interpreter and Ada program run as separate processes and communicate through pipes (a UNIX interprocess communication mechanism.) This implementation is somewhat expensive; for example, there is a five CPU second overhead to start the Prolog interpreter, but this is incurred only once during program execution. A procedure call from Ada to Prolog costs about forty milliseconds excluding parameter conversion.

The combination of algebraic simplification and simple proof tactics implemented in ISLET seems to work very well; in our experience, it can eliminate between fifty and ninety per cent of the verification conditions generated during refinement. The simple methods run very quickly: less than one second response time in all the cases examines so far. The use of TED and theorem provers is very expensive.

Copies of demonstration scripts showing ADA software development in ENCOMPASS and a demonstration version of ENCOMPASS have been ported to NASA Langley machines. Bob Terwilliger presented the ENCOMPASS work to the staff at NASA Langley. Two papers (accepted for publication in the Journal of Software and Systems) describing PLEASE and ENCOMPASS are included in Appendix G and H respectively. Appendix F contains a preliminary report concerning the use of ENCOMPASS in the automatic generation of implementation code from PLEASE specifications using artificial intelligence techniques.

4. Configuration Management

An initial prototype of CLEMMA, an automated configuration librarian, has been completed. The prototype operates using the Troll relational database manager and the

UNIX™ file system. This initial prototype includes facilities for creating project libraries, checking components in and out of libraries, identifying individual components, and storing views of composite software objects.

In its current incarnation, CLEMMA provides the basic functions required of a configuration management system. It allows the user to store and retrieve the components of a software project, to record the relations between the components, and to create attribute–based descriptions of the components in a relational database. CLEMMA presents itself as a set of tools, usable by project managers and workers, for manipulating project libraries. Assuming use in the context of a project management system such as PROMAN, CLEMMA would control the access and modifications to the components of a project. The system would provide identifying, tracking, and derivation information, aiding managers in coordinating the development and maintenance of systems. The interface to the tools is a simple text one, thus facilitating the use of the tools in command scripts in other systems, such as PROMAN.

The development of CLEMMA has uncovered several interesting questions in the area of configuration management. For example, the problems of module description, representation and compatibility of parallel revisions, and component evolution are currently being explored. New, more detailed data models for configurations and versions are required for the system and are being formulated. More sophisticated interfaces to the system are also being considered, to enhance ease of use and applicability to different operating environments. As they reach fruition, these new ideas will be refined and incorporated into the configuration management system. A more detailed overview of CLEMMA can be found in Appendix C. Hal Render, author of CLEMMA, is spending this summer at NASA Langley where he will install the system on a NASA computer. While at NASA, he will deliver a more detailed report on the system and its use.

5. Project Management

The design and development of a project management system for the SAGA environment is now under way. This system, called PROMAN, emphasizes tracking, control, and communication in the project development environment. To support these aspects of project management, PROMAN models a development project as hierarchy of tasks and uses techniques from programming languages, operating systems, and databases in its implementation.

In the past year, we have arrived at PROMAN's basic task model and design. In brief, PROMAN models a software project as a hierarchy of tasks. Each task has associated resources (input and output) and dependencies on other tasks. PROMAN's basic design is as the top level of an integrated environment. Therefore, PROMAN's design has facilities for interfacing to project resource managers such as configuration managers (e.g. CLEMMA) and programming–in–the–small environments (e.g. C shell). It also includes a project browser for viewing the task structure of the development and a report generator for periodically summarizing project status. PROMAN's user interface is based on a form–filling metaphor with a standardized command interface.

Initial implementation has concentrated on support tools and task manipulation. In the area of support tools we have constructed WIN and FORMAN. WIN is a simple multiple screen library that supports building forms out of fields and interacting with them using a standard command mechanism. FORMAN is a form compiler that accepts as input a programming language specification for a form and generates C code that uses WIN to build and manipulate the form. In the area of task manipulation, we have developed the basic interface to the task model for access and work on projects and tasks. Also primitive implementation for task creation and state transition has been implemented. Appendix C contains an overview of PROMAN. Appendix D and E contain draft documents for WIN and FORMAN. More complete documentation is in preparation.

6. Incremental Semantic Analysis

The goal of this research is to find and implement an effective way to do incremental semantics. This will aid in providing user support for program development, and will be used in conjunction with the EPOS language oriented editor. Applications include building intelligent user interfaces for the configuration management and project management systems that can exploit the knowledge stored in the data bases, incrementally processing the contents of forms produced interactively by the FORMAN system and building graphical design tools to support incremental analysis of control flow organization, data flow organization, structural analysis, and coherence and coupling between modules.

The underlying framework we are adapting is that of Graph Grammars, where using graphs gives a useful generalization over trees, and using grammar structuring provides a mechanism to manage the complexity of graph formation and editing. Within this framework, there is a wide variety of specific models. Most of the effort so far has been to explore some of the options, working toward deciding which will be helpful in this particular application. To do this the first step was to build a prototype system, containing a grammar specification analyzer to compile user specified language semantics into tables, and a table–driven graph editing kernel, which automatically computes semantic values as the user edits the graph. Both prototype tools, the analyzer and the kernel, have been completed.

Research is now focussing on using these tools to gain experience in specifying and executing the semantic information propagation on several, all fairly simple, examples. The examples include semantic analysis of data flow graphs, control flow graphs and structure charts. Working with the examples has uncovered some weaknesses in the model which need to be fixed, and has also suggested extensions which are being tried. It is anticipated that the process will continue through several iterations, during which increasingly significant examples can be handled, eventually converging to a model suitable for handling arbitrary programming language semantics.

Perhaps the biggest problem made apparent through experimentation with the prototype is the difficulty of correctly specifying reasonably complex semantics. As we develop experience by working with examples, we will discover common specification

idioms, which may then be facilitated by special syntax and defaults. This consideration, verifiably correct and managable specification of semantics, also provides an important constraint in what models we may use to support the system. Concurrently with exploring the utility of the various models, we are considering their theoretical properties. Appendix I contains a more detailed description of the approach.

7. Towards a General Interface for SAGA Editors and Other Tools

Current plans intend to use GNU Emacs as a uniform user interface to other SAGA tools. The advantages of a uniform user interface are ease of learning, and ease of use. GNU Emacs is a good choice for the user interface because it is widely available and very flexible and powerful. The incremental parser, EPOS, was the first SAGA tool to use Emacs as its user interface.

To support the development of interfaces between GNU Emacs and other SAGA tools, time was spent creating a programmers' manual for GNU Emacs. The extension language for GNU Emacs is a full Lisp augmented with several hundred editor–specific functions and variables.

The Configuration Management system and the Project Management system will interface to the user through GNU Emacs routines for data editing. To support this project, an object–oriented extension to GNU Emacs Lisp was developed. A data specification language will describe both data base aspects for use by the SAGA tool and user interface aspects for use by the Emacs data editor.

8. Leif: EPOS Redesigned

Although Pete Kirslis, the original author of EPOS, is building a C++ "industrial" version of EPOS for AT&T, Denver, the SAGA Project needs access to an efficient, public domain, implementation of EPOS for experimentation. The EPOS language oriented editor is being reimplemented in C. This will improve the portability of the editor and make major improvements in performance and the quality of the design. The new version of the editor, Leif, will use the public domain Bison parser generator and the internal structure will be enhanced.

9. User Interface Prototyping Tool

A support library for user interfaces is in the process of development. It provides a structure for user interfaces that is consistent across all levels of the user's display and provides an infrastructure on which to build both user interfaces and user interface management systems. Further documentation is in preparation.

10. Conclusions

In the first six months of this year's research, we have completed ENCOMPASS, an initial, comprehensive software development environment, used it experimentally to build several small examples and begun to develop a "next generation" environment. The configuration management and project management components of the new environment are well–advanced. We are seeking more powerful executable specification

and prototyping techniques. (The work of Michael Holloway could be of significance to the approach adopted in the next environment.) In the new environment we will seek a tighter coupling between the software methodology and the project management and configuration management tools. We would also like to emphasize code and design reuse, explore design aids, and expert system support for the development process.

ENCOMPASS is the first complete environment constructed by the SAGA Project. ENCOMPASS supports a formal development method similar to VDM, as well as providing basic facilities for configuration control and project management. A VDM–like methodology was chosen because it supports the specification, validation, design, implementation, and verification of software. It also provides completion criteria for the steps in the production process and offers limited, but well–defined, project management goals. The design, construction and use of ENCOMPASS revealed many shortcomings in its project and configuration management systems.

SAGA is now creating new systems both to correct these deficiencies and support more of the life–cycle. CLEMMA is a configuration librarian which maintains software structures and provides views of a project's components. CLEMMA capitalizes on existing data base and file system technology to provide flexible support for abstraction and manipulation of software components. It can be easily updated to provide new facilities and abstractions without reorganizing the project data.

The project management system PROMAN supports the integration and control of the software development and management processes. It implements management policies through the use of interaction protocols and project access permissions. It also supports repositories of project information and components. The management system is based on a process/resource model in which the process hierarchy models the personnel and work breakdown structures of the project. The project management system controls project access, supports resource allocation and usage, and coordinates and synchronizes task activities.

The improved configuration and project management systems are under implementation; many components are complete. The two systems are complementary: efficient automation of the software development process depends on the effective integration of project management and configuration control. The systems must combine to provide users with consistent, task related, functional abstractions of activities and resources. The configuration and project management models have application to existing software development practices; however, the SAGA Project is seeking to apply them to an improved, rigorous software development methodology.

Automating the entire software life–cycle will require continuing research; one reason is the immaturity of the software engineering discipline. For example, future development methodologies must incorporate more formal approaches to requirements analysis, reuse and maintenance. Future environments should also have more advanced system architectures which support knowledge–based tools.

We believe that the configuration and project management models and systems currently proposed can significantly enhance many aspects of the software life–cycle.

However, these models and systems must evolve as we strive for more effective development methodologies; for example, improved implementation methods must be pursued as usage data is gathered. Although life–cycle automation is a long term research problem, the current work in project and configuration management can do much to improve software development as it is practiced today.

APPENDIX A

Project SAGA Bibliography

SAGA Bibliography

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.

23. Sum, Robert, N., Jr. "A Summary of the Software Development Cycle of the AT&T System 75 in Middletown, NJ", Report No. UIUCDCS–R–87–1332, Dept. of Computer Science, University of Illinois at Urbana–Champaign, 1987.

24. Terwilliger, Robert B. and Roy H. Campbell. "ENCOMPASS: an Environment for the Incremental Development of Software", **Journal of Systems and Software**, to appear in 1988.

25. Terwilliger, Robert B. and Roy H. Campbell. "PLEASE: Executable Specifications for Incremental Software Development", revised and submitted to the **Journal of Systems and Software**.

26 Campbell, Roy H., Hal Render, Robert N. Sum, Jr., and R. Terwilliger. "Automating the Software Development Process," Report No. UIUCDCS–R–87–1333, Dept. of Computer Science, University of Illinois at Urbana–Champaign, 1987.

27 Terwilliger, Robert B. "Encompass: An Environment for Incremental Software Development Using Executable, Logic–Based Specifications." Ph.D. Thesis, in preparation, Report No. UIUCDCS–R–87–????, Dept. of Computer Science, University of Illinois at Urbana–Champaign, 1987.

28 Terwilliger, Robert B. "Knowledge–Based Development in ENCOMPASS (Preliminary Report)", Report No. UIUCDCS–R–87–1334, Dept. of Computer Science, University of Illinois at Urbana–Champaign, 1987. Attribute Graph Grammars Appendix:

29 Kaplan, S., S. Goering, R. H. Campbell. "Supporting the Software Development Process with Attributed NLC Graph Grammars", submitted: Proceedings of the 3rd International Workshop on Graph Grammars, June 1987.

APPENDIX B

Encompass: An Environment for Incremental Software Development Using Executable, Logic–Based Specifications

Robert B. Terwilliger

ENCOMPASS: AN ENVIRONMENT FOR INCREMENTAL SOFTWARE DEVELOPMENT USING EXECUTABLE, LOGIC-BASED SPECIFICATIONS

BY

ROBERT BARDEN TERWILLIGER

B.A., Ithaca College, 1980
M.S., University of Illinois, 1982

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, 1987

Urbana, Illinois

© Copyright by
Robert Barden Terwilliger
1987

ENCOMPASS: AN ENVIRONMENT FOR INCREMENTAL SOFTWARE DEVELOPMENT USING EXECUTABLE, LOGIC–BASED SPECIFICATIONS

Robert Barden Terwilliger, Ph.D.
Department of Computer Science
University of Illinois at Urbana–Champaign, 1987
Roy H. Campbell, Advisor

The Vienna Development Method (VDM) supports the top–down development of software specified in a notation suitable for formal verification. 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 may not be formally axiomatized. This dissertation is based on the thesis that the time is ripe for the construction of environments which partially automate development methods similar to VDM, and that such environments will prove useful in industrial settings. ENCOMPASS is an automated environment which supports a formal development method similar to VDM; it supports rapid prototyping and program verification, as well as providing simple facilities for configuration control and project management. In ENCOMPASS, components are specified using a combination of natural language and PLEASE, a wide–spectrum executable specification and design language. PLEASE specifications may be used in proofs of correctness; they may also be automatically transformed into prototypes which use Prolog to "execute" pre– and post–conditions. In ENCOMPASS, PLEASE specifications are incrementally refined into Ada[1] implementations. Each refinement is verified before another is applied; therefore, the final components produced by the development satisfy the original specifications. In ENCOMPASS, the correctness of a refinement step can be verified using either

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

testing, proof or peer review techniques. ENCOMPASS is an environment for the *rigorous* 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. 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. Parts of a project may use detailed mechanical verification while other, less critical parts may be handled using less expensive techniques. We believe the use of future environments similar to ENCOMPASS will enhance the software development process.

DEDICATION

To Life – got something better to do?

Life a river with each man flowing; far downstream he hears the falls.
The wish to stop and ponder before falling through the mist.
But he cannot stop; there is no ground.
Just the river. Only water. Always flowing.

He grasps at twigs, leaves, whatever swirls near him in the flood.
A raft! With a raft he can reach the unseen shore!
But there is no shore.
Just the river. Only water. Always flowing.

Though Man has no shore, He will build his raft
and from there He will reach for higher ground.
This work is dedicated to the future.
Onwards.

Bob Terwilliger, 1977

Yet hath he not root in himself, but dureth for a while: for when tribulation or persecution ariseth because of the Word, by and by he is offended.

Mathew 13:21

Life is what you make it.
It's not much of a raft, but it beats treading water.
Anybody got some spare twigs?

Bob Terwilliger, 1987

Therefore, O Arjuna, surrendering all your works unto Me, with full knowledge of Me, without desires for profit, with no claims to proprietorship, and free from lethargy, fight.

Bhagavad–gita 3.30

ACKNOWLEDGEMENTS

I would like to thank my advisor, Roy Campbell, for his advice and support throughout this research. Roy first introduced me to the ideas on which this work is based. I would also like to thank the members of my committee, particularly Ralph Johnson and Simon Kaplan, for their advice and comments. Ralph especially has greatly influenced my thinking and increased my understanding of the formal aspects and underlying philosophy of the work. My discussions with Lee Benzinger greatly increased my understanding of the problems of incremental development and verification. Phil Roberts was very helpful in designing parts of the ENCOMPASS implementation. This research was funded by NASA grant NAG 1–138; I am grateful for this support. Finally, I would like to thank my parents and my sister, Sue; without a doubt, they are the only reason I ever made it through this period of my life.

TABLE OF CONTENTS

CHAPTER

1. INTRODUCTION 1
   1.1. Software Development 2
   1.2. ENCOMPASS 7
   1.3. Chapter Summary 11

2. FOUNDATIONS 14
   2.1. First–Order Predicate Logic 14
   2.2. Specifying Software Using Pre– and Post–Conditions 19
   2.3. Program Verification 22
   2.4. Resolution Theorem Proving 26
   2.5. Logic Programming 27

3. RELATED WORK 30
   3.1. Specification Methods 31
   3.2. Development Methods 35
   3.3. Tools 38

4. THE PLEASE LANGUAGE 42
   4.1. Development Paradigm 43
   4.2. Design Considerations 47
   4.3. An Example 50
   4.4. Proof Rules 53
   4.5. Data Types 59
   4.6. Objects 63

5. PRODUCING PROTOTYPES FROM PLEASE SPECIFICATIONS 73
   5.1. Translation to Prolog 74
   5.2. Code Optimization 82
   5.3. System Interface 88
   5.4. Software Validation 93

6. THE INCREMENTAL REFINEMENT PROCESS 96
   6.1. An Example 97
   6.2. Refinement Model 100

7. THE IDEAL ENVIRONMENT 107
   7.1. Architecture 107

7.2. ISLET 110
7.3. An Example 112

8. THE ENCOMPASS ENVIRONMENT 126
8.1. Development Model 128
8.2. Configuration and Project Management 132
8.3. An Example 137
8.4. Software Reuse 142
8.5. Change Control 145

9. SUMMARY AND CONCLUSIONS 155
9.1. System Status 157
9.2. Future Research 159

REFERENCES 162

VITA 178

CHAPTER 1.

INTRODUCTION

The production of software is both difficult and expensive. The rising cost of software relative to hardware in complex systems has led some to speak of the "software crisis". The field of *software engineering* has risen to meet this challenge[88,132,258]. The problems involved in software development are complicated and many different solutions have been proposed; these include methods for designing and implementing software[129,134,252], as well as tools to support the development process[177]. Common sense suggests that no tool or method alone will solve the software development problem in the near future[43]; this has led some to propose *software engineering environments*, which combine a number of tools, methods and data structures within a unified framework[119,219].

Beginning in the late sixties, software development methods with strong formal roots were devised[76,98,104,120,134]. Later, attempts were made to construct automated tools for their support[33,110,112,166,203]. Unfortunately, these attempts did not produce immediately practical results. During the seventies and eighties, attempts were made to use formally based methods in industrial settings[35,187,213]. In general, high degrees of formality and automation were sacrificed to achieve useful methodologies. In these experiments, formal specifications served mostly as a tool for precise communication, and the major impact on methodology was that more time was spent on specification and design. However, the methods did prove useful in practice. This dissertation is based on the thesis that the time is ripe for the construction of environments which partially automate formal development methods, and that these environments will eventually prove useful in industrial settings.

In this dissertation we describe a preliminary version of ENCOMPASS[52,230,231], an environment of this type which has been constructed by the SAGA group at the University of Illinois. In ENCOMPASS, software can be specified using PLEASE[232–234], an Ada–based, wide–spectrum, executable specification language. PLEASE specifications can be incrementally refined into Ada implementations using IDEAL, an environment for programming–in–the–small which supports verification using peer review, testing or proof techniques. ENCOMPASS provides simple support for programming–in–the–large, including configuration control[146] and project management[47]. In ENCOMPASS, some modules of a system may be developed using PLEASE and IDEAL, while others are developed using conventional techniques. This allows the practical power of Ada and the formal power of PLEASE to be combined in a single project. Eventually, we plan to extend ENCOMPASS to provide uniform support for the entire life–cycle; however, at present it is primarily an environment for software development.

1.1. Software Development

Figure 1 shows an abstract model of the software development process; many specific paradigms can be analyzed and compared within this framework. At first, a system exists only as an idea in the minds of its users or purchasers. In our model, the first step in the development process is the creation of a *specification* which precisely describes the properties and qualities of the software to be constructed[88]. Unfortunately, with current methods there is no guarantee that the specification correctly or completely describes the customers desires; a specification is *validated* when it is shown to correctly state the customers' requirements[88]. The specification need not be executable; in general, it must be translated into an implementation, in other words a description of the system which has the property of being efficiently executable. Depending on the method used for translation, the exact relationship between the specification and

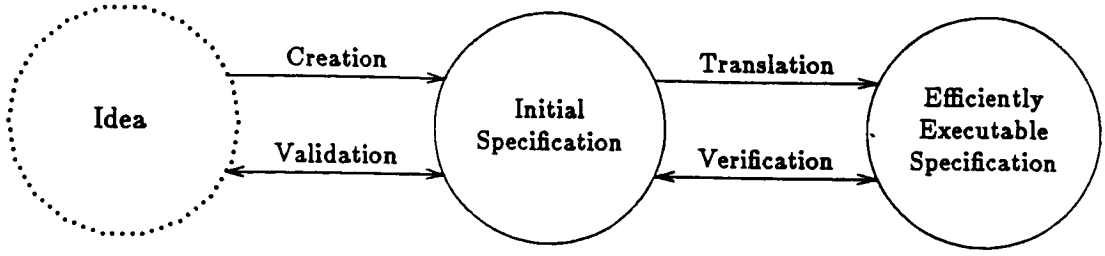


Figure 1. Software development model

implementation may be unknown. An implementation is *verified* when it is shown to satisfy its specification[88].

Many methods for specifying software have been proposed[11,92]. Specifications can be formal, in other words based on mathematics and/or logic, or they can incorporate graphics and natural language. Creating 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. Formal specifications may be an ineffective medium for communication between customers and developers, but natural language specifications are notoriously ambiguous and incomplete. It has been suggested that *prototyping* and the use of *executable specification languages* can enhance the communication between customers and developers[1,140]; providing prototypes for experimentation and evaluation should increase customer/developer communication and enhance the validation process.

The translation from specification to implementation can take many forms. If the specification is in a formal notation, it may be possible to interpret it directly or mechanically

translate it into an executable form. Unfortunately, current technology can not always produce acceptable efficiency; therefore, in most cases a programmer will create the implementation. Many different methods have been proposed to enhance this process; for example, it has been suggested that *modular programming*[138,192,235,238] and *top-down development methods*[76,104,134,178,248] can help reduce the difficulty of software design and implementation. By using *step-wise refinement*[248] to create a concrete implementation from an abstract specification, we divide the necessary decisions into smaller, more comprehensible groups. By encapsulating design and implementation decisions within module boundaries, the clarity and modifiability of software is increased.

Many different techniques can be used to determine if an implementation satisfies a specification. For example, *testing* can be used to check the operation of an implementation on a representative set of input data[91,176]; however, in general, a program cannot be tested on all possible inputs. In a *technical review* process, the specification and implementation are inspected, discussed and compared by a group of knowledgeable personnel[87,242]; unfortunately, there is no guarantee that they will come to the correct conclusions. If the specification is in a suitable notation, formal methods can be used to verify the correctness of an implementation[3,26,110,112,120,121,134,163,250]; however, with the current state of verification technology, many widely used languages are not completely verifiable. Many feel that no one technique alone can ensure the production of correct software[71,75]; therefore, methods which combine a number of techniques have been proposed[9,205].

1.1.1. The Vienna Development Method

The Vienna Development Method (VDM) supports the top–down development of software specified in a notation suitable for formal verification[32,34,35,64,130,133–135,187,213]. In this method, 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. 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. Since each refinement step is small, design and implementation errors can be detected and corrected sooner and at lower cost.

VDM is based on a *model–oriented* or *constructive* approach to specification; components are defined with respect to pre–existing types and operations. To increase the expressive power of specifications, the high–level types *set*, *list*, and *map* are added to the language. In VDM, a procedure or function may be specified using pre– and post–conditions written in first–order, predicate logic. The pre–condition states the properties that the inputs must satisfy, while the post–condition states the relationship of inputs to outputs. VDM may also be used to specify abstract data types. A type has a state, or representation, which can only be modified by certain operations; the operations are defined using pre– and post–conditions. The *invariant* must be true both before and after the execution of each operation; it defines the acceptable states.

In VDM, the refinement process consists of a number of steps. Each step generates *verification conditions* which must always be true for the refinement to be correct. Each step is an instantiation of an abstract refinement; the verification conditions for a step are generated by substituting pre– and post–conditions into the *proof rule* for the abstraction. Refinement steps can be either *decompositions*, which add more detail about the algorithms involved in the solu-

tion, or *refinements*, which add more information about the data structures to be used. In the simplest view, each decomposition changes an unknown program construct into a known structure which may contain other unknowns. Data refinements are more complicated; each step implements the state and operations of one type using the facilities of another.

VDM has been used in industrial environments to enhance the development process[35,187,213]. In this type of environment, the method is not typically applied in all its formality. Pre– and post–conditions are written using operations and predicates which may not be precisely defined. Verification conditions are generated without the aid of automated tools and proved informally using a peer review system. Although this application of the method is useful, a more formal approach could be superior; however, without considerable automated support, a more formal use of VDM would be prohibitively expensive. Many feel the cost of developing automated tools is justified, and environments to support VDM are being constructed[33]. At present, it is unclear how well methods such as VDM can be automated. As the research progresses, the theoretical and practical problems involved will surface.

1.1.2. SAGA

The SAGA (Software Automation, Generation and Administration) project has been active at the University of Illinois at Urbana–Champaign since the early eighties[30,47,48,50,52,53,115,145,146,230–234]; it is investigating both the formal and practical aspects of providing automated support for the full range of software engineering activities. Early efforts in SAGA were devoted to building software development tools such as Notesfiles[83,84], a distributed information base which operates on networks of heterogeneous machines; Epos[145], a language–oriented editor based on an incremental LR(1) parser; and

TED[115], a general purpose tree editor which is interfaced to a number of theorem provers[103] and can serve as a proof management system. ENCOMPASS is the first complete environment to be constructed by the SAGA project; it demonstrates the validity of many of the concepts on which SAGA is based. Specifically, the configuration control and project management systems used in ENCOMPASS serve as prototypes for more comprehensive systems which will be used in future SAGA environments. At present, SAGA is constructing Clemma, a configuration librarian, and PROMAN, a second generation project management system.

1.2. ENCOMPASS

ENCOMPASS (ENvironment for the COMposition of Programs And SpecificationS)[52,230,231] is an integrated environment to support incremental software development in a manner similar to VDM. In ENCOMPASS, software is specified using a combination of natural language and PLEASE[232–234], a wide–spectrum, executable specification and design language. PLEASE specifications may be used in proofs of correctness; they also may be transformed into prototypes which use Prolog[60,152] to "execute" pre– and post–conditions. In ENCOMPASS, PLEASE specifications can be incrementally refined into Ada implementations using IDEAL, an environment for programming–in–the–small which supports verification using peer review, testing, or proof techniques. ENCOMPASS provides simple support for programming–in–the–large, including configuration control[146] and project management[47].

1.2.1. PLEASE

PLEASE (Predicate Logic based ExecutAble SpEcifications) is a wide–spectrum executable specification language which supports incremental software development in a manner similar to VDM. The design of PLEASE is a compromise between logical power, ease of use, applicability

and efficiency. 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[70,241] as the base language. PLEASE permits the development of Ada programs using rapid prototyping and incremental verification techniques.

Executable prototypes can be automatically constructed from PLEASE specifications; these prototypes can enhance both the validation and verification processes. During the validation phase, prototypes produced from PLEASE specifications 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 can increase customer/developer communication and enhance the validation process.

PLEASE prototypes can also be used to verify the correctness of refinements; most simply, the prototype produced from a PLEASE specification can be used as a test oracle against which implementations are compared. In a more complex case, the prototypes produced from the original and refined specifications can be run on the same data and the results compared; this method gives significant assurance that a refinement is correct at low cost. PLEASE specifications also enhance the verification of system components using proof techniques; for the purpose of formal verification, the refinement process can be viewed as the construction of a proof in the Hoare calculus[120,163].

We believe that languages similar to PLEASE can greatly enhance the software development process; however to realize the full benefits of PLEASE an integrated support environment is needed.

1.2.2. IDEAL

IDEAL (Incremental Development Environment for Annotated Languages) is an environment for the specification, prototyping, implementation and verification of single modules; it is a programming–in–the–small environment for development using PLEASE. IDEAL provides facilities to create PLEASE specifications, construct prototypes from these specifications, use the prototypes to validate the specifications, refine the validated specifications into Ada implementations, and verify the correctness of the refinement process. IDEAL contains four main components: ISLET, a language–oriented program/proof editor; a proof management system; a prototyping tool and a test harness.

The central tool in IDEAL is ISLET (Incredibly Simple Language–oriented Editing Tool); it contains three major sub–systems: an algebraic simplifier, a set of simple proof procedures, and an interface to the proof management system. Using ISLET, verification conditions are automatically generated whenever PLEASE specifications are refined into Ada implementations. These verification conditions are first simplified algebraically and then submitted to a number of simple proof tactics. These inexpensive methods can handle a large percentage of the verification conditions generated; if they fail, the verification conditions can be proved using more expensive techniques.

IDEAL is an environment for the *rigorous*[134] 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. Proof techniques may be used that range from a very detailed, completely formal proof using mechanical theorem proving, to a development "annotated" with unproven verification conditions. Parts of a project may use detailed mechanical verification while other, less critical parts may be handled using less expensive techniques. Our experience so

far leads us to believe that the complete, mechanical verification of large programs will be prohibitively expensive; however, inexpensive methods can eliminate a large percentage of the verification conditions generated during a development. By eliminating these "trivial" verification conditions, the total number is reduced so that the verification conditions remaining can be more carefully considered by the development personnel.

While the use of PLEASE and IDEAL alone can enhance the development process, more can be gained with the addition of an environment for programming–in–the–large[200,249]. ENCOMPASS is such an environment; it provides support for all aspects of software development using PLEASE. ENCOMPASS provides facilities to store, track, manipulate and control all the objects used in the software development process: documents, specifications, source code, proofs, test data, and load modules are all supported. ENCOMPASS also provides mechanisms to support the interactions among developers; the system allows the creation, decomposition, distribution, monitoring and completion of tasks.

In ENCOMPASS, the user accesses and modifies components using a set of software development tools. The configuration management system structures the software components developed by a project, while the project management system uses facilities provided by the configuration management system to control both access to data and interactions between developers. The configuration control system is based on a variant of the entity–relationship model[57,58]. The project management system implements a management by objectives approach[106]; each phase in the life–cycle satisfies an objective by producing a *milestone* which can be recognized by the system. ENCOMPASS can be extended with a central repository to support software reuse. We have also constructed an automated change control system based on ENCOMPASS.

ENCOMPASS is based on a traditional life–cycle, modified to support the use of executable specifications and VDM. In ENCOMPASS, we extend the traditional life–cycle to include a separate phase for user validation; we also combine the design and implementation processes into a single refinement phase. In ENCOMPASS, some modules of a system may be developed using PLEASE and IDEAL, while others are developed using conventional techniques. This allows the practical power of Ada and the formal power of PLEASE to be combined in a single project.

1.3. Chapter Summary

The remainder of this dissertation describes PLEASE, IDEAL and ENCOMPASS in detail. In Chapter 2, we present the theoretical foundations of this thesis; they are sound, but not perfect. First, we review the capabilities and limitations of the first–order predicate logic and describe software specification using pre– and post–conditions. We then discuss program verification and describe the Hoare calculus. Finally, we present the resolution principle and its use in automatic theorem proving; this leads naturally into a discussion of logic programming and Prolog.

Chapter 3 discusses previous work in software engineering that is relevant to this thesis. First, we discuss specification methods; formal and informal, model–oriented and axiomatic approaches are considered. We then present some proposed software development methods, including the transformational and "proofs as programs" approaches. Finally, we discuss some of the different tools which have been developed, including environments for programming–in–the–small, program verification systems, and environments for programming–in–the–large.

In Chapter 4, we describe the PLEASE language in detail. First, we present the software development paradigm that PLEASE supports and discuss the tradeoffs present in the language

design. Next, we present an example specification and program. We then present the Hoare–style proof rules for the basic constructs in PLEASE, as well as the pre–defined types and the facilities provided for user type definition. Finally, we describe how PLEASE can be used to specify objects using packages with local variables.

In Chapter 5, we describe the methods used to automatically produce executable prototypes from PLEASE specifications. First, we describe the translation of PLEASE specifications into Prolog procedures; the process is viewed as a sequence of transformations between logically equivalent formulae. We then describe the interface between these prototypes and their environment and discuss their use in the validation process.

In Chapter 6, we present the methods used to refine PLEASE specifications into implementations and verify the correctness of the process. First, we present an example refinement; we describe a single design transformation, which can be decomposed into a number of atomic transformations. Next, we present an abstract model of the incremental development process and use it to define the correctness of a refinement step. Finally, we present the methods used to formally verify the correctness of a refinement.

In Chapter 7 we describe IDEAL in detail and give an example of its use in software development. First, we describe the architecture of the system. Next, we discuss the architecture and operation of ISLET, a language–oriented program/proof editor which is the most important component in IDEAL. Finally we discuss the development of a small program in detail; refinement using ISLET is given the most attention.

In Chapter 8, we describe ENCOMPASS in detail and give examples of its use. First, we describe the life–cycle model ENCOMPASS is designed to support and discuss the configuration and project management systems. We then give an example of software development in

ENCOMPASS; a multi–module system is followed from specification through delivery. Next, we describe how ENCOMPASS can be used to support software reuse, and finally we present an automated change control system based on ENCOMPASS.

Chapter 9 summarizes our experience to date, draws some conclusions, and presents some suggestions for future research. Basically, we are optimistic; we feel the current system demonstrates that a full–scale environment can be constructed. We believe that incremental automation of a proven methodology is a solid research strategy, and we plan to experiment with the addition of knowledge–based techniques. We feel that the use of future environments similar to ENCOMPASS will enhance the software development process.

CHAPTER 2.

FOUNDATIONS

In this chapter we present the theoretical foundations of this thesis; they are sound, but not perfect. The basic limitations and subtle inconsistencies are presented and discussed. We feel these foundations are suitable for a next–generation software development environment. First, we will review the first–order predicate logic; its limitations have a major impact on the design and capabilities of both PLEASE and ENCOMPASS. We then describe software specification using pre– and post–conditions written in predicate logic; this is the method used in PLEASE. We then discuss program verification and describe Hoare calculus, the system used both to specify the semantics of PLEASE and prove that a program satisfies its specification. We then describe the resolution principle and its use in automatic theorem proving. This leads naturally into a discussion of logic programming and Prolog, the language used for prototyping PLEASE specifications.

2.1. First–Order Predicate Logic

The study of *logic*, or the reasoning process itself, goes back at least as far as Aristotle[82]. Over the years, some philosophers have argued that it is the most fundamental of substances. Unfortunately, natural language is typically ambiguous; it does not provide a precise framework for study. During the last hundred years or so, mathematicians have addressed this problem by devising *formal logics* and investigating their properties[117].

Formal logics of the type used in this thesis are concerned with true or false statements, called *formulas*, about a set of objects called the *domain* and *functions* on this set. A formal logic

may contain *predicates* which describe atomic properties of the domain. The functions and predicates together form the *basis*. Formal logics make a division between the symbols used in a formula, the *syntax*, and the meaning of the formula, the *semantics*. The basis consists of symbols; it is given meaning by an *interpretation*. In this thesis, we will make use of a first–order, predicate logic. The logic is *first–order* in that variables may only refer to objects in the domain; no predicate or function variables are allowed.

In this section, we will first review the syntax, or structure, of formulas and then their semantics, or meaning. We will then briefly discuss the basic limitations of the logic; these involve the fact that the validity problem is only partially decidable and the fact that many useful theories are not axiomatizable. This section is not meant for a reader completely unfamiliar with formal logic. Introductory logic texts include[66,81,113]; briefer introductions are given in[163,169]. As much as possible, our notation will be consistent with[163].

2.1.1. Syntax and Semantics

The symbols used to construct formulae in the first–order, predicate logic can be divided into two classes: the *logical symbols*, which may be used in any formula and will have the same meaning in all cases; and the *extra–logical symbols*, which are specific to a particular application. The logical symbols can be divided into: the truth symbols, the variables, the quantifiers, and the connectives. In this thesis, the truth symbols are *true* and *false* and have their usual meanings. Variables will, in general, be denoted by capital letters near the end of the alphabet (for example, X, Y or Z), while formulae will be denoted by possibly subscripted capital letters (for example, W_1, P_1, or Q). The quantifiers are: $\forall$, read *for all*, and $\exists$, read *there exists*. The connectives are: $\neg$, read as *not* ; $\wedge$, read as *and* ; $\vee$, read as *or* ; $\supset$, read as *implies* ; and $\equiv$, read as *if and only if*.

The extra–logical symbols can be divided into the *function symbols* and the *predicate symbols*. The function symbols include the *constants*, which are functions with no arguments, and can be composed into *terms*. Predicates represent fundamental properties; they can take terms, but not other predicates, as arguments. In general, predicates will be denoted by small letters in the middle of the alphabet (for example, p, q(X) or r(X,Y)), while functions will be represented by small letters near the beginning of the alphabet (for example, c, f(X) or g(X,Y)). The meaning of the extra–logical symbols is determined by an interpretation; there may be many possible interpretations for a given basis. Each interpretation maps terms to values from the domain, as well as determining the truth or falsehood of each predicate for all possible arguments.

A formula may consist of a single predicate; in this case, its meaning is simply the value of the predicate. The connectives may be used to construct complex formulae from simpler ones. If W_1 is a formula, then $\neg W_1$ is true if W_1 is false and false if W_1 is true. The formula $W_1 \wedge W_2$ is true if both W_1 and W_2 are true and false otherwise. Similarly, $W_1 \vee W_2$ is true if either W_1 or W_2 is true; it is false if both W_1 and W_2 are false. The formula $W_1 \supset W_2$ is equivalent to the formula $\neg W_1 \vee W_2$, and $W_1 \equiv W_2$ is true if both W_1 and W_2 are true or both W_1 and W_2 are false. The formula $\forall (X) (W_1)$ is true if W_1 is true for all possible values of X, otherwise it is false. The formula $\exists (X) (W_1)$ is true if W_1 is true for some value in the domain, otherwise it is false.

To clarify these concepts further, let us examine an example basis and some formulae referring to it. Consider the system of Peano arithmetic[163]: there are two binary function symbols, + and $*$; two constants, 0 and 1; and a single predicate symbol, $<$. The function symbols can be composed to produce terms such as 0, 1+0, and 1+0*1. Terms, predicates, logical connectives and quantifiers can be used to create formulae such as: $0<1$, $1+1<0+1$, $0<1 \wedge 1+1<0+1$, and $\forall (X) (0<X \vee 0<x+1)$. If we consider an interpretation over the natural numbers with the

usual meanings for the function and predicate symbols, then the formula $0<1$ evaluates to true while the formula $1+1<0$ evaluates to false.

2.1.2. Limitations

While first–order, predicate logic is a powerful formalism, it has severe limitations. When such logics were first devised, it was hoped they might provide a foundation for all of mathematics; unfortunately, this is not possible[117]. Two major limitations will be discussed in this section. First, it can be proved that no algorithm exists which is guaranteed to terminate and correctly determine if an arbitrary formula is true or false. Second, for many interpretations it is not possible to produce a finite set of assumptions from which all true formulae can be deduced. These properties limit both the verification of programs and the automatic creation of prototypes from specifications.

Computer scientists have long been interested in problems which can be answered with either a "yes" or a "no". Such a problem can also be seen as determining if a particular object belongs to the set which yields "yes" answers. If an algorithm exists which will always terminate and correctly answer this question in finite time, then the problem (and the set) are said to be *decidable*[124,163,169]; if no such algorithm exists, then the problem (and the set) are *undecidable*. If an algorithm exists which will terminate with a "yes" in finite time if such an answer is correct, but may either terminate with a "no" or not terminate if a "no" is correct, then the problem (and the set) are *partially decidable*.

The relationship between a problem's decidability and its prospects for practical solution is complex. The question is not whether a solution can always be found, but whether on the average, solutions can be found in a reasonable time. An analogy can be made with combinatorial

optimization and NP-completeness[141]. Although at first NP-complete problems were thought to have no practical solutions, many are routinely solved today. The solutions used have worst-case exponential performance, but on the average yield solutions in polynomial time. The fact that a problem is decidable does not mean it can be solved with today's computers; it may require more resources than are available in current systems. On the other hand, the fact that a problem is not decidable presents fundamental barriers. No matter how technology improves, there will always be cases which yield no solution. Roughly speaking, we can say a formula is *valid* if it is always true[1]. Unfortunately, it is known that the set of all valid formulae for the first-order, predicate logic is only partially decidable.

In general, we are not interested in the formulae which are always true; we are interested in the formula which are true for a particular interpretation. Informally, the set of all formulae which are true for a given interpretation are called its *theory*[2]. In other words, all the formula in the theory are true in the interpretation, and all formula deducible from the formulae in the theory are also contained in the theory. In order to prove the truth of formulae in the interpretation, we must create a set of assumptions from which any formula in the theory can be deduced. For example, if we are interested in the natural numbers, we would like a set of assumptions from which we can deduce whether a given formulae about the natural numbers is in its theory or not, in other words whether it is true or false.

A set of formulae from which a theory can be generated is called its *axiom set*. A theory is *axiomatizable* if it can be generated from a decidable set of axioms[163]. Unfortunately, many common theories are not axiomatizable; for example, Peano arithmetic (which is much simpler

[1]More formally, a formula is valid if it is true in all interpretations[163].

[2] More formally, a theory is a closed set of mutually consistent formulae[163].

than integer arithmetic) is not axiomatizable[93,163]. Even if a theory is axiomatizable, it may not be axiomatizable in first–order logic; for example, the induction axiom, $\forall$ (P) (P(0) $\wedge$ $\forall$ (N) (P(N) $\supset$ P(N+1)) $\supset$ $\forall$ (X) P(X)), is not first order. This complicates the construction of automated deduction systems: the system must use techniques beyond the first–order logic framework to deduce all true formulae. Despite its limitations, first–order, predicate logic is the basis for much of the work in program specification and verification.

2.2. Specifying Software Using Pre– and Post–Conditions

To specify software using predicate logic, we must first describe the semantics of programs. For example, consider a program P which manipulates a set of variables, the values of which are called the *state*. The set of all possible states can be denoted by S. In general, execution of P will change the state: for each state $s_1 \in S$, the execution of P will produce a unique state $s_2 \in S$. Therefore, P can be described as a function, $P : S \rightarrow S$.

Formulae in first–order, predicate logic may be used to describe functions or relations on S. For example, the formulae P and Q define a relation which includes all ordered pairs (s_1, s_2) such that both P(s_1) and Q(s_2) are true. P and Q do not necessarily define a function; there may be more than one tuple with same first element. We may specify a program using formulae called the pre–condition, denoted by P, and the post–condition, denoted by Q. In such a specification, the *pre–condition* specifies the properties that must hold before P begins execution and the *post–condition* specifies the properties that must be true when execution is complete.

To be more precise about the relationship between a program and its pre– and post–conditions, we must differentiate between the notions of partial and total correctness. We say a program is *partially correct* with respect to P and Q if whenever execution begins in a state

where P is true, then if execution terminates normally Q will be true in the state reached. In other words, if P holds when execution begins, then Q will hold on termination. The program is *totally correct* if it is both partially correct and guaranteed to terminate normally.

For example, consider the following program which calculates the value of Z^2 in X.

```
X := 0 ;
Y := 0 ;
while Y < Z loop
      Y := Y + 1 ;
      X := X + Z ;
end loop ;
```

In this example, the state is the values of the variables X, Y and Z. Assuming that all variables range over the natural numbers, *Nat*, then the set of all states is $Nat \times Nat \times Nat$. The program can be viewed as a function taking each state s_1 to a new state s_2 where Z is unchanged, Y is equal to Z, and X is equal to Z^2. We can specify this program using pre-condition $P \equiv (Z=c_1)$ and post-condition $Q \equiv (Z=c_1 \wedge Y=Z \wedge X=Z^2)$. This specification states that if the program begins execution in a state where Z is equal to the constant c_1, then after execution is complete Z will still be equal to c_1, Y will be equal to Z and X will be equal to Z^2.

There are problems with using predicate logic pre- and post-conditions to specify software. First, there is what might be termed *incompleteness*; in general, pre- and post-conditions define relations while programs are functions. It is difficult to ensure that a pre- and post-condition specification defines a unique output for each input, and at times, enforcing this property may result in *over-specification*. For example, consider the following specification of the example program: $P \equiv (\text{true})$, $Q \equiv (X=Z^2)$. In one sense, this specification contains the essence of the program: the value of X should be equal to Z^2; however, it says nothing about the value of Y and Z

after execution is complete. A program which set both X and Z to 0 would satisfy the letter, but not the intent of the specification.

As another example, take the specification $P \equiv (Z=c_1)$ and $Q \equiv (Z=c_1 \wedge X=Z^2)$. This is the specification a human would write for a program to compute the square of a number. The specification states that the number to be squared will not be changed and that the calculated quantity will be correct. This specification does not mention the variable Y; it is an *implementation artifact.* There are many valid implementations which do not use a variable Y, or even a while loop. In general, there will be many correct implementations for any specification; a specification which has only one implementation contains too much implementation detail.

Another problem would arise if there are programs which we cannot specify; in other words, if we cannot express what a program does using predicate logic. Unfortunately, in the strictest sense this is the case[163]. Somewhat informally, we can say that for some interpretations it is possible to compute quantities which cannot be described using the symbols provided. For example, consider the program given earlier in this section and Presburger arithmetic[163], which contains the constants 0 and 1, the function + and the relation $<$. While this basis contains all the symbols necessary to write the program, it does not contain enough symbols to write a formula stating that X is equal to Z^2. In practice, this problem does not arise; one simply defines (possibly recursively) a new predicate which describes the computed values.

Another problem would occur if one could write a specification which no program could satisfy. In a partial correctness system this does not occur; a non–terminating program is correct with respect to any specification. In a total correctness system there are specifications which no program satisfies; for example, $P \equiv (\text{true})$ and $Q \equiv (\text{false})$, or $P \equiv (\text{true})$ and $Q \equiv (X=f(y))$ where f is an uncomputable function. These limitations are not important in practice; in general,

people do not wish to construct programs which terminate in non–existent states or evaluate non–computable functions.

A final point concerns the notation used to reference values in the input state in the post–condition; for example, to specify a program which increments a variable. In the example above, we use the constant c_1 to refer to the value of Z before execution begins; the pre–condition states that Z is equal to c_1. While this offers no theoretical problems, it requires the creation of many new constants and makes pre–conditions more complex than necessary. A similar problem occurs with specifying that the value of a variable does not change. Some specification systems offer notations to address these problems[15,134].

2.3. Program Verification

At times we may wish to prove that a program is correct with respect to a pre– and post–condition specification. To do this, we must understand how the state is changed and the relationship of these changes to the truth of formulae. In the programs we have been considering, only the assignment statement can change the state. For a formula to be true after the assignment of an expression to a variable has completed, the formula with the expression substituted for the variable must be true before the assignment begins. We will denote the formula P, with e substituted for all free occurrences of X, by P^e_X.

With this understanding of the assignment statement we can verify the correctness of implementations; some of the earliest work used the method of *inductive assertions*[89]. In this method, an implementation is modeled as a flow chart with input and output formulae. Each arc in the flow chart is labeled with an *assertion*: a formula which must hold whenever the program reaches that point during execution. Using the input assertion as the basis, induction techniques

are used to prove that the assertions hold for all paths through the flowchart, and therefore for all possible executions. While this method works well, it requires the construction of flow charts and the explicit use of induction. Somewhat later, other methods overcame these difficulties.

2.3.1. The Hoare Calculus

One such system was presented by Hoare[14,120]. Hoare's method is based on axioms and proof rules for common programming language constructs; these can be combined to form deductive proofs of correctness. The method has the advantage of being both easy to use and understand; flowcharts need not be constructed and induction is not used. This method has been used to define the semantics of programming languages[122], and has formed the basis for much of the subsequent work in program verification[134,164,250].

Hoare axioms and proof rules are presented as triples consisting of the pre–condition, the language construct and the post–condition. For example, the axiom for assignment statements is:

$$\{P^{e}_{X}\}\ X := e\ \{P\}$$

for all formulae P, variables X and expressions e.

We can read this as: if the formula P, with e substituted for X, is true before the execution of the assignment X := e, then P will be true on termination. The curly brackets, { }, denote partial correctness; square brackets, [ ], would be used to denote total correctness. As another example, consider the rule for *statement composition*:

$$\frac{\{P\}\ S_1\ \{R\},\ \{R\}\ S_2\ \{Q\}}{\{P\}\ S_1\ ;\ S_2\ \{Q\}}$$

for all formulae P,Q,R and statements S_1,S_2.

We can read this as: if S_1 is partially correct with respect to P and R, and S_2 is partially correct

with respect to R and Q, then S_1 ; S_2 is partially correct with respect to P and Q. To construct any non–trivial proofs in the system we also need the *consequence rule*:

$$\frac{P \supset Q,\ \{Q\}\ S_1\ \{R\},\ R \supset S,}{\{P\}\ S_1\ \{S\}}$$

for all formulae P,Q,R,S and statements S_1.

This rule can be read as: if P implies Q, S_1 is partially correct with respect to Q and R, and R implies S, then S_1 is partially correct with respect to P and S.

Using these axioms and rules we can construct a simple proof of correctness. Consider the initialization code from the square program presented earlier in this chapter:

```
X := 0 ;
Y := 0 ;
```

The program can be described with pre–condition true and post–condition X=0 ∧ Y=0. The first step in the proof is to devise an assertion that holds between the two assignment statements; the formula X=0 will suffice. We can then use the rule for statement composition to perform the following deduction:

$$\frac{\{true\}\ X := 0\ \{X=0\},\ \{X=0\}\ Y := 0\ \{X=0 \wedge Y=0\}}{\{true\}\ X := 0\ ;\ Y := 0\ \{X=0 \wedge Y=0\}}$$

In other words, if X := 0 is partially correct with respect to true and X=0, and Y := 0 is partially correct with respect to X=0 and X=0 ∧ Y=0, then X := 0 ; Y := 0 is partially correct with respect to true and X=0 ∧ Y=0. We now have two sub–proofs to construct: one for each assignment statement.

Since the proofs are similar, we will present only the first assignment. Using the assignment axiom we can deduce:

$$\{0=0\}\ X := 0\ \{X=0\}$$

While at this point it is obvious the proof is correct, to maintain formality we must perform a final step using the consequence rule:

$$\frac{\text{true} \supset 0{=}0,\ \{0{=}0\}\ X := 0\ \{X{=}0\},\ X{=}0 \supset X{=}0,}{\{\text{true}\}\ X := 0\ \{X{=}0\}}$$

This step generates the verification conditions true $\supset$ 0=0 and X=0 $\supset$ X=0; both are true in the standard interpretation.

Although the Hoare logic has greatly influenced the field of program verification, there are a number of problems with the system[186]. One question is whether it is possible to create good Hoare axioms for all language constructs; unfortunately, the answer to this question is no[59]. Another problem is the incompleteness of the Hoare calculus[163]: there are valid formulas in the Hoare logic, in other words programs which are correct with respect to their specifications, for which no proof exists. This is related to the expressiveness of interpretations described earlier in this chapter; we do not feel it is a problem in practice.

A more fundamental problem lies with the notion of partial correctness itself. When a programmer defines and implements a function, it is equivalent to adding a new function to the basis of the underlying first–order logic. If the function is proved only partially correct we can not assume it is total. Most simple treatments of predicate logic assume that all functions are total[163]; therefore, a subtle inconsistency exists and the proof rules are incorrect[186]. This problem does not occur with total correctness.

During the construction of a proof in the Hoare calculus, verification conditions in the underlying first–order logic are generated when the consequence rule is used. It would enhance the construction of proofs if an automated tool could be used to certify the truth of these formulae.

2.4. Resolution Theorem Proving

Over the years this and similar problems have motivated the study of automatic theorem proving[54]. Earlier in the chapter, it was stated that partial decision procedures for the validity problem of first–order logic do exist; one particularly efficient procedure is called *resolution*[54]. Resolution is a *refutation procedure*: instead of proving a formula is true, it proves that the negation of the formula can not be true. Usually, a theorem prover will be used with a set of axioms that describe the interpretation of interest. If a formula is always false, or if it is inconsistent with the axioms being used, then assuming it is true will cause a contradiction. A theorem prover first negates the formula in question and then adds the negated formula to its list of assumptions. It then uses the *resolution principle* to generate all the logical consequences. If the assumptions contain a contradiction, then false will be a logical consequence; this demonstrates that the original formula is true.

Resolution operates on *clauses*, which are a standard form into which all formulae can be transformed[54]. A clause is a disjunction of *literals*, each of which is a predicate or its negation. For example, p, q(X), $p \vee q$, and $\neg p \vee q$ are all clauses. Briefly, the resolution principle states that if one clause contains a literal, and another clause contains the literal's negation, then the clause formed by combining both clauses, with the literal and its negation removed, is a logical consequence of the original clauses. For example, the clauses $p \vee q$ and $r \vee \neg q$ can be resolved to yield the clause $p \vee r$; this is equivalent to saying that $p \vee q \wedge r \vee \neg q \supset p \vee r$. More precisely, in resolution the literal and its negation do not have to match exactly; there must exist a substitution, or *unifier*, that makes the literals equal. This substitution is also applied to the new clause generated. For example, the clauses $p(X) \vee q(X)$ and $r \vee \neg q(1)$ can be resolved to yield the clause $p(1) \vee r$.

From another perspective, the enumeration of logical consequences performed by the theorem prover is a search for an interpretation in which all the assumptions hold; if no such interpretation exists then a contradiction exists. The prover does not consider all possible interpretations, only Herbrand interpretations are examined [54]. This is important because of the notion of equality it provides, two terms are equal only if they are identical; for example, 1+1 does not equal 2. Improving the performance of the prover is a classic problem in heuristic search[185,247]. Many search methods have been investigated[54]; in general, some form of breadth–first search is used.

2.5. Logic Programming

Soon after the resolution principle was presented, it was discovered that it could also be used both to execute and synthesize programs from logic–based specifications[101,102]. Although these applications are interesting, with conventional theorem provers they are too inefficient to be of practical importance. There is a great appeal to the idea of programming in logic: after writing a declarative specification, nothing more is necessary. The promise of these approaches led researchers to consider more efficient implementations; this resulted in the field now known as *logic programming*[40,68,151]. By trading off logical power for efficiency, more practical implementations can be produced; the flagship logic programming language is Prolog[60–62,152].

In Prolog, execution can be viewed as proving a formula of the form $\exists \overline{X}\ p(\overline{X})$ by finding an example $\overline{a}$ such that $p(\overline{a})$ is true. In order to dramatically increase Prolog's efficiency, several concessions were made. First, Prolog is based on a resolution theorem prover for *Horn clauses*[54,60]. A Horn clause may have at most one unnegated literal; for example, p, $p \vee \neg q$, and $p \vee \neg q \vee \neg r$ are all Horn clauses, while $p \vee q$ is not. Horn clauses allow a much more

efficient implementation, but represent only a subset of first–order logic; for example, the formulae $p \supset q \vee r$ can not be written using Horn clauses.

Prolog is also implemented using a depth–first search strategy; this allows a very efficient implementation and allows the programmer to control the search process in a simple manner. However, it makes Prolog *incomplete* as a theorem prover; in some cases an existing interpretation under which all the assumptions hold will not be found. Another concession to efficiency concerns the lack of an *occurs check* in the unification algorithm; this can result in invalid deductions[222]. Another limitation is that in Prolog there can be only one clause with no positive literal, or head; this is called the goal. Therefore, there is no way to state that a predicate is not true for a particular value; for example that $\neg p(1)$ is true. The solution used is the *closed world assumption*: if a goal is not provably true, then it is assumed to be false. While this is acceptable for Horn clauses, it can cause inconsistencies for full first–order logic[202].

The clauses in a Prolog program can be divided into *rules*, which contain negated literals, and *facts*, which do not. For example, consider the following Prolog program:

```
mother(bob,betty).
mother(sue,betty).
mother(betty,rose).
grandmother(N,M) ←
        mother(N,O),
        mother(O,M).
```

There are four clauses in the program. Mother(bob,betty), mother(sue,betty), and mother(betty,rose) are all facts; they state that betty is the mother of bob, betty is the mother of sue, and rose is the mother of betty respectively. The final clause is a rule; it states that for any persons M and N, M is the grandmother of N if there exists a person O such that O is the mother of N and M is the mother of O. The rule is equivalent to the formula grandmother(N,M) $\vee$ $\neg$mother(N,O) $\vee$ $\neg$mother(O,M). A Prolog implementation could be asked to find values of the

mother or grandmother relation. For example, given the query mother(bob,M), it would return with M equal to betty. Given the query grandmother(sue,G), it would return the with G equal to rose.

In this chapter we have presented the theoretical foundations of this thesis. First–order, predicate logic is a formalism for stating and proving statements about a set of objects; it is limited in that it is not possible to mechanically determine if a statement is true. Software can be specified using pre– and post–conditions written in first–order, predicate logic. The pre–condition states the properties that the inputs must satisfy, while the post–condition describes the acceptable outputs. A program can be proven correct with respect to a pre– and post–condition specification using the Hoare calculus; proofs in this method generate verification conditions in the underlying first–order logic. These formulae can sometimes be certified using resolution, a mechanical proof procedure for first–order logic. Resolution can also be used to synthesize programs and execute specifications; Prolog is a language based on a resolution theorem prover. Although these foundations are not perfect, we feel they are suitable for a next–generation software development environment.

CHAPTER 3.

RELATED WORK

In this chapter, we review some of the work performed by other researchers which is related to this dissertation. First, we present some of the specification methods which have been previously proposed, designed, or put into use; they can be formally–based or incorporate natural language and graphics. The formally–based methods may be roughly divided into model–oriented and axiomatic approaches, although languages which combine the two methods have also been proposed. PLEASE is a model–oriented approach; in other words, components are described in terms of pre–defined types and operations. As far as we know, it differs from other work in its combination of Ada, Prolog, and an environment supporting both model–oriented and informal specifications.

We then present some of the software development methodologies which have been proposed. For example, in transformational programming a very–high level specification is translated into an efficient implementation by a series of correctness preserving modifications. Artificial intelligence techniques can be applied to the transformational approach, or used with program schemas or plans. A more radical approach is termed "proofs as programs"; in this method, the development of a program is viewed as the creation of a proof in constructive logic. The work described in this dissertation is not based on a particularly unique development method; in fact it can be viewed as a transformational approach[23,24]. However, it is an attempt to integrate executable specifications and incremental verification into the traditional life–cycle. Work is now underway to extend ENCOMPASS to incorporate artificial intelligence techniques[229].

We then discuss some of the tools which have been developed to automate the software development process. These include systems for the verification of programs, environments for programming–in–the–small, environments for the construction of a program and its proof simultaneously, and environments for programming–in–the–large. As far as we know, ENCOMPASS is unique in its combination of tools and underlying technology. The system combines an incremental verification system and executable specifications based on resolution theorem proving with a test harness and an environment for programming–in–the–large.

3.1. Specification Methods

The *specification* precisely describes the properties and qualities that the software to be produced by a project must satisfy[88]. Parnas gives a number of reasons for the use of specifications in software development[194]. First, specifications allow a programmer to implement a component without understanding how the entire system works; by clearly defining component boundaries, the intellectual effort required for individual component construction is greatly reduced. Second, specifications support the construction of multi–version software; they can record both the capabilities required of all systems and the differences between versions. Third, specifications allow the description and verification of intermediate design decisions; many times, a choice of algorithm or data structure cannot be understood without the context provided by precise specifications.

One of the primary uses of a specification is as a medium for communication between the different people involved in the software development process. Customers, analysts, managers and programmers may have very different backgrounds and perspectives; this can make the choice of a communication medium difficult. Many methods for specifying software have been

proposed[11,92]; for the purposes of this discussion, we will divide them into formal and informal approaches. In our taxonomy, *formal* specifications are based on mathematics and/or logic, while *informal* specifications are based on natural language. Informal specification methods have a number of advantages. First, they are powerful in the sense that the full range of natural language is available to describe the system to be built. Second, formal specifications may be an ineffective medium for communication between customers and developers; with their combination of natural language and graphics, informal specifications are less intimidating to personnel without mathematical backgrounds. Formal specifications have the advantages associated with precise semantics. Natural language specifications are notoriously ambiguous, incomplete, and difficult to analyze automatically. Given a formal specification, it may be possible to automatically check it for ambiguity and incompleteness, generate an executable prototype, or verify an implementation.

3.1.1. Informal Specifications

Informal specifications rely mostly on natural language, but may be highly structured and incorporate graphics. For example, Structured Analysis (SA)[206] combines a blueprint–like notation with any other language to support top–down, hierarchical specification; these specifications can be methodically constructed using the Structured Analysis and Design Technique (SADT)[74,207]. In SA, specifications are divided into *data* and *activities*, which are decomposed independently. The basic unit of specification is the SA box, which can be decomposed to show more detail. In SA, arrows represent input, output, control and mechanism, thereby showing the relationship of boxes. In general, SA will be used with natural language.

A specification method may combine informal and formal components; for example, the Problem Statement Language (PSL)[226] combines a formal component similar to the entity–relationship model[57,58] with natural language. Both PSL and the Problem Statement Analyzer (PSA) are part of the ISDOS system. In PSL, systems consist of *objects* with *properties*; objects may have *relationships* between them. System descriptions include system input/output flow, system structure, data structure, data derivation, system size and volume, system dynamics, system properties, and project management. PSL contains a number of pre–defined objects and relationships geared towards description of these aspects. All the information from the specifications is stored in a project data base; PSA can automatically generate data base modification reports, reference reports, summary reports, and analysis reports.

PSL/PSA provides a good compromise between formal and informal specifications. PSL obtains much of its descriptive power from natural language; much of the information in a specification is stored in the names of objects and relations or in unprocessed text. However, PSL specifications can be extensively processed by PSA. While it cannot understand the significance of variable names or comments, it can perform simple analysis and check for some types of completeness and consistency.

3.1.2. Formal Specifications

Formal specifications have precisely defined semantics and are therefore more suitable for machine processing; however, they may be intimidating to personnel without a strong mathematics background. For the purposes of our discussion, we will divide formal specifications into model–oriented and axiomatic approaches. In a *model–oriented*, or *constructive* approach, a specification is created using pre–defined objects and operations; for example, a stack with opera-

tions push, pop and top might be defined in terms of a list with operations hd and tail. In an *axiomatic* approach a specification defines the relationships of operations to each other, without reference to pre–defined objects or operations; for example, an axiomatic specification of a stack might specify that the result of applying a push and a pop to a stack is equivalent to the original stack.

Many of the approaches to axiomatic specification are based on general algebras[90,97,98,107,181]; for example, the OBJ languages are based on an initial algebra approach[90,97,98]. In OBJ, new types or *sorts* may be added to a many–sorted equational logic; the relationships between the operations on the new sort are defined by a set of equations. The semantics are defined as the algebra that is initial in the category of all algebras for the logic. Some feel that the initial algebra approach adds an unnecessary implementation bias to the specification; therefore, approaches based on the final algebra in the category are also being investigated[139].

Languages such as OBJ have a number of strong points. For one, they are built on a very solid theoretical foundation; the semantics of the language and the deduction methods for the logic are well understood. Second, mechanical execution and proof procedures exist and are reasonably efficient. Their draw backs are that they are (arguably) not well suited to specifying some common programming constructs; for example, an algebraic approach is based on sets of values while software systems normally contain objects with internal states. While OBJ can be used to specify objects, much of the elegance and simplicity of the approach is lost.

In a model–oriented approach, components are specified in terms of pre–defined types and operations[15,26,29,105,118,162,193]; for example, ASLAN and RT–ASLAN are model–oriented languages for sequential and real–time systems respectively[15]. In both these languages, systems

are specified as state machines using predicate logic formulae; *invariants* state properties that all states must satisfy, while *constraints* describe requirements between consecutive states. The system generates the lemmas needed for an inductive proof of correctness. A specification consists of a sequence of *levels*; each level is a view of the system being described. The top level is a very abstract model of the system components, transitions, and requirements. Lower levels are more detailed; the lowest level might correspond to high–level code.

Languages such as ASLAN have a number of advantages. For one, they are (arguably) easy to write and understand; almost any software construct can be described using these methods. It is also reasonably easy to extend these methods with procedural and performance specifications, and mechanical execution and proof procedures do exist. The disadvantage of these methods is that they have unavoidable implementation bias: a specification is an implementation, although possibly in terms of very–high level primitives. In light of these problems, languages which combine both model–oriented and axiomatic approaches have been proposed[109,245].

The best specification language or method alone will not solve the problem of software development. Constructing a specification is just the first step; an implementation must then be created.

3.2. Development Methods

A number of methods for software development have been proposed[16,76,104,129,134,143,179,199,243,252]; many of these methods are explicitly or implicitly based on a particular model of the software life–cycle. A *life–cycle model* describes the sequence of distinct stages through which a software product passes during its lifetime[88]; there is no single, universally accepted model of the software life–cycle[8,17,20,36,254]. One point of contention

is how early in the development process an executable system should be produced. Creating a valid specification is a difficult task; the users of the system may not really know they want, and they may be unable to communicate their desires to the development team. It has been suggested that *prototyping* and the use of *executable specification languages* can enhance the communication between customers and developers[1,22,95,108,118,140,144,153,198,237,255–257]; providing prototypes for experimentation and evaluation should increase customer/developer communication and enhance the validation process. One technique is the use of a logic programming language such as Prolog for specification and/or prototyping[67,79,149]; this approach combines reasonably efficient prototypes with fairly declarative specifications.

Prototyping has been used on large projects; for example, the NYU Ada compiler was first prototyped in SETL and then rewritten in C[209]. In experiments comparing specifying and prototyping[38], it was found that prototyping required 45% less effort to produce systems with equivalent performance but 40% less code. On the other hand, it was found that systems which were specified rather than prototyped had more coherent designs and were easier to integrate. It was also discovered that systems produced using prototyping rated lower on functionality and robustness, but higher on ease of use and understanding.

Two widely used methods are *modular programming*[138,192,235,238] and *top-down development*[76,104,134,178,248]. By using *step-wise refinement*[248] to create a concrete implementation from an abstract specification, we divide the necessary decisions into smaller, more comprehensible groups. By encapsulating design and implementation decisions within module boundaries, the clarity and modifiability of software is increased. A number of modern programming languages support modular programming[70,157,161], and methods to support the top-down development of programs have been both devised and put into

use[13,35,41,42,64,134,159,179,206,213]. Environments to support such methods have also been both proposed and constructed[46,220,223,253].

Others have proposed that software development be viewed as a sequence of transformations between different, but somehow equivalent, specifications[17,18,45,56,158,184,195]. These transformations can be between specifications written in the same language or between different *linguistic levels*[158]. Systems can be based on on an extensible *catalog* of transformations, or on a small set which can be used to *generate* more complex modifications[195]. For example, Burstall and Darlington have developed a system in which first–order recursion equations are modified into a more efficient form by the application of correctness preserving transformations[45]. The system generates all modifications from six primitive transformations: definition, instantiation, unfolding, folding and laws (a set of data structure specific rules).

Other methods combine a *knowledge–base* and/or *artificial intelligence* techniques to support software engineering[16,19,100,165,212,215,240]. One such technique is *deductive synthesis*, the use of theorem proving techniques to create verified code from specifications[72,100,101,123,170]. As another example, IDeA[165] is an environment to support a data flow design and refinement technique using knowledge–based tools. In IDeA, design information is represented using reusable, domain–oriented *schemas*; the environment uses domain knowledge and various rules to assist the designer in the construction of designs from specifications.

The combination of artificial intelligence and transformational techniques can be termed *automatic programming*. For example, researchers at ISI have been working on an extended automatic programming paradigm for fifteen years[16]; this includes acquiring a high–level specification, validating the specification, and an interactive means of translating the specification

into an implementation. Their efforts use a specification language called GIST, which is based on an extended entity–relationship model. In another long term effort, a system has been built at the Kestrel Institute to support the transformational development of a predicate logic based specification language called V[215]. The system includes an integrated environment based on the language; the user can update or query a data base containing all the objects produced and used in the development process

Another approach views the development process as the creation of a proof in a constructive logic[21,63,172,221]; such a proof can be "executed" using an appropriate interpreter, or can be used to create an executable program. For example, the PRL system[21] supports the construction of proofs in a logic of the same name. The system provides an integrated environment for proof construction, including a "smart" editor and a hierarchical library of lemmas. The system allows users to ask "experts" for advice during the proof construction process; experts may implement guaranteed proof procedures, or be based on heuristic techniques.

Although many methods require no automated support, most can be enhanced by the use of specialized tools.

3.3. Tools

A number of different tools have been proposed, constructed, or used to enhance the software development process[55,77,110,114,137,177,210,215,218]. For example, the Cornell Program Synthesizer[227] is an environment for programming–in–the–small based on a language–oriented editor; it provides facilities to create, edit, execute and debug programs. The language–oriented editor is based on a *generator* approach; programs are created top–down by chosing and instantiating templates. Expressions are parsed as they are entered and structure–oriented com-

mands are supported. In more recent work, the Synthesizer Generator[204] allows a editor of this type to be generated from a language description. The editor designer specifies the constructs of the language, their relationships, how they are to be displayed, and the feedback to be given when errors are encountered; the Synthesizer Generator then creates a full-screen editor for manipulating programs in the language.

As another example, a number of different systems to support mechanical program verification have been proposed[112,166,203,253]; in such systems, the program and its proof can be created separately or simultaneously. For example, the Stanford Pascal Verifier[166] is an interactive system for program verification based on the Hoare logic. In this system, verification conditions are first generated from an annotated Pascal program and then submitted to algebraic simplification and proof methods. Verification conditions which can not be proved are then displayed for analysis by the programmer. Unproven conditions may indicate an error in the program, or merely the absence of necessary axioms or lemmas. The programmer may then modify the input to correct these deficiencies and repeat the generate, prove and inspect cycle until a complete proof is produced.

Although the Pascal Verifier is interactive, the construction of the program and its proof are really two separate processes. The synthesizer generator has been used to create a program/proof editor based on the Hoare calculus [203]. In this editor, a program and its partial correctness proof are constructed simultaneously; verification conditions are proved using the sequent calculus. In the system, proofs are treated as objects with constraints on them; rules of inference are implemented as attribute grammars. The editor checks the constraints on the proof after each editing command and keeps the user informed of errors and inconsistencies.

It may be difficult to integrate and coordinate the different tools used in a software engineering process; a number of tools can be integrated into a *software engineering environment*[6,33,37,50,56,69,73,78,99,111,127,148,180,228,239]. The integration provided by an environment can create a synergistic effect between the environment's components; the support provided by the environment is greater than the sum of that provided by its individual parts. A high degree of integration can be achieved by basing an environment on a particular development paradigm; since the methods to be employed are known, more support can be provided for a larger part of the development process.

An environment can be tightly integrated, or based on a number of small, composible tool fragments. An example of the latter approach is Toolpack[65,189], an environment to support the production, testing, transportation and analysis of mathematical software written in Fortran. Toolpack contains a number of tools including a compiling/loading system, an intelligent editor, a formatter, a structurer, a dynamic testing and validation aid, a dynamic debugging aid, a static error detection and validation aid, a static portability checking aid, a document generation aid, and a program transformer. The tools are integrated using a common file system and command interpreter.

An example of a tightly integrated environment is Cedar [224,225], which incorporates high quality graphics, a sophisticated editor and document preparation facilities, and a number of other tools such as an interpreter and debugger. The Cedar environment supports development using the Cedar language, a strongly typed, compiler oriented language of the Pascal family. The project may be seen as an attempt to bring features found in environments for dynamically typed languages such as Smalltalk or Lisp into an environment for languages such as Pascal.

The growing interest in Ada has prompted a number of projects to investigate environments to support the language[46,220,249]. For example, Arcturus[220] is an environment combining tools for template assisted editing, performance measurement, and automatic formating. The system supports an Ada-based Program Design Language (PDL) and automated refinement from PDL into executable code. Another example is PIC[249], an environment for programming-in-the-large using Ada. It provides tools to specify the structure of large systems using either graphical or textual representations. The environment allows easy movement between these forms and provides an integrated set of tools for analyzing and managing the interface control aspects of large systems.

In this chapter, we have reviewed some of the work performed by other researchers which is related to this dissertation. PLEASE is a formal, model-oriented specification language; in other words, components are described in terms of pre-defined types and operations with precise semantics. As far as we know, it differs from other work in its combination of Ada, Prolog, and an environment supporting both model-oriented and informal specifications. Many different software development methodologies have also been proposed. The work described in this dissertation is not based on a particularly unique development method; in fact it can be viewed as a transformational approach. However, it is an attempt to integrate executable specifications and incremental verification into the traditional life-cycle. Work is now underway to extend ENCOMPASS to incorporate artificial intelligence techniques. Many different tools have been developed to automate the software development process. As far as we know, ENCOMPASS is unique in its combination of tools and underlying technology. The system combines an incremental verification system and executable specifications based on resolution theorem proving with a test harness and an environment for programming-in-the-large.

CHAPTER 4.

THE PLEASE LANGUAGE

The first step in the program of research described in this thesis is the design of a wide-spectrum, executable specification language to support incremental software development. This language is called PLEASE (Predicate Logic based ExecutAble SpEcifications). PLEASE permits the construction of programs in a conventional language using rapid prototyping and incremental verification techniques. In PLEASE, software components are specified using pre- and post-conditions written in a subset of first order, predicate logic. Prototypes can be automatically constructed from these specifications, and the specifications can be incrementally refined into conventional implementations. Each refinement can be verified before another is applied; therefore, errors can be detected sooner and corrected at lower cost.

In this chapter, we describe PLEASE in some detail. First, we present the software development paradigm PLEASE is designed to support; it is basically the traditional life-cycle extended to support the use of executable specifications and VDM. Next we discuss the tradeoffs present in the design of PLEASE: logical power, ease of use, applicability and efficiency were all considered. We then present and discuss an example specification and program; this allows us to describe the main features of our approach. Next, we present the Hoare-style proof rules for the basic constructs in PLEASE including assignment and *if-then-else* statements, *while* loops, procedure calls and user-defined functions. These rules assume variables of a single type; we then describe the pre-defined types in PLEASE and the facilities provided for user type definition. Finally, we describe how PLEASE can be used to specify objects, in other words encapsulated types with an internal state.

4.1. Development Paradigm

Figure 2 shows the software development paradigm PLEASE was designed to support; a different perspective is presented in Chapter 8. In this 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[88]. A *systems analyst* produces a *software requirement specification*[88], 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*[230], 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 that 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 can improve customer/developer communication and enhance the validation process. If it is found that the specification does not satisfy the customers, then it is

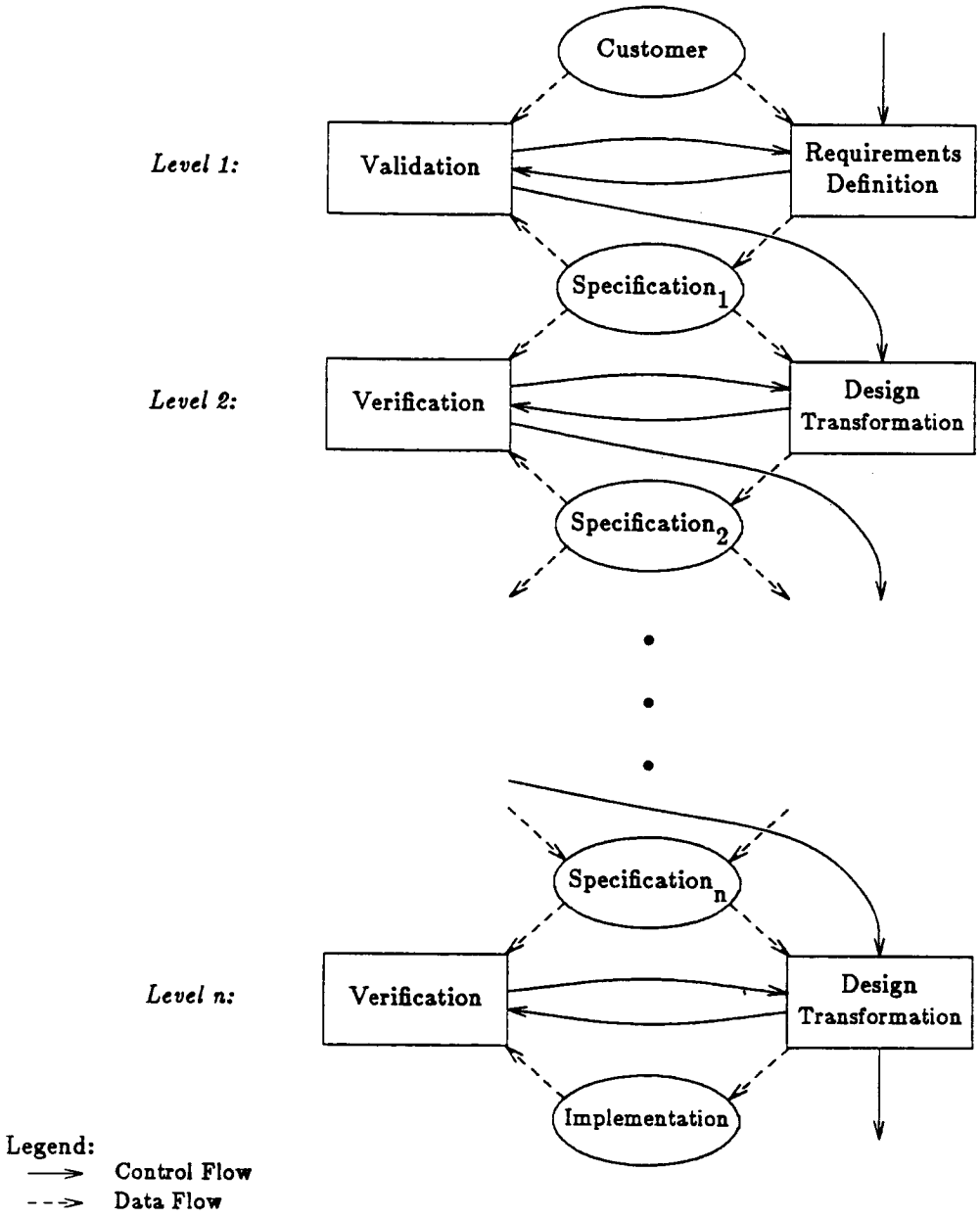


Figure 2. Software development paradigm

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 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*[88], 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*[134] 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 develop-

ment is located at a given node. A design decision chooses an arc 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 that 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 that 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" [244].

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.

Now that we have seen the development paradigm PLEASE is designed to support, we can better understand the tradeoffs involved in its design.

4.2. Design Considerations

The design of PLEASE is a compromise between a number of conflicting goals. First, the language must allow the implementation of software using a conventional programming

language. Early experiments were performed using Path Pascal[51], a variant of Pascal in which the interactions between concurrent processes are specified using *path expressions*[49]. Although this language provided valuable insights, it had a number of drawbacks. Pascal itself does not provide support for modular programming; Path Pascal provides *objects* which were designed with concurrent programming rather than modularity in mind. Although a reasonable implementation is available on Berkeley Unix®, the language is not widely implemented or used.

The examples given in this thesis use Ada [70,241] as the implementation language; it seems a good choice for a number of reasons. Ada is (arguably) a well designed, modern language. It contains more than enough features, including support for modular programming. Others are researching Ada–based specification languages[167]; much of their work can be reused. Ada is enthusiastically supported by the Department of Defense. Commercial compilers have already been produced and it seems likely that more will be developed for many different architectures. Ada is currently in industrial use and promises to become widely used in the near future[182]; this decreases the distance between the somewhat academic work described in this thesis and the real world of software development.

The second design requirement is that PLEASE must allow the specification of software using pre– and post–conditions written in predicate logic; the more powerful the specification method, the better. Third, the language must allow the rapid, automatic construction of executable prototypes from these specifications; the prototypes should be as efficient as possible. Unfortunately, there is a conflict between the second and third goals. A fairly powerful specification method would use pre– and post–conditions written in the full first–order, predicate logic. These specifications would use a number of very high–level data types such as sets. Unfortunately, the

Unix® is a trademark of AT&T.

validity problem for first–order logic is undecidable. Therefore, if we allow full first–order logic to be used for the specifications, we will be unable to construct totally correct prototypes in all cases.

A resolution theorem prover for first–order logic could be used to construct partially correct prototypes; however, the performance of these prototypes would be very poor. The axiom sets for types such as sets would result in a further decrease in performance. The emergence of logic programming as a technology, most notably Prolog[60], suggests that Horn clauses may provide a good compromise. Although not as powerful as full first–order logic, Horn clauses allow much more efficient implementation techniques to be used. Commercial Prolog implementations are available which provide support for machine types such as integers and floating point numbers[5]. By restricting the types used to those with efficient Prolog implementations, reasonable specification power is combined with implementation efficiency.

The fourth design goal is that PLEASE specifications can be incrementally refined into verified implementations. Given Ada as the base language, problems arise. Ada was not designed with program verification as a goal; therefore, it contains constructs for which no formal semantics have been developed. For example, the Ada Language Reference Manual (ALRM)[70] states that *in out* parameters may be implemented using either a copy/restore or pointer strategy. Most of the work on Hoare axioms for procedure call assumes one implementation or the other.

Although the examples in this thesis use Ada syntax, the constructs do not necessarily have the Ada semantics; the semantics of PLEASE are defined using Hoare calculus proof rules. PLEASE is designed for use only within an encapsulated environment; special tools manipulate and display the abstract syntax in a format suitable for humans. In the current implementation,

Ada programs with behavior matching the semantics defined by the proof rules are created automatically from the PLEASE abstract syntax trees. This approach allows programs in different languages to be created from the same abstract syntax, and constructs to be provided which are implementable, but not supported, in the implementation language.

In order to further clarify the concepts, design and implementation of PLEASE, we will present and discuss an example specification and program.

4.3. An Example

Figure 3 shows the PLEASE specification of a component to compute the factorial of a given number. In Ada, *packages* are used to group logically related components[70,241]. The

```
package factorial_pkg is

     --: predicate is_fact( X,Y : in out 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 ),
          --|       out( is_fact(X,Y) ) ;

end factorial_pkg ;
```

Figure 3. Specification of procedure *factorial*

specification defines a package *factorial_pkg*, which provides a procedure *factorial*. To increase readability and understandability, the syntax of PLEASE is similar to Anna[167,168]. Like Anna, a concrete PLEASE program is an Ada program with *formal comments*, which are ignored by Ada compilers but are meaningful to other tools in the IDEAL environment. Formal comments are divided into virtual program text, each line of which begins with the symbol "--:", and annotations, each line of which begins with "--|". In PLEASE, *annotations* contain predicate logic formulas which should be true at various points in the program's execution; for example, annotations are used to write pre- and post-conditions. In PLEASE, *virtual program text* defines constructs that are used only in assertions, not in the actual program being constructed; for example, virtual program text is used to define predicates needed to write assertions.

The specification in Figure 3 has two parts: the definition of the predicate *is_fact* and the specification of the *factorial* procedure. In PLEASE, a *predicate* syntactically resembles a procedure and may contain local type, variable, function or predicate definitions. The declaration of a new predicate is equivalent to extending the basis of the underlying first-order logic; the predicate definition is translated into axioms which are added to the set defining the theory for the basis. For example, the definition of *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 $(X{=}0 \wedge Y{=}1) \vee ((X{-}1)!{=}T1 \wedge Y{=}T1{*}X)$).

Predicates are specified using Horn clauses; this approach allows a simple translation from predicate definitions into Prolog procedures. A major drawback is that pure Horn clause programming has no way to specify the falsehood of formulae; for example, the fact that *is_fact(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[202]; therefore, there is no way to specify negative information in PLEASE. At present, the best solution using PLEASE is to define a new predicate that is understood to be the negation of the predicate in question. Unfortunately, this relationship can not be recorded in a formal manner. We feel that pure Horn clauses are inadequate as the basis for a practical specification language; therefore, we plan to extend PLEASE to support a more powerful logic.

The second part of the specification states that *factorial* is a procedure with an input parameter *X* and an output parameter *Y*, both of type *natural*. In PLEASE, the state before execution begins is designated by *in(...)*, while the state after execution is complete is designated by *out(...)*. Procedures are defined using pre– and post–conditions; the pre–condition for a procedure specifies the conditions the input data must satisfy before procedure execution begins, while the post–condition for a procedure states the conditions the output data must satisfy after procedure execution has completed. The pre– and post–conditions can be expressed as formulae surrounded by *in(...)* and *out(...)* respectively. For example, the pre–condition for *factorial* is *true*; the type declarations for the parameters give all the requirements for the input. 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. This could be written in Hoare style notation as:

$$[\text{true}]\ S_1\ [\text{is_fact}(X,Y)]$$

where S_1 is the body of *factorial*.

The PLEASE specification of *factorial* can serve many purposes during the development of the procedure. In the validation phase, the specification can serve as the basis for precise discussions between customers and developers. The specification can be used to produce an executable prototype, which can be delivered to the customers for experimentation and evaluation, or

possibly even installed for use on a trail basis. The specification enhances the verification of refinements using either testing or proof techniques. The prototype produced from the specification can be used as a test oracle against which an implementation can be compared; since the specification is formal, proof techniques can also be used to verify implementations.

For example, Figure 4 shows a complete implementation of the *factorial* procedure. The implementation first checks for the special case of zero, and then uses a *while* loop to calculate the factorial of X in Y. The loop has invariant *is_fact(I,Y)* and terminates when I is equal to X. The body of *factorial* is completely annotated; in other words, there is an assertion both before and after each executable statement. Each assertion states the conditions that must be satisfied whenever execution reaches that point in the procedure. For example, when execution enters the *then* branch of the *if–then–else* the assertion $true \wedge X=0$ must hold. The assertions plus the executable statements form a proof in the Hoare calculus[120,163,169]; this approach allows a proof of correctness to be constructed when a specification is refined into an implementation.

Now that the basic concepts and philosophy of PLEASE have been presented, we can describe the language in a more formal manner.

4.4. Proof Rules

We describe the semantics of PLEASE using Hoare calculus proof rules. There are two things to notice about our general approach. First, the rules are stated in terms of total correctness; this avoids the problems inherent in the partial correctness of user–defined functions[186]. Second, PLEASE does not allow functions with side effects.

```
package body factorial_pkg is

    procedure factorial( X : in natural ; Y : out natural ) is
        --| where in( true ),
        --|       out( is_fact(X,Y) ) ;

        I : natural ;

    begin
        --| true ;
        if X = 0 then
            --| true and X = 0 ;
            Y := 1 ;
            --| is_fact(X,Y) ;
        else
            --| true and not X = 0 ;
            I := 1 ;
            --| I = 1 ;
            Y := 1 ;
            --| is_fact(I,Y) ;
            while I /= X loop
                --| is_fact(I,Y) and I /= X and X-I = c1 ;
                I := I + 1 ;
                --| is_fact(I-1,Y) and X-I+1 = c1 ;
                Y := Y * I ;
                --| is_fact(I,Y) and X-I < c1 ;
            end loop ;
            --| is_fact(X,Y) ;
        end if ;
        --| is_fact(X,Y) ;
    end factorial ;

end factorial_pkg ;
```

Figure 4. Annotated implementation of *factorial*

One of the most basic constructs in PLEASE is the *assignment statement*; its semantics are defined by the axiom schema:

$$[P^{e}_{X}]\ X := e\ [P]$$

where P is a formula, X is a variable, and e is an expression.

In other words, for the formula P to be true after execution of the statement assigning e to X, the formula P with e substituted for X must be true before execution begins.

PLEASE also includes the *if–then–else statement*; its semantics are defined by the rule:

$$\frac{[P \wedge E]\ S_1\ [Q],\ [P \wedge \neg E]\ S_2\ [Q]}{[P]\ \textit{if}\ E\ \textit{then}\ S_1\ \textit{else}\ S_2\ \textit{end if}\ [Q]}$$

where P and Q are formulae, E is a quantifier free formula, and S_1,S_2 are statements.

In other words, for an *if–then–else* statement with branch condition E to be correct with respect to pre–condition P and post–condition Q, the *then* branch must be correct with respect to $P \wedge E$, Q and the *else* branch must be correct with respect to $P \wedge \neg E$, Q.

PLEASE also includes a *while loop* construct; its proof rule is more complicated because non–termination is possible. Ignoring the problem of non–termination, the semantics of the *while* loop are defined by the following rule:

$$\frac{\{P \wedge E\}\ S_1\ \{P\}}{\{P\}\ \textit{while}\ E\ \textit{loop}\ S_1\ \textit{end loop}\ \{P \wedge \neg E\}}$$

where P is a formula, E is a quantifier free formula, and S_1 is a statement.

In other words, for a *while* loop with condition E to be partially correct with respect to pre–

condition P and post-condition $P \wedge \neg E$, the body of the loop must be partially correct with respect to $P \wedge E$ and P.

We can extend the rule to total correctness:

$$\frac{P \wedge E \supset W_X^e,\ [P \wedge E \wedge e{=}c]\ S_1\ [P \wedge e < c]}{[P]\ \textit{while}\ E\ \textit{loop}\ S_1\ \textit{end loop}\ [P \wedge \neg E]}$$

where P and W are formulae, E is a quantifier free formulae, e is an expression, c is a constant not used elsewhere, and S_1 is a statement.

The rule is fully explained in[163]. Briefly, it relys on a ***well founded set*** defined by the formula W; if W_X^e is true, then e is a member of the set. Since there are no infinite decreasing sequences in a well founded set, the fact that e equals c before the body of the loop begins and e is less than c after the body completes implies that the loop will terminate.

Statements can also be sequentially composed in PLEASE; the semantics are defined by the usual Hoare rule:

$$\frac{[P]\ S_1\ [R],\ [R]\ S_2\ [Q]}{[P]\ S_1\ ;\ S_2\ [Q]}$$

where P,Q,R are formulae and S_1,S_2 are statements.

In other words, for the sequence of two statements S_1 and S_2 to be correct with respect to pre-condition P and post-condition Q, there must exist a formula R such that S_1 is correct with respect to P, R and S_2 is correct with respect to R, Q.

PLEASE also includes procedure calls; the semantics can be defined with the rule:

$$\frac{[I \wedge (Q_1)^{A}_{X} \supset (Q_2)^{Y,Z}_{B,C}],\ [P_1]\ S_1\ [Q_1]}{[(I \wedge P_1)^{A,B}_{X,Y}]\ p(A,B,C)\ [Q_2]}$$

where P,Q,I are formulae, X,Y,Z are variables, A,B,C are expressions, and p is a procedure with statement S_1 as its body.

Note that A, B and C are disjoint, I does not depend on B or C, and that the rule must be extended for recursive procedures; for a more complete discussion see[171].

This rule requires considerable explanation. First, we must differentiate between the different types of formal parameters. In PLEASE, formal parameters may be of type *in*, in which case their value may be referenced but not set in the procedure; type *out*, in which case their value is copied to the actual parameter when the procedure returns; or type *in out*, in which case the value of the actual parameter is copied to the formal parameter when the procedure is called, and the value of the formal parameter is copied back to the actual when the procedure returns. In other words, PLEASE has *copy–restore* or *value–result* semantics for parameter passing.

In the rule above, X represents the *in* formal parameters, Y represents the *in out* formal parameters and Z represents the *out* formal parameters. Similarly, A, B and C are the *in*, *in out* and *out* actual parameters respectively. S_1 is the body of the procedure p and is correct with respect to pre–condition P_1 and post–condition Q_1. The proof of the procedure call rule is based on the equivalence between the call p(A,B,C) and the statement sequence:

```
X := A ;
Y := B ;
S1 ;
B := Y ;
C := Z ;
```

The formula I is called the *invariant*; it must be true both before and after the procedure is called. The invariant states properties which may be necessary to prove other parts of the program, but are not necessary for the correct execution of the procedure call. The rule states that if the procedure body is correct with respect to P_1,Q_1 and the invariant and Q_1, both with the *in* actuals substituted for the formals, imply the post–condition for the call with the *out* formals substituted for the actuals, then if the invariant and P_1, both with the *in* actuals substituted for the formals, are true before the call begins, then the call will terminate so that the post–condition will be true.

For example, consider the *factorial* procedure specified earlier in this chapter; its body has pre–condition true and post–condition is_fact(X,Y). A proof of the correctness of a call factorial(I,J) with respect to invariant is_fact(K,L), and post–condition is_fact(K,L) ∧ is_fact(I,J) would reduce to the following:

$$\frac{[\text{is_fact}(K,L) \wedge \text{is_fact}(I,Y) \supset \text{is_fact}(K,L) \wedge \text{is_fact}(I,Y)],\ [\text{true}]\ S_1\ [\text{is_fact}(X,Y)]}{[\text{is_fact}(K,L) \wedge \text{true}]\ \text{factorial}(I,J)\ [\text{is_fact}(K,L) \wedge \text{is_fact}(I,J)]}$$

where S_1 is the body of *factorial*

PLEASE also supports user–defined functions; the semantics of which can be explained with the following rule:

$$\frac{[P]\ S_1\ [Q]}{P \supset Q_F^{f(X)}}$$

where P,Q are formulae and f is a function with body S_1 and return variable F.

In other words, if the function f has body S_1 which is correct with respect to pre–condition P and

post–condition Q, then for any X, P implies Q with f(X) substituted for the return variable. For example, the body of the *factorial* procedure could also be used as the body of a function. Since the pre–condition is true and the post–condition is is_fact(X,Y), we can deduce that true $\supset$ is_fact(X,f(X)); this can be used as an axiom with which to reason about the function.

The proof rules presented in this section assume that variables are of a single type; while this is sufficient for many purposes, PLEASE currently provides significantly more power.

4.5. Data Types

The current version of PLEASE provides a number of pre–defined types which may be composed to form larger constructs: characters, naturals and lists are supported. In PLEASE, as in Ada, the type *natural* implements a finite subset of the natural numbers; the operations addition, subtraction, multiplication and division are pre–defined. The PLEASE type *character* implements the ASCII character set and has no pre–defined operations except equality, which is defined on all types. In PLEASE, *list* is a generic; an instantiation of *list* will have elements of a particular type. Lists have pre–defined operations cons, hd, tail, append, extract, and length.

For example, Figure 5 shows the please specification of a component to sort a list of natural numbers; 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 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, in PLEASE 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

```
with natural_list_pkg ; use natural_list_pkg ;

package sort_pkg is

     --: predicate permutation( 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
     --:           permutation(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( permutation(Input,Output) and sorted(Output) ) ;

end sort_pkg ;
```

Figure 5. Specification of *sort* procedure

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 || L_2 denotes the concatenation of the elements of L_1 and L_2.

The specification defines the predicates *permutation* and *sorted*, as well as giving pre– and post–conditions for the procedure. The definition of 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 definition of the predicate *sorted* states that a list is sorted if the list is empty, or if the list has one element, or if the first element in the list is less than or equal to the second element and the tail of the list is sorted. 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.

While the pre–defined types in PLEASE provide reasonable expressive power, even more can be obtained with the addition of a user type definition facility.

4.5.1. User–Defined Types

It has been proposed that the use of *abstract data types* can enhance software specification, validation and verification[98,107,139,162,181]. In PLEASE a new type can be defined with another type as its *representation*; for example, Figure 6 shows the PLEASE specification of an Ada package defining the type *natural_stack* with representation *natural_list*. *Natural_stack* implements a stack of natural numbers using a list of natural numbers; each object of type *natural_stack* is represented by an object of type *natural_list*. In PLEASE, as in VDM[134], 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 type are specified with pre– and post–conditions in a manner similar to procedures. For example, the function *top* has S /= *empty_stack* as a pre–condition; the function is only defined on stacks with at least one element[1]. The post–condition for *top*

[1]This makes *top* a partial function, creating a subtle inconsistency.

```
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 push( E : natural ; S : natural_stack )
                return natural_stack ;
           --| where return Ns:natural_stack => Ns = cons(E,S) ;

        function pop( S : natural_stack ) return natural_stack ;
           --| where in( S /= empty_stack ),
           --|        return Ns:natural_stack => Ns = tl(S) ;

        function top( S : natural_stack )  return natural ;
           --| where in( S /= empty_stack ),
           --|        return E : natural => E = hd(S) ;

end natural_stack_pkg ;
```

Figure 6. *Natural_stack* in terms of *natural_list*

states that the value returned by the function is the head of the list given as an argument. As mentioned in the previous section, 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.

Natural_stack_pkg defines four functions on the type *natural_stack*. The function *empty_stack* returns an empty list to be interpreted as an empty stack, while 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* 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.

In PLEASE, both pre– and user–defined types are types in the mathematical sense: sets of values with functions. Although this gives reasonable expressive power and good formal properties, encapsulated components with internal state are necessary for the specification of large software systems.

4.6. Objects

Many feel that *object–oriented techniques* can greatly enhance the specification, design and implementation of software[7,39,175]. The definition of "object–oriented" is somewhat vague; however, it usually contains the notions of state, encapsulation, inheritance, and specialization. An object has a *state* which is internal, changeable and retained between operations on the object; this differentiates it from a value which is not changeable. An object is *encapsulated* if its state can only be modified by certain operations; in other words if the representation is hidden. In a system with *inheritance*, objects can inherit operations or values from (possibly multiple) parents. This is subtly different from *specialization*, in which an object can override as well as inherit the properties of its parent. PLEASE provides support for state, encapsulation and inheritance, but not specialization.

In PLEASE, objects are specified using Ada packages; the local variables form the state which can be modified only by procedures defined in the package. A package may *use* other packages to implement a simple type of inheritance. To better understand the specification of

objects in PLEASE, we will consider an example[144], a small library data base which supports the following transactions:

1 – Check out or return a book.
2 – Add or remove a book from the library.
3 – Get all the books by a particular author.
4 – Get the books checked out to a borrower.
5 – Find out who has a book checked out.

Users of the system are divided into *staff* and *non-staff* categories. Only staff users can perform transactions one, two, four or five, except that any one can perform transaction four to find the books they currently have checked out. The data base must satisfy the following integrity constraints:

1 – All books must be available or checked out.
2 – No book may be both available and checked out.
3 – No user may have more than a set number of books checked out at any time.

The PLEASE specification of the library data base is a single package including both the data structures and operations. Figure 7 shows the PLEASE specification of the data structures for the library data base. The specification of the library data base uses the PLEASE type *list* to define the type *book_list* as *list of book*. The type *book* defines a record including fields for unique identifier, title and author. The type *borrower* defines a record including name and number of books checked out. The type *check_out_rec* defines a record that relates a borrower and a book. The data base consists of four data structures. *Shelf_list* is a list of all the books owned by the library, while *Available* is a list of all the books currently available for check out. *Checked_out* contains a record of each book currently checked out, while *Borrowers* records the number of books currently checked out by each borrower.

```
type book is record
        id     : book_id ;
        title  : strng ;
        author : strng ;
    end record ;

type borrower is record
        name            : user ;
        num_checked_out : natural ;
    end record ;

type check_out_rec is record
        u  : user ;
        bk : book ;
    end record ;

type book_list      is list of book ;
type borrower_list  is list of borrower ;
type check_out_list is list of check_out_rec ;

Shelf_list  : book_list ;
Available   : book_list ;
Checked_out : check_out_list ;
Borrowers   : borrower_list ;
```

Figure 7. Specification of the library data base

The integrity constraints on the data base are expressed as an *invariant*; the invariant must be true both before and after any transaction is performed. Figure 8 shows the PLEASE specification of the invariant for the library data base. Figure 8 contains an assertion that specifies the invariant for the data base, as well as virtual program text which defines a number of predicates used in the invariant. The function *books_checked_out* returns a list of all the books currently out to any borrower; its specification uses the predicate *is_books_checked_out*, which is true only if the list L contains all the books currently on *Checked_out*. The predicate *all_available_or_checked_out* is true if all books on the shelf list are either available or checked

```
--: predicate is_books_checked_out(
--:     L           : in out book_list ;
--:     Checked_out : in out check_out_list
--: ) is true if
--: begin
--:     Checked_out = [] and L = []
--:         or
--:     hd(L) = hd(Checked_out).bk and
--:     is_books_checked_out(tl(L),tl(Checked_out))
--: end ;

function books_checked_out return book_list ;
    --| where  return L : book_list => is_books_checked_out(L,Checked_out) ;

--: predicate all_available_or_checked_out is true if
--: begin  permutation(Shelf_list, Available || books_checked_out)  end ;

--: predicate none_available_and_checked_out is true if
--: begin  member_both(Available,books_checked_out) = []  end ;

--: predicate under_limit( Borrower : in out borrower ) is true if
--: begin  Borrower.num_checked_out <= borrow_limit  end ;

--: predicate all_under_limit( Borrowers : in out borrower_list ) is true if
--: begin
--:     Borrowers = []
--:         or
--:     under_limit(hd(Borrowers)) and all_under_limit(tl(Borrowers))
--: end ;

--| where all_available_or_checked_out,
--|       none_available_and_checked_out,
--|       all_under_limit(Borrowers) ;
```

Figure 8. Invariant for the library data base

out, while the predicate *none_available_and_checked_out* is true if no book is both available and checked out. The predicate *all_under_limit* is true is all borrowers currently have less than the maximum number of borrowed books. The invariant for the library data base states that the predicates *all_available_or_checked_out*, *none_available_and_checked_out*, and *all_under_limit(Borrowers)* must be true both before and after the execution of each operation. The is equivalent to having the conjunction of these predicates in both the pre– and post–condition of each operation.

Figure 9 shows the specification of procedures to perform the check in and check out transactions. The procedure *check_out* implements the check out operation; it is called with the identity of the user performing the operation, as well as the identity of the borrower and book in question. The pre–condition for *check_out* states that the user performing the transaction must be a staff member and that the book to be checked out must be available. The post–condition

```
procedure check_out( U : in user ; B : in borrower ; Bk : in book ) ;
    --| where in( is_staff(U) and is_available(Bk) ),
    --|       out( book_is_checked_out(B.name, Bk, in(Checked_out)) and
    --|            Available = extract(in(Available),Bk) and
    --|            borrower_is_updated(B,1,in(Borrowers)) and
    --|            under_limit(Borrower) ) ;

procedure check_in( U : in user ; B : in borrower ; Bk : in book ) ;
    --| where in( is_staff(U) and is_checked_out(Bk) ),
    --|       out( Available = cons(Bk,in(Available)) and
    --|            book_is_not_checked_out(B.name,Bk,in(Checked_out)) and
    --|            borrower_is_updated(B,-1,in(Borrowers)) ) ;
```

Figure 9. Procedures to check books in and out

for *check_out* states that the book must be checked out to the borrower, that the book must not be on the available list[2], that the borrower's record is updated to reflect the new book checked out, and that the borrower is still under the limit for number of books checked out. The procedure *check_in* is similar. The pre–condition for *check_in* states that the user performing the transaction must be a staff member and that the book in question must be checked out, while the post–condition for *check_in* states that the book must be on the available list, that the book is not checked out to the borrower, and that the borrower's record is updated to reflect the check in.

When a book is checked out of the library, the state of the library data base is changed; the specification of *check_out* refers to the state of the data base both before and after the operation is performed. A predicate is evaluated in a single state; therefore, in order to refer to both the initial and final states, the value of one of the states must be passed as a parameter. For example, in the post–condition for *check_out* the initial value of *Checked_out*, denoted by *in(Checked_out)*, is used as an argument to the predicate *book_is_checked_out*. This allows the predicate to reference both the initial and final values of the list.

Figure 10 shows a number of user–defined predicates which are used in the specification of *check_out*. All the predicates take the initial value of a variable as one of their parameters; they are true if the current value reflects correct modifications of the original. For example, the predicate *book_is_checked_out* takes the initial value of *Checked_out* as a parameter, called *Checked_out_0*, and is true if the current value of *Checked_out* is the initial value plus the new check out record. The predicate *book_is_not_checked_out* is similar, but is true if the check out record has been removed. The predicate *borrower_is_updated* is true if the number of books out to the borrower has been updated correctly.

[2]In PLEASE, the function *extract(list,member)* returns a list with all instances of *member* removed.

```
--: predicate book_is_checked_out(
--:     B             : in out user ;
--:     Bk            : in out book ;
--:     Checked_out_0 : in out check_out_list
--: ) is true if
--:     New_record : check_out_rec ;
--: begin
--:     New_record.u = B and New_record.bk = Bk and
--:     Checked_out = cons(New_record,Checked_out_0)
--: end ;

--: predicate book_is_not_checked_out(
--:     B             : in out user ;
--:     Bk            : in out book ;
--:     Checked_out_0 : in out check_out_list
--: ) is true if
--:     New_record : check_out_rec ;
--: begin
--:     New_record.u = B and New_record.bk = Bk and
--:     Checked_out = extract(Checked_out_0,New_record)
--: end ;

--: predicate borrower_is_updated(
--:     B           : in out borrower ;
--:     Inc         : in out integer ;
--:     Borrowers_0 : in out borrower_list
--: ) is true if
--:     New_b : borrower ;
--:     Borrowers_tail : borrower_list ;
--: begin
--:     Borrowers_tail = extract(Borrowers_0,B) and
--:     New_b.name = B.name and
--:     New_b.num_checked_out = B.num_checked_out + Inc and
--:     Borrowers = cons(New_b,Borrowers_tail)
--: end ;
```

Figure 10. Predicates to support check in operation

Figure 11 shows the specification of procedures that add books to or remove books from the library. The procedure *add_book* adds a new book to both *Available* and *Shelf_list*, while *remove_book* removes a book from them. Figure 12 shows the PLEASE specification of a function that returns a list of all the books by a particular author. There is no pre–condition specified for this function; the type declarations for the parameters and the invariant for the data base give all the requirements for the input. The post–condition for the function specifies that any list returned must satisfy the predicate *all_by_author*, which defines the relationship between *Shelf_list* and the list to be produced. If *Shelf_list* is empty, then *List* is also; otherwise, if the author of a book on *Shelf_list* is the author in question then the book is on *List*.

Figure 13 shows the specification of functions that return the borrower to whom a book is checked out, as well as the list of all books checked out to a borrower. The function *what_checked_out* takes a borrower as input and returns the list of all books checked out to him; its specification uses the predicate *out_to_borrower*, which is true if *List* contains all the books

```
procedure add_book( U  : in user ; Bk : in book ) ;
    --| where in( is_staff(U) ),
    --|       out( Available = cons(Bk,in(Available)) and
    --|            Shelf_list = cons(Bk,in(Shelf_list)) ) ;

procedure remove_book( U  : in user ; Bk : in book ) ;
    --| where in( is_staff(U) and is_available(Bk) ),
    --|       out( Available = extract(in(Available),Bk) and
    --|            Shelf_list = extract(in(Shelf_list),Bk) ) ;
```

Figure 11. Procedures to add and remove books

checked out to a borrower B and no others. The function *who_has* returns the borrower to whom a particular book is checked out; its specification uses the predicate *has_book* which is true if book Bk is indeed checked out to borrower B.

In this chapter, we have described PLEASE in some detail. PLEASE is designed to support a software development paradigm which is basically the traditional life–cycle, extended to support the use of executable specifications and VDM. The design of PLEASE is a tradeoff between logical power, ease of use, applicability and efficiency. PLEASE allows the specification of software using Horn clauses, an efficiently implementable subset of first–order logic. The

```
--: predicate all_by_author(
--:         Shelf_list : in out book_list ;
--:         Author     : in out strng ;
--:         List       : in out book_list
--: ) is true if
--:     Tail : book_list ;
--: begin
--:     Shelf_list = [] and List = []
--:         or
--:     hd(Shelf_list).author = Author and
--:     all_by_author(tl(Shelf_list),Author,Tail) and
--:     List = cons(hd(Shelf_list),Tail)
--:         or
--:     all_by_author(tl(Shelf_list),List)
--: end ;

function books_by_author( U : user ; Author : strng ) return book_list ;
    --| where return List : book_list =>
    --|              all_by_author(Shelf_list,Author,List) ;
```

Figure 12. Function to return all books by an Author

```
--: predicate out_to_borrower(
--:         Checked_out : in out check_out_list ;
--:         B           : in out user ;
--:         List        : in out book_list
--: ) is true if
--:     Tail : book_list ;
--: begin
--:     Checked_out = [] and List = []
--:         or
--:     hd(Checked_out).u = B and
--:     out_to_borrower(tl(Checked_out),B,Tail) and
--:     List = cons(hd(Checked_out).bk,Tail)
--:         or
--:     out_to_borrower(tl(Checked_out),List)
--: end ;

function what_checked_out( U : user ; B : user ) return book_list ;
--| where in( is_staff(U) or U = B ),
--|       return List : book_list => out_to_borrower(Checked_out,B,List) ;

--: predicate has_book( B : in out user ; Bk : in out book ) is true if
--:     Temp : check_out_rec ;
--: begin
--:     member(Checked_out,Temp) and Temp.bk = Bk and Temp.u = B
--: end ;

function who_has( U : user ; Bk : book ; B : user ) return user ;
--| where in( is_staff(U) ),
--|       return B : user => has_book(B,Bk) or B = none ;
```

Figure 13. Functions to examine check out status

language includes the assignment and *if-then-else* statements, *while* loops, procedure calls and user-defined functions. The semantics are defined by Hoare-style proof rules. Although the proof rules assume variables of a single type, PLEASE contains a number pre-defined types and a facility for user type definition. PLEASE can also be used to specify objects, in other words encapsulated types with an internal state, using packages with local variables.

CHAPTER 5.

PRODUCING PROTOTYPES FROM PLEASE SPECIFICATIONS

The PLEASE language provides facilities for the specification of procedures, functions, types and objects; the use of PLEASE aids in problem understanding and enhances communication between the different members of a project. PLEASE specifications can be even more useful if they can be used to automatically create executable prototypes. The prototypes should have a clearly defined relationship to the specification. The programmer should be able to produce the prototypes with little effort, but should be able to optimize them if higher performance is required. The prototypes should be fully compatible with components written in the implementation language; this allows them to be used in the development of embedded systems. Such methods have been devised for PLEASE and implemented in the IDEAL environment; they are based on the translation of PLEASE into Prolog.

In this chapter we describe these methods in detail, give examples of their application, and discuss the use of the prototypes produced in the validation of PLEASE specifications. First, we describe the translation of PLEASE specifications into Prolog procedures. The process can be viewed as a sequence of transformations between logically equivalent formulae; the implementation of equality creates complications. The Prolog procedures produced by this process are only partially correct with respect to the PLEASE specifications; next, we describe some optimizations performed on the procedures to increase the chance of termination. We then describe the interface between these prototypes and their environment; in IDEAL, they are externally indistinguishable from conventional Ada components. We then discuss the use of these prototypes in the validation of PLEASE specifications; problems occur because of incomplete specifications and the implementation of equality.

C-2

5.1. Translation to Prolog

In our system, prototypes are produced from PLEASE specifications by translating predicates and pre– and post–conditions into Prolog procedures; tools in the IDEAL environment perform the translation and generate code to handle I/O and other implementation level details. Although many implementations show significant deviations[222], a "pure" Prolog interpreter can be viewed as a resolution theorem prover for Horn clauses[54,60]. For example, a Prolog procedure $q(\overline{X})$ can be viewed as a set of Horn clause axioms for the relation q; a Prolog implementation trys to find values for $\overline{X}$ such that $q(\overline{X})$ is provable from the axioms. The procedure may terminate with some of the variables only partially instantiated; this corresponds to a set of solutions. In this case, any instantiation of the variables will be acceptable.

Using this model, the translation process becomes a sequence of transformations between logically equivalent formulae; it consists of three 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 definition

```
predicate gr( X, Y : in out natural ) is true if
    T1 : natural  ;
begin
    X = Y + T1 and T1 /= 0
end
```

is converted to the logical formula

$$\forall (X,Y,T1) (X=Y+T1 \wedge T1/=0 \supset gr(X,Y))$$

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 example, the

formula given above is unraveled into

$$\forall\ (X,Y,T1,Temp_1)\ (\\ \quad plus(X,Y,Temp_1) \wedge \\ \quad eq(X,Temp_1) \wedge \\ \quad not_eq(T1,0) \\ \quad \supset gr(X,Y))$$

Finally, the standard transformations to clause form are used to convert the resultant formulae to Prolog procedures. For example, the definition of *gr* given above would produce the following Prolog code

$$gr(X,Y) \leftarrow \\ \quad plus(X,Y,Temp_1), \\ \quad eq(X,Temp_1), \\ \quad not_eq(T1,0)$$

The Prolog procedures produced by this process are partially correct[163,169] with respect to the formulae; although they have the proper logical properties, there is no guarantee that they will terminate. In practice, this is not always a problem; for example, all the specifications in this thesis produce prototypes which terminate in normal use. The set of all logically valid formulae of predicate logic is not decidable[163,169]; therefore, in general it is not possible to extend our approach to total correctness.

To see why, assume that such prototypes can be constructed. We can then determine the validity of an arbitrary formula W in the following manner. First, we specify a procedure having true as the pre-condition and $\neg$W as the post-condition. We then construct a prototype from this specification and execute it. Since the prototype is totally correct, it will find values of the output variables that make $\neg$W true, or if there are no such values it will return an error indication. If the prototype finds a model of $\neg$W then we know that W is not valid; if it returns the error indication, then we know that W is valid. Therefore, we have a decision procedure for the

validity problem of the first–order predicate logic. But the validity problem is known to be undecidable; therefore, totally correct prototypes cannot always be constructed.

This problem can be addressed in a number of ways. First, one can say there is a problem with the logic itself; what does it mean to say that something is true, but that a proof cannot be found by mechanical means? In fact, the situation is somewhat worse: the type of logic we use allows the existence of statements which can not be proven either true or false. Problems such as these have led some to propose the use of *constructive logics*[21,63,221], in which a formula is not considered true unless a proof can be constructed. Unfortunately, this does not solve the problem of a mechanical proof procedure.

5.1.1. An Example

To further clarify the translation process we will consider an example; Figure 14 shows a simplified version of the Prolog code produced from the *factorial* specification presented in Chapter 4[1]. There is a Prolog procedure for the predicate *is_fact* as well as the pre– and post–conditions. The Prolog procedure for *factorial* 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 values satisfy a formula, while executing a post–condition means finding values such that a formula is true. The pre–condition can only reference the input parameters to the procedure, while the post–condition specifies a relation between the inputs and outputs. For example, the *pre_condition* procedure in Figure 14 checks that the variable X satisfies the formula *true*, while *post_condition* must find a value for Y such that *is_fact(X,Y)* is true.

[1] Figure 3 shows the specification of *factorial*.

```
is_fact(X,Y) ←
      eq(X,0), eq(Y,1).
is_fact(X,Y) ←
      minus(Y,1,Temp1),
      is_fact(Temp1,T1),
      times(T1,Y,Temp2),
      eq(Y,Temp2).

pre_condition(X) ← true.
post_condition(X,Y) ← is_fact(X,Y).

factorial(X,Y) ←
      pre_condition(X),
      post_condition(X,Y).
```

Figure 14. Prolog code for the *factorial* procedure

Let us consider the translation of the *is_fact* predicate. The definition of *is_fact* is equivalent to the formula:

$$
\begin{aligned}
&\forall\,(X,Y,T1)\,(\\
&\qquad X{=}0 \wedge Y{=}1\\
&\qquad\qquad \vee\\
&\qquad \mathrm{is_fact}(X{-}1,T1) \wedge Y{=}T1{*}X\\
&\qquad \supset \mathrm{is_fact}(X,Y))
\end{aligned}
$$

This is unraveled into the formula:

$$
\begin{aligned}
&\forall\,(X,Y,T1)\,(\\
&\qquad X{=}0 \wedge Y{=}1\\
&\qquad\qquad \vee\\
&\qquad \mathrm{minus}(X,T1,\mathrm{Temp}_1) \wedge \mathrm{is_fact}(\mathrm{Temp}_1,T1) \wedge\\
&\qquad\qquad \mathrm{times}(T1,X,\mathrm{Temp}_2) \wedge \mathrm{eq}(Y,\mathrm{Temp}_2)\\
&\qquad \supset \mathrm{is_fact}(X,Y))
\end{aligned}
$$

The standard transformations to clause form produce the Prolog procedure in Figure 14. In the

formula above, the functions $-$ and $*$ have been unraveled into the relations minus and times respectively, while the constants 0 and 1 have not been modified; this is due to our implementation of equality.

5.1.2. Equality

The relation of equality is fundamental in mathematics; to qualify as equality, a relation must satisfy certain properties[113]:

i) $X = X$
ii) $t_1 = t_2 \supset f(..t_1..) = f(..t_2..)$
iii) $t_1 = t_2 \supset p(..t_1..) \supset p(..t_2..)$

for all variables X, terms t_1 and t_2, and predicate p.

Property i states that the relation is reflexive; in other words every element is equal to itself. Property ii states that if two terms are equal, then when they are substituted for each other in any other term the resultant terms will be equal. Property iii states that if two terms are equal, then when one is substituted for the other as an argument to a predicate the resultant predicate is implied by the original. We can summarize properties ii and iii as: substitution of equals for equals does not change meaning. The symmetry and transitivity of equality follow from the above.

For the sake of efficiency, Prolog is based on a resolution theorem prover without equality; therefore, Prolog considers all terms to be distinct. For example, it does not consider 1+1 and 2 to be equivalent[2]. We solve this problem by dividing the functions into "constructors" and "non–constructors"; the constructors define the space of all possible terms while the non–

[2]Practically, Prolog solves this problem with the *is* operator, which implements equality for a restricted set of terms.

constructors are implemented as relations (for other solutions to this problem see[96,150]). More precisely, we assume that for each non-constructor function $f(\overline{X})$, there exists a relation $f'(\overline{X},Y)$ such that $f(\overline{X})=Y \equiv f'(\overline{X},Y)$. Axioms which characterize the relation $f'(\overline{X},Y)$ are part of the Prolog run-time library. We unravel the formula $p(..f(\overline{X})..)$ into the equivalent formula $\exists T\ (f'(\overline{X},T) \wedge p(..T..))$.

For example, Figure 15 shows the Prolog procedures implementing the type *natural*. The type has constructors *0* and *succ*; relations *eq*, *not_eq* and *ls*; and non-constructors *plus*, *minus*, *times*, and *divide*. The *eq* relation is equality; it obviously satisfies property i given above. For the relation $eq(T_1,T_2)$ to be true, in other words for the procedure $eq(T_1,T_2)$ to succeed, T_1 and T_2 must be unified. The implementation of Prolog is such that if two terms are unified then they are for all purposes identical; therefore, properties ii and iii are also satisfied. The procedure *not_eq* implements the negation of the equality relation while *ls* implements $<$.

The procedures may at first seem unnecessarily complex; for example, it would seem that the following is sufficient to implement *times*:

```
times(N,0,0).
times(0,N,0).
plus(succ(N),succ(M),0) ←
      times(N,succ(M),P),
      plus(succ(M),P,0).
```

While this is a correct implementation, the procedure does not perform as well during the backtracking process. For example, it responds to the query "times(X,Y,0)" with the solutions (0,N) and (N,0); in other words, X=0, Y=N and X=N, Y=0. The procedure in Figure 15 responds to the same query with the solutions (0,0), (succ(N),0), and (0,succ(N)). While in one sense both solutions are equivalent, the variables are "more instantiated" in the second solution; in other words they contain more information. When the output from *times* is used as input for other

```
eq(N,N).

not_eq(0,succ(N)).
not_eq(succ(N),0).
not_eq(succ(N),succ(M)) ←
      not_eq(N,M).

ls(N,succ(N)).

plus(0,0,0).
plus(0,succ(N),succ(N)).
plus(succ(N),0,succ(N)).
plus(succ(N),succ(M),succ(succ(L))) ←
      plus(N,M,L).

minus(0,0,0).
minus(succ(N),0,succ(N)).
minus(succ(N),succ(M),L) ←
      minus(N,M,L).

times(0,0,0).
times(0,succ(N),0).
times(succ(N),0,0).
times(succ(N),succ(M),L) ←
      times(N,succ(M),P),
      plus(succ(M),P,L).

divide(0,succ(N),0).
divide(succ(N),succ(M),succ(L)) ←
      minus(succ(N),succ(M),P),
      divide(P,succ(M),L).
divide(succ(N),succ(M),0) ← ls(N,M).
```

Figure 15. Prolog procedures for type *natural*

procedures, this can result in success where failure would occur if less information were present.

As described in Chapter 4, PLEASE provides facilities for user type definition. All the functions on user–defined types are implemented as relations; the Prolog code created is similar to that for procedures. For example, Figure 16 shows the Prolog implementation of the type

```
empty_stack_pre ← true.
empty_stack_post(S) ← eq(S,[]).

empty_stack(S) ←
      empty_stack_pre,
      empty_stack_post(S).

push_pre(E,S) ← true.
push_post(E,S,Ns) ← cons(E,S,Ns).

push(E,S,Ns) ←
      push_pre(E,S),
      push_post(E,S,Ns).

pop_pre(S) ←
      empty_stack(Temp1),
      not_eq(S,Temp1).
pop_post(S,Ns) ← tl(S,Ns).

pop(S,Ns) ←
      pop_pre(S),
      pop_post(S,Ns).

top_pre(S) ←
      empty_stack(Temp1),
      not_eq(S,Temp1).
top_post(S,E) ← hd(S,E).

top(S,E) ←
      top_pre(S),
      top_post(S,E).
```

Figure 16. Prolog code for type *natural_stack*

natural_stack[3], which is based on the procedures *cons*, *hd*, *tl*, *eq* and *not_eq* implemented for type *natural_list*. The procedure for each function on *natural_stack* simply "executes" the pre– and post conditions; this is equivalent to stating that the relation corresponding to the function is

[3] Figure 6 , in Chapter 4, shows the specification of *natural_stack*.

implied by the conjunction of the pre– and post–conditions. This follows from the Hoare axiom for user–defined functions under the assumption that the pre– and post–conditions specify a function and not a relation. For example, when the procedure for *top* is invoked, it first calls the pre–condition to ensure that the input list contains at least one element. It then invokes the post–condition to instantiate the result variable to the head of the list.

The translation process described in this section produces procedures that are only partially correct; there is no guarantee they will terminate. Although no translation can guarantee termination, the process previously described produces programs that are too inefficient to be practical; therefore, a number of heuristics are applied to the Prolog procedures to improve their efficiency and increase their chances of termination.

5.2. Code Optimization

For example, Figure 17 shows a simplified version of the Prolog code that is produced from the *sort* specification in Chapter 4[4]. The Prolog procedure for the post–condition must find a value for the output list such that the input and output are permutations of each other and the output is sorted. It accomplishes this by performing a naive sort; the procedure *permutation* functions as a "generator" and the procedure *sorted* as a "selector". When *sort_post* is invoked, *permutation* 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 *permutation* generates the next permutation to be evaluated. This continues until a sorted permutation is generated. The performance of this algorithm is quite poor; however, it can be improved by transformation techniques applied to the logical formulae involved[116,123].

[4] Figure 5 contains the specification of *sort*.

```
permutation(L1,L2) ←
        eq(L1,[]),eq(L2,[]).
permutation(L1,L2) ←
        eq(L1,Temp3),
        hd(L2,Temp1),
        cons(Temp1,Back,Temp2),
        append(Front,Temp2,Temp3),
        append(Front,Back,Temp4),
        tl(L2,Temp5),
        permutationTemp4,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) ← true.
sort_post(Input,Output) ←
        permutation(Input,Output),
        sorted(Output).

sort(Input,Output) ←
        sort_pre(Input),
        sort_post(Input,Output).
```

Figure 17. Prolog code for *sort* procedure

The translation process described in the previous section produces the following procedure for *permutation*:

```
permutation(L1,L2) ←
      eq(L1, []), eq(L2, []).
permutation(L1, L2) ←
      hd(L2, Temp1),
      cons(Temp1, Back, Temp2),
      append(Front, Temp2, Temp3),
      eq(L1, Temp3),
      append(Front, Back, Temp4),
      tl(L2, Temp5),
      permutation(Temp4, Temp5).
```

Unfortunately, it will not function as a generator. The problem lies with the procedure call *append(Front,Temp*$_2$*,Temp*$_3$), which is the third call in the second clause of the *permutation* procedure. Consider a typical invocation. *Permutation* is called with L_1 instantiated and L_2 uninstantiated; in other words, we know everything about L_1 but nothing about L_2. If L_1 is the empty list then L_2 is instantiated to the empty list and the procedure returns. If L_1 is not empty then the second clause is used. After the call to *hd* we know that L_2 has at least one element, and after the call to *cons* we know that *Temp*$_2$ has at least one element. *Append* is then called with *Front* and *Temp*$_3$ still uninstantiated; as we will see, it can not function correctly in this situation and goes into an infinite loop.

We can understand why by examining Figure 18 which shows the Prolog implementation of the type *list*[5]. The procedure for *append* is normally called with two of its arguments instantiated. When *append* is called with its first and third arguments uninstantiated it can return an infinite number of solutions; for example, *append(X,[1],Y)* has solutions X = *[]*, Y = *[1]*; X = *[_]*,

[5] *List* has constructors *[]* and *ccons*; relations *eq* and *not_eq*; and non–constructors *empty_list*, *cons*, *hd*, *tl*, and *append*.

```
eq(L,L).

not_eq([],ccons(E,L)).
not_eq(ccons(E,L),[]).
not_eq(ccons(E1,L1),ccons(E2,L2),[])) ←
      not_eq(E1,E2).
not_eq(ccons(E1,L1),ccons(E2,L2),[])) ←
      not_eq(L1,L2).

empty_list([]).

cons(E,L,ccons(E,L)).

hd(ccons(E,L),E).

tl(ccons(E,L),L).

append([],L,L).
append(ccons(E,La),Lb,ccons(E,Lc)) ←
      append(La,Lb,Lc).
```

Figure 18. Prolog procedures for type *list*

$Y = [_,1]$[6]; $X = [_,_]$, $Y = [_,_,1]$; and so on. In some cases this is acceptable; however, *permutation* calls itself recursively. When *permutation* is called with the first solution from *append*, it calls *append*, which generates an infinite number of solutions. Since the second call to *append* never terminates, the second solution to the first call to *append* is never generated.

To be more specific, when called from *permutation* each solution to *append* has $Temp_3$ instantiated to a "template" with the first element of L_2 equal to a different element of L_1. Successive calls to *append* return solutions with the first element in L_2 occurring later and later in L_1. For example, the first solution has the first element in L_2 equal to the the first element in

[6]In Prolog, "_" refers to a variable which cannot be referenced by name elsewhere.

L_1, while the second solution has the first element in L_2 equal to the the second element in L_1.

All permutations can be generated by using each succeeding element in L_1 as the first element in L_2. However, *append* continues to generate longer and longer templates, even though it has tried all the elements in L_1; therefore, when *permutation* is called recursively, an infinite loop can result before all permutations are generated. For example, all permutations with the first element of L_2 equal to the first element of L_1 must be explored before generating any others. This produces a recursive call to *permutation* which produces a call to *append*. The call to *append* produces an infinite number of solutions so the call to *permutation* never terminates. Therefore, permutations with the first element of L_2 equal to the second element of L_1 are never created.

As a more detailed example, consider the call *permutation([],L_2)*. The solution L_2 = [] is found using the first clause of the *permutation* procedure. When the second clause of the procedure is invoked, the call *hd(L_2,$Temp_1$)* instantiates L_2 to *ccons($Temp_1$,_)* and the call *cons($Temp_1$,Back,$Temp_2$)* instantiates $Temp_2$ to *ccons($Temp_1$,Back)*. Therefore, the call *append(Front,$Temp_2$,$Temp_3$)* reduces to *append(Front,ccons($Temp_1$,Back),$Temp_3$)*. The first solution to this call has *Front* instantiated to [] and $Temp_3$ instantiated to *ccons($Temp_1$,Back)*; in other words, $Temp_3$ can be any list with $Temp_1$ as the first element. Therefore, the call *eq(L_1,$Temp_3$)* reduces to *eq([],ccons($Temp_1$,Back)*, which cannot succeed.

The second solution to *append(Front,ccons($Temp_1$,Back),$Temp_3$)* has *Front* instantiated to *ccons(E,La)* and $Temp_3$ instantiated to *ccons(E,Lc)*; it also generates a recursive call: *append(La,ccons($Temp_1$,Back,Lc)*. The first solution to the recursive call has *La* instantiated to [] and *Lc* instantiated to *ccons($Temp_1$,Back)*; therefore $Temp_3$ is equal to *ccons(E,ccons($Temp_1$,Back))*. In other words, $Temp_3$ can be any list with at least two elements,

the second of which is equal to the first element in L_2. In this case the call $eq(L_1, Temp_3)$ reduces to $eq([],ccons(E,ccons(Temp_1,Back))$, which also cannot succeed.

Similarly, the third solution to the call of *append* allows $Temp_3$ to be any list with at least three elements, the third of which is equal to the first element in L_2. This solution also will not produce a successful call to *eq*. Although there are no more solutions to the original call to *permutation*, *append* will keep returning longer and longer "templates" until some implementation bound is reached. In our example, all the valid permutations were generated before the infinite loop was encountered; however, *permutation* is called recursively. Therefore, in some cases all the valid permutations will not be generated by the procedure.

This problem can be solved by the application of a simple heuristic. Moving procedure calls within the body of a clause does not change the logical meaning; therefore, we can reorder the calls to increase the information available to procedures and increase the chances of termination. In other words, we can improve the chances of termination by sorting the calls within a clause by "chance of termination"; the procedures which are more likely to terminate should be invoked first. For example, in our current implementation, the *eq* predicate always terminates and can instantiate one of its arguments, thereby increasing the amount of information available to subsequent procedures. By moving all calls to *eq* to the "front" of the clause we do not change the logical meaning, but we do increase the chances of termination. When this heuristic is applied to the above procedure, the *permutation* procedure shown in Figure 17 results.

The translation of logic specifications into Prolog procedures alone is not sufficient to create useful prototypes; an interface to the outside world is also necessary.

5.3. System Interface

The first thing to notice is that the specifications in this thesis contain no explicit I/O statements; at present all I/O is handled implicitly by the system. In IDEAL, a program can be automatically generated which reads the *in* parameters to a procedure from input, executes the procedure, and writes the *out* parameters 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 IDEAL is greatly simplified.

Although PLEASE cannot formally specify systems with complex I/O, it can be used in their development. Modules implementing user or system interfaces can be directly implemented in Ada, while other modules are first specified and prototyped in PLEASE. This allows the practical power of Ada and the formal power of PLEASE to be combined in the development of complex, embedded systems. It is also possible to use PLEASE with other systems which provide support for I/O specification and prototyping. The ENCOMPASS environment allows systems to be decomposed into modules which are developed using different languages, methods or tools.

The simple I/O facilities provided by IDEAL are still not enough to turn Prolog procedures into practical prototypes; an interface between Prolog and the implementation language is also needed. The present implementation provides an Ada to Prolog interface which allows procedures to be called transparently. The UNSW Prolog interpreter[208] and the Ada program run as separate processes and communicate through pipes[7]. When a procedure or function is called,

[7]Pipes are a buffering mechanism implemented in Unix.

the *in* parameters are converted to their Prolog representations and the call is passed to the interpreter. When the corresponding Prolog procedure completes, the *out* parameters are converted to the Ada representation and the original call returns. Because the Prolog axiom sets are not complete, it is possible that a prototype will not find existing values which satisfy the specification; in this case the current implementation will terminate with an error indication.

For example, Figure 19 shows a simplified version of the code produced for the *factorial*

```
with ada_to_prolog ;     use ada_to_prolog ;
with types ;             use types ;

package factorial_pkg is

      procedure factorial( X : in natural ; Y : out natural ) ;

private

      Input_buf  : atp_ptr := atp_new ;
      Output_buf : atp_ptr := atp_new ;

end factorial_pkg ;


with ada_to_prolog ;     use ada_to_prolog ;
with types ;             use types ;
with factorial_pkg ;     use factorial_pkg ;

procedure factorial_prog is
      X : natural ;
      Y : natural ;
begin
      get(X) ;
      factorial(X,Y) ;
      put(Y) ;
end factorial_prog ;
```

Figure 19. Ada code for *factorial* prototype

prototype. The figure shows two compilation units: *factorial_pkg*, which implements the *factorial* procedure; and *factorial_prog*, which provides a stand–alone program based on *factorial*. The prototype uses the packages *ada_to_prolog*, which implements the Ada to Prolog interface, and *types*, which provides the types pre–defined in PLEASE. *Factorial_pkg* contains the private variables *Input_buf* and *Output_buf* which serve as buffers for the interface.

Figure 20 shows the body of *factorial_pkg* which contains both initialization code and the implementation of the *factorial* procedure. The initialization code loads the Prolog procedure for *factorial* into the interpreter; clauses for both the pre– and post–conditions and the *is_fact* predicate are asserted at start up time. The body of *factorial* calls Prolog twice: once to execute the pre–condition and once to execute the post–condition. When *factorial* is called, it first clears the input buffer by calling *atp_reset*. It then loads the call "factorial_pre(X)" into the input buffer using *atp_put*; the value of X is converted to the Prolog representation when it is placed in the buffer. The Prolog procedure is invoked using *atp_call*. When the call returns, the procedure *atp_chkpre* determines if the call was successful; if not, then the procedure terminates with an exception. The post–condition is executed in a similar manner; if the call is successful then the value of Y is read from the buffer into the output parameter.

More machinery is necessary if the Prolog procedure is to modify non–local variables; for example, Figure 21 shows the Prolog code for the *add_book* procedure specified in Chapter 4[8]; The procedure adds a book to a library data base by modifying the global variables *Available* and *Shelf_list*. The Prolog procedure for *add_book* has parameters for the initial and final values of all the global variables; this allows the pre– and post–conditions to check that the invariant for the data base holds. The pre–condition for *add_book* simply checks that the procedure is being

[8] Figure 11 shows the specification of *add_book*.

```
package body factorial_pkg is

    procedure factorial( X : in natural ; Y : out natural ) is
        begin
            atp_reset(Input_buf) ;
            atp_put(Input_buf,"atp_call( factorial_pre(") ;
            atp_put(Input_buf,X) ; atp_put(Input_buf,"))!") ;
            atp_call(Input_buf,Output_buf) ;
            atp_chkpre(Output_buf,"factorial") ;
            atp_reset(Input_buf) ;
            atp_put(Input_buf,"atp_call( factorial_post(") ;
            atp_put(Input_buf,X) ; atp_put(Input_buf,",") ;
            atp_put(Input_buf,"Y") ; atp_put(Input_buf,"))!") ;
            atp_call(Input_buf,Output_buf) ;
            atp_chkpost(Output_buf,"factorial") ;
            atp_get(Output_buf,Y) ;
        end factorial ;

begin
     atp_reset(Input_buf) ;
     atp_put(Input_buf,"is_fact(X,Y) :- ") ;
     atp_put(Input_buf,"        eq(X,0),") ;
     atp_put(Input_buf,"        eq(Y,1).") ;
     atp_call(Input_buf,Output_buf) ; atp_reset(Input_buf) ;
     atp_put(Input_buf,"is_fact(X, Y) :-          ") ;
     atp_put(Input_buf,"        minus(X,1,Temp1), ") ;
     atp_put(Input_buf,"        is_fact(Temp1,T1),") ;
     atp_put(Input_buf,"        times(T1,X,Temp2),") ;
     atp_put(Input_buf,"        eq(Y,Temp2).      ") ;
     atp_call(Input_buf,Output_buf) ;
     atp_reset(Input_buf) ;
     atp_put(Input_buf,"factorial_pre(X) :- true.") ;
     atp_call(Input_buf,Output_buf) ;
     atp_reset(Input_buf) ;
     atp_put(Input_buf,"factorial_post(X,Y) :- is_fact(X,Y).") ;
     atp_call(Input_buf,Output_buf) ;
end factorial_pkg ;
```

Figure 20. Body of prototype *factorial* procedure

```
add_book_pre(
            User, Book,
            Shelf_list, Available, Checked_out, Borrowers
      ) ←
      is_staff(User),
      invariant(Shelf_list,Available,Checked_out,Borrowers).

add_book_post(
            User, Book,
            Shelf_list0, Shelf_list,
            Available0, Available,
            Checked_out0, Checked_out,
            Borrowers0, Borrowers
      ) ←
      cons(Book,Available0,Temp1),
      eq(Available,Temp1),
      cons(Book,Shelf_list0,Temp2),
      eq(Shelf_list,Temp2),
      invariant(Shelf_list,Available,Checked_out,Borrowers).


add_book(
            User, Book,
            Shelf_list0, Shelf_list,
            Available0, Available,
            Checked_out0, Checked_out,
            Borrowers0, Borrowers
      ) ←
      add_book_pre(User,Book,
            Shelf_list0,Available0,
            Checked_out0,Borrowers0),
      add_book_post(User,Book,Shelf_list0,Shelf_list,
            Available0,Available,Checked_out0,Checked_out,
            Borrowers0,Borrowers).
```

Figure 21. Prolog code for *add_book* procedure

invoked by a staff user and that the invariant is satisfied. The post–condition for *add_book* updates the data base by adding the new book to both the available and shelf lists before checking the invariant.

The prototypes produced using these methods are useful in the development process; except for lower performance, they are indistinguishable from Ada implementations. The prototype programs can be invoked from a terminal or used with the IDEAL test harness; these facilities provide significant support for the validation process.

5.4. Software Validation

To aid in the validation process, the prototypes produced from PLEASE specifications can be submitted to a series of tests, delivered to the customers for experimentation and evaluation, or possibly even installed for production use on a trial basis. If the software development effort is part of the construction of a large embedded system, PLEASE prototypes can allow the hardware–software interfaces to be debugged earlier and more thoroughly. Unfortunately, the use of prototypes for the validation of PLEASE specifications is complicated due to incomplete specifications and the implementation of equality.

We say that the specification of a procedure or function is *complete* if it specifies a unique output for each input. In general, PLEASE specifications will define relations; there may be more than one output for a particular input. A number of behaviors are possible for the Prolog procedure constructed from such a specification. One option would be to have the procedure return a list of all possible outputs, or to have it return each allowable output on demand (as do many Prolog interpreters). These approaches are useful for single procedures, but do not work well for large systems containing many interacting Prolog prototypes. First, there is the problem of maintaining the "back track context" when control is passing back and forth between the implementation language and Prolog. While this does not seem impossible, it would be very difficult to extend our current implementation to support it. Second, there is a "combinatoric explosion"

problem. If the output of one prototype serves as input for another, the second prototype can produce an output list for each member of its input list; if results flow through among many procedures, the number of outputs generated will quickly become unwieldy

In the current implementation, the prototypes produced will return only one of the outputs allowed for each input. It is possible that this particular output will satisfy the customers, but that other outputs allowed by the specification will not; therefore, the satisfactory performance of a prototype does not guarantee that any implementation which satisfies the specification will produce similar results. For example, assume the customers and analyst create a set of acceptance tests which are correctly executed by the prototype produced from the PLEASE specification. It is possible for an implementation to be constructed which satisfies the specification, but does not execute the test cases correctly.

Another set of problems arise if the implementation of equality is expensive. In the simple scheme now in use, when we say that a prototype correctly executes a test case, we mean that it produces output which has either been inspected for correctness, or is equivalent to output which has. In the current implementation, there are three representations for any value: the "text" representation displayed on a terminal or stored in the test harness; the representation used by Ada procedures; and the representation used by the Prolog prototypes. There is only one way to represent each value in a representation (in other words, we can implement equality using simple comparison) and each value in any representation corresponds to one and only one value in any other. Therefore, no matter how may times we convert a set of values between representations, equality is still inexpensive.

More complex types and representations may require the use of more expensive implementations of equality. For example, consider sets of natural numbers implemented as lists. The com-

parison of two sets for equality can be very expensive; for example, there are six lists which are valid representations of the set containing three distinct elements. The cost will be even higher if the set consists of elements for which equality is also expensive. The use of representations in which distinct values are equivalent will increase the cost of parameter conversion in the Ada to Prolog interface as well as the cost of equality comparison in the test harness.

In this chapter we have described the methods used to construct prototypes from PLEASE specifications, given examples of their application, and discussed the use of the prototypes produced in the validation process. The translation of PLEASE into Prolog is viewed as a sequence of transformations between logically equivalent formulae; the implementation of equality creates complications. The Prolog procedures produced by this process are only partially correct with respect to the PLEASE specifications; optimizations are performed on the procedures to increase the chance of termination. The prototypes can interact with other components in the development environment; in IDEAL, they are externally indistinguishable from conventional Ada components. The prototypes can be used in the validation of PLEASE specifications; problems occur because of incomplete specifications and the implementation of equality.

CHAPTER 6.

THE INCREMENTAL REFINEMENT PROCESS

We have seen that PLEASE can be used to formally specify software, that prototypes can be automatically produced from PLEASE specifications, and that these prototypes can be used in the validation process. While the prototypes produced from PLEASE specifications ocasionally provide high enough performance; in most cases a conventional implementation will be required. PLEASE will be more useful if implementations with a well understood relationship to the specification can be constructed in an orderly manner. The process should be decomposable into small steps which can be independently checked for correctness; this allows each step to be verified before the next is begun, so that errors can be detected as soon as possible and corrected at the lowest possible cost. Such methods have been developed for PLEASE that are similar to VDM and support the verification of refinements using peer review, testing or proof techniques.

In this chapter, we present the methods used to refine PLEASE specifications into implementations and verify the correctness of the process. First, we present an example refinement; we describe a single design transformation, which can be decomposed into a number of atomic transformations. The design transformation involves the choice of algorithm with which to implement a specification; each atomic transformation might be implemented as a single command in a language oriented editor. Next, we present an abstract model of the incremental development process; it is based on viewing a specification as the set of programs which satisfy it. Using this model we can define a refinement step as correct if the set it produces is a subset of the original. Finally, we present the methods used to formally verify the correctness of a refinement; they are based on viewing the process as the construction of a proof in the Hoare calculus.

6.1. An Example

The refinement process can best be explained with an example; consider the *sort* specification given in Chapter 4[1]. Assume that a decision is made to implement the sort procedure using the quicksort algorithm. Figure 22 shows the body of *sort* after the refinement is complete. *Sort_pkg* contains three procedures which are called by *sort*: *select_elmt*, *partition*, and *combine*; *sort* has the same specification as before, but now implements an abstraction of the quicksort algorithm. 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. Figure 23 shows the definitions of the new procedures.

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 abstraction. 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

[1] Figure 5 shows the specification of *sort*.

```
procedure sort( Input : in natural_list ; Output : out natural_list )  is
      --| where in( true ),
      --|       out( permutation(Input,Output) and sorted(Output) ) ;

    Low, High, Sorted_l, Sorted_h : natural_list ; Elmt : natural ;

begin -- sort
    --| true ;
    if Input = [] then
        --| true and Input = [] ;
        Output := [] ;
        --| permutation(Input,Output) and sorted(Output) ;
    else
        --| true and Input /= [] ;
        select_Elmt(Input,Elmt) ;
        --| member(Elmt,Input) ;
        partition(Input,Elmt,Low,High) ;
        --| is_partition(Input,Elmt,Low,High) ;
        sort(Low,Sorted_l) ;
        --| is_partition(Input,Elmt,Low,High) and
        --|     permutation(Low,Sorted_l) and sorted(Sorted_l) ;
        sort(High,Sorted_h) ;
        --| is_partition(Input,Elmt,Low,High) and
        --|     permutation(Low,Sorted_l) and sorted(Sorted_l) and
        --|     permutation(High,Sorted_h) and sorted(Sorted_h) ;
        combine(Sorted_l,Elmt,Sorted_h,Output) ;
        --| permutation(Input,Output) and sorted(Output) ;
    end if ;
    --| permutation(Input,Output) and sorted(Output) ;
end SORT ;
```

Figure 22. Initial refinement of *sort* specification

any implementation which satisfies the new specification will also satisfy the original.

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

```
procedure select_elmt(
                List : in natural_list ;
                Elmt : out natural
) is separate ;
    --| where in( List /= [] ),
    --|       out( member(Elmt,List) ) ;

--: predicate is_partition(
--:              List      : in out natural_list ;
--:              Elmt      : in out natural ;
--:              Low, High : in out natural_list
--: ) is true if
--: begin
--:     permutation(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_partition(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 ) ;
```

Figure 23. Definitions to support refinement of *sort* specification

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. We can best understand the methods used to formally verify a refinement step in the context of an abstract model of the refinement process.

6.2. Refinement Model

We will use a simplification of the model presented in [23,24]. In our model, a *development* begins with a specification and incrementally refines it into an implementation; the process consists of a number of *steps*, each of which further constrains the allowable implementations. At each stage in the process we have an *abstract program*: a specification that can be satisfied by a number of different implementations. The final step in the process produces a *concrete program*: a specification that has only one implementation.

For the purposes of this thesis, an abstract program is a PLEASE specification; for example, both the original specification of *sort* and the refinement presented in the last section are abstract programs. Similarly, a concrete program is an Ada implementation; for example, the implementation of *factorial* given in Chapter 4[2] is a concrete program. The distinction between abstract and concrete programs is in one sense a matter of viewpoint. For example, the refined *sort* specification given in Figure 22 is an annotated Ada implementation; in other words a concrete program. However, it depends on a number of external procedures whose implementations have not been defined; in other words, the combination of Figure 22 and Figure 23 is an abstract program.

We can say that an abstract program represents the set of implementations that are correct with respect to it; from this viewpoint, a concrete program is simply an abstract program with only one implementation. We can view each step in the refinement process as taking a set of programs as input and producing a new set as output; we say that a step is correct if its output is a proper subset of its input. A development is correct if each of its steps is correct.

---

[2] Figure 4 shows the *factorial* implementation.

Depending on the methods to be employed, it is useful to view the refinement process in more or less detail; for this thesis, two perspectives are interesting. We can profitably view a development as a sequence of *design transformations*. For example, the refinement of *sort* presented in the previous section is a *design transformation*: it implements a major choice of algorithm or data structure. In our model, each design transformation can be decomposed into a number of *atomic transformations*. Atomic transformations represent the smallest significant changes to an abstract program; for example, in IDEAL each editor command is an atomic transformation. A design transformation is correct if each of its atomic transformations is correct just as a development is correct if each of its design transformations is correct.

The difference in "size" causes these transformations to be treated differently in IDEAL. 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.

Many different methods can be used to verify the correctness of a refinement. For example, in a *technical review* process the specification and implementation are inspected, discussed and compared by a group of knowledgeable personnel[87,242]. These methods would typically be applied to a design transformation or an even larger refinement unit; the use of peer review on each atomic transformation would be extraordinarily expensive.

Testing can also be used to check the correctness of a refinement[91,176]; prototypes produced from the abstract programs input and output by a step can be run on a representative set

of input data and the results compared. This method can give significant assurance at low cost; however, in general a program cannot be tested on all possible inputs. Further complications are caused by the incompleteness of specifications. In general, an abstract program represents a relation between inputs and outputs, while the prototype produced from an abstract program implements a function. Therefore, it is possible that the abstract programs for a correct refinement step can produce prototypes which perform differently on the test data. If we require that the prototypes performs identically we may eliminate correct refinements.

Since PLEASE specifications are mathematically based, formal methods can also be used to verify the correctness of a refinement[26,110,112,120,121,134,163,250]; unfortunately, formal methods are devised by humans and are often found to be in error[186]. This has led some to feel that no one technique alone can ensure the production of correct software[71,75] and to propose methods which combine a number of different techniques[9,205]. Despite their limitations, formal methods have a number of advantages: first, they can symbolically determine that a program will perform correctly for all possible inputs; second, although they can be tedious to use, many of the processes involved can be automated. The formal methods used with PLEASE are based on viewing the refinement process as the construction of a proof in the Hoare calculus.

6.2.1. Formal Verification

For example, consider the refined *sort* specification given in Figure 22 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[120,163,169]; this proof was incrementally created as the design transformation was performed. Each atomic transformation corresponds to at most two proof steps; the transformation between Figure 5 and Figure 22 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 24 shows the first step in the refinement of the *sort* procedure from both the procedure

Program View

```
begin -- sort
     --| true ;
     <Statement(1)>
     --| permutation(Input,Output)
     --|      and sorted(Output) ;
end sort ;
```

≡

Proof View

$[P]\ S_1\ [Q]$

Where $P \equiv$ true, $S_1 \equiv$ Statement(1),
$Q \equiv$ permutation(Input,Output) $\wedge$ sorted(Output)

Refine Statement(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 = [] ;
        <Statement(2)>
        --| permutation(Input,Output)
        --|      and sorted(Output) ;
    else
        --| true and Input /= [] ;
        <Statement(3)>
        --| permutation(Input,Output)
        --|      and sorted(Output) ;
    end if ;
    --| permutation(Input,Output)
    --|      and sorted(Output) ;
end sort ;
```

≡

$$\frac{[P \wedge E]\ S_2\ [Q],\ [P \wedge \neg E]\ S_3\ [Q]}{[P]\ \textit{if}\ E\ \textit{then}\ S_2\ \textit{else}\ S_3\ \textit{end if}\ [Q]}$$

Where $P \equiv$ true,
$Q \equiv$ permutation(Input,Output) $\wedge$ sorted(Output)
$E \equiv$ Input = [],
$S_i \equiv$ Statement(i).

Figure 24. Refinement as proof construction

and proof views. From the program view, an atomic transformation takes an incomplete program and produces a more concrete one; from the proof view, an atomic transformation adds steps 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. For more discussion on the relationship of proofs and programs see[21].

From this perspective, each abstract program corresponds to a Hoare calculus proof tree. Given an expressive interpretation[3], each implementation which satisfies a specification corresponds to a tree with Hoare axioms or provable first–order logic formulae at all of its leaves; we will call such trees *complete*. An atomic transformation is correct if it results in a legal tree; in other words if it consists of correct applications of Hoare rules or axioms. Since each step forms a legal tree, the tree produced by the development will be legal; unfortunately, there is no guarantee that a complete tree can be produced. Our notion of correctness does not insure that a satisfactory program can be produced; however, it does prevent a program which does not satisfy the specification from being constructed.

In IDEAL, this view of the refinement process is supported by ISLET, a language oriented editor similar to[203]. 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[115].

[3]For a full explanation see[163]

The use of general purpose theorem provers is quite expensive[4]; 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 IDEAL for use during the corresponding verification phase. For example, Figure 25 shows the verification conditions for the transformation from Figure 5 to Figure 22 which can not be proven by algebraic simplification and simple proof tactics alone; out of twenty six atomic transformations, 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.

In this chapter, we have presented the methods used to refine PLEASE specifications into implementations and to verify the correctness of the process. The refinement process can be viewed as a sequence of design transformations, each of which implements a major choice of data structure or algorithm. Design transformations can be decomposed into atomic transformations, which might be implemented as a single command in a language oriented editor. The correctness

---

```
Input = [] ⊃
        permutation(Input, [] ) ∧ sorted( [] )

is_part(Input,Elmt,Low,High) ∧
        permutation(Low,Sorted_l) ∧ sorted(Sorted_l) ∧
        permutation(High,Sorted_h) ∧ sorted(Sorted_h) ∧
        List = Sorted_l || [Elmt] || Sorted_h ⊃
        permutation(Input,List) ∧ sorted(List)
```

Figure 25. Verification conditions for refinement

---

of a refinement step can be understood in the context of an abstract model of the incremental development process; it is based on viewing a specification as the set of programs which satisfy it. Using this model we can define a refinement step as correct if the set it produces is a subset of the original. The methods used to formally verify the correctness of a refinement are based on viewing the refinement process as the construction of a proof in the Hoare calculus.

CHAPTER 7.

THE IDEAL ENVIRONMENT

We believe that languages similar to PLEASE can greatly enhance the software development process; we also believe that to realize the full benefits of PLEASE an integrated support environment is needed. The environment must provide facilities to create PLEASE specifications, construct prototypes from these specifications, validate the specifications using the prototypes produced, refine the validated specifications into Ada implementations, and verify the correctness of the refinement process. IDEAL (Incremental Development Environment for Annotated Languages) is an environment concerned with the specification, prototyping, implementation and verification of single modules; it is a programming–in–the–small environment for development using PLEASE. In this chapter we describe IDEAL in detail and give an example of its use in software development. First we describe the architecture of the system; it contains four main components: ISLET, a language–oriented program/proof editor; a proof management system; a prototyping tool; and a test harness. We then discuss the operation of ISLET; it allows the construction of PLEASE specifications and their incremental refinement into Ada implementations. It contains three major sub–systems: an algebraic simplifier, a set of simple proof procedures, and an interface to the proof management system. Finally we discuss the development of a small program in detail; refinement using ISLET is given the most attention.

7.1. Architecture

Figure 26 shows the top–level architecture of IDEAL, which contains four tools: TED[115], a proof management system that is interfaced to a number of theorem provers; ISLET (Incredi-

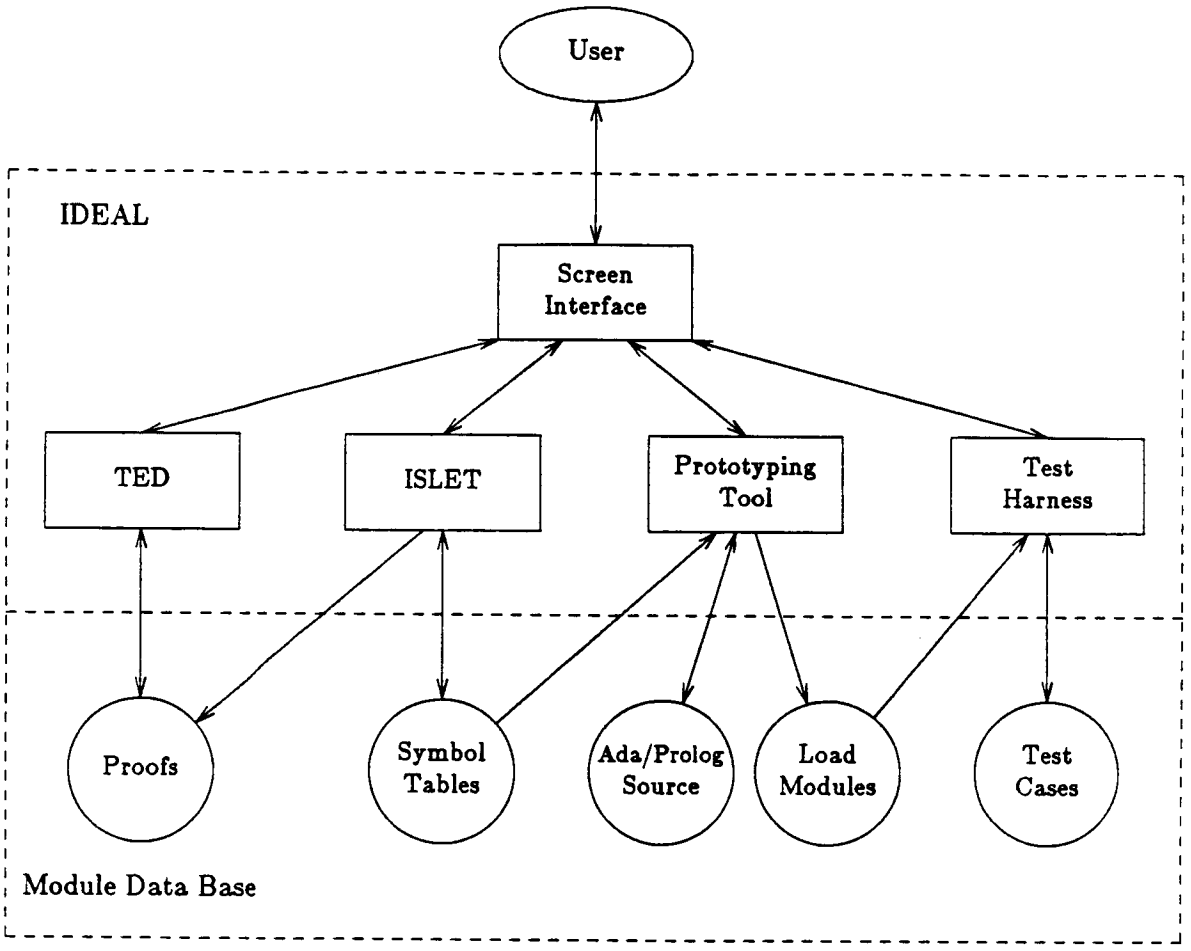


Figure 26. Architecture of IDEAL

bly 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 that are stored in a *module data base*. When IDEAL is used with ENCOMPASS, the module data base is stored as part of a project data base by the ENCOMPASS configuration

control system; IDEAL receives a capability to the module data base from the ENCOMPASS 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 implementations being developed. These symbol tables are displayed and manipulated by ISLET, which can be used to create PLEASE specifications and incrementally refine them into Ada implementations. This process can also be viewed as the construction of a proof in the Hoare calculus[120,163]. 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 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.

7.2. 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[120,163]. The refinement process is a sequence of *atomic transformations*, which can be grouped into *design transformations*. An atomic transformation cannot be decomposed. From the program view, an atomic transformation changes an unknown statement into a particular language construct; from the proof view, an atomic transformation adds more steps 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 27 shows the architecture of ISLET. The user interacts with ISLET through a simple language–oriented editor similar to[203]. 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

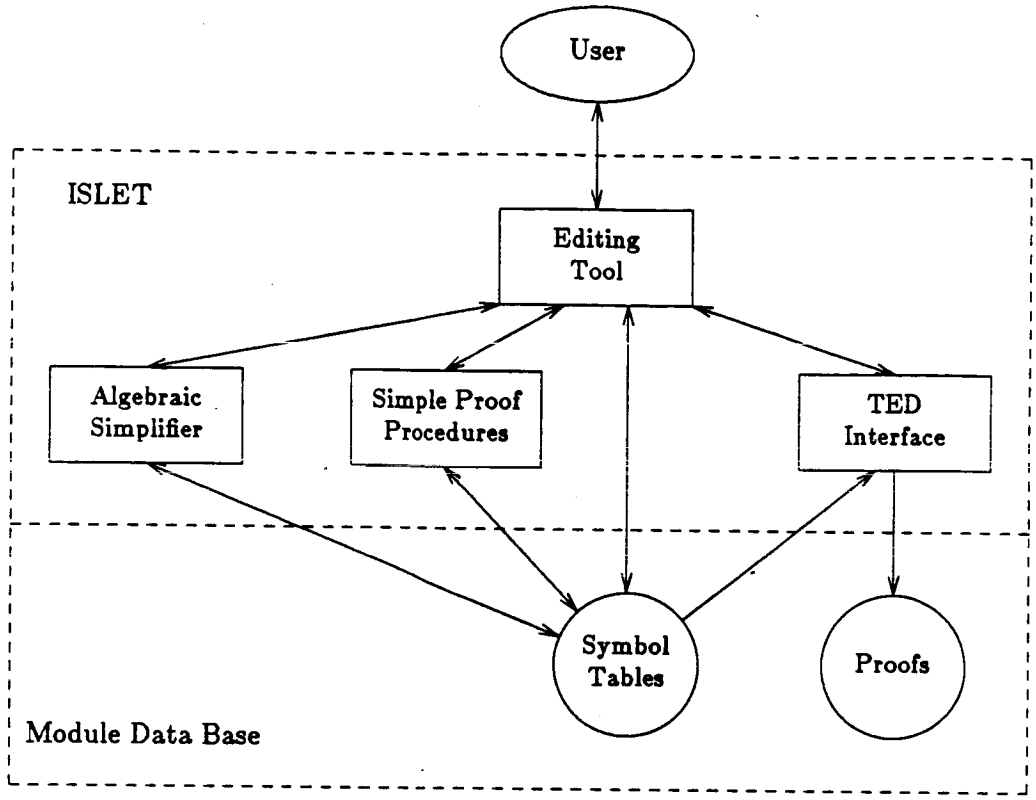


Figure 27. Architecture of ISLET

inexpensive; however, they are not very powerful. The combined use of these two methods supports the *rigorous*[134] 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.

The algebraic simplifier is implemented as a term rewriting system[147,181]; it contains a knowledge-base of rules which are assumed to be convergent. The simple proof procedures rely

on a knowledge base containing information such as: if the formulae F_1 and F_2 are equivalent under renaming of variables, then the formula $F_1 \supset F_2$ is always true. Other rules implement simple knowledge of equality; for example, if F(X) and X=c are both true then so is F(c). At present, it is difficult to examine, analyze or change the contents of these knowledge bases; for example, algorithms exist to determine if a set of rewrite rules are convergent, but they are not implemented in ISLET. In the future, we plan to develop tools to correct these deficiencies.

To further clarify both the principles behind IDEAL and the limitations of the current implementation, we will consider an example of software development. We will follow the development of a procedure through the specification, refinement and verification processes.

7.3. An Example

Assume that a programmer must implement a procedure that takes a natural number as input and produces its factorial as output[1]. The programmer first creates an empty module and then invokes IDEAL, which produces a display showing an empty package. The programmer then invokes ISLET to specify the procedure. Figure 28 shows the completed specification, which includes both the pre– and post–conditions for the procedure and the definition of the *is_fact* predicate. This figure, and the others in this section, show the actual output from the current implementation; therefore, they do not alway follow the syntax conventions used elsewhere in this thesis.

The top line of the display gives a menu of ISLET commands. ISLET has a *focus of attention* which is always on a particular symbol table scope; for example, in Figure 28 ISLET's focus

[1]The specification of such a procedure was given in Chapter 4.

```
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 28. ISLET display showing *factorial* specification

of attention is on the *factorial* package. The visible objects are the *factorial* procedure and the *is_fact* predicate. The *open* command changes the focus of attention to an inner scope; for example, the command "open! procedure factorial." would shift the focus of attention to the body of the *factorial* procedure[2]. The *clos* command shifts the focus to the containing scope, while the

[2]The "!" after the command name is an artifact of ISLET's Prolog-based implementation.

display command presents the focus of attention on the screen. The *decl* command allows declarations, and the *put* and *get* commands support the saving and restoring of the editor's state. The *help* command provides on–line assistance, while the *quit* command exits ISLET. The *refine* command allows more design or implementation detail to be added to a specification, while the *undo* command reverses the last refinement step. The *use* command allows separately developed modules to be used in a specification or implementation, and the *list* command displays all the verification conditions which have not been certified.

As the programmer enters the specification, the syntax and static semantics are checked for correctness; unfortunately, at present the granularity is somewhat coarse. For example, the programmer must enter the definition of *is_fact* as a unit; if a mistake is made the entire definition must be re–entered. Also, the definition of *is_fact* must be entered before the pre– and post–conditions for *factorial*. This is because *is_fact* is referenced in the post–condition; an undeclared identifier error would result if the order of entry were reversed. The utility of ISLET would be dramatically increased by a finer grained implementation; for example, one based on an editor, such as Epos[145], that allows an arbitrary number of text–oriented commands to be performed before the syntax and semantics are checked.

Figure 29 shows the ISLET display as the programmer opens the factorial procedure to begin the refinement process. At this point, the procedure consists of an unknown statement sequence, denoted by *unknown_0*, surrounded by assertions *true* and *is_fact(x,y)*. The goal of the refinement process is to produce an implementation of *unknown_0* which is correct with respect to pre–condition *true* and post–condition *is_fact(x,y)*. This problem is simplified by the fact that the factorial calculation can be divided into two cases: if the input is 0 then the result is 1, otherwise more computation is needed.

```
MENU: Clos, DEcl, DIsp, Get, Help, List, Open, Put, Quit, Refine, Undo, USe

 procedure factorial( x : in natural ; y : out natural ) is

 begin
     --| true ;
        <unknown_0> ;
     --| is_fact(x, y) ;
 end factorial ;

 :
```

Figure 29. ISLET display at beginning of refinement process

With this in mind, the programmer refines the unknown statement sequence into an *if-then-else*; he types the command "refine! 0 if x=0.", which can be read as: refine unknown zero into an *if-then-else* on condition x equal to zero. In ISLET, each refinement step corresponds to at most two proof steps in the Hoare calculus: a step using the rule for the appropriate language construct, and possibly a step using the consequence rule. For example, the current refinement uses the rule for the *if-then-else* construct, but does not make use of the consequence rule; therefore, no verification conditions are generated.

Figure 30 shows the ISLET display after the refinement is compete; *unknown_0* has been transformed into an *if-then-else* with an unknown at each branch. The refinement process now has two sub-goals. An implementation of *unknown_1* must be found which is correct with respect to pre-condition *true and* $x = 0$ and post-condition *is_fact(x,y)*. Similarly, *unknown_2* must be refined into a statement sequence which is correct with respect to *true and not* $x = 0$ and

```
MENU: Clos, DEcl, DIsp, Get, Help, List, Open, Put, Quit, Refine, Undo, USe

procedure factorial( x : in natural ; y : out natural ) is

begin
     --| true ;
     if x = 0 then
          --| true and x = 0 ;
             <unknown_1> ;
          --| is_fact(x, y) ;
     else
          --| true and not x = 0 ;
             <unknown_2> ;
          --| is_fact(x, y) ;
     end if ;
     --| is_fact(x, y) ;
end factorial ;

:
```

Figure 30. ISLET display after initial refinement of *factorial*

is_fact(x,y). These goals can be pursued in any order: the programmer can perform a number of refinements on *unknown_1*, switch his attention to *unknown_2*, and then return to finish *unknown_1*. Fortunately, this is not necessary: *unknown_1* has a simple implementation.

Knowing that the factorial of zero is one, the programmer refines *unknown_1* into a statement assigning one to *y*; Figure 31 is produced by ISLET after the command "refine! 1 y := 1.". The proof of this refinement step makes use of both the Hoare axiom for assignment statements and the consequence rule; therefore, verification conditions are generated. The algebraic

```
MENU: Clos, DEcl, DIsp, Get, Help, List, Open, Put, Quit, Refine, Undo, USe

Verification conditions are:

    true and x = 0 =>
         is_fact(x, 1)

Simplified verification conditions are:

    x = 0 =>
         is_fact(x, 1)

Trying simple proof procedure ...

Simple proof procedure failed - generating ted file ...


Should I invoke TED (y/n) ? n

Type ".<cr>" to continue .
```

Figure 31. ISLET display showing verification conditions for first assignment

simplifier is able to reduce the verification conditions by realizing that for any formula F, true and F is equivalent to F. Although the verification conditions are true, the simple proof procedures are not able to certify this. The problem is that the current implementation does not add user definitions to the rule base; the proof methods have no knowledge of *is_fact*. We plan to correct this problem in the future.

At this point the programmer examines the verification conditions; ISLET has created input for TED, which can be invoked if the programmer desires. In this case the programmer is convinced that the verification conditions are correct; he decides to wait until after the refinement process is complete to formally certify them. ISLET displays the completed refinement and the development process continues. The programmer decides to implement the *else* branch using a sequence consisting of a *while* loop and its initialization. He realizes he needs a loop counter and declares a variable *i*.

In ISLET, a loop has both an *invariant*, which is maintained by the body, and a *condition*, which controls termination. If the invariant is true before the loop begins execution, then both the invariant and the negation of the condition will be true when the loop terminates. The programmer's strategy involves the invariant *is_fact(i,y)*; he must initialize *i* and *y* to make this true. He decides that the loop initialization will consist of two statements: one to initialize *i*, and one to initialize *y*.

Figure 32 shows the display after all these refinements, none of which generate verification conditions, have been performed. To achieve the invariant *is_fact(i,y)*, the programmer refines *unknown_5* into a statement assigning one to *i* and *unknown_6* into a statement assigning one to

```
MENU: Clos, DEcl, DIsp, Get, Help, List, Open, Put, Quit, Refine, Undo, USe

procedure factorial( x : in natural ; y : out natural ) is

    i : natural ;

begin
    --| true ;
    if x = 0 then
        --| true and x = 0 ;
        y := 1 ;
        --| is_fact(x, y) ;
    else
        --| true and not x = 0 ;
          <unknown_5> ;
        --| i = 1 ;
          <unknown_6> ;
        --| is_fact(i, y) ;
          <unknown_4> ;
        --| is_fact(x, y) ;
    end if ;
    --| is_fact(x, y) ;
end factorial ;


:
```

Figure 32. ISLET display after sequence of refinements

y. The verification conditions for the first assignment can be certified using simple methods; the second assignment generates verification conditions:

```
i = 1 => is_fact(i,1)
```

These conditions can not be certified using the simple methods in ISLET alone; TED must be invoked. As with the previous conditions, this is due to lack of knowledge of the *is_fact* predicate.

The programmer must now implement the *while* loop. He types the command "refine! 4 while i /= x.", to refine *unknown_4* into a *while* loop with condition i not equal to x; Figure 33 shows the ISLET display. ISLET generates verification conditions for this refinement based on both the rule for *while* loops and the consequence rule; the current ISLET implementation only verifies partial correctness. The verification conditions can be algebraically simplified, and the result can be proved using basic knowledge of equality: if two terms are equal, then we can substitute one for the other in any formula without changing its meaning.

The programmer realizes that the body of the loop must both increment the loop counter and update the calculated quantity. Figure 34 shows the ISLET display after he refines the body

```
MENU: Clos, DEcl, DIsp, Get, Help, List, Open, Put, Quit, Refine, Undo, USe

Verification conditions are:

    is_fact(i, y) and not i /= x =>
         is_fact(x, y)

Simplified verification conditions are:

    is_fact(i, y) and i = x =>
         is_fact(x, y)

Trying simple proof procedure ...

Simple proof procedure successful
```

Figure 33. ISLET display showing verification conditions for loop creation

```
MENU: Clos, DEcl, DIsp, Get, Help, List, Open, Put, Quit, Refine, Undo, USe

procedure factorial( x : in natural ; y : out natural ) is

    i : natural ;

begin
    --| true ;
    if x = 0 then
         --| true and x = 0 ;
         y := 1 ;
         --| is_fact(x, y) ;
    else
         --| true and not x = 0 ;
         i := 1 ;
         --| i = 1 ;
         y := 1 ;
         --| is_fact(i, y) ;
         while i /= x loop
              --| is_fact(i, y) and i /= x ;
                 <unknown_8> ;
              --| is_fact(i - 1, y) ;
                 <unknown_9> ;
              --| is_fact(i, y) ;
         end loop ;
         --| is_fact(x, y) ;
    end if ;
    --| is_fact(x, y) ;
end factorial ;

:
```

Figure 34. ISLET display showing decomposition of loop body

into a statement sequence. The programmer then refines *unknown_8* into an assignment which increments *i* and produces verification conditions:

```
is_fact(i, y) and i /= x =>
     is_fact(i + 1 - 1, y)
```

These can be solved using the simple proof methods. *Unknown_9* is more difficult. The programmer refines it into a statment assigning y the value of $y * i$; this refinement produces the verification conditions:

```
is_fact(i - 1, y) =>
      is_fact(i, y * i)
```

These verification conditions can not be solved by the simple methods alone; their proof requires a reasonably deep knowledge of both the *is_fact* predicate and the natural numbers.

The implementation of *factorial* is now complete; Figure 35 shows the completed procedure. The body of *factorial* is completely annotated; in other words, there is an assertion both before and after each executable statement. The assertions plus the executable statements form a proof in the Hoare calculus. Before the proof is really complete, the programmer must certify the verification conditions which did not yield to the simple methods implemented in ISLET; these are:

```
x = 0 =>
      is_fact(x, 1)
i = 1 =>
      is_fact(i, 1)
is_fact(i - 1, y) =>
      is_fact(i, y * i)
```

The first two conditions could be proved in a more advanced implementation of ISLET: one which added user defined predicates to the appropriate rule bases. The third condition will probably always require a general purpose theorem prover.

```
MENU: Clos, DEcl, DIsp, Get, Help, List, Open, Put, Quit, Refine, Undo, USe

procedure factorial( x : in natural ; y : out natural ) is

    i : natural ;

begin
    --| true ;
    if x = 0 then
        --| true and x = 0 ;
        y := 1 ;
        --| is_fact(x, y) ;
    else
        --| true and not x = 0 ;
        i := 1 ;
        --| i = 1 ;
        y := 1 ;
        --| is_fact(i, y) ;
        while i /= x loop
            --| is_fact(i, y) and i /= x ;
            i := i + 1 ;
            --| is_fact(i - 1, y) ;
            y := y * i ;
            --| is_fact(i, y) ;
        end loop ;
        --| is_fact(x, y) ;
    end if ;
    --| is_fact(x, y) ;
end factorial ;

:
```

Figure 35. ISLET display showing completed implementation of *factorial*

To certify the verification conditions, the programmer first exits ISLET; Figure 36 shows the IDEAL display at this point in the development. The top line of the display gives a list of IDEAL commands. The *dn* and *up* commands scroll the screen down or up respectively. The *set* and *unset* commands alter internal flags. The *ted* command causes TED to be invoked on a set of

```
MENU:  dn,  Help,  List,  Open,  Make,  set,  Ted,  unset,  up


This module contains:

symtab_0        symtab_11       symtab_2        symtab_5        symtab_8
symtab_1        symtab_12       symtab_3        symtab_6        symtab_9
symtab_10       symtab_13       symtab_4        symtab_7

ted_0           ted_13          ted_4           ted_9

test_cases


:
```

Figure 36. IDEAL display after completion of *factorial* refinement

verification conditions. The *help* command provides on-line assistance, while the *list* command displays the contents of the module. The *make* command constructs a load module from either a specification or an implementation, while the *open* command can be used to invoke either ISLET or the test harness.

The module contains a number of symbol tables: one to record each significant point in the development process. These symbol tables provide a simple implementation of the *undo* command, as well as allowing the examination of the entire refinement process during peer review. The module also contains a number of TED files: one for each step in the refinement process which produced verification conditions not proven by the simple methods. These files are complete with axiom sets including user defined predicates. The programmer invokes TED on these files and certifies the verification conditions. At this point, the development is complete.

In this chapter we have described IDEAL, an environment concerned with the specification, prototyping, implementation and verification of single modules using PLEASE. We have also given an example of software development using the system. IDEAL contains four main components: ISLET, a language–oriented program/proof editor; a proof management system; a prototyping tool; and a test harness. ISLET is the central tool in IDEAL; it allow the construction of PLEASE specifications and their incremental refinement into Ada implementations. It contains three major sub–systems: an algebraic simplifier, a set of simple proof procedures, and an interface to the proof management system. Although the current implementation of IDEAL is quite limited, it demonstrates much of the potential of such environments. We feel the use of future environments similar to IDEAL will enhance the software development process.

CHAPTER 8.

THE ENCOMPASS ENVIRONMENT

IDEAL provides an integrated environment for programming–in–the–small using the PLEASE executable specification language; while this alone can enhance the development process, more can be gained with the addition of an environment for programming–in–the–large[200,249]. The environment should support all the objects produced and used in software development; it should save, record and update their versions and relationships. The environment should also support the interactions necessary for the development of multi module systems by multiple programmers. The environment should support the rigorous [134] development of programs: it should be possible to develop parts of a project using an environment such as IDEAL, while other, less critical parts are developed using less expensive methods.

ENCOMPASS is such an environment; it provides support for all aspects of software development using PLEASE. In ENCOMPASS, software is decomposed into modules which can be developed using either IDEAL or more conventional tools. ENCOMPASS provides facilities to store, track, manipulate and control all the objects used in the software development process: documents, specifications, source code, proofs, test data, and load modules are all supported. ENCOMPASS also provides mechanisms to support the interactions among developers; the system allows the creation, decomposition, distribution, monitoring and completion of tasks. Figure 37 shows the top–level architecture of the system. In ENCOMPASS, the user accesses and modifies components using a set of software development tools. The *configuration management system* structures the software components developed by a project and stores them in a *project data base*. The *project management system* uses facilities provided by the configuration

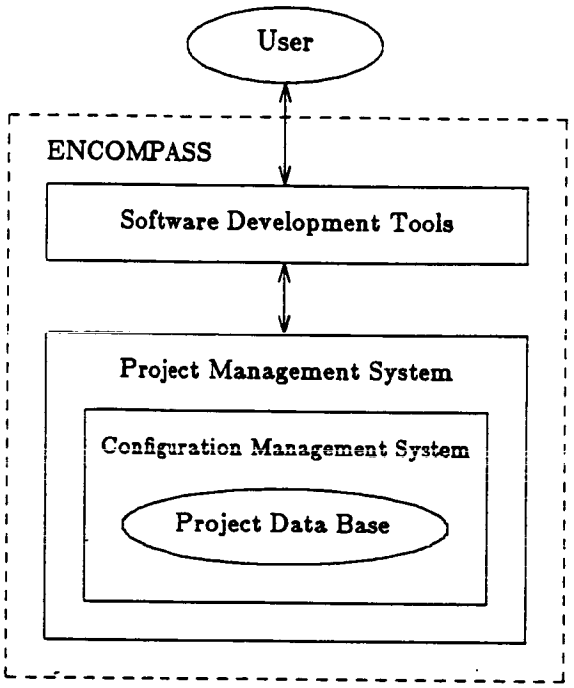


Figure 37. Architecture of ENCOMPASS

management system to control both access to the data base and interactions between developers.

In this chapter, we describe ENCOMPASS in detail and give examples of its use. First, we describe the life–cycle model ENCOMPASS is designed to support; this is Fairley's traditional or *waterfall* life–cycle[88], extended to support the use of executable specifications and VDM. In ENCOMPASS, we extend the traditional life–cycle to include a separate phase for user validation; we also combine the design and implementation processes into a single refinement phase. Next, we describe the ENCOMPASS configuration and project management systems. In the configuration management system, software is modeled as *entities* which have *relationships*

between them. These entities can be structured into complex hierarchies which may be accessed through different *views*. The project management system implements a *management by objectives*[106] approach to software development; each phase in the life–cycle satisfies an objective by producing a *milestone* which can be recognized by the system. We then give an example of software development in ENCOMPASS; the construction of a multi–module system is followed from specification through the delivery of a validated and verified implementation. Next, we describe how ENCOMPASS can be used with a central repository to support software reuse, and finally we present an automated change control system which incorporates ENCOMPASS, a central repository, and Notesfiles.

8.1. Development Model

Figure 38 shows the software development model that ENCOMPASS is designed to support. In this model, a development passes through the phases: planning, requirements definition, validation, refinement, and system integration. The refinement phase may be decomposed into a number of steps, each consisting of a design transformation and its verification. This model can be profitably viewed from two perspectives. On one hand, it is Fairley's traditional, or *phased* life–cycle[88], extended to support the Vienna Development Method and the use of an executable specification language. On the other hand, it is the IDEAL life–cycle, extended to support multiple modules and programmers.

In the ENCOMPASS development model, the *planning phase* defines the problem to be solved and determines if a computer solution is feasible and cost effective[88]. Alternative solutions to the problem are considered and compared for cost effectiveness, and preliminary plans and schedules for the project are created. In the *requirements definition* phase, the functions and

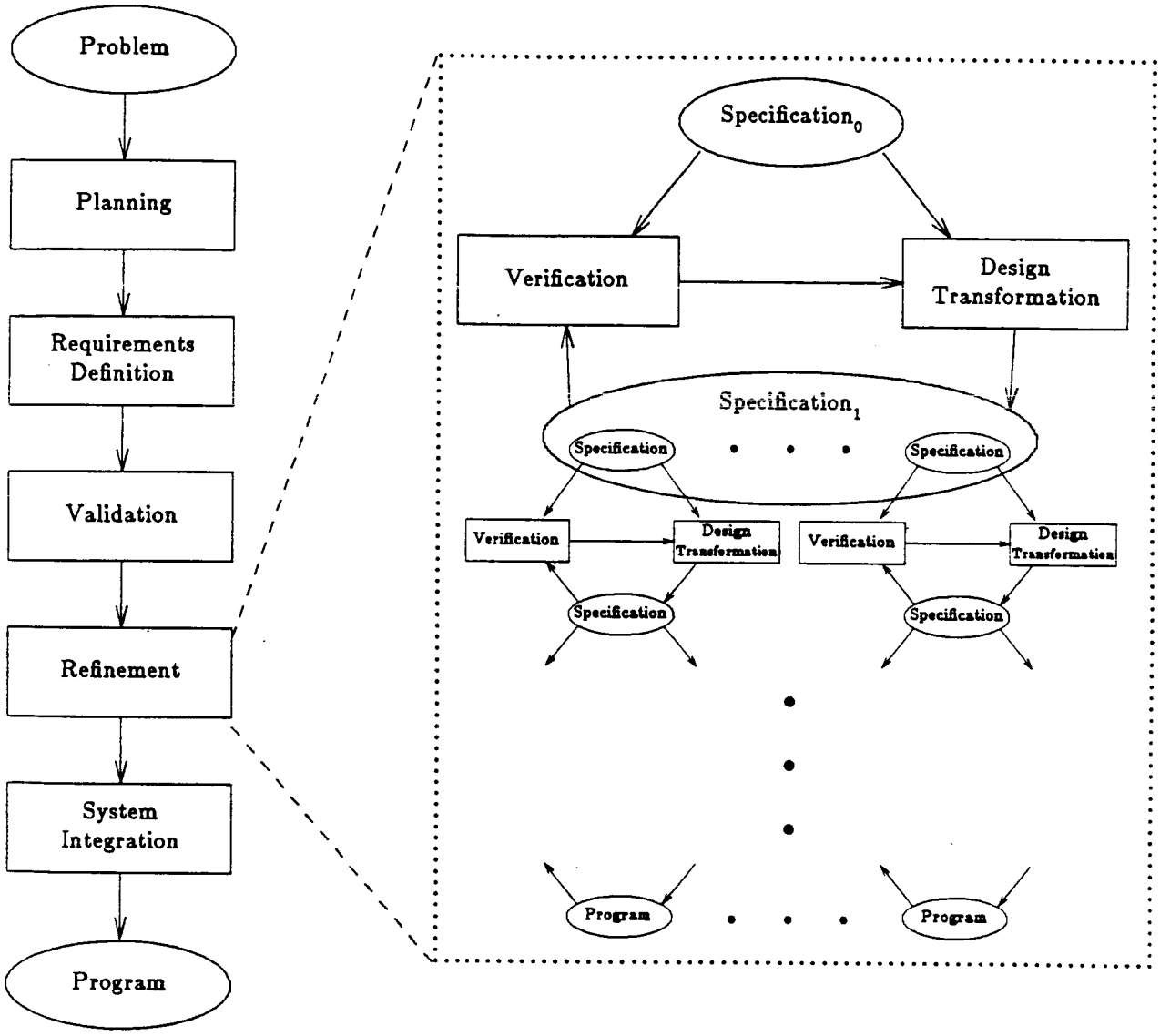


Figure 38. ENCOMPASS software development model

qualities of the software to be produced by the development are precisely described[88]. Requirements definition produces a high-level specification, which concentrates on the needs and desires

of the customers as they affect the external system interface rather than the internal structure of the software to be produced. 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 that satisfies the software requirements specification will also satisfy the customers, in other words, that the requirements specification is valid. If not, it should be corrected before the development proceeds to the costly phases of refinement and system integration. During the validation phase, prototypes produced from PLEASE specifications 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. We believe the early production of executable prototypes enhances the validation process.

In the *refinement phase*, the Vienna Development Method[134] is used to incrementally transform the abstract specification into a concrete implementation. The refinement phase can be decomposed into a number of steps, each of which consists of a *design transformation* and its verification. If a design transformation is especially complex, it may be decomposed into a *design phase*, in which more of the structure of the system is described, and an *implementation phase*, in which components of the system are constructed.

In ENCOMPASS, a design transformation may be verified using any combination of mathematical reasoning[110,163,250], testing[91,131,176], technical review [87,242], and inspection. The use of PLEASE specifications enhances the verification of system components using

either testing or proof techniques. A prototype produced from the specification for a component can 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 formal argument presented as in a mathematics text. ENCOMPASS supports the *rigorous*[134] 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.

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 larger and larger units, each of which is shown to satisfy the specifications[88]. If errors are found and corrected in a low level module, the correctness of any previously verified modules which use the low level module may have to be redetermined. ENCOMPASS provides tools to aid in the hierarchical integration and testing of programs. These tools ensure that before a module is tested, all modules that it uses are tested before tests of that module are begun. 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 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. The tracking and control of a project is further enhanced by the use of configuration and project management systems.

8.2. Configuration and Project Management

A project management system must identify, control, and monitor the tasks that comprise the software development and maintenance processes. Different models of these processes lead to different approaches to project management[211]. For example, Osterweil uses a "process program" approach[190]; he feels the tasks involved can be described using a notation similar to programming languages.

The ENCOMPASS project management system is based on a *management by objectives* approach[106]: each step in the development process satisfies an objective by producing a milestone. The 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. For example, the objective of the requirements definition phase is to describe the properties that the software sys-

tem to be constructed must satisfy; the milestone for this phase is a specification which lists these properties. In ENCOMPASS, the PLEASE[232–234] executable specification language provides many machine–recognizable milestones; for example, the existence of a PLEASE specification, the production of a verified implementation, and the correct execution of a set of test cases by a prototype can all be recognized by the system.

Configuration management is concerned with the identification, control, auditing, and accounting of the components produced and used in software development and maintenance[2,27,28,44]. A number of different configuration management systems have been proposed, developed and/or used[85,86,126,160,183,197,214,236,239,246,259]. In ENCOMPASS, the configuration management system is responsible for maintaining the consistency of, integrity of, and relationships between the products of software development. Many models of software configurations have been proposed[12,94,128,154,155,173,188,196,251]. A configuration model which is understood and accepted by everyone involved can enhance communication, aid project management and increase product quality.

In ENCOMPASS, software configurations are modeled using a variant of the *entity–relationship model*[57,58,191] which incorporates the concept of *aggregation*[216,217]. At present, most databases do not provide the features necessary to support integrated software development environments[25]; our model provides us with a natural way to describe software and also has a convenient implementation on conventional computer systems.

8.2.1. Configuration Management in ENCOMPASS

In ENCOMPASS, an *entity* is a distinct, named component; not all components are named. In one sense, an entity is any component important enough to be recognized by the configuration

management system. An example of an entity is a file, which could contain the source code for a program, some test data, or an executable program. Entities may belong to different *entity sets*; in other words, there may be different types of entities. For example, the current implementation supports entities of type "program", which may be decomposed into modules manipulated by IDEAL, and entities of type "document", which consist of sections and are manipulated by the ENCOMPASS document tool. An entity may have *attributes* which describe its properties or qualities. For example, a file could have attributes such as "size", "owner", "permissions", and "modify time". An entity may be decomposed into smaller components, which may or may not be entities themselves. For example, a file might be composed of paragraphs of text or statements in a programming language.

Two or more entities may have a *relationship* between them. For example, the entities containing the source and object code for a routine might have the relationship "compiled–from" between them. A relationship may also have attributes, for example the time the compile took place. A group of entities with a relationship between them may be abstracted into an *aggregate* entity; this entity would have entities as the values of some or all of its attributes. For example, the symbol tables, proofs, source code, load modules and test cases used by IDEAL are all grouped together into a single entity known as a "module". An entity which does not have entities as the values of any of its attributes is known as a *simple entity*. An example of a simple entity is a file containing the source code for a routine, which has the attributes "language", "modify time", and "size".

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 include *images* of entities in this base

view. In ENCOMPASS, access to components is controlled through the use of views. For example, different views of the software system being developed may be used by the development and quality assurance teams. The development team may use a view which includes all the specifications and software being developed. However, the quality assurance team may use a different view which contains only the specifications, executable code and, in addition, the test cases. Different views may be used during different phases of a development project; views may also be used to restrict the activities of a programmer to a particular group of modules.

In general, a *version* is the state of an entity at a particular point in time; more precisely, a distinction can be drawn between *linear revisions* and *parallel versions*[52]. The current implementation of ENCOMPASS allows a sequential revision of any object to be saved or restored at any time, but does not provide support for parallel versions. When saving or restoring revisions of aggregate objects, a "dangling pointer" problem arises; for example, consider a view of an aggregate object A which contains an image of an object S. When a revision of the view is stored, should a reference to, or copy of S be stored? If a reference to S is stored, should it refer to a particular revision or the latest copy?

The current implementation of ENCOMPASS does not address these problems in a deep way; images are implemented as symbolic links[1] to the latest copy of an object. In the example above, the version of the view contains a symbolic link which will point to the latest copy of S when the version is restored. The SAGA project is developing a configuration librarian called Clemma to address these and other issues[52]. The combination of Clemma and the ENCOMPASS project management system should provide significant support for software development.

---

[1] A symbolic link contains the name of the file to which it is linked. Symbolic links may span file systems and may refer to directories. The file to which the link refers need not exist at the time the link is created.

8.2.2. Project Management in ENCOMPASS

The ENCOMPASS project management system is based on a hierarchical project organization: each member of the team has at least one immediate supervisor and may have a number of subordinates. The system is organized around *work trays*[47], which provide a mechanism to manage and record the allocation, progress, and completion of work within a software development project. Each member of the project has a *workspace* containing a number of trays. Each tray holds *tasks* containing the *products* produced and used during software development; these products are stored as entities within the ENCOMPASS configuration management system.

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*; he may then transfer these tasks to an *in–progress* tray where he will perform the actions required of him and produce new products. An *output tray* can be used to return a task to its originator. A user may create a new task in an in–progress tray that he owns and transfer it to other user's input tray; for example, a team leader can decompose a task into sub–tasks and send the sub–tasks to his subordinates. A task that has been transferred back into the in–progress tray of its creator may be marked as complete and transferred to a *file tray* for long term storage.

To further clarify the operation of the configuration and project management systems, we will present an example of software development. We will follow the construction of a small, multi–module system by a team of programmers; the example will include specification, validation, refinment and verification.

8.3. An Example

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 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 assignment by electronic mail, he creates a project library called *combinatorics* 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 39 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

[2]The number of k–combinations of n items is equal to n!/(k!(n–k)!)

ORIGINAL PAGE IS
OF POOR QUALITY

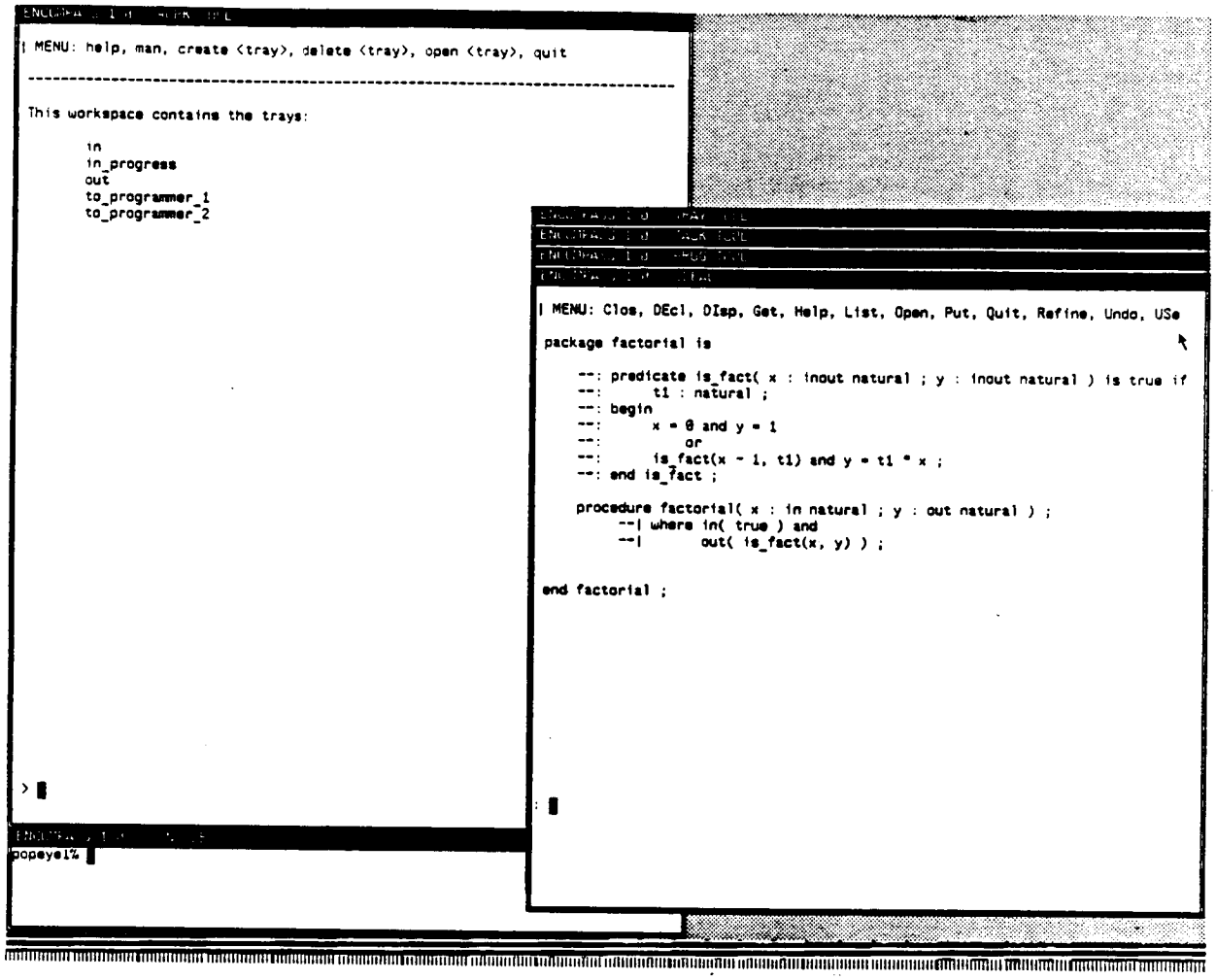


Figure 39. Team leader's screen after specifying *factorial*

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 *combinatorics* 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.

After *factorial* is specified, it is prototyped. From IDEAL, the team leader issues a command that 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*[88] 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 40 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*. 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

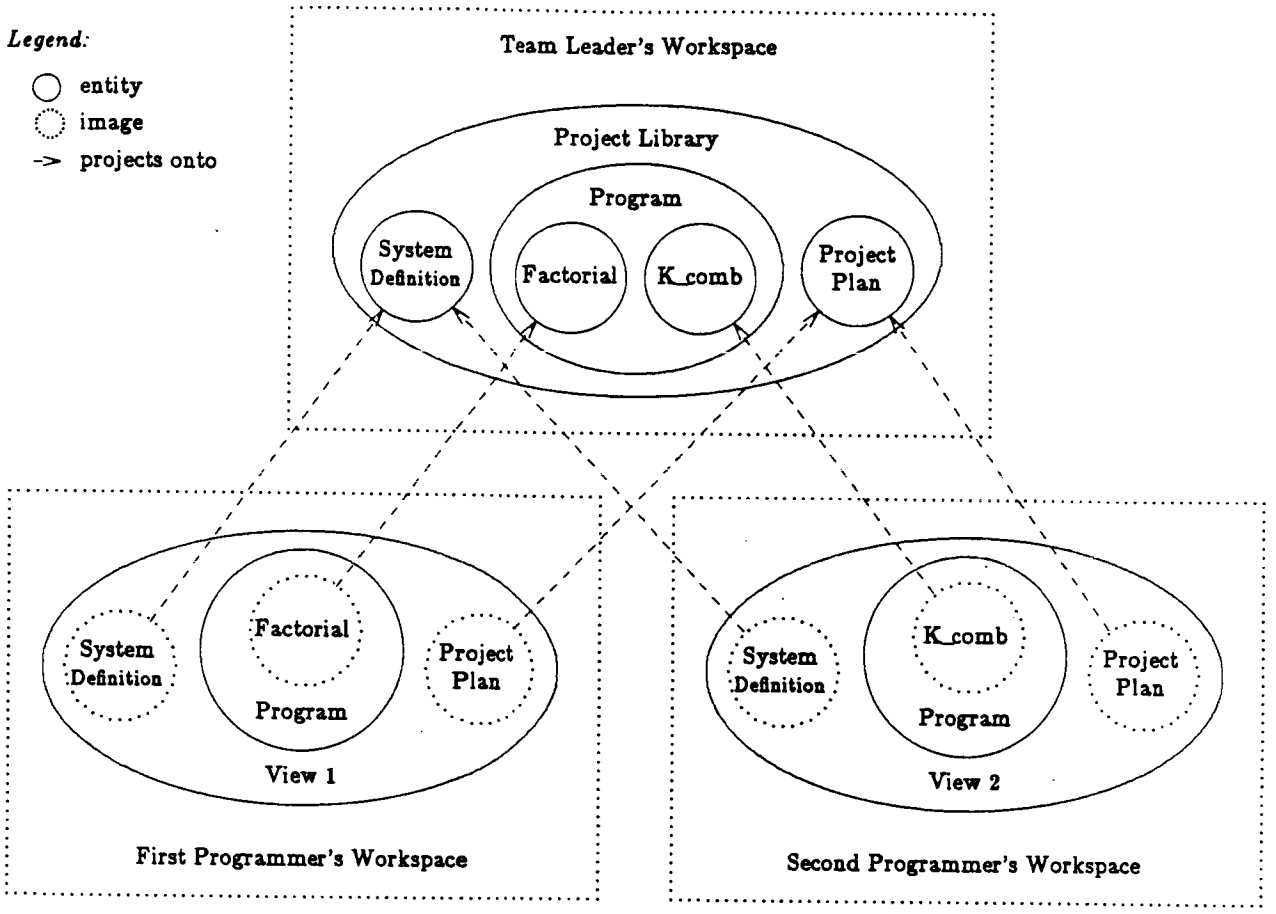


Figure 40. Different Views of the Project Library

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.

At present, ENCOMPASS is primarily an environment for software development; however, it can easily be combined with other tools produced by the SAGA project to provide support for a larger portion of the life–cycle. For example, the addition of a global repository allows ENCOMPASS to provide support for software reuse.

8.4. Software Reuse

It has been suggested that the *reuse* of software can significantly reduce the cost of program development[31,125], and systems which contain libraries of previously coded modules and/or a number of standard designs or program schemas have been proposed[80,142,156,165,174]. Some have suggested that the combination of reuse with object oriented design is particularly effective[175]. When ENCOMPASS is used with a *global repository*, 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 global repository and later retrieved. The repository can support a number of projects, both accepting and supplying components for reuse in all phases of development.

Figure 41 shows the flows of control and data among the global repository and a number of projects using ENCOMPASS. As is usual in ENCOMPASS, there is a workspace for each programmer and a project library for each project; the global repository is common to all projects. Each programmer controls his own workspace, while the *project leader* controls the library for

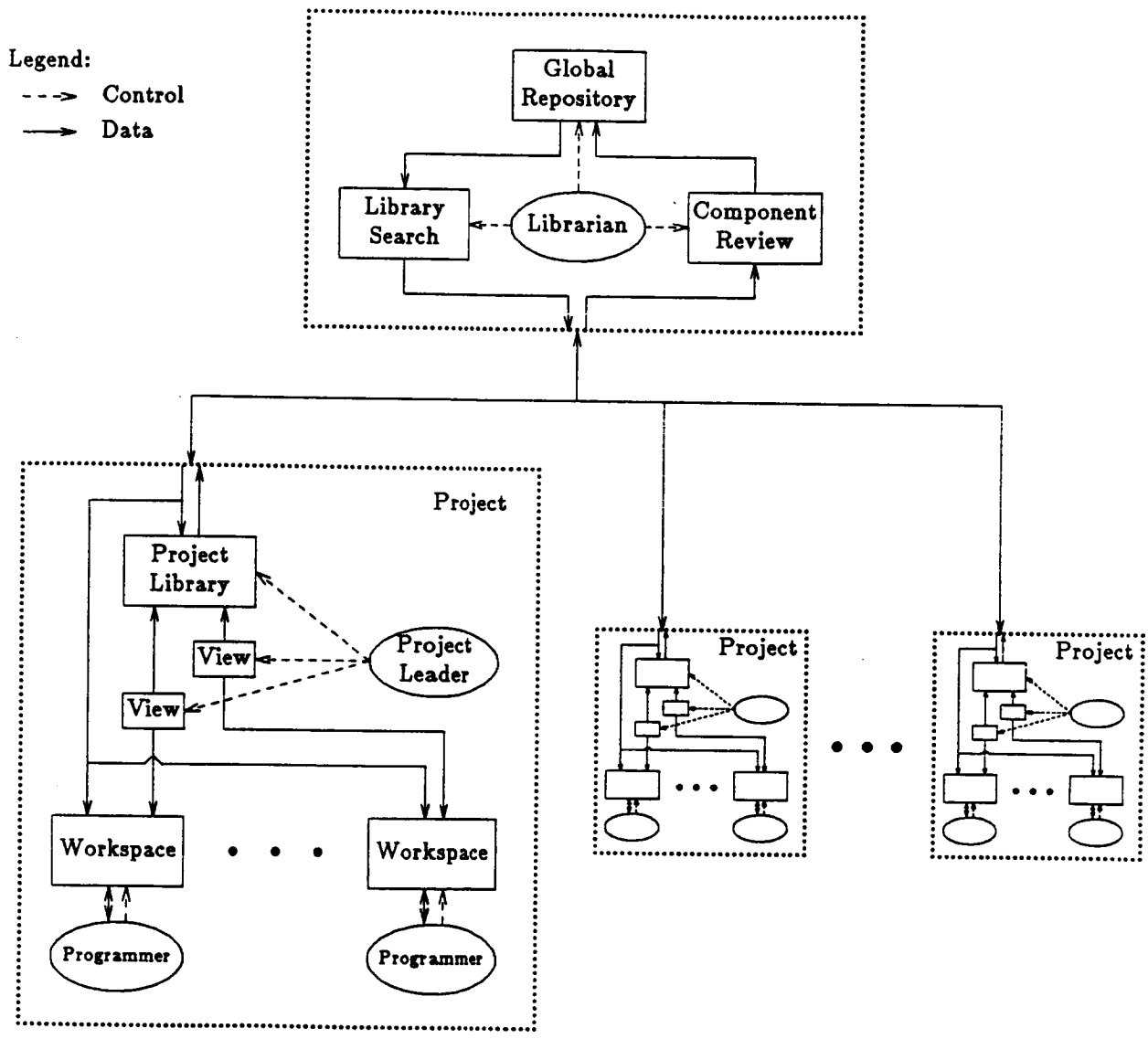


Figure 41. Global repository used with ENCOMPASS

his project and the *librarian* controls the repository. All components which are accessed by more than one programmer reside in either the project libraries or global repository where they are

controlled by either the project leaders or the librarian. A programmer accesses the components he is working with through his workspace. The workspace may actually contain these components, or it may reference components in the project library through a view. The project library contains all the components associated with a particular project; the project leader controls access to components by controlling the views of the library.

The global repository contains components available for reuse on all projects and is read–only to all but the librarian. The librarian controls which components will be saved for reuse and how they will be available. When a project leader feels that a component may be useful for reuse on other projects he submits it to the librarian who performs a *component review* to determine if the component meets the minimum standards for correctness, reliability, documentation, and generality. If the component meets these standards then the librarian must decide how to index the component for later retrieval.

Each component available for reuse is associated with a number of *key words* which describe its structure, function and quality[3]. To search the library for components that may be useful, a programmer uses simple retrieval tools, specifying the key words in which he is interested using a regular expression. The tool returns a list of components, each of which is associated with the key words he specified. The programmer may then create an image or copy of any components which are of interest in his workspace and examine them in more detail.

For example, suppose a programmer needs a verified module which implements a stack of strings. By searching the library on the key words "stack" and "verified" he might discover that a verified module implementing a stack of integers existed in the global library. Assuming he

[3] For example a module might have met technical review standards, be well tested, be proven by a period of use, or possibly even be formally verified with respect to its specification.

had the proper access permissions, he could then make a copy of this module in his workspace and modify it to implement a stack of strings. The programmer may be able to reuse more than just the source code for the module. Associated documentation, test cases, and proof of correctness would also be retrieved, and could be possibly be modified and reused in the new development.

With the addition of a global repository, ENCOMPASS can provide support for a much wider range of software engineering activities. As another example, by combining ENCOMPASS, the global repository, and Notesfiles, a system to provide basic support for software maintenance can be constructed.

8.5. Change Control

Software typically remains in use long after it is developed; as operating environment and user needs change, the system must be modified to meet new demands. To ensure system reliability, integrity and availability, these modifications must be performed in a controlled manner. A *change control system* provides methods and tools to record, effect, and monitor changes to an installed software system. For example, assume analysts and programmers are responsible to a *Change Control Board* for their contributions to the maintenance activity; bugs and requests for modifications are received by the Board, which decides whether requests should be satisfied or ignored and manages the necessary resources.

Figure 42 shows a simplified diagram of the flow of information that occurs within the maintenance group. Customers submit *user change requests*, which may be either bug reports or proposals for enhancements to the software; the Change Control Board assigns these requests to an analyst for further examination. The analyst reviews the requests and produces *program*

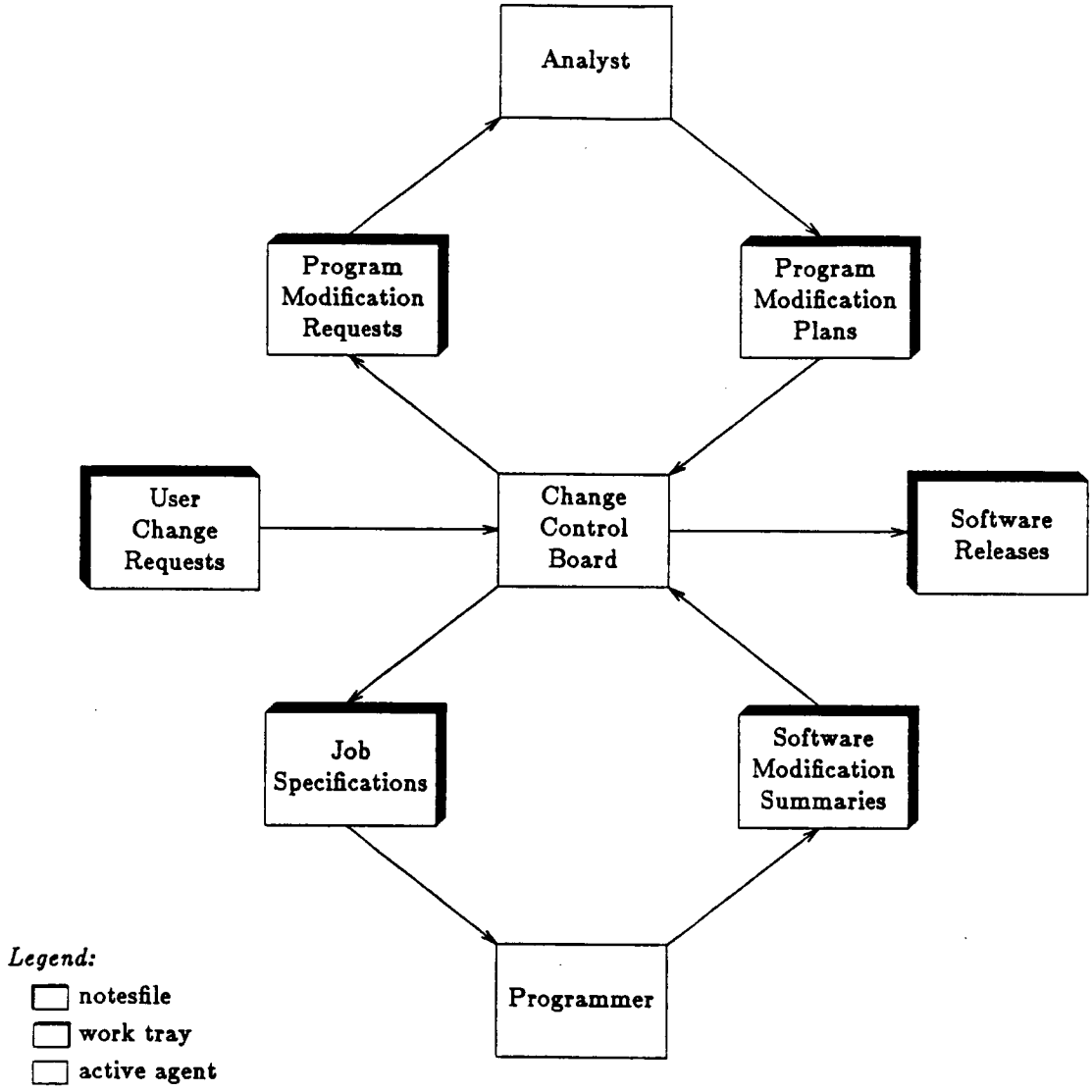


Figure 42. Data flow for change requests

modification plans for those that are valid; these plans are forwarded to the Board for approval and scheduling. The Board may either assign 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 performs the appropriate software modifications and submits a *software modification summary* to the Change Control Board. The Board examines the summary and may either produce a new software release or generate a new job specification for further 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. While a manual system can prove valuable in controlling the software maintenance process, it can be further enhanced by the addition of automated tools.

8.5.1. Automating Change Control

The *Notesfiles* system is a distributed project information base which operates on networks of heterogeneous machines under the Unix operating system [83,84]. Within the SAGA project, we have used the Notesfile system to organize technical discussions and product reviews, track problems and grievances, keep agendas and minutes, and maintain documentation. A notesfile is a sequence of *notes*, each of which may have a sequence of *responses*; each note or response has a title, author, and creation time. To maintain consistency, updates to notesfiles are transmitted among networked systems using the standard electronic mail facility. A library and standard interface permits user programs to submit notes and responses. This library has been used in the construction of automatic logging and error reporting facilities in both software and test harnesses. Notesfiles are used with ENCOMPASS to automate the change control system.

Figure 42 also shows the basic implementation of the ENCOMPASS change control system. User change requests enter the system by electronic mail and are stored in the *User Change Requests* notesfile. In the automated system, a change request is a form that can be filled in manually or can be generated by software error reporting tools; using the notesfile/mail interface, change requests can be generated from either local or remote sources. Program modification requests and program modification plans are tasks, which are passed between the Change Control Board and the analysts via work trays of the same names. The Board and the programmers also use work trays to exchange job specifications and software modification summaries. The notesfile *Software Releases* allows new versions of controlled software to be distributed to both local and remote sites.

To explain the operation of the change control system in more detail, we will look at an example of its use. We will follow a change request from its entry into the system through performance of the required modifications and installation of a new system.

8.5.2. An Example of Change Control

Assume that the *combinatorics* project described earlier in this chapter has been completed; versions of *factorial* and *k_comb* are stored in the global repository. Figure 43 shows that conceptually these versions appear in the repository as independent entities. Actually, they are interdependent; a storage management system can make use of this fact to optimize disk usage.

Assume that a user is dissatisfied with the performance of the *factorial* procedure; he enters a request for enhancement into the change control system by placing a note in *User Change Requests*. When the Change Control Board manager inspects the notesfile, he finds the request and creates a task called *modify_factorial* in his workspace. This task contains not only a

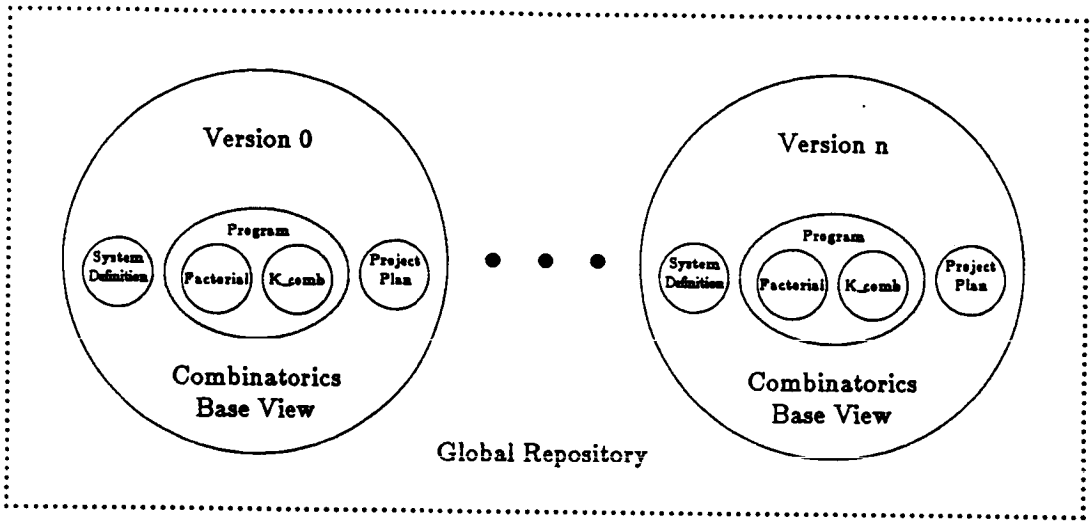


Figure 43. Global repository containing versions of *k_combinations* base view

description of the problem, but a view of the *combinatorics* project_library which is stored in the global repository; Figure 44 shows that the components of *combinatorics* can be accessed read-only using this view. After the task is created, the manager transfers it to the *Program Modification Request* tray for examination by an analyst; the analyst transfers the task to an in-progress tray in order to process it. The analyst examines the user's request and uses the view of *combinatorics* to inspect the current implementation. When his study is complete, the analyst creates a document describing the steps necessary to effect the desired changes and transfers the task to the Change Control Board through the *Program Modification Plan* tray.

When the Change Control Board manager receives the program modification plan, he transfers the task to his in-progress tray and convenes the Change Control Board. The Board discusses the plan and accepts the proposed modifications; the manager must then produce a job

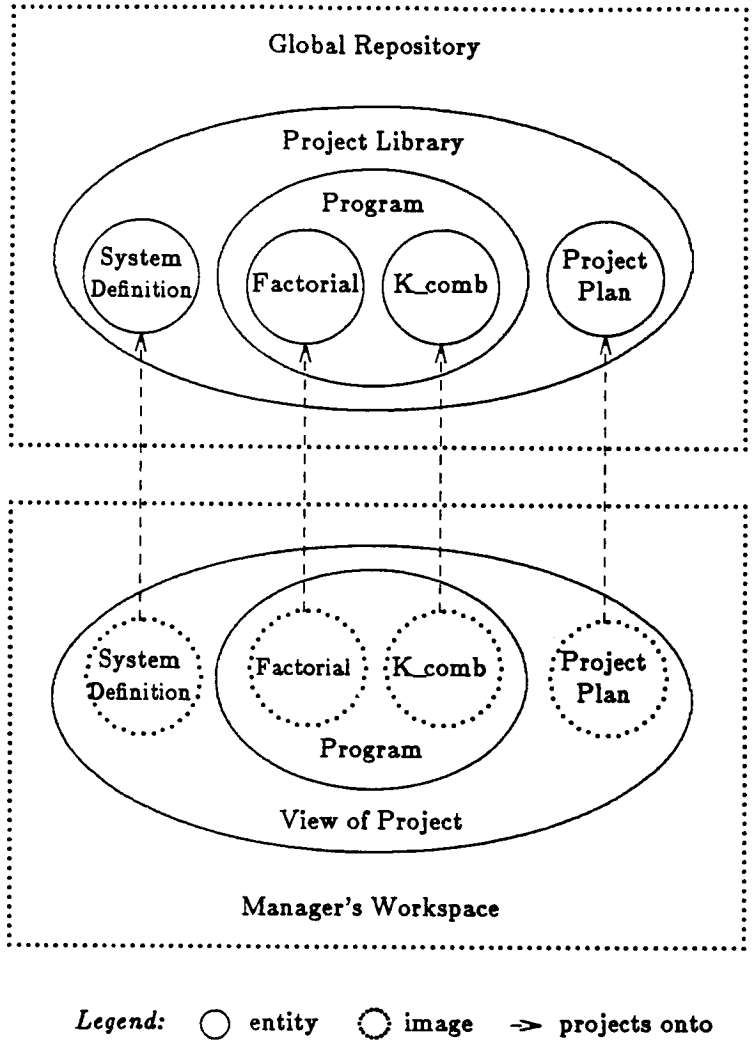


Figure 44. Workspace containing view of *combinatorics*

specification for the programmer. In order to effect the changes, the necessary components are checked out of the global repository into the task. This involves both replacing the images by actual objects and locking the modules in the repository to prevent conflicting modifications by other programmers. If desired, the next version number for the objects to be modified can also

be assigned at this time. When check out is completed, the task is transferred to a programmer using the *Job Specification* tray.

When the programmer receives the task he transfers it into an in-progress tray in his workspace and performs the required changes. The programmer not only modifies the source code for *factorial*, but checks that the documentation, test cases and proof of correctness are all up to date. Figure 45 shows the new version of *combinatorics* developed by the programmer; only the *factorial* module is modified. When the modifications are complete, the programmer creates a document summarizing the modifications and transfers the task to the Change Control Board using the *Software Modification Summary* tray.

When the software modification summary is received, the manager again convenes the Change Control Board. The Board evaluates the modifications and makes a recommendation as to whether the work constitutes a valid version[4]. In our example, the modifications pass the review, and the manager checks the new version of *combinatorics* into the global repository. Check in involves both storing the modified software and releasing the locks on the modified components. Figure 46 shows the global repository after check in is complete. The manager then announces a new release of *combinatorics* through the *Software Release* notesfile; if desired, the source code could also be distributed automatically.

ENCOMPASS is an environment for programming-in-the-large using the PLEASE executable specification language. It supports the use of IDEAL as an environment for programming-in-the-small. In this chapter, we have described ENCOMPASS in detail and given examples of its use. ENCOMPASS is based on Fairley's traditional or *waterfall* life-cycle[88], extended to

[4] In a more complex change control system, the evaluation of the new software might be performed by a quality assurance group; our model and implementation are easily extended to support this.

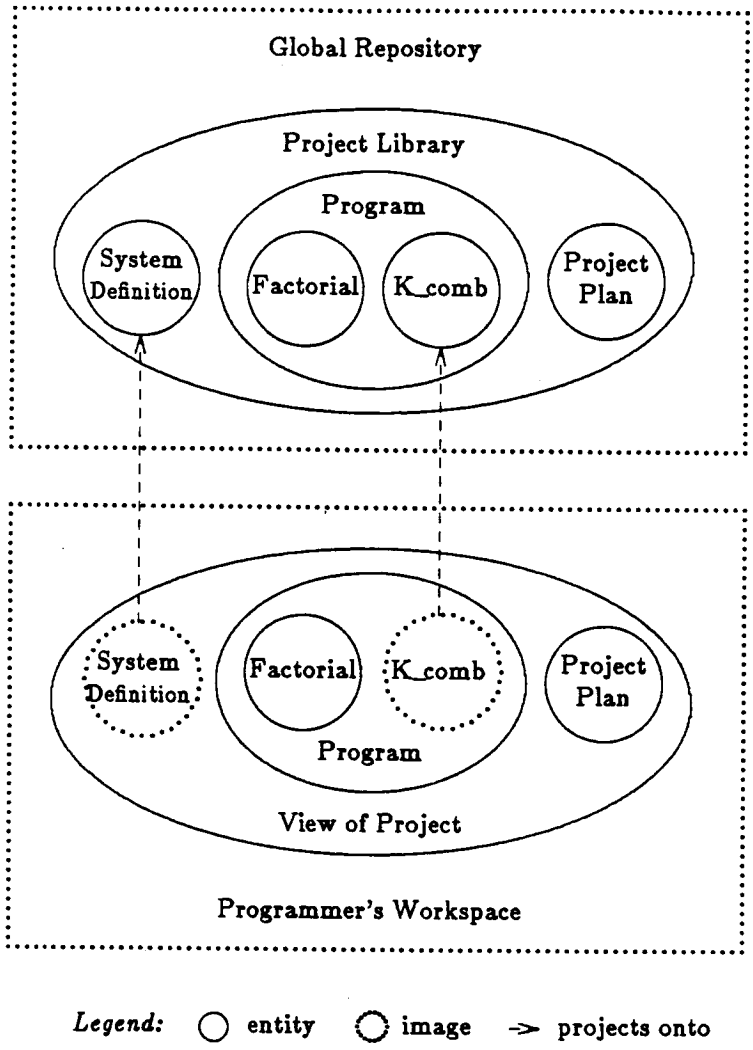


Figure 45. Workspace containing modified view of *combinatorics* project

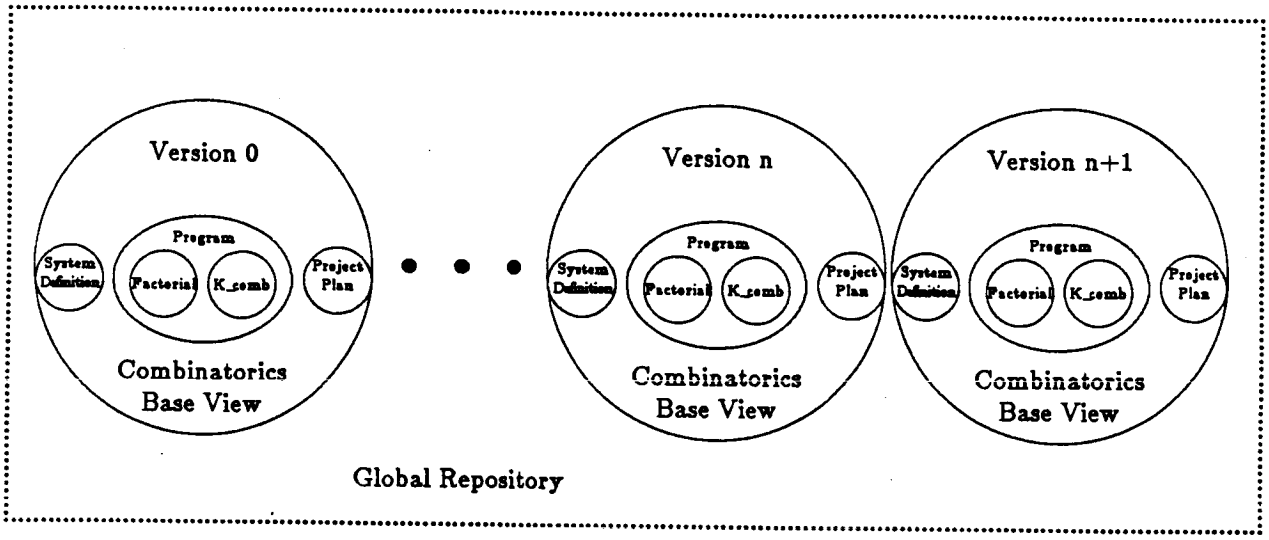


Figure 46. New version of *combinatorics* installed in global repository

support the use of executable specifications and VDM. In ENCOMPASS, the traditional life-cycle is extended to include a separate phase for user validation; the design and implementation steps are also combined into a single refinement phase. In ENCOMPASS, the user accesses and modifies components using a set of software development tools. The configuration management system structures the software components developed by a project and stores them in a project data base. The project management system uses facilities provided by the configuration management system to control both access to the data base and interactions between developers.

In ENCOMPASS, software is modeled as *entities* which have *relationships* between them. These entities can be structured into complex hierarchies which may be accessed through different *views*. The project management system implements a *management by objectives*[106] approach to software development; each phase in the life-cycle satisfies an objective by producing

a *milestone* which can be recognized by the system. ENCOMPASS can be used with a central repository to support software reuse; specifications, source and object code, documentation, test cases, and proofs of correctness can all be stored and retrieved. An automated change control scheme can be implemented using ENCOMPASS, the global repository and Notesfiles.

At present, the implementation of ENCOMPASS is skeletal; the major components have all been implemented, but are not particularly robust or user–friendly. Even with this limitation, we feel the work is promising; we believe that the use of future environments similar to ENCOMPASS will enhance the development, reuse, and maintenance of software.

CHAPTER 9.

SUMMARY AND CONCLUSIONS

In this dissertation we have described PLEASE[232–234], an Ada–based, wide–spectrum, executable specification and design language; IDEAL[231], an environment for programming–in–the–small using PLEASE; and ENCOMPASS[52,230,231], a simple environment for programming–in–the–large. Together, these form an integrated system to support incremental software development in a manner similar to VDM. In ENCOMPASS, software is specified using a combination of natural language and PLEASE. In PLEASE, software can be specified using Horn clauses: a subset of first–order, predicate logic. In ENCOMPASS, PLEASE specifications can be incrementally refined into Ada implementations. Each step is verified before the next is applied; therefore, errors can be detected and corrected sooner and at lower cost. In ENCOMPASS, refinement steps can be verified using peer review, testing, or proof techniques.

Executable prototypes can be automatically constructed from PLEASE specifications by translating pre– and post–conditions into Prolog procedures. 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[5]). As logic programming progresses, new versions of PLEASE can be built based on more powerful logics.

PLEASE prototypes can enhance the validation, design, and verification processes. During the validation phase, 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. PLEASE prototypes can also be used to

verify the correctness of refinements. Most simply, the prototype produced from a PLEASE specification can be used as a test oracle against which implementations are compared. In a more complex case, the prototypes produced from the original and refined specifications can be run on the same data and the results compared. PLEASE specifications also enhance the verification of system components using proof techniques; for the purpose of formal verification, the refinement process can be viewed as the construction of a proof in the Hoare calculus[120,163].

IDEAL is an environment concerned with the specification, prototyping, implementation and verification of single modules. IDEAL provides facilities to create PLEASE specifications, construct prototypes from these specifications, validate the specifications using the prototypes produced, refine the validated specifications into Ada implementations, and verify the correctness of the refinement process. IDEAL is an environment for the *rigorous*[134] 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. Proofs may range from a very detailed, completely formal proof using mechanical theorem proving, to a development "annotated" with unproven verification conditions. Parts of a project may use detailed mechanical verification while other, less critical parts may be handled using less expensive techniques. Our experience so far leads us to believe that the complete, mechanical verification of large programs will be prohibitively expensive; however, inexpensive methods can eliminate a large percentage of the verification conditions generated during a development. By eliminating these "trivial" verification conditions, the total number is reduced so that the verification conditions remaining can be more carefully considered by the development personnel.

ENCOMPASS provides facilities to store, track, manipulate and control all the objects used in the software development process: documents, specifications, source code, proofs, test data,

and load modules are all supported. ENCOMPASS also provides mechanisms to support the interactions among developers; the system allows the creation, decomposition, distribution, monitoring and completion of tasks. In ENCOMPASS, the configuration management system structures the software components developed by a project, while the project management system uses facilities provided by the configuration management system to control both access to data and interactions between developers. ENCOMPASS is based on a traditional life–cycle, modified to support the use of executable specifications and VDM. In ENCOMPASS, some modules of a system may be developed using PLEASE and IDEAL, while others are developed using conventional techniques. This allows the practical power of Ada and the formal power of PLEASE to be combined in a single project. ENCOMPASS can be extended with a central repository to support software reuse; we have also constructed an automated change control system based on ENCOMPASS.

9.1. System Status

The ENCOMPASS environment has been under development since 1984. A prototype implementation has been operational since 1986; it is written in a combination of C, Csh[1], Prolog and Ada. The prototype implementation of IDEAL 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.

[1] Csh is a command interpreter on Unix which supports many of the features found in modern programming languages. A sequence of shell commands may be saved and run as a program.

PLEASE, IDEAL and ENCOMPASS have been used to develop a number of programs, including specification, prototyping, and mechanical verification. At present, all the programs developed have been less than one hundred lines in length, but some have included more than one module, allowing demonstrations of the ENCOMPASS configuration control and project management systems.

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 and characters. The current implementation of PLEASE is based on the UNSW Prolog interpreter[208] and the Verdix Ada Development System[10]; it runs under Berkeley Unix on a Sun 2/170. The Prolog interpreter and Ada program run as separate processes and communicate through pipes[2]. This implementation is somewhat expensive; for example, there is a five CPU second overhead to start the Prolog interpreter, but this is incurred only once during program execution. A procedure call from Ada to Prolog costs about forty milliseconds excluding parameter conversion. As an example of actual performance, the sort prototype produced from the specification given in Chapter 4 can process a list of length four in an average of .9 seconds and a list of length five in an average of 4.7 seconds.

The combination of algebraic simplification and simple proof tactics implemented in ISLET seems to work very well; in our experience, it can eliminate between fifty and ninety per cent of the verification conditions generated during refinement. For example, the design transformation presented in Chapter 6 consists of twenty–six steps, only two of which generated verification conditions that could not be certified by these methods. The example presented in Chapter 7 also consists of twenty six steps, only four of which generated verification conditions that did not

[2]Pipes are a buffering mechanism implemented in Unix.

yield to the simple approach. The simple methods run very quickly: less than one second response time in all the cases examines so far. The use of TED is very expensive; for example, the first verification condition in Figure 25 can be certified in about five CPU seconds simply by invoking the theorem prover on the file produced by ISLET. The second verification condition in Figure 25 can not be proved in this manner; it requires a considerable investment of user time to decompose it into a number of lemmas.

9.2. Future Research

The research described in this dissertation is presently at the "proof of concept" stage: we have demonstrated that an integrated environment to partially automate a development method similar to VDM can be constructed, and it seems likely our implementation can scale up. We see the project developing in five main directions. For one, the current implementation is primitive at best and skeletal at worst. We plan to continue a straight forward expansion of the functionality of ENCOMPASS, as well as constructing a more realistic implementation of the system. For example, the current configuration and project management systems are rudimentary in both concept and implementation; the SAGA group is constructing second generation systems with enhanced functionality and performance[52].

As another example, although commercial Prolog compilers are available, the current implementation of PLEASE is interpreter–based; we plan to experiment with a Prolog compiler based implementation of PLEASE. This system will allow us to evaluate a number of technical problems in a state–of–the–art environment. For example, how easy will it be to freely mix Ada and Prolog modules? In the current implementation, Ada procedures can not be called during the execution of a Prolog prototype; there is an Ada to Prolog, but not a Prolog to Ada interface.

This means that even if an Ada implementation is available, a Prolog implementation must sometimes be used. Second, it is unclear how easily machine–level data structures can be shared between Prolog and Ada code. In the current Ada to Prolog interface, all the parameters to a procedure are converted on both call and return; this is a major barrier to scaling up the implementation.

A second major direction for future research is the extension of PLEASE itself. For example, at present PLEASE supports a small set of pre–defined types; these types were chosen to expedite the implementation of the translator and proof procedures for the language. We plan to experiment with extensions to PLEASE that incorporate more complex and difficult to implement types; we hope to create a new version of the language which strikes a balance between simplicity, expressiveness and efficiency. We also plan to expand the facilities for user type definition in PLEASE; at present, some type of algebraic approach seems the most promising. We also plan to experiment with derivatives of PLEASE based on different deduction engines or more powerful logics; for example, an extension of Prolog for full first–order logic[222] or some form of *narrowing*[136,201].

The third major direction is the extension of ENCOMPASS to support artificial intelligence techniques and knowledge–based tools. In the present implementation, the algebraic simplifier and simple proof procedures in ISLET incorporate knowledge–bases, but they are difficult to examine, debug, or extend. We plan to upgrade these implementations as well as investigating the use of knowledge–based techniques for program synthesis and configuration control. For example, in a current experiment[229] we are extending ISLET with a knowledge–based assistant which uses *deductive synthesis*[72,100–102,123,170] techniques. During the refinement process, the assistant can give advice on routine design and implementation decisions. The assistant also contains a library of *program schemas* which can be instantiated during development.

The fourth major area is acquiring experience with the system and collecting experimental data on its performance. For example, our experience so far leads us to believe that the simple methods in ISLET can certify fifty to ninety per cent of the verification conditions generated during development; however, much more data is needed to substantiate this claim. Our plans in this area involve three phases. In the first phase, a slightly modified version of the current system will be used in a classroom environment to generate data on the suitability of data types and proof procedures using small test programs. When this phase is completed, the results will be used in the design and construction of a new set of tools which can be used in the construction of medium sized components. In the third phase, these tools will be put into use.

The fifth major thrust will be the extension of PLEASE, IDEAL and ENCOMPASS to support more of the software life–cycle; although software development is important, the maintenance phase is currently more costly. At first our work will be mostly conjecture, but much of the material developed during experimental evaluation of the system can be used in the creation of a maintenance test bed. Although the work described in this dissertation is preliminary, we feel it is promising; it will continue at both the University of Illinois and the University of Colorado. We feel that the use of future environments similar to ENCOMPASS will enhance the specification, design, implementation and maintenance of software.

REFERENCES

1. *Special Issue on Rapid Prototyping: Working Papers from the ACM SIGSOFT Rapid Prototyping Workshop.* **Software Engineering Notes** (December 1982) vol. 7, no. 5.

2. "Software Configuration Management", Standard 828–1983, IEEE Computer Society, Los Angeles, California, 1983.

3. *Special Issue: Proceedings of VERkshop III — A Formal Verification Workshop.* **Software Engineering Notes** (February 1984) vol. 10, no. 4.

4. "Peer Review of a Formal Verification / Design Proof Methodology", NASA Conference Publication 2377, 1985.

5. "Quintus Prolog Users Guide and Reference Manual (Version 3)", Quintus Computer Systems, Palo Alto, California, 1985.

6. *Special Issue on the Gandalf Environment.* **Journal of Systems and Software** (May, 1985) vol. 5, no. 2.

7. **Proceedings of the Conference on Object–Oriented Programming Systems, Languages and Applications.** ACM, Baltimore, MD, 1986.

8. *Proceedings of the International Workshop on the Software Process and Software Environments.* **Software Engineering Notes** (August 1986) vol. 11, no. 4.

9. *Proceedings of the NRL Invitational Workshop on Testing and Proving: Two Approaches to Assurance.* **ACM Software Engineering Notes** (October 1986) vol. 11, no. 5, pp. 63–85.

10. "VADS Reference Manual", Verdix Corporation, Chantilly, Virginia, 1986.

11. **Proceedings of the 4th International Workshop on Software Specification and Design** (April 1987).

12. 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.

13. Alagic, Suad and Michael A. Arbib. **The Design of Well–Structured and Correct Programs.** Springer–Verlag, New York, 1978.

14. 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.

15. Auernheimer, Brent and Richard A. Kemmerer. *RT–ASLAN: A Specification Language for Real–Time Systems.* **IEEE Transactions on Software Engineering** (September 1986) vol. SE–12, no. 9, pp. 879–889.

16. Balzer, Robert. *A 15 Year Perspective on Automatic Programming.* **IEEE Transactions on Software Engineering** (November 1985) vol. SE–11, no. 11, pp. 1257–1268.

17. 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.

18. 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.

19. ——. *Artificial Intelligence and Software Engineering.* **Proceedings of the 9th International Conference on Software Engineering** (1987) pp. 200–211.

20. Baskette, Jerry. *Life Cycle Analysis of an Ada Project.* **IEEE Software** (January 1987) vol. 4, no. 1, pp. 40–47.

21. 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.

22. Belkhouche, Boumediene and Joseph E. Urban. *Direct Implementation of Abstract Data Types from Abstract Specifications.* **IEEE Transactions on Software Engineering** (May 1986) vol. SE–12, no. 5, pp. 649–661.

23. Benzinger, Lee A. "An Abstract Model for the Stepwise Development of Programs", Report No. UIUCDCS–R–87–1335, Dept. of Computer Science, University of Illinois at Urbana–Champaign, 1987.

24. ——. "A Model and a Method for the Stepwise Development of Verified Programs", Report No. UIUCDCS–R–87–1339, Dept. of Computer Science, University of Illinois at Urbana–Champaign, 1987.

25. Bernstein, Philip A. *Database System Support for Software Engineering.* **Proceedings of the 9th International Conference on Software Engineering** (1987) pp. 166–178.

26. Berry, Daniel M. *Towards a Formal Basis for the Formal Development Method and the Ina Jo Specification Language.* **IEEE Transactions on Software Engineering** (February 1987) vol. SE–13, no. 2, pp. 184–201.

27. Bersoff, Edward H. *Elements of Software Configuration Management.* **IEEE Transactions on Software Engineering** (January 1984) vol. SE–10, no. 1, pp. 79–87.

28. Bersoff, E. H. *Software Configuration Management: A Tutorial.* **IEEE Computer** (January, 1979).

29. Berzins, Valdis and Michael Gray. *Analysis and Design in MSG.84: Formalizing Functional Specifications.* **IEEE Transactions on Software Engineering** (August 1985) vol. SE–11, no. 8, pp. 657–670.

30. 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.

31. Biggerstaff, Ted and Charles Richter. *Reusability Framework, Assessment, and Directions.* **IEEE Software** (March 1987) vol. 4, no. 2, pp. 41–49.

32. Bjorner, Dines. *On The Use of Formal Methods in Software Development.* **Proceedings of the 9th International Conference on Software Engineering** (1987) pp. 17–29.

33. Bjorner, D., T. Denvir, E. Meiling and J. S. Pedersen. "The RAISE Project – Fundamental Issues and Requirements", RAISE/DDC/EM/1, Dansk Datamatik Center, 1985.

34. Bjorner, D. and Cliff B. Jones. **Formal Specification and Software Development.** Prentice–Hall, Englewood Cliffs, N.J., 1982.

35. 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.

36. Blum, Bruce I. *The Life-Cycle - A Debate Over Alternative Models.* **Software Engineering Notes** (October 1982) vol. 7, pp. 18–20.

37. ——. *The Tedium Development Environment for Information Systems.* **IEEE Software** (March 1987) vol. 4, no. 2, pp. 25–34.

38. Boehm, Barry W., Terence E. Gray and Thomas Seewaldt. *Prototyping Vs. Specifying: a Multi-Project Experiment.* **Proceedings of the 7th International Conference on Software Engineering** (1984) pp. 473–484.

39. Booch, Grady. *Object-Oriented Development.* **IEEE Transactions on Software Engineering** (February 1986) vol. SE–12, no. 2, pp. 211–221.

40. Bowen, Kenneth A. *New Directions in Logic Programming.* **Proceedings of the ACM Computer Science Conference** (February 1986) pp. 19–27.

41. 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.

42. 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.

43. Brooks, Frederick P., Jr. *No Silver Bullet.* **IEEE Computer** (April 1987) vol. 20, no. 4, pp. 10–19.

44. Buckle, J. K. **Software Configuration Management.** Scholium International Inc., Great Neck, N.Y., 1982.

45. Burstall, R. M. and John Darlington. *A Transformation System for Developing Recursive Programs.* **Journal of the ACM** (January 1977) vol. 24, no. 1, pp. 44–67.

46. Buxton, J. N. and V. Stenning. "Requirements for ADA Programming Support Environments, *Stoneman*", U.S. Dept. Defense, 1980.

47. 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.

48. 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.

49. 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.

50. 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.

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

52. Campbell, R. H., H. Render, R. N. Sum, Jr. and R. B. Terwilliger. "Automating the Software Development Process", Report No. UIUCDCS–R–87–1333, Dept. of Computer Science, University of Illinois at Urbana–Champaign, 1987.

53. 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.

54. Chang, Chin–Liang and Richard Char–Tung Lee. **Symbolic Logic and Mechanical Theorem Proving.** Academic Press, New York, 1973.

55. Cheatham, Thomas E., Jr. *Reusability Through Program Transformations.* **IEEE Transactions on Software Engineering** (September 1984) vol. SE–10, no. 5, pp. 589–594.

56. 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.

57. 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.

58. ——. *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.

59. Clarke, Edmund Melson, Jr. *Programming Language Constructs for Which It Is Impossible To Obtain Good Hoare Style Axiom Systems.* **Journal of the ACM** (January 1979) vol. 26, no. 1, pp. 129–147.

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

61. Cohen, Jacques. *Describing Prolog by Its Interpretation and Compilation.* **Communications of the ACM** (December 1985) vol. 28, no. 12, pp. 1311–1324.

62. Colmerauer, Alain. *Prolog in 10 Figures.* **Communications of the ACM** (December 1985) vol. 28, no. 12, pp. 1296–1310.

63. Constable, Robert L. and Michael J. O'Donnell. **A Programming Logic.** Winthrop Publishers, Cambridge, Massachusetts, 1978.

64. 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.

65. Cowell, Wayne R. and Leon J. Osterweil. *The Toolpack/IST Programming Environment.* **Proceedings IEEE Softfair** (1983) pp. 326–333.

66. Dalen, Dirk van. **Logic and Structure.** Springer–Verlag, New York, 1983.

67. 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.

68. ——. *Logic Programming and Prolog: A Tutorial.* **IEEE Software** (September 1985) vol. 2, no. 5, pp. 53–62.

69. 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.

70. Defense, U. S. Dept. **Reference Manual for the ADA Programming Language ANSI/MIL–STD–1815A–1983.** Springer–Verlag, New York, 1983.

71. 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.

72. Dershowitz, Nachum. *Synthetic Programming.* **Artificial Intelligence** (1985) vol. 25, pp. 323–373.

73. Diaz–Gonzalez, Jose P. and Joseph E. Urban. *ENVISAGER: A Visual, Object–Oriented Specification Environment for Real–Time Systems.* **Proceedings of the 4th International Workshop on Software Specification and Design** (April 1987) pp. 13–20.

74. Dickover, Melvin E., Clement L. McGowan and Douglas T. Ross. *Software Design Using SADT.* **Proceedings of the ACM National Conference** (October 1977) pp. 125–133.

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

76. ——. **A Discipline of Programming.** Prentice Hall, Englewood Cliffs, New Jersey, 1976.

77. 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.

78. Dowson, Mark. *ISTAR – An Integrated Project Support Environment.* **Proceedings of the Second ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments** (December 1986) pp. 27–33.

79. Dwiggins, Don. *Prolog as a System Design Tool.* **Proceedings of the 16th Annual Hawaii International Conference on System Sciences** (1983) pp. 14–23.

80. Embley, David W. and Scott N. Woodfield. *A Knowledge Structure for Reusing Abstract Data Types.* **Proceedings of the 9th International Conference on Software Engineering** (1987) pp. 360–368.

81. Enderton, Herbert B. **A Mathematical Introduction to Logic.** Academic Press, New York, 1972.

82. Engel, S. Morris. **The Study of Philosophy.** Holt, Rinehart and Winston, New York, 1981.

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

84. 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.

85. Estublier, J. *Experience with a Data Base of Programs.* **Proceedings of the Second ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments** (December 1986) pp. 27–33.

86. 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.

87. Fagan, Michael E. *Advances in Software Inspections.* **IEEE Transactions on Software Engineering** (July 1986) vol. SE–12, no. 7, pp. 744–751.

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

89. Floyd, R. W. *Assigning Meanings to Programs.* **Proceedings of the AMS Symposium on Applied Mathematics** (1967) vol. 19, pp. 19–32.

90. Futatsugi, Kokichi, Joseph Goguen, Jose Meseguer and Koji Okada. *Parameterized Programming in OBJ2.* **Proceedings of the 9th International Conference on Software Engineering** (1987) pp. 51–60.

91. 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.

92. Gehani, Narain and Andrew D. McGettrick (eds.). **Software Specification Techniques.** Addison Wesley, Reading, Massachusetts, 1986.

93. Godel, Kurt. *Uber Formal Unentscheidbare Satze der Principia Mathematica und Verwandter Systeme I.* In: **From Frege to Godel**, J. van Heijenoort, ed. Harvard University Press, Cambridge, Mass., 1967.

94. Goguen, Joseph A. *Reusing and Interconnecting Software Components.* **Computer** (February 1986) vol. 19, no. 2, pp. 16–28.

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

96. ——. *Equality, Types, Modules and (why not?) Generics for Logic Programming.* **Logic Programming** (1984) vol. 1, no. 2, pp. 179–210.

97. ——. *Remarks on Remarks on Many–Sorted Equational Logic.* **SIGPLAN Notices** (April 1987) vol. 22, no. 4, pp. 41–48.

98. 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.

99. Goldberg, Adele. **Smalltalk–80: The Interactive Programming Environment.** Addison–Wesley, Reading, MA, 1984.

100. Goldberg, Allen T. *Knowledge–Based Programming: A Survey of Program Design and Construction Techniques.* **IEEE Transactions on Software Engineering** (July, 1986) vol. SE–12, no. 7, pp. 752–768.

101. Green, Cordell. *Application of Theorem Proving to Problem Solving.* **Proceedings of the First IJCAI** (1969) pp. 219–239.

102. ——. *Theorem–Proving by Resolution as a Basis for Question–Answering Systems.* In: **Machine Intelligence 4**, B. Meltzer and D. Michie, ed. American Elsevier, New York, 1969, pp. 183–205.

103. Greenbaum, Steven. "Input Transformations and Resolution Implementation Techniques for Theorem Proving In First–Order Logic", Ph. D. Dissertation, Dept. of Computer Science, University of Illinois at Urbana–Champaign, 1986.

104. Gries, David. **The Science of Programming.** Springer–Verlag, New York, 1981.

105. Gries, David and Jan Prins. *A New Notion of Encapsulation.* **Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments** (June, 1985) pp. 131–139.

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

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

108. ——. *Formal Specification as a Design Tool.* **Proceedings of the 7th ACM Symposium on the Principles of Programming Languages** (1980) pp. 251–261.

109. 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.

110. 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.

111. Habermann, A. Nico and David Notkin. *Gandalf: Software Development Environments.* **IEEE Transactions on Software Engineering** (December 1986) vol. SE–12, no. 12, pp. 1117–1127.

112. Halpern, J. Daniel, Sam Owre, Norman Proctor and William F. Wilson. *Muse - A Computer Assisted Verification System.* **IEEE Transactions on Software Engineering** (February 1987) vol. SE–13, no. 2, pp. 151–156.

113. Hamilton, A. G. **Logic for Mathematicians.** Cambridge University Press, Cambridge, 1978.

114. Hamilton, Margaret and Saydean Zeldin. *Higher Order Software – A Methodology for Defining Software.* In: **Tutorial: Automated Tools for Software Engineering**, Edward Miller, ed. IEEE Computer Society, New York, 1979, pp. 72–95.

115. 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).

116. 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.

117. Hatcher, William S. **Foundations of Mathematics.** W. B. Saunders, Philadelphia, 1968.

118. Henderson, Peter. *Functional Programming, Formal Specification, and Rapid Prototyping.* **IEEE Transactions on Software Engineering** (February, 1986) vol. SE–12, no. 2, pp. 241–250.

119. Henderson, Peter B. (ed.). **Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.**, 1986.

120. Hoare, C. A. R. *An Axiomatic Basis for Computer Programming.* **Communications of the ACM** (October 1969) vol. 12, no. 10, pp. 576–580.

121. ——. *Proof of Correctness of Data Representations.* **Acta Informatica** (1972) vol. 1, pp. 271–281.

122. Hoare, C. A. R. and N. Wirth. *An Axiomatic Definition of the Programming Language PASCAL.* **Acta Informatica** (1973) vol. 2, pp. 335–355.

123. Hogger, C. J. *Derivation of Logic Programs.* **Journal of the Association for Computing Machinery** (April 1981) vol. 28, no. 2, pp. 372–392.

124. Hopcroft, John E. and Jeffrey D. Ullman. **Introduction to Automata Theory, Languages and Computation.** Addison–Wesley, Reading, MA, 1979.

125. Horowitz, Ellis and John B. Munson. *An Expansive View of Reusable Software.* **IEEE Transactions on Software Engineering** (September 1984) vol. SE–10, no. 5, pp. 477–487.

126. 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.

127. Howden, William E. *Contemporary Software Development Environments.* **Communications of the ACM** (May 1982) vol. 25, no. 5, pp. 318–329.

128. 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.

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

130. Jackson, M. I. *Developing Ada Programs Using the Vienna Development Method (VDM).* **Software – Practice and Experience** (March 1985) vol. 15, no. 3, pp. 305–318.

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

132. Jensen, Randall W. and Charles C. Tonies. **Software Engineering**. Prentice–Hall, Englewood Cliffs, New Jersey, 1979.

133. Jones, Cliff B. *Constructing a Theory of a Data Structure as an Aid to Program Development.* **Acta Informatica** (1979) vol. 11, pp. 119–137.

134. ——. **Software Development: A Rigorous Approach**. Prentice–Hall International, Engelwood Cliffs, N.J., 1980.

135. ——. *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.

136. Josephson, Alan and Nachum Dershowitz. *An Implementation of Narrowing: The RITE Way.* **Proceedings of the Symposium on Logic Programming** (1986).

137. Kaiser, Gail E. and Peter H. Feiler. *An Architecture for Intelligent Assistance in Software Development.* **Proceedings of the 9th International Conference on Software Engineering** (1987) pp. 180–188.

138. Kamel, Ragui F. *Effect of Modularity on System Evolution.* **IEEE Software** (January 1987) vol. 4, no. 1, pp. 48–55.

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

140. 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).

141. Karp, Richard M. *Combinatorics, Complexity, and Randomness (1985 Turing Award Lecture).* **Communications of the ACM** (February 1986) vol. 29, no. 2, pp. 98–109.

142. Katz, Shmuel, Charles A. Richter and Khe–Sing The. *PARIS: A System for Reusing Partially Interpreted Schemas.* **Proceedings of the 9th International Conference on Software Engineering** (1987) pp. 377–385.

143. Kelly, John C. *A Comparison of four Design Methods for Real–Time Systems.* **Proceedings of the 9th International Conference on Software Engineering** (1987) pp. 238–252.

144. 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.

145. Kirslis, Peter A. "The SAGA Editor: A Language–Oriented Editor Based on an Incremental LR(1) Parser (Ph.D. Dissertation)", Report No. UIUCDCS–R–85–1236, Dept. of Computer Science, University of Illinois at Urbana–Champaign, 1985.

146. 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.

147. Knuth, D. E. and P. E. Bendix. *Simple Word Problems in Universal Algebra.* In: **Computational Problems in Abstract Algebra**, J. Leech, ed. Pergamon, New York, 1970, pp. 263–297.

C-3

148. Komorowski, Henryk Jan. "A Declarative Logic Programming Environment", Report TR-06-86, Center for Research in Computing Technology, Harvard University (also to appear in the **Journal of Systems and Software**), Cambridge, MA, 1986.

149. Komorowski, Henryk Jan and Jan Maluszynski. "Logic Programming and Rapid Prototyping", Report TR-01-86, Center for Research in Computing Technology, Harvard University (also to appear in the **Science of Computer Programming**), Cambridge, MA, 1986.

150. Kornfeld, William A. *Equality for Prolog.* **Proceedings of the International Joint Conference on Artificial Intelligence** (1983).

151. Kowalski, Robert. *Predicate Logic as a Programming Language.* **IFIP Proceedings** (1974) pp. 569–574.

152. ——. *Logic as a Computer Language.* In: **Logic Programming,** K. L. Clark and S. A. Tarnlund, ed. Academic Press, London, 1982, pp. 3–16.

153. 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.

154. Lampson, Butler W. and Eric E. Schmidt. *Organizing Software in a Distributed Environment.* **SIGPLAN Notices** (June 1983) vol. 18, no. 6, pp. 1–13.

155. ——. *Practical Use of a Polymorphic Applicative Language.* **Proceedings of the 10th ACM Symposium on Principles of Programming Languages** (January 1983) pp. 237–255.

156. Lanergan, Robert G. and Charles A. Grasso. *Software Engineering with Reuseable Designs and Code.* **IEEE Transactions on Software Engineering** (September 1984) vol. SE-10, no. 5, pp. 498–501.

157. 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.

158. 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.

159. Levitt, Karl, Larry Robinson and Brad Silverberg. "The HDM Handbook", Computer Science Lab, SRI International, Menlo Park, CA, 1979.

160. 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.

161. 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.

162. 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.

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

164. London, R. L., J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell and G. J. Popek. *Proof Rules for the Programming Language Euclid.* **Acta Informatica** (1978) vol. 10, pp. 1–26.

165. Lubars, Mitchell D. and Mehdi T. Harandi. *Knowledge-Based Software Design Using Design Schemas.* **Proceedings of the 9th International Conference on Software Engineering** (1987) pp. 253–262.

166. Luckham, David C., S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak and W. L Sherlis. "Stanford Pascal Verifier User Manual", Report No. STAN-CS-79-731, Computer Science Department, Stanford University, Stanford, CA, 1979.

167. Luckham, David C. and Friedrich W. von Henke. *An Overview of Anna, a Specification Language for Ada.* **IEEE Software** (March, 1985) vol. 2, no. 2, pp. 9–22.

168. Luckham, David C., Friedrich W. von Henke, Bernd Krieg-Brueckner and Olaf Owe. "Anna, a Language for Annotating Ada Programs (Preliminary Reference Manual)", Technical Report CSL-84-261, Computer Systems Laboratory, Stanford University, Stanford, CA, 1984.

169. Manna, Zohar. **Mathematical Theory of Computation.** McGraw-Hill, New York, 1974.

170. Manna, Zohar and Richard Waldinger. *A Deductive Approach to Program Synthesis.* **ACM Transactions on Programming Languages and Systems** (January 1980) vol. 2, no. 1, pp. 90–121.

171. Martin, Alain J. *A General Proof Rule for Procedures in Predicate Transformer Semantics.* **Acta Informatica** (1983) vol. 20, pp. 301–313.

172. Martin-Lof, P. *Constructive Mathematics and Computer Programming.* **Proceedings of the 6th International Congress for Logic, Method, and Philosophy of Science** (1982) pp. 153–175.

173. Marzullo, Keith. *Jasmine: A Software System Modelling Facility.* **Proceedings of the Second ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments** (December 1986) pp. 27–33.

174. Matsumoto, Yoshihiro. *Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels.* **IEEE Transactions on Software Engineering** (September 1984) vol. SE-10, no. 5, pp. 502–512.

175. Meyer, Bertrand. *Reusability: The Case for Object-Oriented Design.* **IEEE Software** (March 1987) vol. 4, no. 2, pp. 50–64.

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

177. Miller, Edward (ed.). **Tutorial: Automated Tools for Software Engineering.** IEEE Computer Society, New York, 1979.

178. Mills, Harlan D. *Structured Programming: Retrospect and Prospect.* **IEEE Software** (November 1986) vol. 3, no. 6, pp. 58–66.

179. 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.

180. Muller, Carlo. *Modula--Prolog: A Software Development Tool.* **IEEE Software** (November 1986) vol. 3, no. 6, pp. 39–45.

181. 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.

182. Myers, Ware. *Ada: First users - pleased; prospective users - still hesitant.* **IEEE Computer** (March 1987) vol. 20, no. 3, pp. 68–73.

183. Narayanaswamy, K. and Walt Sacchi. *Maintaining Configurations of Evolving Software Systems.* **IEEE Transactions on Software Engineering** (March 1987) vol. SE–13, no. 3, pp. 324–334.

184. 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.

185. Nilsson, Nils J. **Principles of Artificial Intelligence.** Tioga, Palo Alto, CA, 1980.

186. O'Donnell, Michael J. *A Critique of the Foundations of Hoare Style Programming Logics.* **Communications of the ACM** (December 1982) vol. 25, no. 12, pp. 927–935.

187. Oest, Ole N. *VDM From Research to Practice.* **Information Processing** (1986) pp. 527–533.

188. 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.

189. 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.

190. ——. *Software Processes Are Software Too.* **Proceedings of the 9th International Conference on Software Engineering** (1987) pp. 2–13.

191. 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.

192. 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.

193. ——. *A Technique for Software Module Specification with Examples.* **Communications of the ACM** (May 1972) vol. 15, no. 5, pp. 330–336.

194. ——. *The Use of Precise Specifications in the Development of Software.* **IFIP Congress Proceedings** (1977) pp. 861–867.

195. Partsch, H. and R. Steinbruggen. *Program Transformation Systems.* **Computing Surveys** (September 1983) vol. 15, no. 3, pp. 199–236.

196. Perry, Dewayne E. *Software Interconnection Models.* **Proceedings of the 9th International Conference on Software Engineering** (1987) pp. 61–69.

197. ——. *Version Control in the Inscape Environment.* **Proceedings of the 9th International Conference on Software Engineering** (1987) pp. 142–149.

198. Prywes, Noah, Yuan Shi, Boleslaw Szymanski and Jine Tseng. *Supersystem Programming with Model.* **Computer** (February 1986) vol. 19, no. 2, pp. 50–60.

199. Rajlich, Vaclav. *Refinement Methodology for Ada.* **IEEE Transactions on Software Engineering** (April 1987) vol. SE–13, no. 4, pp. 472–478.

200. 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.

201. Reddy, Uday S. *Narrowing as the Operational Semantics of Functional Languages.* **Proceedings of the Symposium on Logic Programming** (1985) pp. 138–151.

202. Reiter, Raymond. *On Closed World Data Bases.* In: **Logic and Data Bases**, H. Gallaire and J. Minker, ed. Plenum Press, 1978.

203. 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.

204. Reps, Thomas and Tim Teitelbaum. *The Synthesizer Generator.* **Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments** (1984) pp. 42–48.

205. 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.

206. 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.

207. Ross, Douglas T. and Kenneth E. Schoman, Jr. *Structured Analysis for Requirements Definition.* **IEEE Transactions on Software Engineering** (January 1977) vol. SE–3, no. 1, pp. 6–15.

208. 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.

209. Schonberg, Edmond and David Shields. *From Prototype to Efficient Implementation: a Case Study using SETL and C.* **Proceedings of the 19th Hawaii International Conference on System Sciences** (1986) pp. 75–88.

210. Schwan, K., R Ramnath, S. Vasudevan and D. Ogle. *A System for Parallel Programming.* **Proceedings of the 9th International Conference on Software Engineering** (1987) pp. 270–282.

211. Schwartz, David P. *Software Evolution Mangement: An Integrated Discipline for Managing Software.* **Proceedings of the 9th International Conference on Software Engineering** (1987) pp. 388–397.

212. Seviora, Rudolf E. *Knowledge-Based Program Debugging Systems.* **IEEE Software** (May 1987) vol. 4, no. 3, pp. 20–32.

213. 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.

214. 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.

215. 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.

216. Smith, John M. and Diane C. P. Smith. *Database Abstractions: Aggregation.* **Communications of the ACM** (June, 1977) vol. 20, no. 6, pp. 405–413.

217. ——. *Database Abstractions: Aggregation and Generalization.* **ACM Transactions on Database Systems** (June 1977) vol. 2, no. 2, pp. 105–133.

218. Sneed, Harry M. and Andras Merey. *Automated Software Quality Assurance.* **IEEE Transactions on Software Engineering** (September 1985) vol. SE–11, no. 9, pp. 909–916.

219. Sommerville, Ian (ed.). **Software Engineering Environments.** Peter Perigrinus Ltd., 1986.

220. 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.

221. Stansifer, Ryan. "Representing Constructive Theories in High-Level Programming Languages (Ph.D. Thesis)", TR 85–664, Department of Computer Science, Cornell University, Ithaca, New York, 1985.

222. Stickel, Mark E. *A Prolog Technology Theorem Prover.* **Proceedings of the International Symposium on Logic Programming** (February 1984) pp. 211–217.

223. Sweet, Richard E. *The Mesa Programming Environment.* **ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments** (June 1985) pp. 216–229.

224. Swinehart, Daniel C., Polle T. Zellweger, Richard J. Beach and Robert B. Hagmann. *A Structural View of the Cedar Programming Environment.* **ACM Transactions on Programming Languages and Systems** (October 1986) vol. 8, no. 4, pp. 419–490.

225. 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.

226. Teichroew, Daniel and Ernest A. Hershey, III. *PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems.* **IEEE Transactions on Software Engineering** (January 1977) vol. SE–3, no. 1, pp. 41–48.

227. 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.

228. Teitelman, W. and L. Masinter. *The Interlisp Programming Environment.* **Computer** (April 1981) vol. 14, no. 4, pp. 25–33.

229. Terwilliger, Robert B. "Knowledge-Based Development in ENCOMPASS (Preliminary Report)", Report No. UIUCDCS-R-87-1334, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1987.

230. 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.

231. ——. "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 (also to appear in the **Journal of Systems and Software**), 1986.

232. ——. "PLEASE: Executable Specifications for Incremental Software Development", Report No. UIUCDCS–R–86–1295, Dept. of Computer Science, University of Illinois at Urbana–Champaign (also to appear in the **Journal of Systems and Software**), 1986.

233. ——. *PLEASE: Predicate Logic based ExecutAble SpEcifications.* **Proceedings of the 1986 ACM Computer Science Conference** (February, 1986) pp. 349–358.

234. ——. *PLEASE: a Language for Incremental Software Development.* **Proceedings of the 4th International Workshop on Software Specification and Design** (April 1987) pp. 249–256.

235. Tichy, Walter F. *Software Development Control Based on Module Interconnection.* **Proceedings IEEE 4th International Conference on Software Engineering** (1979) pp. 29–41.

236. ——. *Design, Implementation, and Evaluation of a Revision Control System.* **Proceedings of the 6th IEEE International Conference on Software Engineering** (September 1982) pp. 58–67.

237. 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.

238. 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.

239. Wartik, Steven. *Rapidly Evolving Software and the OVERSEE Environment.* **Proceedings of the Second ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments** (December 1986) pp. 77–83.

240. Waters, Richard C. *The Programmer's Apprentice: A Session with KBEmacs.* **IEEE Transactions on Software Engineering** (November 1985) vol. SE–11, no. 11, pp. 1296–1320.

241. Wegner, Peter. **Programming with Ada: an Introduction by Means of Graduated Examples.** Prentice–Hall, Englewood Cliffs, New Jersey, 1980.

242. 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.

243. Whitehurst, R. Alan. *The Need for an Integrated Design, Implementation, Verification and Testing Methodology.* **Software Engineering Notes** (February 1984) vol. 10, no. 4, pp. 97–100.

244. Wile, David S. *Program Developments: Formal Explanations of Implementations.* **Communications of the ACM** (November 1983) vol. 26, no. 11, pp. 902–911.

245. Wing, Jeannette M. *Writing Larch Interface Language Specifications.* **ACM Transactions on Programming Languages and Systems** (January 1987) vol. 9, no. 1, pp. 1–24.

246. Winkler, Jurgen F. H. *Version Control in Families of Large Programs.* **Proceedings of the 9th International Conference on Software Engineering** (1987) pp. 150–161.

247. Winston, Patrick H. **Artificial Intelligence.** Addison–Wesley, Reading, MA, 1977.

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

249. 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.

250. 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.

251. Yau, Stephen S. and Jeffery J. Tsai. *Knowledge Representation of Software Component Interconnection Information for Large-Scale Software Modifications.* **IEEE Transactions on Software Engineering** (March 1987) vol. SE–13, no. 3, pp. 355–361.

252. 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.

253. 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.

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

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

256. Berliner, Edward F., and Pamela Zave. *An Experiment in Technology Transfer: PAISLey Specification of Requirements for an Undersea Lightwave Cable System.* **Proceedings of the 9th International Conference on Software Engineering** (1987) pp. 42–50.

257. 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.

258. Zelkowitz, Marvin V. *Perspectives on Software Engineering.* **Computing Surveys** (June, 1978) vol. 10, no. 2, pp. 197–216.

259. Zucker, Sandra. *Automating the Configuration Management Process.* **Proceedings SOFTFAIR, Arlington, Virginia** (July, 1983) pp. 164–172.

VITA

Robert Barden Terwilliger was born in Ithaca, New York in August 1956. In 1980 he received the B.A. degree in chemistry from Ithaca College and began graduate studies in the same field. At that time, he received both a University of Illinois Fellowship and an N.S.F. Graduate Fellowship Honorable Mention. He received the M.S. degree in computer science from the University of Illinois at Urbana–Champaign in 1982 and attended the University of Wisconsin–Madison from 1982 until 1984. He received his Ph.D. in computer science from the University of Illinois at Urbana–Champaign in 1987. Dr. Terwilliger will be an assistant professor in the Department of Computer Science, the University of Colorado at Boulder beginning in the Fall of 1987. His research interests include software engineering environments, executable specification languages, formal development methods and logic programming. Dr. Terwilliger is a member of the IEEE Computer Society and the Association for Computing Machinery. He has co–authored over ten papers on his work with SAGA, PLEASE and ENCOMPASS.

APPENDIX C

Automating the Software Development Process

R. H. Campbell
H. Render
R. N. Sum, Jr.
R. Terwilliger

Automating the Software Development Process

R. H. Campbell
H. Render
R. N. Sum, Jr.
R. Terwilliger

Report No. UIUCDCS-R-87-1333
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
and a grant from AT&T Information Systems.

Automating the Software Development Process

R. H. Campbell
H. Render
R. N. Sum, Jr.
R. Terwilliger

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

Abstract: Much of the software development process is repetitive, tedious to perform, but *possible to automate*. Our research on the SAGA Project includes building models of software development tasks that both accurately reflect the processes involved and have direct and efficient implementations. In this paper we assume the desirability of a software engineering environment that supports the entire life-cycle; automation should greatly enhance the quality and efficiency of software production and maintenance. To study the problems involved, SAGA has constructed ENCOMPASS, a prototype environment which supports software development. ENCOMPASS has provided valuable insights and experience; however, during its development and use many limitations have surfaced. In this paper we emphasize the configuration and project management aspects of our work. We discuss the current capabilities and limitations of ENCOMPASS, as well as describing the new systems being constructed to both overcome its limitations and extend its life-cycle coverage.

1. Introduction

In a typical development shop, software engineers use poorly integrated tools which cannot control the complexity of software development and maintenance. To help remedy this situation, the SAGA Project is investigating both the formal and practical aspects of providing automated support for the entire life-cycle. SAGA has constructed ENCOMPASS, a prototype environment which supports software development. In this paper we emphasize the configuration and project management aspects of our work. We discuss the capabilities of ENCOMPASS, as well as the new systems being developed.

A *life-cycle model* describes the sequence of distinct stages through which a software product passes during its lifetime [Fairley, 85]; there is no single, universally accepted model of the software life-cycle. The stages of the life-cycle generate *software components*, such as code written in programming languages,

This work was funded by NASA Langley Grant NSG-138 and a grant from AT&T Information Systems.

test data or results, and many types of documentation. *Configuration management* is concerned with the identification, control, auditing, and accounting of the components produced and used in software development and maintenance [Babich, 86]. *Project management* controls the software development process: setting objectives, coordinating development activities, creating schedules, allocating resources, monitoring milestones and reporting on progress [Gunther, 78]. To be effective, the mechanisms and policies involved in configuration and project management must be integrated with the methodology used to develop and maintain the software. In a large project producing many components, automating management should have a major impact on quality and productivity.

In section two of this paper we describe a typical software development environment as it might be found in industry today. In section three we describe both the capabilities and limitations of ENCOMPASS, a prototype environment to support software development. In section four, we outline our current efforts to develop a more advanced configuration control system and in section five, we discuss our current work in project management. Finally, in section six we summarize and draw some conclusions.

2. A Typical Software Development Shop

As part of our research, we have investigated both the capabilities and limitations of some existing software development environments. One such environment is used by AT&T to develop and support its System 75® telephone switching software [Sum, 87]. System 75 is a very large development effort, with approximately one million lines of source code produced to date. A variation of the traditional *waterfall* life-cycle model [Fairley, 85] is used on the project. In this model, the life-cycle is divided up into discrete, sequential phases. Each phase produces some combination of documents and code which are used in subsequent phases. For example, the output of the design phase, the system design documents, acts as the input to the implementation phase.

The example environment contains many tools to support the development and maintenance processes. For example, the *document library* stores documents with release numbers and completion

System 75® is a trademark of AT&T.

status, such as draft, final, and obsolete. Source code is stored with a hierarchical version control system built on top of the UNIX™ file system. *Makefiles* are used to record compilation dependencies and SCCS [Dolotta, et al, 77] acts as the low level version maintainer. During the latter phases of development and during maintenance, an error reporting and tracking system holds error reports along with their status and resolutions.

The problem with many existing environments is that they are not well enough integrated. For example, in the AT&T case study, the document library does not necessarily provide on-line access to project members. Generally, project members must request paper copies from a document librarian. The version control system lacks flexibility in the types of data (files) whose dependencies it can maintain and in the hierarchical structure in which those files may be organized. To exacerbate the problem, the version control system is composed of several loosely-coupled tools that require a project integrator to ensure that new revisions added to a module are consistent and complete. Even though the tools in such environments work, they can be awkward to use and require that management enforce their use.

Many environments contain management tools, but they usually support only a small subset of the management tasks. Common tools include a global project milestone scheduler and tracking system, and smaller personal computer programs like Timeline™. Most work assignments and progress monitoring are performed by managers using manual procedures. The milestone system allows schedules to be kept on-line by the developers. Milestones record such information as the contractor, the producer, the consumer, and the due date. Timeline performs dependency analysis augmented with some critical path and cost analysis capabilities. The work assignment and monitoring procedure consists of a form in which a worker and his manager order the worker's tasks by priority. The worker and his manager frequently meet to review the tasks and set personal milestones for the worker, thereby keeping each other informed.

As one would expect, there are problems with existing management tools. For example, the milestone system provides only a tabular representation of its output and does not have a good mechanism for

UNIX™ is a trademark of AT&T.

Timeline™ is a trademark of Breakthrough Software.

creating and maintaining task dependencies. Also, the system is used across entire projects without a mechanism to view subsets of interrelated milestones. The Timeline program is too small for large projects, 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 the milestone system, selective viewing of dependencies, and automatic notifications of approaching milestones to the workers producing deliverables and to the managers. One would also like an automated system to monitor individual work loads and help with task breakdown and assignment. Similar automation goals are also alluded to in [Howes, 84].

Although tools exist that aid the various phases of conventional software development, for the most part they require a great deal of manual effort to use and do not satisfy all of the developers' needs. Automating many existing approaches to software development requires a more formal specification of the software production process than is available. The research performed by SAGA is motivated by our observations of deficiencies in existing environments. In order to clarify our thinking and test our architectural concepts, SAGA has constructed a prototype environment for software development.

3. ENCOMPASS

Our early efforts in SAGA were devoted to building software development tools, documentation systems (Notesfiles[1]), and an interactive language–oriented editor that controls access to the software development system. A hierarchy of different languages offered project management and configuration control as well as design and program entry [Campbell & Richards, 81]. Editing would change specifications in these languages, automatically advancing the project through a series of development activities. However, the lack of adequate models for many of the development processes involved has led us to broaden our research. In particular, although software development and maintenance methodologies, configuration control, and project management clearly interact, no model for the interaction appears suitable as the basis for an automated environment. To clarify our ideas, we have devised an experimental environment

[1] Notesfiles are now distributed as part of Berkeley UNIX.

based on a rigorous software development methodology and integrated it with simple configuration and project management schemes. We have automated many aspects of this environment using existing tools from SAGA and elsewhere.

The environment we have used to study the interaction between development methodologies, project management and configuration control is called ENCOMPASS [Terwilliger & Campbell, 86a]. ENCOMPASS supports small to medium-scale projects using an incremental development methodology similar to VDM. The methodology was chosen because it supports the specification, validation, design, implementation, and verification of software. It also provides acceptance criteria for the steps in the production process and offers limited but well-defined project management goals.

3.1. Support for VDM

VDM (the Vienna Development Method) supports the top-down development of software specified in a notation suitable for formal verification [Jones, 80]. In this method, components are first written in a combination of 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.

ENCOMPASS is designed to support a particular model of the software life-cycle. It extends the waterfall life-cycle model described in [Fairley, 85] with the use of executable specifications and VDM. The ENCOMPASS life-cycle model includes a separate phase for customer validation; a development passes through the stages planning, requirements definition, validation, refinement and system integration. Project management and configuration management incorporate features to support the activities that occur and record the data generated in these stages.

In ENCOMPASS, requirements specifications are a combination of natural language documents and components written in PLEASE [Terwilliger & Campbell, 86b], a wide-spectrum, executable specification language. 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, and may interact with other modules written in the target programming language. We believe that the early production of prototypes for experimentation and evaluation will enhance the software development process. PLEASE also provides many machine-recognizable milestones for the project management system. For example, the existence of an implementation, its correct execution of a test case, and the proof of its correctness can all be recognized by the system.

In ENCOMPASS, components specified in PLEASE are incrementally refined into Ada™ implementations. Since PLEASE specifications are both executable and formal, refinements can be verified using either testing or proof techniques. ENCOMPASS is an environment for the *rigorous* development of programs. 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. 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 ENCOMPASS environment is coupled to four tools for programming in the small: TED, a proof management system which is interfaced to a number of theorem provers; ISLET a simple 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.

3.2. Configuration and Project Management

In ENCOMPASS, the configuration management system structures the software components developed by a project and stores them in a project data base. It also provides a primitive form of software capabilities to control access. The project management system distributes these capabilities to implement a *management by objectives* [Gunther, 78] approach to software development; each phase in the life-cycle satisfies an objective by producing a *milestone* which can be recognized by the system.

Ada™ is a registered trademark of the U.S. Government, Ada Joint Program Office.

The project management system is organized around *work trays* [Campbell & Terwilliger, 86] which provide a mechanism 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*. 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 then transfers these tasks to an in-progress tray where he performs the actions required of him and produces new products. The user returns the task via an output tray. A user may create new tasks in in-progress trays that he owns. File trays are used for long term storage.

In ENCOMPASS, software configurations are modeled using a variant of the *entity-relationship model* [Chen, 76]. An *entity* is a distinct, named component. Two or more entities may have a *relationship* between them. Both entities and relationships may have *attributes* to describe their properties or qualities. A group of entities may be abstracted into an *aggregate*. 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 include *images* of entities in the base view. In ENCOMPASS, access to components is controlled through the use of views; the project management system uses views to implement tasks.

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." However, the underlying storage mechanisms (chosen for implementation expediency) impose limitations on its use. In particular, the view mechanism is based on the use of symbolic links and shell scripts under UNIX and this imposes performance, consistency, and flexibility constraints.

The configuration and project management systems provide a basic capability to store and retrieve modules, create milestones and acceptance tests, and create, monitor, and complete tasks. A task can require a PLEASE specification be refined into a more concrete Ada program. The task would have, as its completion criteria, a milestone which verifies the refinement using predetermined test data. A further task might be created to verify the refinement step using TED. Its completion criteria could be one of the

following milestones, a complete proof that can be automatically checked by the theorem prover, the discovery of an error, or a partially completed proof together with an informal argument as to why the proof is correct. The individual refinement, testing, and proof steps are saved by ENCOMPASS and may be reused in maintenance or future design projects.

The completed experimental environment demonstrates the interconnectivity between the phases in the life-cycle, the interaction between the development methodology and project/configuration management, and the way in which automated tools can take advantage of an integrated system. For the next step in the SAGA research program, we are refining the project and configuration management ideas of ENCOMPASS to produce more sophisticated and general-purpose support. Our subsequent goal will be to integrate these tools with a revised, less abstract and more powerful software development methodology.

4. Scaling up Configuration Management

The ENCOMPASS environment demonstrates the potential benefits from the identification, tracking and control of the components produced and used in software development. This need is common to all projects, large or small, and lasts throughout their lifetime. However, the ENCOMPASS configuration management tools were designed as prototypes with limited capabilities; as a response, the CLEMMA system has been developed.

4.1. A Software Librarian

CLEMMA is a *librarian*, providing operations to create and maintain project libraries. A library contains files that store a software project's components and a relational data base that holds a catalog and cross reference. CLEMMA operations retrieve and store components and update and maintain the consistency of the database. When the software project develops a new component, the librarian is used to create and *catalog* a corresponding library *item*. Subsequently, implementations of the component will be checked into the library as *versions* of the item. Keyword indexing of library items is provided by CLEMMA, allowing quick searching and retrieval of project components and encouraging re-use.

We refer to all the versions of an item as a *version group*. Versions may be revisions or variations. A *variation* may have little or no textual correspondence to any other variation of the item it implements, while a *revision* is directly derived from an existing version of the item [Babich, 86]. Given that any version may be modified several times, with each modification producing a new revision, the version group for an item may be described as a forest of *derivation trees*. Each variation of an item is the root of a derivation tree. The use of *delta storage* for the revisions within a derivation tree optimizes the space required to save versions of an item.

During modification, a version is *locked*, preventing other users from modifying it concurrently. This reduces the likelihood of duplicate modifications, and lessens the need for merging of overlapping changes. *Parallel* revisions of items are possible, but their occurrence is controlled. The compatibility of parallel revisions, being a language-oriented issue, is left to the users for the moment, and CLEMMA makes no

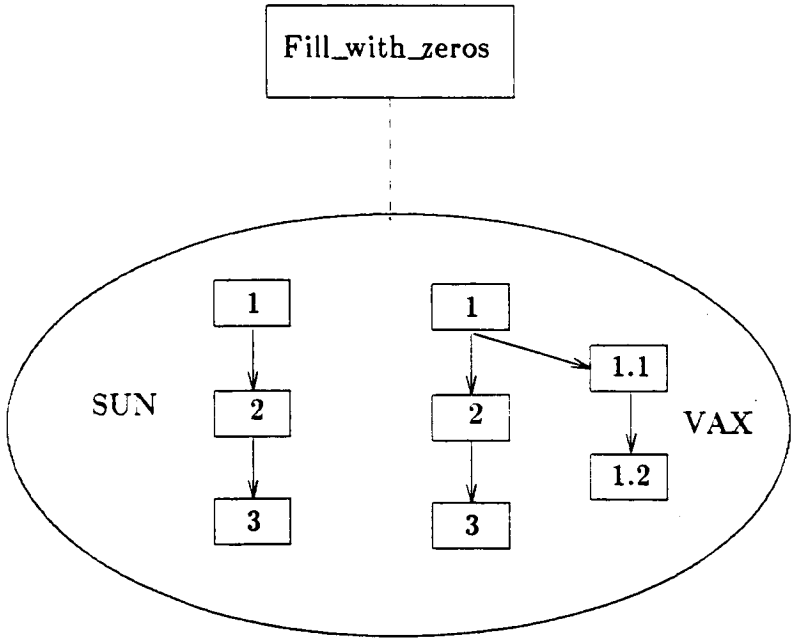


Figure 4.1: An Example Version Group for Item **Fill_with_zeros**.

restrictions on the types of modifications allowed on item versions. Because many project members may request access to an item's versions, permission lists are used to manage an item's usage and modification. A list of permitted managers and users is maintained for each item. *Managers* may add, modify, and delete versions of an item. *Users* may only check out versions of an item.

For example, Figure 4.1 shows the version group for a routine in the paging software of an operating system. The routine is written in assembly code and is used to fill a newly created virtual memory page with a pattern of zeros. The version group of **Fill_with_zeros** contains separate variations for a SUN and a VAX. The derivation tree for the VAX code contains a number of revisions, including a parallel revision which creates an independent sequence within the tree. In CLEMMA, a derivation tree is described using the *derived_from* relation between versions. Solid arcs represent the derived_from relation in the figure.

CLEMMA, like ENCOMPASS, is based on the entity-relationship model [Chen, 76]. The elements of a software project are entities; for example, items, versions and users are all entities. An entity may have attributes; for example, an item has attributes "name", "type", "owner", and "identification number". Entities may have relationships between them; for example, "version_of", "contains", "derived_from", "manager_of", and "user_of" are all relations used in CLEMMA. All of the entities, attributes and relations comprising a CLEMMA library are represented in the relational data base. Using a data base as the storage mechanism saves space, allows greater flexibility in structuring the data, and makes accessing the information easier than is possible with ENCOMPASS. This, in turn, enhances the tools that can be built to automate the software development process.

4.2. Views

Versions of items may be grouped into aggregate items called *views*, which have many applications. Views allow abstractions of a project's components to be constructed, manipulated and maintained. For example, a view may specify a *baseline* of a module. A baseline is a configuration of the components of a module that satisfies all of the acceptance criteria of a milestone in the production of the module. In practice, a baseline may often be followed by a *release*. A release is a view of the module and its components that is made available to users other than the module's development team.

Views may also represent a selected subset of the components, chosen by a functional abstraction of the development process. For example, a test view of a module may contain a specification of the module, the binary object files, test data and results and a test harness. A quality assurance group may use this view of the module to perform acceptance testing. A documentation view may contain the specification and source code for the module along with documentation of the program. Such a view would facilitate the production of a user's manual for the module and might also be used in code and documentation inspections. A view may also be constructed to select parts of a software system that will be reused in the construction of a different software system.

Each of these different views of a module may be created and stored as an item in the library and will have a version group. Fully hierarchical systems may be represented by views that include versions of other views. This facility is more general than that of ENCOMPASS, which allows for only single-level module hierarchies. For example, Figure 4.2 shows a hierarchy of modules which implement the page fault handler for an operating system. The **fault_handler** module is an item that has a version group. A particular version of **fault_handler** is a view that contains items such as **machine_check** and views such as the **VM** module.

Like other items, views have associated catalog information and can be checked out of the library, modified, and checked back into the library as a revision. They may also be used to check out the components they contain from the library. A test view may be used to check out the components required in an acceptance test. A release of **fault_handler** may be used to check out a stable, released version of all the code associated with the fault handler module and submodules.

The mechanism to perform a check out is implementation dependent. Under the UNIX operating system, we have used the ENCOMPASS approach. The contents of a view may be checked out read only in which case a user work space is supplied with an image of the contents of the view. The work space is a directory structure in which symbolic links are mapped to read-only copies of the files that are stored in a central repository in the library. When the contents of a view are checked out for modification, copies of the files are created in the workspace and the versions in the library are locked. If modifications are made

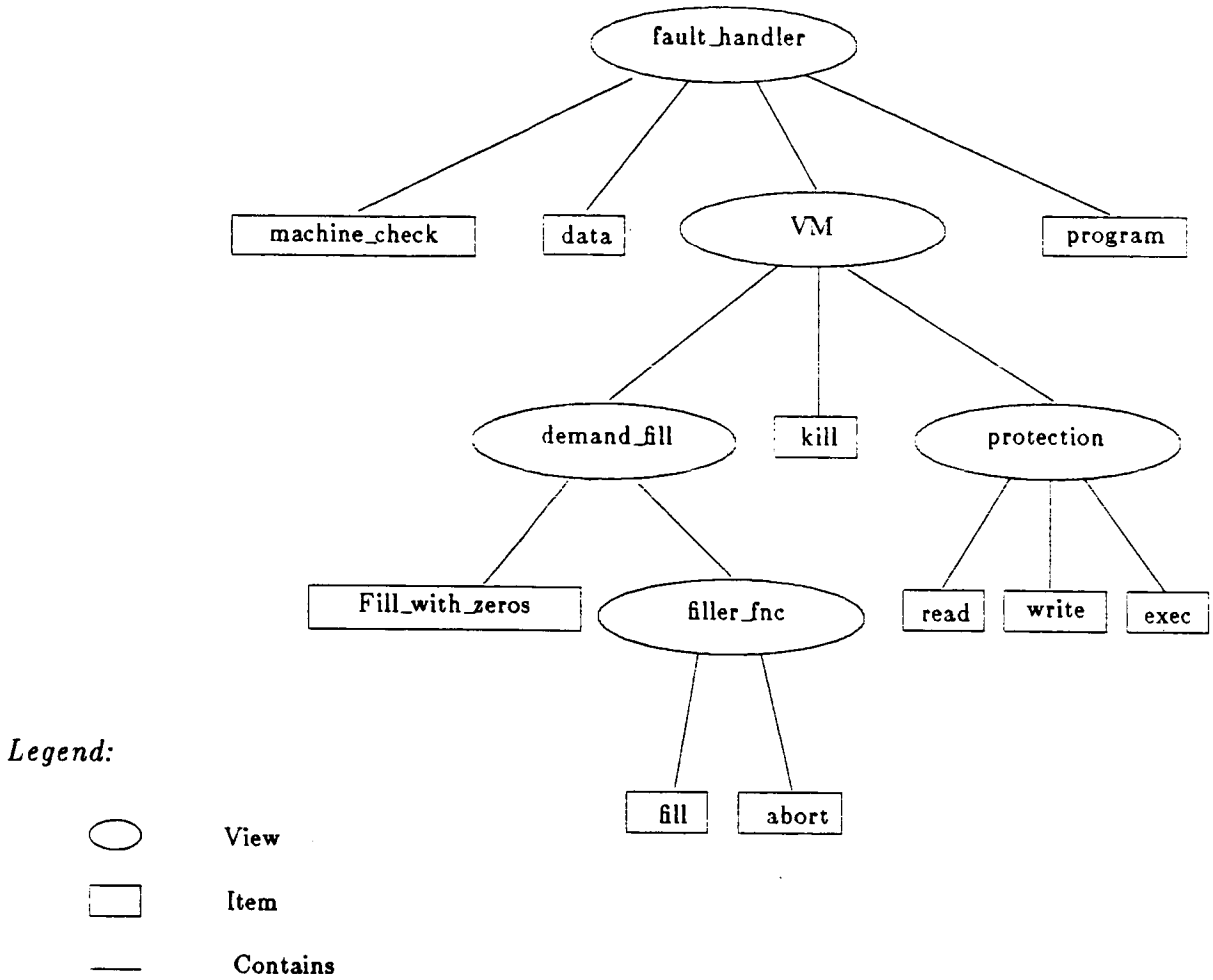


Figure 4.2: Fault-Handler Subsystem

to the contents of a view and the revisions are checked in, a new version of the view does not necessarily have to be created. A later section discusses how views may be parameterized.

The view in CLEMMA is much more powerful than in ENCOMPASS. A version of a view is implemented by a *view descriptor*. A view descriptor is a list of specifications, one for each component of the aggregate, giving the component's name, an item identification number, and a list of attribute matching expressions. The attribute matching expressions may be constant (as in a view that describes a specific

release) or may vary depending upon when a view is checked out of a library. For example, attribute matching expressions may specify the most recent version, the most recent release, or the version created by a specific author.

When the contents of a view are checked out of a library, the associated descriptor is evaluated and the user is given a workspace containing an image of each component in the view. For each component, the version with attribute values equal to those given in the matching expression is the one whose image is supplied. A component specification is said to be *well-formed* and *complete* if it is syntactically correct and it resolves to a single version of an existing library item. For example, a descriptor for the SUN implementation of the **demand_fill** module in Figure 4.2 could specify the **Fill_with_zeros** component as being the most recent SUN variation that has been released.

Using attribute matching expressions to specify a view's components extends the simple idea of a module configuration as a list of items and version numbers [Babich, 86]. The facility permits views of a system that satisfy particular dynamic functional requirements. Views and their attribute matching expressions may be parameterized by symbolic names that are replaced by arguments at check-out time. For example, a view of test results may be parameterized by specifying the version of the software which was tested. Additionally, the data base implementation of views as sets of components allows them to be intersected or combined. For example, one can define a high-level view of all the items which have changed between one stable configuration and another.

CLEMMA has several advantages over the configuration management facilities of ENCOMPASS. First, it provides more flexible support for the abstraction and manipulation of software components stored in the library. Second, by combining data base and file system technology, CLEMMA capitalizes on the efficiency of existing tools. Third, CLEMMA may be easily updated to support additional entities, relations, or attributes without requiring the whole library to be reorganized. Finally, the configuration model is represented more directly in the configuration management system.

5. Scaling Up Project Management

ENCOMPASS also demonstrates the potential benefits of an on-line project management system; however, the facilities it provides are minimal. To track, audit, and control a software development project, one needs to make project structure visible and accessible, maintain deadlines, ease scheduling, maintain task dependencies, coordinate and synchronize task activities, control access to project resources, collect statistics, and produce a project archive. A project management system and the project it controls can be compared to an operating system and the underlying hardware. The project management system organizes and supports the task structure, scheduling, and resource management systems. It provides resource allocation and other services to its users while implementing management policies. With this metaphor in mind, the SAGA Project is developing a project management system which greatly extends the capabilities of ENCOMPASS.

5.1. A Resource/Process Model of Project Management

In our model, a software development project consists of *tasks* that are *executed* by project members. These tasks produce and use *resources*, such as specifications, documents, code, test cases and reports. The tasks form a *dynamic hierarchy* corresponding to the manager-worker relationships and work breakdown structures within the project; this is similar to the hierarchy of processes in a UNIX-like operating system. Although these tasks are similar to the process models proposed by [Osterweil, 87] and [Dowson, 87], our model differs in that it attempts both to enforce resource allocation and to provide task synchronization. Even though one may compare the execution of a task to the execution of a program, we do not attempt the detailed level of programming in [Osterweil, 87]. Instead, one might think of the task execution as a communication protocol followed by the developers.

With the project management system, tasks may only be created, executed, and destroyed in well-defined ways. As a task is executed, resources are created, acquired and released. Resources have access and ownership properties which determine their use. In general, resources are stored and accessed through subsystems which provide appropriate services; for example, a configuration mangement system for source code. Such subsystems resemble an operating system's memory managers or device drivers.

Task execution follows a protocol based on a management by objectives approach [Gunther, 78]. For example, Figure 5.1 presents the interaction between manager and worker in the protocol. In our approach, each task has objectives which its products must satisfy. Often, some negotiation between the manager and worker is needed to arrive at the objectives for a task. Once the objectives has been agreed upon, the worker is left to complete the task. Completion of a task means that the manager and worker agree that the task meets its objectives. During task execution, the system checks for simple forms of deadlock, delays, or deadlines. These correspond to circular dependency and schedule slippage problems in PERT/CPM methods. The task protocol may be likened to executing a batch job in an operating system. The job must acquire the resources it needs, create independent subtasks to accomplish its goals, provide those subtasks with the appropriate resources, wait for their completion, and check their return codes.

The data dependencies in the management system may also be described using an entity relationship model[Chen, 76]. As a project progresses, the system manipulates the entities and relations to reflect the current structure and status of the project. A task is created and assigned according to the project's work breakdown structure. The management system monitors task dependencies to ensure that inputs are available and sequencing constraints are obeyed. For example, the initiation of a programming task may depend on a successful design document walkthrough; therefore, the system will not allow the programming task to proceed before the walkthrough is complete.

The system also monitors a task's completion date and informs managers and workers of impending and missed deadlines. The system can provide appropriate resource views, for example access to file structures maintained by a configuration manager, for the performance of different tasks. Finally, accounting information is accumulated by the system; for example, the time spent working on a task by its manager and worker are stored for use in cost calculations and scheduling.

5.2. From The User's Viewpoint

In a similar manner to the views in CLEMMA, the project management system provides views of the projects and tasks in the environment. As in ENCOMPASS, access to project management facilities,

Manager	Worker
{ Task Specification - create/revise specifications and objectives assign the task	
	Accept/Reject Assignment - review specifications and objectives }+
	{ Task Active - work being done
.	.
.	.
.	.
{ a problem may result	
	in an amendment or revision }
.	.
.	.
.	.
Task Complete - product is accepted }+	
Task Archived - for future reference	

Notation: {} ≡ repeat 0 or more times, {}+ ≡ repeat one or more times

Figure 5.1: Task Execution Protocol

including browsers and reports, is provided through the tray mechanism[2]. This ensures that workers and managers are provided with the appropriate views of the project and the management operations that they may perform. A browser supports views of the project task hierarchy based on specific dependencies or attributes such as deadlines or user names. Report generation facilities automatically retrieve and format information on the project.

To start a project, a supervisor creates a new project, gives a user project manager capability, and initiates a root task. During the life of the project, sub-tasks may be created and additional workers may be added. For example, a hierarchy of tasks might be created to model the structure of the fault handler subsystem shown in Figure 4.2. A manager might create a task **fault_handler** to develop the entire subsystem. This task would have a sub-task for each of the **machine_check**, **data**, **VM** and **program** handlers. These sub-tasks could themselves be decomposed until the smallest tasks act as leaves of the work breakdown structure. It is possible for one to assign oneself several tasks, possibly to break a large task into smaller units.

Returning to the larger scale, the task hierarchy for a project does not necessarily model the structure of the software being developed. For example, Figure 5.2 shows the task hierarchy for a project to develop the virtual memory and fault handler modules of an operating system. The project consists of three phases: specification, development and testing. The root task represents the entire project, while each phase has an associated sub-task. Each phase is divided into sub-tasks according to the structure of the software being developed; there are separate sub-tasks for the **VM** and **fault_handler** sub-systems. The dependencies in a task hierarchy are unlikely to follow the work breakdown structure because resources such as specifications, designs, code, and tests may be produced by tasks in separate sub-trees. One way for a task to be dependent on another task is by completion; for example, appropriate code and specification tasks must complete before the testing task can begin. This situation might be represented by the arrows in Figure 5.2.

[2]For more information on trays see section 3.

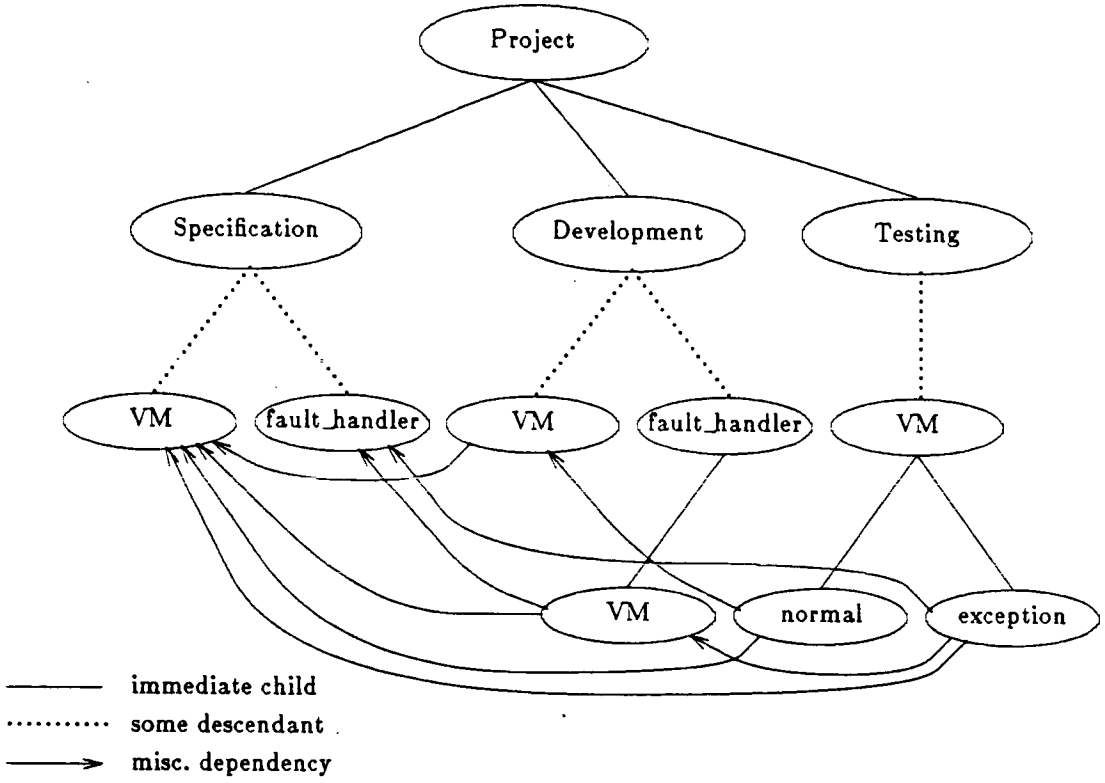


Figure 5.2: Sketch of a Project Task Hierarchy

Within a task, the user's view of the project is the resources allocated to the task. When using CLEMMA as a resource, access to components such as source code is easily controlled. The manager of a task is given a complete view of the task and objectives for its completion. He creates and assigns sub-tasks to his workers, giving them access derived from his view. The sub-tasks' resources are fetched by invoking the configuration manager when the worker accepts the task. The worker performs his task, and when he believes that he is finished, he informs his manager. The manager then checks the completed work; for example, by inspection, by running tests in a test harness, or by running a proof checker. If the manager accepts the task as complete, then the system stores the task's output resources using the configuration manager. A task may be archived when it is no longer useful to the manager's or worker's views of the project.

There are advantages to the use of configuration managers which support abstract naming. For example, using CLEMMA, views need not be specified by exact version numbers; therefore, when a task becomes active, it receives the intended resources without knowledge of past revisions. More specifically, if an active task has a view including abstract names for the specifications, and a serious error in the specifications causes them to change, then the view of the task need not change; the system can re-fetch the specification to re-establish the correct view. Moreover, if a testing task had been specified but not activated, even the first fetch of the tester's view would be correct.

We have described a management system by analogy with an operating system, discussed the data being managed and its relationships, and presented examples of its intended use. The current prototype is designed to handle basic task structures, resources and minimal accounting information. It assumes an available programming-in-the-small environment, for example the IDEAL environment of ENCOMPASS. Planned extensions will support various development methods and detailed project accounting.

6. Summary and Conclusions

Current software development shops are characterized by poorly integrated tools which are inadequate to control the complexity of software development and maintenance. This paper assumes that an automated environment which supports the entire life-cycle will improve the quality and efficiency of the software production process. The SAGA Project is investigating models of software development which support automation. At present, we are emphasizing models of project management and configuration control.

ENCOMPASS is the first complete environment constructed by the SAGA Project. ENCOMPASS supports a formal development method similar to VDM, as well as providing basic facilities for configuration control and project management. A VDM-like methodology was chosen because it supports the specification, validation, design, implementation, and verification of software. It also provides completion criteria for the steps in the production process and offers limited, but well-defined, project management goals. The design, construction and use of ENCOMPASS revealed many shortcomings in its project and configuration management systems.

SAGA is now creating new systems both to correct these deficiencies and support more of the life-cycle. CLEMMA is a configuration librarian which maintains software structures and provides views of a project's components. CLEMMA capitalizes on existing data base and file system technology to provide flexible support for abstraction and manipulation of software components. It can be easily updated to provide new facilities and abstractions without reorganizing the project data.

The project management system supports the integration and control of the software development and management processes. It implements management policies through the use of interaction protocols and project access permissions. It also supports repositories of project information and components. The management system is based on a process/resource model in which the process hierarchy models the personnel and work breakdown structures of the project. The project management system controls project access, supports resource allocation and usage, and coordinates and synchronizes task activities.

The improved configuration and project management systems are under implementation; many components are complete. The two systems are complementary: efficient automation of the software development process depends on the effective integration of project management and configuration control. The systems must combine to provide users with consistent, task related, functional abstractions of activities and resources. The configuration and project management models have application to existing software development practices; however, the SAGA Project is seeking to apply them to an improved, rigorous software development methodology.

Automating the entire software life-cycle will require continuing research; one reason is the immaturity of the software engineering discipline. For example, future development methodologies must incorporate more formal approaches to requirements analysis, reuse and maintenance. Future environments should also have more advanced system architectures which support knowledge-based tools.

We believe that the configuration and project management models and systems currently proposed can significantly enhance many aspects of the software life-cycle. However, these models and systems must evolve as we strive for more effective development methodologies; for example, improved implementation methods must be pursued as usage data is gathered. Although life-cycle automation is a long term

research problem, the current work in project and configuration management can do much to improve software development as it is practiced today.

7. REFERENCES

[Babich, 86]. Babich, Wayne A. **Software Configuration Management.** Reading: Addison-Wesley, 1986.

[Campbell & Richards, 81]. Campbell, Roy H. and Paul Richards, *SAGA: A System to Automate the Management of Software Production,* **Proceedings of 1981 National Computer Conference,** Chicago IL, May, pp. 231-234.

[Campbell & 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, pp. 145-159.

[Chen, 76]. 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.

[Dolotta, et al, 77]. Dolotta, T.A., Haight, R.C., and E.M. Piskorik, editors. **Documents for the PWB/UNIX Time-Sharing System,** Edition 1.0. Bell Laboratories, Inc., October 1977.

[Dowson, 87]. Dowson, Mark. *ISTAR - An Integrated Project Support Environment.* **Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,** ACM SIGPLAN Notices (January 1987) vol. 22, no. 1, pp. 27-33.

[Fairley, 85]. Fairley, Richard. **Software Engineering Concepts.** McGraw-Hill, New York, 1985.

[Gunther, 78]. Gunther, R. **Management Methodology for Software Product Engineering.** Wiley Interscience, New York, 1978.

[Howes, 84]. Norman R. Howes. *Managing Software Development Projects for Maximum Productivity,* **IEEE Transactions on Software Engineering,** (January, 1984) vol. SE-10, no. 1.

[Jones, 80]. Jones, Cliff B. **Software Development: A Rigorous Approach.** Prentice-Hall International, Engelwood Cliffs, N.J., 1980.

[Osterweil, 87]. Osterweil, Leon. *Software Processes are Software, too.* **Proceedings of the 9th International Conference on Software Engineering.** (March 1987), pp. 2-13.

[Sum, 87]. Robert N. Sum, Jr. "A Summary of the Software Development Cycle of the AT&T System 75 in Middletown, NJ", Report No. UIUCDCS-R-87-1332, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1987.

[Terwilliger & Campbell, 86a]. Terwilliger, Robert B. and Roy H. Campbell. "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, (also to appear, with revisions, in the **Journal of Systems and Software**).

[Terwilliger & Campbell, 86b]. "PLEASE: Executable Specifications for Incremental Software Development", Report No. UIUCDCS-R-86-1295, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1986, (also to appear, with revisions, in the **Journal of Systems and Software**).

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-87-1333	2.	3. Recipient's Accession No.

4. Title and Subtitle
Automating the Software Development Process

5. Report Date
May 1, 1987

6.

7. Author(s)
R. H. Campbell, H. Render, R. N. Sum, Jr., R. Terwilliger

8. Performing Organization Rept. No. R-87-1333

9. Performing Organization Name and Address
University of Illinois
Department of Computer Science
1304 W. Springfield, 240 Digital Computer Lab
Urbana, Illinois 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.
NASA NAG 1-138
AT&T IL SOFTWARE ENG.

12. Sponsoring Organization Name and Address
NASA Langley Research Center
Hampton, Virginia 23665

AT&T Information Systems
307 Middletown-Lincroft Rd.
Lincroft, NJ 07738

13. Type of Report & Period Covered

14.

15. Supplementary Notes

16. Abstracts

Much of the software development process is repetitive, tedious to perform, but possible to automate. Our research on the SAGA Project includes building models of software development tasks that both accurately reflect the processes involved and have direct and efficient implementations. In this paper we assume the desirability of a software engineering environment that supports the entire life-cycle; automation should greatly enhance the quality and efficiency of software production and maintenance. To study the problems involved, SAGA has constructed ENCOMPASS, a prototype environment which supports software development. ENCOMPASS has provided valuable insights and experience; however, during its development and use many limitations have surfaced. In this paper we emphasize the configuration and project management aspects of our work. We discuss the current capabilities and limitations of ENCOMPASS, as well as describing the new systems being constructed to both overcome its limitations and extend its life-cycle coverage.

17. Key Words and Document Analysis. 17a. Descriptors

software engineering, software development environments, configuration control, project management

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
unlimited	UNCLASSIFIED	24
	20. Security Class (This Page) UNCLASSIFIED	22. Price

APPENDIX D

FORMAN
A Form Compiler

R. N. Sum, Jr.

FORMAN:

A Form Compiler

Reference Manual

Robert N. Sum, Jr.

1. Introduction

FORMAN is a system for creating forms for use in interactive programs. It consists of a compiler and a run-time library. From an input file specifying the form, the compiler produces several files of C code that realize the form using the FORMAN run-time library and the *win* screen interface library.

This document is the FORMAN reference manual describing the structure of FORMAN forms, the FORML form language and its compiler, the FORMAN run-time library, programming with FORMAN, and the FORMAN implementation.

The following terminology is used throughout this document: *FORMAN* refers to the entire form system whereas *forman* refers to the FORML compiler. Also, a *user* is one using a program containing forms made with FORMAN whereas a *programmer* is one using FORMAN to write a program containing forms.

2. Form Structure

This section describes a form with respect to the form's displayed appearance and user interaction. To do this, brief note about the implementation is needed first. Then, a description of the displayed appearance is presented. Finally, user interaction is described.

2.1. Implementation Note

At the implementation level, forms are made up of fields. Every field has a *name*, a *prompt*, some *data*, and up to three *actions*. The name is an internal (not displayed) identifier often used to interface to C. It is discussed further in sections on FORML and programming. The prompt and data are part of the display as the displayed name and value of a field. The actions are determined by the programmer as the commands available to the user. For some fields, the programmer also specifies the starting column of the field on the display. (The reader may note the rather direct relationship between FORMAN fields and win fields.)

2.2. Displayed Appearance

The displayed appearance of a form has four partitions which are the title, menu, message, and body. The typical form display shown in the figure *An Example Form Display* illustrates the form partitions. The figure is an empty form containing only a system assigned number for a "new module". The remainder of this section describes the form partitions from the user level.

2.2.1. Title

The title partition is a single field. The title's prompt string is centered on the first line of the screen as the title of the form and it does not scroll. The three actions specify initialization, quit, and exit actions for the form. These will

An Example Form Display

```
                          Module Form
                          *** MENU ***
                 browse  report
Messages:  new module
                          *** FORM ***
                           ... top ...
Module Body:
     module name:
     number: 1103527590
     project:
     manager:
End Module Body.
List of Users:
End list of Users.
List of Imports:
End list of Imports.
```

be described during the cursor interaction section.

2.2.2. Menu

The menu partition of the form is optional. When it does exist, it consists of a field indicating the beginning of the menu and a menu made of matrix of programmer defined fields. The menu does not scroll. The prompt of each field is displayed as the action available to the user. A menu field usually has only one action associated with it.

2.2.3. Messages

Forman automatically supplies this non-scrolling field in every form with its prompt "Messages: ". It is used by the command interpreter for error messages. It is also available to actions as will be described in the run-time library section.

2.2.4. Body

The body of a form consists of two fields and a list of records. The first field indicates the beginning of the form area and does not scroll. The second field indicates the top of the form and does scroll. The list of records contains most of the fields for user data. The list of records scrolls.

2.2.4.1. Records

The list of records contains two kinds of records: single records and list records. Single records have exactly one occurrence in the body of the form. They have begin and end record fields which bracket their data fields. In the example, the record starting at "Module Body:" and ending at "End Module Body." is such a record. List records may have zero or more occurrences in the body of a form. Therefore, they have begin and end list records in order to represent empty lists. The other two records in the example are list records. Because the example is a new empty form, there are no records in the lists. Records in the list records are similar to single records. One difference is that forman automatically builds insertion and deletion actions for user interaction with list records.

2.2.4.2. Fields

Regardless of the kind of a record, it is composed of fields. Each field has programmer specifiable insert, delete, and pick actions. The insert action is used to insert data into a field. Additionally, each field has a type that is checked by FORMAN. FORMAN supports the predefined types identifier, integer, path, static, and string. It also supports programmer defined scalar types similar to Pascal scalar types. These types are used to aid the user in interactively entering correct data.

2.3. User Interaction

This section describes form initialization, cursor motion, menu actions, list of records manipulation, inserting field data, and exiting from a form.

2.3.1. Initialization

Each form may have an initialization action which is performed after the form is built internally, but before the user input is accepted so that the programmer may place initial values in various fields, if he so desires. In the example, the number was entered by the initialization action. The user really has no control over form initialization.

2.3.2. Cursor Motion

The cursor motion on the screen is governed by the win command interpreter. A summary of its commands with comments about forms is:

i) h, <backspace> – move to the field to the left,
ii) j, <return> – move to the field below,
iii) k – move to the field above,
iv) l, <space> – move to the field to the right,
v) d – if the field (or record) has a delete action, invoke it,
vi) i – if the field (or record) has an insert action, invoke it,

vii) p – if the field (or record) has a pick action, invoke it,
viii) r, <ctrl-L> – refresh (redraw) the form from scratch,
ix) b – move to the bottom of the form,
x) B – move to the bottom of the display,
xi) t – move to the top of the body of the form,
xii) T – move to the top of the display,
xiii) q – quit the form. If the form has a quit action, invoke it,
xiv) e – exit the form. If the form has an exit action, invoke it.
xv) otherwise, post an error message to the message field and try to ring the display's bell!

The interpreter erases the message field after each successful command in order to prevent lingering error messages.

2.3.3. Menu Actions

The menu actions are determined by the programmer; however, the are following convention is encouraged. Of a menu field's three actions, the first two are conventionally unused, but the third is the application action with which the field is associated. This third action coincides with the pick (p) command of the win command interpreter.

2.3.4. List of Records

In the form body, single records have only programmer defined actions associated with their begin record. There is no convention for these other than that they are almost never used. When they are used, they correspond to the insert, delete, and pick actions of the command interpreter.

List records, however, have actions associated with the list begin field and record begin field (of each record in the list) which are generated by forman. The insert action is attached to the begin list field and the deletion action is attached to the begin record field of each record in the list. These actions allow the user to insert records into the list and delete records from the list. The programmer may also specify three actions (insert, delete, and pick) to be performed at the user's command. These actions are attached as follows: The programmer insert and delete actions are called from the forman generated actions noted above. The insert action is called after the fields have been inserted, but before the display is redrawn. The delete action is called before the record and its fields are deleted. The pick action is attached to the begin record field of each record in the list.

2.3.5. Field Data

Each field in a record has a type which FORMAN checks during the insertion action as noted earlier. FORMAN *type-checks* the user data by calling a run-time routine to check and display the user's data as it is entered. To enter data. the user uses the insert command of the interpreter, enters data, and terminates it with a <return>. The FORMAN run-time command will print

error messages (and possibly ring the bell) for erroneous input. It will not let the user terminate input without entering a correct value. To aid the user, the FORMAN routines support two additional commands during insertion:

i) <ctrl-T> — print a message describing the type of the field.
ii) <ctrl-R> — if the field is a scalar type, allow the user to run through the possible values as if he had entered them. The value is cycled to next one on each <ctrl-R>.

The type static specifically does not allow the user to input data. The programmer may still specify an insert action and it will be called by FORMAN. Details of the types may be found in the section on the run-time library.

2.3.6. Exiting a Form

Exiting from a form is done when using the quit or exit command of the command interpreter. Conventionally, the quit and exit commands act as follows: The quit command leaves the state of the form as if the user had never entered it. The exit command updates the state of the form with any new data that the user may have entered. In either case, if a programmer action for one of these commands fails, the user will remain in the form so that he can correct a mistake and see any error message if displayed.

This section has described the appearance of a form to a user in terms of both its display appearance and it command actions. The rest of the paper is devoted to the programmer's use of FORMAN.

3. FORML and Forman

This section describes the syntax and semantics of compiling the form specification language FORML. It includes the lexical constructs of the language (e.g. character set, reserved words), common parse elements (e.g. composite identifiers), types and naming (e.g. type checking), and finally describes an entire form (with an example). The EBNF syntax for FORML is included in the Appendix.

3.1. Lexical Constructs

The following tokens and directives are processed and executed by the lexical analyzer.

3.1.1. Character Set

The character set for FORML is the character set (usually ASCII) as supported by the host C programming language. FORMAN is case sensitive, if the host C programming language is. One is discouraged from using non-printable characters except where specifically needed (e.g. new lines, tabs for indentation).

3.1.2. White Space

Blanks and all non-printable characters are insignificant in multiplicity and generally serve as delimiters between tokens. The only exception to this is blanks when they appear in quoted strings.

3.1.3. Comments

FORML comments start with a % and end with a new line character. Comments act like white space.

3.1.4. Options

Options are embedded in comments in a manner similar to Pascal. A comment beginning %$ is followed with the third character indicating an option. Currently, the only option available is the include option indicated by the letter I. It is then followed by a path name enclosed in single quotes, "'", that indicates another FORML file. After the file name is acquired, the remainder of the line is treated as a comment and the file is inserted after the line, until the file ends. At the end-of-file, the lexer returns to the file from which the include was made. An example:

```
%$I '../h/common_types.h'
```

There currently is a nesting limit of 19 on include files (which probably exceeds the open file limit on many UNIX™ systems).

3.1.5. Special Symbols

The following characters are used as special punctuation in FORML:

```
; : " [ ]
```

3.1.6. Reserved Words

The following are FORML reserved words:

```
END FORM ID INTEGER IS LINE LIST MENU
NULL OF PATH RECORD STATIC STRING TYPE
```

3.1.7. Strings

A string is a double quote, followed by any number of printable characters, followed by a double quote. To put a double quote in a string, use two consecutive double quotes. In other words, the string consisting of only a double

UNIX is a trademark of AT&T Bell Laboratories.

quote is four double quotes. Theoretically, strings must be less than 1024 characters in length, but practically, they must not extend beyond the edge of a display when used in a form.

3.1.8. Integers

An integer is a sequence of digits. It is unsigned (i.e. a whole number) that also follows the rules of C's atoi(3) function.

3.1.9. Identifiers

An identifier is one of [_A-Za-z] followed by any number of [_0-9A-Za-z] using the common notation for ranges and alternatives of characters. The theoretical length limit on identifiers is 1024 characters.

3.2. Composite Identifiers

The composite identifier is the basic structure used in the FORML language declarations. A composite identifier consists of an identifier, a string, and a function array. The identifier is the name of the field or record in the form. This name is used by the compiler to generate names for functions and to interface with the file system when necessary. The string is the description (prompt) of the field or record that is displayed to the user. The function array is a list of three identifiers that correspond to programmer defined semantic actions which are functions written in C. They should have the C type "pointer to function returning integer" when used alone. (In other words, they are the names of C functions that return an integer.) If the programmer does not supply a semantic function, then he must use the reserved word NULL for each function that is not supplied.

3.3. Types and Naming

Because FORML is primarily a language of declarations, there is almost no type structure within itself. Most of the type checking is designed to occur at run-time during user data entry into the fields of a form. This type checking will be discussed in the section on the run-time library.

There are, however, two checks that forman could do in FORML. One is that programmer defined scalar types are defined before they are used in field declarations. Forman does this correctly. The other is ensuring unique names for functions that forman must generate. Unfortunately, forman currently does not do this correctly. Future versions of forman will ensure that generated function names are unique within a form. For now, the programmer should follow the following guidelines:

1. The identifier for each form within a program should be unique.
2. The identifier for each record within a form should be unique.
3. The identifier for each field within a record should be unique.

4. The programmer should not mix the above identifiers with the suffixes *_form*, *_insert*, or *_delete*.

These guidelines arise from the fact that the main function for a form is its title identifier with the suffix _form, and other function names are constructed by concatenating the title identifier, record identifiers, field identifiers, with the suffixes _insert and _delete.

3.4. A Form

In the previous section, the form structure was presented as it appears when displayed. Here, the form structure is presented from a syntactic viewpoint. A form also has four syntactic partitions: title, menu, type, and body. Except for message and type, they correspond directly to the partitions of the previous section. The figure *An Example Form* will be used throughout the following description.

3.4.1. Title

For this example, the file of FORML specifications would be *module.f* because forman uses the title identifier with a ".f" appended as the file name. Its displayed title is "Module Form". It has semantic functions for initialization ("module_init") and exiting ("module_exit") which will be supplied by the programmer. The quit function is NULL, meaning that no function will be called.

3.4.2. Menu

The menu partition for this form results in a displayed menu of a single line with two fields. The reserved word LINE is used to start a new line of the menu. (LINE is not needed for the first line of a menu, but it is good practice to use it.) The two fields share a common semantic function "sorry_not_imp" which probably posts a message to the user that the menu functions have not yet been implemented. Finally, the programmer has specified a starting column for each field. If omitted, forman just tries to make sure that fields do not overlap. If one does not wish to have a menu in a form, one merely omits this partition including the reserved words MENU and (the closest) END.

3.4.3. Type

The type partition is optional in the same way as the menu partition in that it is denoted by the pair of reserved words TYPE and END. The type partition is used when the programmer wishes to define scalar types for input by the user. Each scalar type consists of an identifier, a colon, a list of white-space-separated identifiers, and a semicolon. The example has one scalar type ("acc_type") with two identifier values ("read_acc" and "modify_acc"). When a user enters a value for a field of a scalar type, it must be one of the identifiers in the list for the scalar type.

An Example Form

```
FORM module "Module Form" [ module_init NULL module_exit ] IS
     MENU
          LINE browse "browse" [NULL NULL sorry_not_imp] 16;
               report "report" [NULL NULL sorry_not_imp] 24;
     END
     TYPE
          acc_type : read_acc modify_acc;
     END
     RECORD module_body "Module Body" [NULL NULL NULL] OF
          name    "module name:" [NULL NULL NULL] ID;
          number  "number:"      [NULL NULL NULL] STATIC;
          project "project:"     [NULL NULL NULL] ID;
          manager "manager:"     [name_check NULL NULL] ID;
     END
     RECORD LIST user_record "Users" [NULL NULL NULL] OF
          user    "user name:"     [name_check NULL NULL] ID;
          access  "access:"        [NULL NULL NULL] acc_type;
          comment "brief comment:" [NULL NULL NULL] STRING;
     END
     RECORD LIST import_record "Imports" [NULL NULL NULL] OF
          item "item name:" [NULL NULL NULL] ID;
          file "from file:" [NULL NULL NULL] PATH;
     END
```

3.4.4. Body

The body of the form begins with the first record and ends at the end of file.

3.4.4.1. Records

Each record begins with the reserved word RECORD, optionally is a list record by using reserved word LIST. has a composite identifier, the reserved word OF, a list of fields, and ends with the reserved word END. When displayed, a single record has a begin field and end field which use the prompt string in the composite identifier. The functions of the composite identifier are assigned to the begin field. In the example, the "module_body" record is a single record. When a list record is displayed, initially only list begin and list end fields which use the prompt string of the composite identifier are displayed. The insert function of the composite identifier is attached to the begin list field. When the user inserts a

record, the identifier of the composite identifier is used in begin and end record files. Also, the delete and pick functions are attached to the begin record field. The "user_record" and "import_record" records are list records in the example.

3.4.4.2. Fields

Each record consists of fields of various types. The exact implementation of these types is done in the run-time library to be described in the next section. All the types except integer are included in the example. (The author could not think of a reasonable integer to be input here.) Finally, forman automatically assigns the starting column on the display both fields and records.

4. Run-Time Library

The FORMAN run-time library provides the run-time support for the form code produced by forman. It includes routines that type-check user input and allow forms to be stored in files. The library also includes a few environment variables that are useful to the programmer.

4.1. Type Checking

Type checking in the run-time library is designed to ease the burden of the programmer when acquiring user input. Earlier, the user commands were presented. Here, the FORMAN functions that support the field types of identifier, integer, path, scalar, string, and static are described.

4.1.1. Identifier

In FORML, this type is represented by the keyword ID. The run-time library has a routine called *ID_insert* that provides this type checking. It prevents the user from entering characters other than those legal for an identifier. An ID_insert identifier is identical to a FORML identifier except that "." is also a legal identifier character. This reason for adding "." is to facilitate using identifiers as UNIX file names.

4.1.2. Integer

In FORML, this type is represented by the keyword INTEGER. The run-time library has a routine called *INT_insert* that provides this type checking. It prevents the user from entering characters other than those legal for an integer. An INT_insert integer is identical to a FORML integer.

4.1.3. Path

In FORML, this type is represented by the keyword PATH. The run-time library has a routine called *PATH_insert* that provides this type checking. A *path* is that of a UNIX file system path. It consists of a pattern of ID_insert identifiers and slashes that must end with an identifier.

1) <path> → [/] { <id> / } <id>

PATH_insert ensures that a path has the above form.

4.1.4. Scalar

In FORML, this type is represented by a user defined scalar type. The run-time library has a routine called *SCA_insert* that provides this type checking. It prevents the user from entering identifiers other than those legal for the particular scalar type. To do this forman generates a type table containing the legal identifiers and at run-time SCA_insert is called with the appropriate table as a parameter. An SCA_insert identifier is identical to a FORML identifier.

4.1.5. String

In FORML, this type is represented by the keyword STRING. The run-time library has a routine called *STR_insert* that provides this type checking. It prevents the user from entering characters other than those legal for a string. An STR_insert string is identical to a FORML string.

4.1.6. Static

The forman run-time library support for this type is by not having any way for the user to input to a field of this type. Usually, static fields are initialized in the programmer's form initialization function. Unfortunately, the current version of forman does not provide run-time support for this. The programmer must use the form structure and follow the pointer links of win to use this type. The code for the example of the previous section shows this. Future versions of forman will generate an access function that will return a pointer to the field for use with win functions, thereby eliminating programmer traversal of the win structure.

4.2. Form Storage

The run-time library contains two functions for file storage of forms: *FORM_fetch* and *FORM_store*. FORM_fetch takes as its parameter a file descriptor that has been opened using fopen(3) and fills the current form from the file. Similarly, FORM_store takes as its parameter a file descriptor that has been opened using fopen(3) and writes the current form to the file. There are more details on using these functions in the section on programming with FORMAN.

4.3. Environment Variables

There are two environment variables that forman sets that can be very helpful to the programmer. They are *current_FORM_win* and *current_FORM_msg*. The former is useful for manipulating the data in a form and the latter is useful for displaying error messages and other useful information. There are more details on using these variables in the section on programming with FORMAN.

5. Programming with FORMAN

This section describes how to access the FORMAN system and presents some programming hints.

5.1. FORMAN Access

FORMAN was developed as a screen interface mechanism for the PROMAN project management system. As a result, this library resides in the project management research area of the author. (For convenience, we will use the tilde '~' in the usual manner throughout this paper to denote a home directory in UNIX.) Access to the compiler and the run-time library are presented below.

5.1.1. Compiler Access

The FORMAN compiler, *forman*, is located in

~sum/management/bin

In order to avoid the full path name one should include it in his UNIX PATH environment variable. It is possible that the compiler could be located in different places on different machines, so one might check with the person in charge of the FORMAN system if he has problems with access to it. To use forman, the form's specification in the language FORML (described later) should be in a file whose name is the same as the name of the form it contains with ".f" as its suffix. (This is an overload of the .f suffix. But, forman was not intended for use with FORTRAN. This also means that one may use a ".f.c" rule in make(1).)

To compile the source file into C code in the source file's directory, one does:

forman my_form.f

A successful compilation results in the generation of four files of C code. One file is an include file; the other three are code files. The file *my_form.h* contains all the external declarations of the C functions generated by the compiler and the programmer's external semantic functions. The file *my_form.c* contains the main function for the form and is probably the only function that the programmer's program will call. The file *my_form_i.c* contains all the functions needed to handle data insertions by the program's user. The file *my_form_d.c* contains all the functions needed to handle data deletions by the program's user. An unsuccessful compilation results in an error message that is cryptic except for the line number and character at which the error occurred. Because forman currently has no error recovery, it dies on the first error it finds. Fortunately, this is not much of a problem because form specifications are not very long and the language is fairly simple.

5.1.2. Library Access

The FORMAN run-time library is needed when compiling and linking forman's code into a program. (As described later, the programmer must supply at least a minimal main program. Also, note that compilers require either a full path name or an include option '-I/partial/path/name' to work correctly.) To use the FORMAN run-time library, one must include the files

```
~sum/management/include/FORM.h
my_form.h
```

and link the library files

```
~sum/management/lib/FORM.a
~sum/management/lib/IO.a
~sum/management/lib/win.a
```

into one's program. Because win is built on top of curses(3x) which is built on top of termcap(3x), a sample input line for cc(1) might look like:

```
cc -I/mntb/3/srg/sum/management/include -o my_prog my_prog.c
my_form.c my_form_i.c my_form_d.c ~sum/management/lib/FORM.a
~sum/management/lib/IO.a ~sum/management/lib/win.a -lcurses
-ltermcap
```

which probably justifies using a "makefile" and make(1). Please note that one need not, and probably should not, include files related to curses(3x) and termcap(3x) other than their libraries, as shown above. More information about win is available in "WIN: A Simple Field-Oriented Screen-Interface Library, *Reference Manual.*" Finally, it is possible that the library could be located in different places on different machines, so one might check with the person in charge of the FORMAN system if he has problems with access to it.

5.2. Programming Hints

These programming hints describe both necessities and suggestions when using FORMAN. Necessities will be clearly stated. The example that has been used throughout this manual is complete and included in the FORMAN distribution. The implementation section has details of its location. These programming hints are for the main program, form initialization and storage, and user (programmer defined) actions.

5.2.1. Main Program

The FORMAN system does NOT provide a main program. This is the responsibility of the user. The minimal main program is pictured in the figure *Main Program.* Basically, the main program must initialize and stop the win

library and call the main routine for a form. The main routine for a form is always the name of the form with the suffix *_form* appended to it. It has no arguments. It returns an integer; however, it rarely returns a value other than 0. On an error, it would return a non-zero value.

5.2.2. Initialization and Storage

The following are merely suggestions and examples of putting data into static fields and using the form storage routines. The figures *A Form Initialization* and *A Form Exit* show basic use of these ideas. For the initialization function, FORMAN gives it a win field pointer to the title field as an argument. For the exit function, FORMAN passes a pointer to the field where the user (cursor) was. This can be awkward and may change in the future.

One should note that one can NOT use the *si_type* slot in win fields when using the form storage routines because they depend on the values that FORMAN puts in them. The storage functions save the slots *si_type*, *si_data*, and *si_user*. FORMAN uses only the first two of these slots. The programmer may use the last two.

5.2.3. User Actions

Examples of user (programmer defined) actions added to a menu field and an insert data field are shown in figures *An Menu Function* and *A Data Function*, respectively. The menu function demonstrates how one can display error messages to the user. The data field indicates how to access the data of a field. Essentially, all user functions are passed a pointer to the win field structure and can directly access the data.

Main Program

```
#include "win.h"
#include "module.h"

main()
{
     init_screen_window();
     module_form();
     stop_screen_window();
}
```

A Form Initialization

```
#include <stdio.h>
#include "win.h"
#include "FORM.h"

int  module_init( si )
     SI *     si;
{
     FILE *     fetch;

     if ((fetch = fopen( "module.sav", "r" )) == NULL ){

          int     i;
          char     num[24];

          /* initialization of i is semi magic based on lines in form */
          /* future versions of FORMAN will have access functions */
          for( i = 8; i > 0; i-- ){
               si = si->si_down;
          }
          sprintf( num, "%d", rand() );
          insert_data( si, num );

          insert_data( current_FORM_msg, "new module" );
          return 0;
     }

     FORM_fetch( fetch );
     fclose( fetch );
     return 0;
}
```

A Form Exit

```
#include <stdio.h>
#include "win.h"
#include "FORM.h"

int  module_exit( si )
     SI *      si;
{
     FILE *      store;

     if ((store = fopen( "module.sav", "w" )) == NULL ){
          post_field( current_FORM_msg, "module_exit: can't open file.");
          return 1;
     }
     FORM_store( store );
     fclose( store );
     return 0;
}
```

A Menu Function

```
#include "win.h"
#include "FORM.h"

int  sorry_not_imp( si )
     SI *      si;
{
     post_field( current_FORM_msg, "Sorry command not implemented." );
     return 1;      /* return 1 so message is not erased */
}
```

A Data Function

```
#include <stdio.h>
#include <pwd.h>
#include "win.h"
#include "FORM.h"

int  name_check( si )
     SI *     si;
{
     struct passwd *     buf;

     if ((buf = getpwnam( si->si_data )) == NULL){
          post_field( current_FORM_msg, "WARNING:  user has no login." );
          return 1;
     }
     return 0;
}
```

6. Implementation

This section describes the directory structure and code characteristics of the implementation of the FORMAN system. It will do so by describing the directory structure and then some characteristics of the each component of the forman system.

6.1. Directory Structure

The parent directory for FORMAN is

~sum/management/src/lib/forman

and it contains the following files and directories:

a) Makefile - the input file for make(1) to make the entire system.
b) doc - a directory containing all external documentation such as this one.
c) example - a directory containing the code for the example form in this manual.
d) h - a directory of include files that are used in more than one FORMAN component.
e) lex - a directory containing forman's lexical analyzer.
f) lib - a directory containing the FORMAN run-time library.

g) main – a directory containing code for the forman main program and man(1) manual page.
h) parse – a directory containing forman's parser

Occasionally, there are miscellaneous directories with "test" as part of their names. These are transient and used to test FORMAN and other related programs. They are just for fun.

6.2. Makefile

The FORMAN system can be made with one command from the top level directory:

make ROOTDIR=*/rootdir* install

FORMAN requires the following structure of */rootdir*:

a) It must be an absolute path name (i.e. start with a "/").
b) It must have sub-directories: *bin*, *include*, *lib*, and *man*.
c) The *man* sub-directory must have sub-directories *man1* and *man3*.

Otherwise, paths inside each of FORMAN's sub-directories' makefiles must be changed.

Although distributed with all the C source files, FORMAN uses a version of S/SL to generate parsers in C for forman's lexical analyzer and parsers. S/SL is not distributed with FORMAN or PROMAN although it is available. Finally, FORMAN depends on the win library and the IO library included in the distribution. They must exist before FORMAN is made. This is ensured when FORMAN is made from a PROMAN system or PROMAN library distribution.

6.3. Documents

Several documents are available about FORMAN. The main document is this reference manual. The others are subsets of this manual. A makefile in the doc directory describes how to produce these documents using some form of troff or nroff.

6.4. Forman

Forman uses files in the h, lex, parse, and main directories. The h directory contains several include files of S/SL and C code that are used by the lexical analyzer and parser. The lex directory contains all the S/SL and C code for the lexical analyzer plus a makefile for building and installing it. Similarly, the parse directory contains all the S/SL and C code for the parser plus a makefile for building and installing it. The main directory contains the csh(1) script that drives the compiler, the man(1) manual page (*roff source), and of course, a makefile.

When run, the compiler consists of the driving csh(1) forman script, the lexical analyzer and the parser. The forman script checks the argument and constructs temporary file names for the two passes. It then runs the each pass and cleans up any left over temporary files. The lexical analyzer breaks the input file into tokens including the processing of include files. The parser recognizes the language and calls semantic functions to build the C code of the form. The current version of the compiler has no error recovery. It does, however, try to print reasonable error messages and quit gracefully.

6.5. Run-Time Library

The FORMAN run-time library contains C code to implement all the run-time support functions and the include file for user programs. It also contains a makefile for building and installing the library.

6.6. Example

The sub-directory example contains an example form program with its makefile. Depending on the installation of FORMAN, one may need to change the values of path names in the make file. To make and run the example do:

```
make
module
```

The example has an exit action that stores the form in a file *module.sav* and a quit function that leaves the module alone (NULL in FORML). One should not be concerned with the "module number" for a new module as it is a random number.

7. Closing

The FORMAN reference manual has described the FORMAN system from the user and programming points of view. It has presented user interaction, the FORML language, the run-time library, an example of a form, and a brief overview of the implementation structure. Hopefully, you will find FORMAN to be a useful tool.

8. Appendix: FORML Syntax

This appendix contains the EBNF description of the syntax of the FORMAN form language, FORML.

1) <form> → **FORM** <composite_id> **IS** <menu> <type_decl> <record>

2) <menu> → [**MENU** { <menu_field> } **END**]

```
3) <menu_field>      → { LINE | <composite_id> | <integer> } ;

4) <type_decl>       → { TYPE { <id> : <id> { <id> } ; } END }

5) <record>          → { RECORD | LIST | <composite_id> OF { <field>
                       } END }

6) <field>           → <composite_id> <type> ;

7) <composite_id>    → <id> <string> <functions>

8) <functions>       → [ <func_id> <func_id> <func_id> ]

9) <func_id>         → <id>
10)                    | NULL

11) <type>           → ID
12)                    | INTEGER
13)                    | PATH
14)                    | STATIC
15)                    | STRING
16)                    | <id>

17) <id>             → [A-Za-z_] { [A-Za-z0-9_] }

18) <string>         → " { [A-Za-z0-9_] } "

19) <integer>        → [0-9] { [0-9] }
```

Please note that strings can include any printable character; not just numbers, digits, and blanks. The double quote is doubled to include it in a string.

APPENDIX E

WIN
A Simple Field–Oriented Screen–Interface Library

R. N. Sum, Jr.

WIN:

A Simple Field-Oriented Screen-Interface Library

Reference Manual

Robert N. Sum, Jr.

1. Introduction

The *win* library is a simple, field-oriented, screen-interface library. It is simple in that it provides basic multiple screen manipulations, but does not do windows. Field-oriented means that the user partitions the screen area into fields that have a static prompt string and a dynamic data string. Screen-interface means that it is designed for building terminal interfaces for interactive programs. Finally, of course, library means that it is a collection of subroutines.

This manual is brief, but it will describe win access, data structures functions, and implementation. Most programmers will not need to bother with the implementation section. But, because win is still developing, it may be that the code is more useful than expected. A simple closing rounds out the paper.

2. Access

Win was developed as a simple, portable screen interface mechanism for the PROMAN project management system. As a result, this library resides in the project management research area of the author. (For convenience, we will use the tilde '~' in the usual manner throughout this paper to denote a home directory in UNIX.™ Note that compilers require either a full path name or an include option '-I/partial/path/name' to work correctly.) To use win, one must include the file

~sum/management/include/win.h

and link the library file

~sum/management/lib/win.a

into one's program. Also, because win is built on top of curses(3x) which is built on top of termcap(3x), a sample input line for cc(1) might look like:

```
cc -I/mntb/3/srg/sum/management/include -o myprog myprog.c
~sum/management/lib/win.a -lcurses -ltermcap
```

which probably justifies using a "makefile" and make(1). Please note that one need not, and probably should not, include files related to curses(3x) and termcap(3x) other than their libraries, as shown above. Finally, it is possible that the library could be located in different places on different machines, so one might check with the person in charge of the library if he has problems with access to it.

UNIX is a trademark of AT&T Bell Laboratories.

3. Data Structures

In this section we will describe the win data structures from a programmer's viewpoint, i.e. the description will be conceptual rather than solely about implementation details. Win has two data structures: the *screen* also known as a *screen_window* and the *field* also known as a *screen_item*. Each of these will be described in turn and a final section will note the implementation details needed for programming. (When reference to the actual physical terminal display is needed, display will be used.)

3.1. Screens

Even though win does not support windows, it does support multiple screens. That is, even though one can not partition the terminal display to view more than one screen simultaneously, one can have several screens with a *current* screen displayed. Win keeps a list of screens and has several functions that allow the programmer to manipulate them. Except for these functions, all functions act on the current screen.

3.2. Fields

The field is the basic division of the screen. One may think of a screen as a matrix of fields. Each field is one line in height but any number of characters in length. (The programmer must manage field lengths.) Win supports screens longer than the number of lines of the terminal display when the programmer sets all or part of the screen as scrollable. Win does not support screens wider than the terminal display (see remark above). It does, however, make the length and width of the terminal display available through the integer variables LINES and COLS as in curses(3x).

Each field has slots associated with it for programmer data and functions. There are three groups of slots; one each for character display, general data, and user functions.

3.2.1. Character Display

Each field has three special data slots for displaying characters: *prompt*, *x*, and *data*. The prompt slot is used to hold the prompt displayed to the user concerning the field. It is set at field insertion (creation) time and never changed. The x slot is the column number where the prompt starts to be displayed. It is also set at field insertion time and never changed. Finally, the data slot is where the user's input data is placed. It may change fairly often, usually as a result of user functions.

3.2.2. General Data

Each field has two general data slots for the programmer's own use: *type* and *user*. They are both suitable for integers and can be set directly by the programmer via a pointer to the field.

3.2.3. Function Fields

Each field has three function slots: *insert*, *delete*, and *pick*. These slots may contain pointers to functions which may be called based on the program's user's request. They are named as such by convention with user actions of inserting more fields or data into fields, deleting fields or data from fields, and invoking operations from a menu, respectively. These are set at field creation time and are not changed thereafter in the manner of the prompt and x slots.

3.3. Programming

The table, *Concepts to C Code*, associates the terms used to describe the screens and fields with the C code needed. For further details, see the include file *win.h* mentioned in the section 2.

4. Functions

This section is a listing of win functions (subroutines) with descriptions of their parameters and actions. There are three general categories of functions: screen management, field operations, and cursor motion.

4.1. Screen Management

The following functions are used for creating, destroying, setting, and otherwise managing screens.

void init_screen_window()

Perform all the necessary initialization for the win library. This routine must be called once before any other calls or references to the library are made.

SW * create_screen_window()

Create a screen, set it to be the current screen, and return a pointer to it.

Concepts to C Code		
win concept	C type	C typedef or field id
screen	struct screen_window	SW
field	struct screen_item	SI
field slot prompt	char *	si_prompt
field slot data	char *	si_data
field slot type	int	si_type
field slot user	int	si_user
pointer to function	int (*PICK_ACTION)()	PICK_ACTION
field slot insert	PICK_ACTION	si_insert
field slot delete	PICK_ACTION	si_delete
field slot pick	PICK_ACTION	si_pick

SW * set_screen_window(SW *)

Set the current screen to the screen pointed to by its argument. It returns its argument.

void destroy_screen_window(SW *)

Destroy the screen window pointed to by its argument, reclaiming all of its storage.

void refresh_screen_window()

Redraw the terminal display from scratch.

void refresh_screen_on()

Enable the refresh routine to redraw the terminal screen. Otherwise, refresh_screen_window merely updates internal structures. This is sometimes useful when using the motion routines *behind the scenes.*

void refresh_screen_off()

Disable the refresh routine from redrawing the terminal screen.

void suspend_screen_window()

Temporarily suspend action of the win library and reset the terminal display modes. This is useful if the application is using system(3) to run another program that uses the terminal's display.

void resume_screen_window()

Reset the terminal display modes and the win library. It also refreshes the screen.

void top_screen_window()

Move to the cursor to the field at the top of the terminal's display. This function is listed here because it is associated with the physical display.

void bot_screen_window()

Move to the cursor to the field at the bottom of the terminal's display. This function is listed here because it is associated with the physical display.

void stop_screen_window()

Stop the win library. This should be used before a program exits to clean up and reset the terminal display modes.

4.2. Field Operations

These functions are concerned with inserting, deleting, filling, and printing fields.

SI * insert_line(type, x, prompt, insert, delete, pick)

SI * insert_right(type, x, prompt, insert, delete, pick)

Win assumes that one will initially build a screen like a matrix in row major order. Therefore, there are insert_line (new row, down) routines and insert_right (next col) routines. Insertions always occur with respect to the current field (where the cursor is). The parameters are:

i) type - the value for integer general data slot.
ii) x - the column at which the prompt is to begin.
iii) prompt - a character pointer to the string of characters that will be the prompt. This character string is copied.
iv) insert, delete, pick - pointers to functions returning integers that are usually associated with user actions.

They return a pointer to the new field.

SI * insert_data(SI *, char *)
This function inserts the data pointed to by the second argument into the data slot pointed to by the first argument. It copies the data and returns its first argument. It does not print the data on the terminal's display.

SI * top_of_scroll(SI *)
This function sets the first line of the scrollable part of the screen to the line containing its argument. This argument should be the first field in a line. The remainder of the screen will then be scrolled as necessary by motion commands if the screen is longer than the physical display.

SI * delete_field(SI *)
This function deletes the field pointed to by its argument which must also be the position of the cursor. It then moves the cursor to a neighboring field, if there is one, in the order: right, left, down, up.

SI * post_field(SI *, char *)
This function is like insert_data, except that it does print the data on the terminal's display.

SI * fetch_field(SI *)
This is a simple routine (that might be used for a field's insert slot) that accepts a string of characters as data from the user at the field of its argument. It returns its argument.

4.3. Cursor Motion

This group of functions is used for moving the cursor around and figuring out where you are in a screen window. They all refresh the screen when necessary and unfortunately, sometimes when it is not necessary.

SI * up_field()

SI * down_field()

SI * left_field()

SI * right_field()
The above functions all move the cursor to the next field in the indicated direction, if possible. Otherwise, they do not move. The return a pointer to the new field.

SI * top_form()

This function moves the cursor to the first field in the scrollable part of the screen. It returns a pointer to the new field.

SI * bot_form()

This function moves the cursor to the first field in the scrollable part of the screen. It returns a pointer to the new field.

SI * current_field()

This function just returns a pointer to the field that the cursor is at.

int screen_command(message, quit, exit)

This function is a fairly standard command interpreter with motions similar to vi(1). Its arguments are a pointer to a field to post error messages, a PICK_ACTION to use for the quit command, and a PICK_ACTION to use for the exit command. The commands are:

i) h, <backspace> – move to the field to the left,
ii) j, <return> – move to the field below,
iii) k – move to the field above,
iv) l, <space> – move to the field to the right,
v) d – if the field's delete function exists, call it with a pointer to the field as its argument,
vi) i – if the field's insert function exists, call it with a pointer to the field as its argument,
vii) p – if the field's pick function exists, call it with a pointer to the field as its argument,
viii) r, <ctrl-L> – refresh (redraw) the terminal's display from scratch,
ix) b – move the bottom of the screen,
x) B – move to the bottom of the terminal's display,
xi) t – move to the top of the scrollable part of the screen,
xii) T – move to the top of the terminal's display,
xiii) q – quit the command interpreter, if the quit function is not NULL, then execute it. If it returns an error (return value != 0), then stay in the interpreter. The quit function is by convention usually NULL and acts as an abort.
xiv) e – exit the command interpreter, if the exit function is not NULL, then execute if. If if returns an error (return value != 0), then stay in the interpreter. The exit function is by convention usually present and acts as the completion of the work session.
xv) otherwise, post an error message to message argument and try to ring the terminal's bell!

The interpreter erases the error message field after each successful command. For the field functions of the d, i, and p commands, if the function returns a non-zero value, then the error message field will not be erased. This

facilitates posting error messages from those routines. Also, because of the actions on the quit and exit commands, the interpreter almost always returns 0. On an extreme error it would return some non-zero value.

5. Implementation

This section contains an overview of the win library implementation. Win is written in the C programming language. Initially, an attempt to modularize the code was made, but as it was hacked much of that modularity was lost. This section will describe what remains of the modularity.

First, the high level structure is a main directory with sub-directories for source code and documentation. The main directory is

~sum/management/src/lib/win

which contains a make file *Makefile* for making the library, directory *lib* with source code, and a directory *doc* with the documentation. Note that because the man page is installed and treated like code, it is in the directory with the source code. Other documentation such as this manual is in the doc directory.

Within each sub-directory are make files that indicate the products available. Make(1) will make a product given its name and any noted special arguments. Changing the location of the source for win requires changes to paths in the make files. Making documents may also require changing the names of the processors in the make files.

The C code for win was structured so that there are two include files and four source files. The first include file is the user interface file *win.h*. It contains the data structure and function definitions. The second include file is an internal interface among the four source files called *internal.h*, of course. It contains internal data and function definitions. The first source file *win.c* contains all the user available functions (many of which do little more than call internal functions) except for the command interpreter. The file *command.c* contains the command interpreter. The remaining files contain internal functions that implement most of the win functionality. The file *internal_si.c* contains code to do most of the field (screen_item) manipulations concerning insertion, deletion, etc. The file *internal_sw.c* contains code to do most of the screen (screen_window) manipulations including initialization, window list management, display refreshing, and cursor motion.

6. Closing

This has been a rough introduction to the concepts and functionality of the win library for screen interfaces. Obviously, this is not a product, but has proven useful in the development of PROMAN. One may also wish to check out FORMAN a compiler for forms that takes quite a bit of drudgery out of using win.

APPENDIX F

Knowledge–Based Development in ENCOMPASS
(Preliminary Report)

Robert B. Terwilliger

Knowledge–Based Development in ENCOMPASS

(Preliminary Report)

Robert B. Terwilliger

Report No. UIUCDCS–R–87–1334
Department of Computer Science
University of Illinois at Urbana–Champaign
1304 W. Springfield Ave.
Urbana, Illinois 61801
217–333–4428
email: terwilli@a.cs.uiuc.edu

Preprint July 12, 1987

This research is supported by NASA Grant NAG 1–138
Roy H. Campbell Principal Investigator

Knowledge–Based Development in ENCOMPASS
(Preliminary Report)

Robert B. Terwilliger

Department of Computer Science
University of Illinois at Urbana–Champaign

This research is supported by NASA grant NAG 1–138
Roy H. Campbell Principal Investigator

Abstract

PLEASE is an executable specification language which supports incremental software construction in a manner similar to the Vienna Development Method. PLEASE is part of the ENCOMPASS environment which provides automated support for all aspects of the development process. In ENCOMPASS, PLEASE specifications are incrementally refined into Ada® implementations. ENCOMPASS is now being extended with a knowledge–based assistant which uses deductive synthesis techniques. During the refinement process, the assistant can provide advice on design and implementation decisions. The assistant also contains a library of program schemas which can be instantiated during development. In this paper, we give an overview of ENCOMPASS and present an example of development using the environment.

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[2,7,22,42,43,51]. Another solution is the use of tools which combine a *knowledge–base* and/or *artificial intelligence* techniques to support software engineering[3,4,19,32,39,41,49]. One such technique is *deductive synthesis*, the use of theorem proving techniques to create verified code from specifications[16,19,20,24,34]. ENCOMPASS[44,45] is an integrated environment to support incremental software development. ENCOMPASS is being extended with a *knowledge–based assistant* which uses deductive synthesis techniques. In this paper, we give an overview of ENCOMPASS and present an example of development using the environment.

The Vienna Development Method (VDM) supports the top–down development of software specified in a notation suitable for formal verification[5,6,14,25–27,40]. In this method, components are first specified using a combination of conventional programming languages and mathematics. These

Ada® is a trademark of the US Government, Ada Joint Program Office.

specifications 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 [46–48] is a wide–spectrum, executable specification language which supports a development method similar to VDM. PLEASE extends Ada[15,50] 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 may also be transformed into prototypes which use Prolog[13,30] to "execute" pre– and post–conditions. We believe that the early production of prototypes will enhance the software development process.

ENCOMPASS[44,45] is an integrated environment to support the incremental development of software using PLEASE. ENCOMPASS is a descendant of the SAGA project[8–11,28], which is investigating both the formal and practical aspects of providing automated support for the full range of software engineering activities. 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; refinements can be verified using either testing, proof, or peer review methods. ENCOMPASS provides facilities for specification, prototyping, refinement, testing, mechanical verification, configuration control and project management.

IDEAL[45] is an environment for the specification, validation, refinement and verification of single modules; it is a programming–in–the–small environment for software development using PLEASE. IDEAL is now being extended with a knowledge–based assistant which uses deductive synthesis techniques. During the refinement process, the assistant can give advice on routine design and implementation decisions. The assistant also contains a library of *program schemas*; the programmer can browse this library and instantiate schemas with the aid of the assistant.

In section two of this paper, we describe the development methodology PLEASE, IDEAL and ENCOMPASS are designed to support, and in section three we describe the system architecture. In section four we present an example of software development using the tools, including specification of a com-

ponent, validation of the specification, and a single design transformation. The design transformation consists of a number of atomic transformations, some of which are generated automatically by the knowledge–based assistant. In section five we describe the status of the system and in section six we summarize and draw some conclusions.

2. Incremental Software Development

ENCOMPASS is based on a *traditional* or *waterfall* life–cycle[18], extended to support the use of executable specifications and a development method similar to VDM. In ENCOMPASS, the *requirements definition phase* determines the functions and properties of the software to be produced[18]; software requirements specifications are a combination of natural language and components specified in PLEASE. The fact that a software system satisfies its specifications does not necessarily imply that the system will satisfy the customers' requirements. The *validation phase* attempts to show any implementation which satisfies the specification will also satisfy the customers. If the specification is not valid, then it should be corrected before the development proceeds any further.

To aid in the validation process, the PLEASE components in the specification can be transformed into executable prototypes. These prototypes can be used in interactions with the customers; they can 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 can 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 the refinement phase, the validated specification is incrementally transformed into a program in the implementation language. The refinement process consists of a number of steps. Each step is small and is verified before the next is applied; therefore, errors are detected early and corrected at low cost. Since each step is correct, the final implementation satisfies the original specification. Each refinement step adds more information about the data structures or algorithms used in the system; each step produces a more detailed specification, until an implementation is finally 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[36], technical review[17], and formal verification[31]. Many feel that no one technique alone can ensure the production of correct software; therefore, combinations of techniques are desirable.

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*[26] 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..

In ENCOMPASS, systems can contain modules developed using PLEASE and IDEAL as well as components constructed using conventional techniques. The allows the formal power of PLEASE and the practical power of Ada to be combined in a single project. Parts of a project may use detailed mechanical verification while other, less critical parts may be handled using less expensive techniques. ENCOMPASS provides an integrated set of tools to support such a development method.

3. System Architecture

The tools in ENCOMPASS can be divided into two main groups: support for programming-in-the-large, including the configuration and project management systems; and IDEAL, an environment for programming-in-the-small using PLEASE. IDEAL is an environment for the specification, validation, incremental refinement, and verification of single modules. It contains four tools: TED[21], 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. A set of *symbol tables* represent the PLEASE specifications and Ada implementations being developed.

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. 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[23,31]. The refinement process is a sequence of atomic transformations, which can be grouped into design transformations.

A *design transformation* implements a choice of data structure of algorithm; for example, whether to use a hash table or B–tree to implement a data base. *Atomic transformations* are the smallest distinguishable changes to the system; in ISLET, editor commands are 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 more steps 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 1 shows the architecture of ISLET. The user can interact with ISLET through a simple language–oriented editor similar to[38]. 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 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. 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

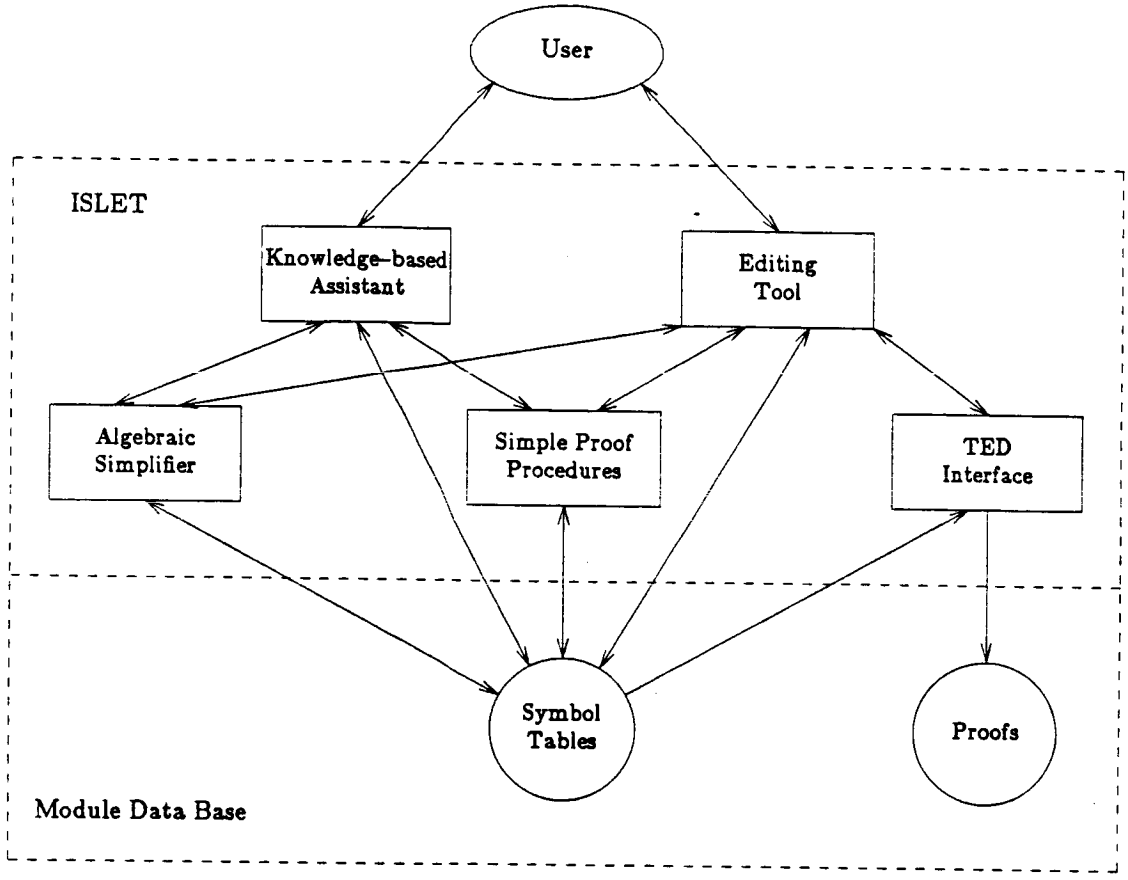


Figure 1. Architecture of ISLET

simple proof tactics used in ISLET are very inexpensive; however, they are not very powerful.

The algebraic simplifier is implemented as a term rewriting system[29,37]; it contains a knowledge-base of rules which are assumed to be convergent. The simple proof procedures rely on a knowledge base which contains information such as: if the formulae F_1 and F_2 are equivalent to the renaming of variables, then the formula $F_1 \supset F_2$ is always true. Other rules implement simple knowledge of equality; for example, if F(X) and X=c are both true then so is F(c). At present, it is difficult to examine, analyze or change the contents of these knowledge bases; for example, algorithms exist to determine if a set of rewrite rules

are convergent, but they are not implemented in ISLET. We are developing tools to correct these deficiencies.

The user can also interact with ISLET through a *knowledge-based assistant* based on deductive synthesis techniques. During the refinement process, the user can ask the assistant for advice on how to implement an undefined construct. The assistant attempts to solve this problem by first searching for values that satisfy the pre– and post–conditions for the construct and then synthesizing Ada code to set these values. The assistant can access the information stored in the symbol table and invoke the algebraic simplifier and simple proof procedures. The assistant also contains a library of *program schemas* which can be instantiated to produce code fragments. The user can browse this library and instantiate schemas with the aid of the assistant; he can then use the instantiated schema in a refinement.

To further clarify our approach and to better illustrate the use of PLEASE and IDEAL, we will consider an example of software development.

4. An Example of Software Development

Assume that a customer needs a component that 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. 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.

4.1. Specifying a Procedure

For example, Figure 2 shows the PLEASE specification of a package, *sort_pkg*, which provides a procedure called *sort*. To increase readability and understandability, the syntax of PLEASE is similar to Anna[33]. 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 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 || L_2 denotes the concatenation of the elements of L_1 and L_2.

```
with natural_list_pkg ; use natural_list_pkg ;

package sort_pkg is

     --: predicate permutation( 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
     --:            permutation(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( permutation(Input,Output) and sorted(Output) ) ;

end sort_pkg ;
```

Figure 2. Specification of *sort* procedure

The *sort* 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 *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 *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. At present, predicates are specified using Horn clauses: a subset of predicate logic which is also the basis for Prolog[12,13]. This approach allows a simple translation from predicate definitions into Prolog procedures; however, there are drawbacks[46].

Although this specification describes a procedure, it is not known if any procedure which satisfies the specification will satisfy the customers. Before the development proceeds further, we should show that the specification is valid. To aid in the validation process, the PLEASE specification can be transformed into an executable prototype which can be used for experimentation and evaluation. Producing a valid specification is a difficult task; the customers may not really have defined what they want, and they may be unable to communicate their desires to the development team. Prototyping and the use of executable specification languages have been suggested as partial solutions to these problems[1]. Providing the customers with prototypes for experimentation and evaluation early in the development process can increase customer/developer communication and enhance the validation and design processes.

4.2. Refining the Specification

After the specification has been validated, it is refined into a concrete implementation. In ENCOMPASS this is performed using ISLET, a language–oriented editor which views the refinement process as the construction of a proof in the Hoare calculus[23,31]. ISLET contains a knowledge–based assistant based on deductive synthesis techniques[16,19,20,24,34]; during the refinement process the programmer can invoke the assistant to request advice or have it perform routine refinments. The assistant also contains a library of *program schemas*; the programmer can search the library and instantiate a schema with the aid of the assistant.

Continuing our example, at the beginning of the refinement phase the body of *sort* consists of an unknown statement sequence with pre–condition *true* and post–condition *permutation(Input,Output) and sorted(Output)*; this can be displayed as follows:

```
begin
        --| true ;
        < unknown >
        --| permutation(Input,Output) and sorted(Output) ;
end ;
```

At this point, the programmer invokes the assistant and requests advice on how to refine the unknown into an implementation. The assistant searches for Ada code which satisfies the pre– and post–conditions for the unknown; if code to handle part or all of the problem can be automatically generated, the complexity of the refinement task performed by the programmer will be reduced.

First, the assistant searches for general or specific values which satisfy the pre– and post–conditions; if specific values are found, simple constructs may be used to implement a solution. In our example, the assistant searches for values of *Input* and *Output* such that the formula

true and permutation(Input,Output) and sorted(Output)

evaluates to true. Using its knowledge of the types and predicates involved, the assistant finds the two solutions:

Input = [], *Output* = []
Input = [X], *Output* = [X] (for any natural number X)

The assistant is aware that these solutions do not cover all possibilities.

Deducing Ada code from these solutions is difficult. First, the assistant realizes that *Input* is an *in* parameter, while *Output* is an *out* parameter; therefore, the value of *Input* can not be set in the procedure, but the value of *Output* should be. Based on a heuristic, the assistant trys to use a sequence of *if–then–else* statments on the value of *Input* to handle the individual cases. However, the expressions used in the statements must be representable in Ada. For example, the formula *Input* = [] can be represented, but *Input* =

[X] can not[1]. The assistant can automatically generate a partial solution to the problem: if *Input* is equal to the empty list then *Output* is set to the empty list and the procedure returns. The assistant modifies the symbol table to effect the necessary refinement; Figure 3 shows the result.

The procedure's actions when called with a non–empty list are still not determined. The assistant can not automatically implement the second branch of the *if–then–else*; the programmer must perform this task himself. However, the assistant has successfully handled the special case of an empty list as input. At this point the programmer can search the assistant's library for a schema which can help him to implement the *else* branch; otherwise, he simply constructs an implementation using language–oriented editor commands.

For example, Figure 4 shows a *divide and conquer* schema. The schema can be instantiated to produce an *annotated code fragment*; in other words, a sequence of statements with a predicate logic formula

```
procedure sort( Input : in natural_list ; Output : out natural_list )  is
      --| where in( true ),
      --|       out( permutation(Input,Output) and sorted(Output) ) ;

begin -- sort
    --| true ;
    if Input = [] then
        --| Input = [] ;
        Output := [] ;
        --| permutation(Input,Output) and sorted(Output) ;
    else
        --| Input /= [] ;
        < unknown >
        --| permutation(Input,Output) and sorted(Output) ;
    end if ;
    --| permutation(Input,Output) and sorted(Output) ;
end SORT ;
```

Figure 3. Automatically generated refinement of *sort* specification

[1]It might be possible to represent this formula as *length(Input) = 1*, but at present the assistant is not intelligent enough to deduce this.

```
generic

        with formula Pre_condition  is <> ;
        with formula Post_condition is <> ;

        N                  : natural ;              --  number of sub-parts

        type Input_type     is private ;
        type Output_type    is private ;
        type In_Part_type   is array (1..N) of Input_type ;
        type Out_Part_type  is array (1..N) of Output_type ;

        Input              : Input_type     ;
        In_Parts           : In_Part_type   ;       --  sub-parts
        Out_Parts          : Out_Part_type  ;       --  sub-solutions

        with predicate Is_divided(
                     Input  : Input_type ;
                     Parts(1) .. Parts(N) : Output_type
              ) is <> ;
        with procedure Conquer(
                     Input  : Input_type ;
                     Output : Output_type
              ) is <> ;

fragment divide_and_conquer is

        --| Pre_condition ;
        < Divide >
        --| Is_divided(Input,In_Parts(1)..In_Parts(N)) ;
        Conquer(In_Parts(1),Out_Parts(1)) ;
        .
        .
        Conquer(In_Parts(N),Out_Parts(N)) ;
        < Combine >
        --| Post_condition ;

end divide_and_conquer ;
```

Figure 4. *Divide_and_conquer* schema

before and after each executable construct. The statements (or unknowns) plus the assertions form an incomplete proof in the Hoare calculus. The schema assumes that the code to be constructed will take a single input and process it to produce a single output; however, the input and output can be very complicated data structures. The schema states the the output will be computed by dividing the input into a

number of sub–parts, finding a solution for each sub–part, and then combining the sub–solutions to generate the output.

The schema takes a number of parameters. *Pre_condition* and *Post_condition* are the pre– and post–conditions for the fragment, while N is the number of sub–parts into which the input is to be divided. *Input_type* and *Output_type* are the types of the input and output respectively, while *Input* is the variable which contains the input. *In_Parts* and *Out_Parts* are the variables to be used to store the sub–parts of the input and the corresponding sub–solutions respectively, while *In_Parts_type* and *Out_Parts_type* are the types of these variables. The predicate *Is_divided* defines the proper division of the input into sub–parts, while the procedure *Conquer* will be used to solve the sub–problems. The notation *Parts(1) .. Parts(N)* is shorthand for all the elements of *Parts*; therefore the number of parameters to *Is_divided* is dependent on the number of sub–parts into which *Input* is divided. Similarly, when instantiated the schema will call *Conquer* once for each sub–part of the input.

The parameters to the schema must be provided when it is instantiated; some are entered by the programmer, others are automatically generated by the assistant. For example, *Pre_condition* and *Post_condition* are inherited from the context of the instantiation. The programmer must specify the variable to be used for *Input*; the system looks up its type and sets *Input_type* automatically. The programmer must declare variables for the sub–parts and the sub–solutions. He gives the variables for *In_Parts* and *Out_parts* when the schema is instantiated; the system looks up their types and sets *In_Parts_type*, *Output_type*, N, and *Out_Parts_type*. The programmer must also specify the predicate to be used for *Is_divided* and the procedure to be used for *Conquer*.

Continuing our example, assume the programmer decides to implement the sort procedure using the quicksort algorithm; he realizes this can be accomplished by replacing the unknown in Figure 3 with an instantiation of *divide_and_conquer*. In the quicksort algorithm, an element is selected and the input list is divided into two sub–lists such that all the items in one list are less than or equal to the element and all the items in the other list are greater than or equal to the element. The *sort* procedure is then recursively called with these sub–lists and the results are combined to form a sorted output. The quicksort algorithm

can therefore be seen as an instantiation of *divide_and_conquer* with the selection and partitioning as *Divide*, the recursive calls to *sort* as *Conquer*, and the merging of the sorted sub-lists as *Combine*

To instantiate the schema and perform the refinement, the programmer must first define decide how *Input* is to be divided, declare variables for the sub-parts, and define a predicate which describes the proper division of *Input*. The programmer declares the variables *Elmt*, *Low* and *High* to hold the partitioning element, the list of all members less than or equal to *Elmt* and the list of all members greater than or equal to *Elmt* respectively. He also declares the following predicate to define the proper division of *Input* into sub-parts:

```
--: predicate is_partition(
--:             List,Low,Elmt,High : in out natural_list ;
--: ) is true if
--: begin
--:     permutation(List,Low || Elmt || High) and
--:     lseqall(Low,hd(Elmt)) and greqall(High,hd(Elmt))
--: end ;
```

The programmer must also define variables to hold the sub-solutions; he declares *Sorted_l*, *Sorted_e* and *Sorted_h* for this purpose.

With the aid of the assistant, he can now replace the unknown with an instantiation of *divide_and_conquer*. The pre- and post-conditions are determined by the context of the instantiation; *Pre_condition* becomes *Input /= []*, while *Post_condition* becomes *permutation(Input,Output) and sorted(Output)*. The programmer specifies that the input variable is *Input* and that *Low*, *[Elmt]*, and *High* are the sub-parts. He also states that *Sorted_l Sorted_e* and *Sorted_h* are to be used for the sub-solutions, that *is_partition(Input,Low,[Elmt],High)* defines when the input is properly divided, and that *sort* should be used to compute the sub-solutions. The rest of the parameters are generated automatically by the assistant.

The assistant first instantiates the schema to the following:

```
--| Input /= [] ;
< Divide >
--| is_partition(Input,Low,[Elmt],High) ;
sort(Low,Sorted_l) ;
sort([Elmt],Sorted_e) ;
sort(High,Sorted_h) ;
< Combine >
--| permutation(Input,Output) and sorted(Output) ;
```

In PLEASE, there is normally an assertion before and after each executable statement in the program; however, at this point the system has not placed assertions around the calls to *sort*. This is because the generation of these assertions may be expensive, and the assistant will try to optimize the instantiated schema before performing the necessary actions. To optimize the schema, the assistant examines each procedure call to determine if it can be replaced with a simpler construct; this involves checking if there is a specific solution to the pre- and post–conditions for the given inputs.

For example, to optimize the call *sort([Elmt],Sorted_e)* the assistant trys to find a value of *Sorted_e* such that the formula *true and permutation([Elmt],Sorted_e) and sorted(Sorted_e)* evaluates to true; it discovers the solution *Sorted_e = [Elmt]*. It now realizes it can replace the call *sort([Elmt],Sorted_e)* with the assignment *Sorted_e := [Elmt]*. However, the assistant can do even better than this. It can access the symbol table and determine that *Sorted_e* is not referenced anywhere else in the code generated so far; it therefore asks the programmer if the variable is really necessary. The programmer replies that it is not, and both the declaration of *Sorted_e* and the call are removed.

The instantiated schema now appears as follows:

```
--| Input /= [] ;
< Divide >
--| is_partition(Input,Low,[Elmt],High) ;
sort(Low,Sorted_l) ;
--| is_partition(Input,Low,[Elmt],High) and
--| permutation(Low,Sorted_l) and sorted(Sorted_l) ;
sort(High,Sorted_h) ;
--| is_partition(Input,Low,[Elmt],High) and
--| permutation(Low,Sorted_l) and sorted(Sorted_l) and
--| permutation(High,Sorted_h) and sorted(Sorted_h) ;
< Combine >
--| permutation(Input,Output) and sorted(Output) ;
```

This fragment is completely annotated; the assertions were automatically generated by the assistant as it instantiated the schema. Some care was necessary to correctly handle the procedure calls. The procedure call rule used in ISLET is a variant of the one developed in[35]. To use it, one needs to define an *invariant* for each call. The invariant states properties that are necessary for proof of the rest of the program, but are independent of the procedure call. The invariant must be true both before and after the call. While in this case the invariants can be generated quite simply, in general the process is very difficult.

For example, to generate the invariant for the call *sort(Low,Sorted_l)*, the assistant first checks if the *out* parameter from *sort*, *Sorted_l*, occurs in the assertion preceding the call. Since it does not, the call can not invalidate this assertion. The assistant then checks the pre–condition for *sort*; since it is *true* the assertion preceding the call can safely be used as the invariant. The post–condition for the procedure, with the proper substitutions performed, can be added to the invariant to produce the assertion after a call. Continuing the example,

is_partition(Input,Low,[Elmt],High) and permutation(Low,Sorted_l) and sorted(Sorted_l)

is the assertion following the call *sort(Low,Sorted_l)*. Similar reasoning gives this assertion as the invariant for the second call to *sort* and produces the assertion between the call and *Combine*.

The programmer now defines three procedures which are called by *sort* and uses them to implement *Divide* and *Combine* from the schema. Figure 5 shows the body of *sort* after the design transformation is complete. *Sort* has the same specification as before, but now implements an abstraction of the quicksort

```
procedure sort( Input : in natural_list ; Output : out natural_list )  is
      --| where in( true ),
      --|       out( permutation(Input,Output) and sorted(Output) ) ;

    Low, High, Sorted_l, Sorted_h : natural_list ; Elmt : natural ;

begin -- sort
    --| true ;
    if Input = [] then
        --| Input = [] ;
        Output := [] ;
        --| permutation(Input,Output) and sorted(Output) ;
    else
        --| Input /= [] ;
        select_Elmt(Input,Elmt) ;
        --| member(Elmt,Input) ;
        partition(Input,Elmt,Low,High) ;
        --| is_partition(Input,Low,[Elmt],High) ;
        sort(Low,Sorted_l) ;
        --| is_partition(Input,Low,[Elmt],High) and
        --|     permutation(Low,Sorted_l) and sorted(Sorted_l) ;
        sort(High,Sorted_h) ;
        --| is_partition(Input,Low,[Elmt],High) and
        --|     permutation(Low,Sorted_l) and sorted(Sorted_l) and
        --|     permutation(High,Sorted_h) and sorted(Sorted_h) ;
        combine(Sorted_l,Elmt,Sorted_h,Output) ;
        --| permutation(Input,Output) and sorted(Output) ;
    end if ;
    --| permutation(Input,Output) and sorted(Output) ;
end SORT ;
```

Figure 5. Abstract implementation of quicksort algorithm

algorithm. 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. Figure 6 shows the definitions of *select_elmt*, *partition*, and *combine*.

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

```
procedure select_elmt(
                List      : in natural_list ;
                Elmt      : out natural
) is separate ;
    --| where in( List /= [] ),
    --|       out( member(Elmt,List) ) ;

procedure partition(
                List      : in natural_list ;
                Elmt      : in natural ;
                Low, High : out natural_list
) is separate ;
    --| where in( member(Elmt,List) ),
    --|       out( is_partition(List,Low,[Elmt],High) ) ;

procedure combine(
                Sorted_l  : in natural_list ;
                Elmt      : in natural ;
                Sorted_h  : in natural_list ;
                List      : out natural_list
) is separate ;
    --| where in( true ),
    --|       out( List = Sorted_l || [Elmt] || Sorted_h ) ;
```

Figure 6. Definitions to support refinement of *sort* specification

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 abstraction. 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.

In ISLET, as a specification is refined into an implementation, a Hoare calculus proof is simultaneously constructed. Many refinements will generate verification conditions in the underlying first–order logic. These refinements are first algebraically simplified and then submitted to a number of simple proof tactics; these methods can eliminate a large percentage of the verification conditions generated at a very

low cost. For example, the design transformation presented in this section produced a number of verification conditions. Only the following could not be proved using the inexpensive techniques:

```
Is_partition(Input,Low,[Elmt],High) and
      permutation(Low,Sorted_l) and Sorted(Sorted_l) and
      permutation(High,Sorted_h) and Sorted(Sorted_h) and
      List = Sorted_l || [Elmt] || Sorted_h =>
      permutation(Input,List) and Sorted(List)
```

This formula can be certified using TED, or by some form of peer review process. When all the verification conditions have been certified, 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.

5. System Status

The SAGA project has been active at the University of Illinois at Urbana–Champaign since the early eighties. The ENCOMPASS environment has been under development since 1984. A prototype implementation of ENCOMPASS has been operational since 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; 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. PLEASE and ENCOMPASS have been used to develop a number of small programs, including specification, prototyping, and mechanical verification. An experimental implementation of the knowledge–based assistant has been written in Prolog. It shares many components with the full implementation of ISLET, but in general uses much simpler data structures. It is not a completely general tool, but does perform the deductions described in this paper. Work is now underway to integrate the assistant into the full implementation.

6. Summary

ENCOMPASS [44,45] is an integrated environment which provides automated support for all aspects of a development method similar to VDM. In ENCOMPASS, software is first specified using a combination of natural language and PLEASE [46–48], a wide spectrum, executable specification and design language. In ENCOMPASS, components specified in PLEASE are incrementally refined into components in Ada. The refinement process consists of a number of steps. Each step is small and is verified before another is applied; therefore, errors can be detected early and corrected at low cost. Since each step is correct, the final components produced by the development satisfy the original specifications. In ENCOMPASS, refinements can be verified using any combination of testing, peer review, or formal methods.

ENCOMPASS is being extended with a knowledge–based assistant which uses deductive synthesis techniques. During the refinement process, the programmer can ask the assistant for advice on how to implement an undefined construct. The assistant attempts to solve this problem by first searching for values that satisfy the pre– and post–conditions for the construct and then synthesising Ada code to set these values. The assistant can access the information stored in the symbol table and invoke an algebraic simplifier and simple proof procedures. The assistant also contains a library of *program schemas* which can be instantiated to produce code fragments. The user can browse this library and instantiate schemas with the aid of the assistant; he can then use the instantiated schema in a refinement. We believe the use of future environments similar to ENCOMPASS will enhance the design, development, validation and verification of software.

7. References

1. *Special Issue on Rapid Prototyping: Working Papers from the ACM SIGSOFT Rapid Prototyping Workshop.* **Software Engineering Notes** (December 1982) vol. 7, no. 5.
2. *Special Issue on the Gandalf Environment.* **Journal of Systems and Software** (May, 1985) vol. 5, no. 2.
3. Balzer, Robert. *A 15 Year Perspective on Automatic Programming.* **IEEE Transactions on Software Engineering** (November 1985) vol. SE–11, no. 11, pp. 1257–1268.
4. Barstow, David R. *Artificial Intelligence and Software Engineering.* **Proceedings of the 9th International Conference on Software Engineering** (1987) pp. 200–211.
5. Bjorner, D. and Cliff B. Jones. **Formal Specification and Software Development.** Prentice–Hall, Englewood Cliffs, N.J., 1982.
6. 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.

7. Buxton, J. N. and V. Stenning. "Requirements for ADA Programming Support Environments, *Stoneman*", U.S. Dept. Defense, 1980.

8. 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.

9. 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.

10. 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.

11. Campbell, R. H., H. Render, R. N. Sum, Jr. and R. B. Terwilliger. "Automating the Software Development Process", Report No. UIUCDCS-R-87-1333, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1987.

12. Chang, Chin-Liang and Richard Char-Tung Lee. **Symbolic Logic and Mechanical Theorem Proving**. Academic Press, New York, 1973.

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

14. 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.

15. Defense, U. S. Dept. **Reference Manual for the ADA Programming Language ANSI/MIL-STD-1815A-1983**. Springer-Verlag, New York, 1983.

16. Dershowitz, Nachum. *Synthetic Programming.* **Artificial Intelligence** (1985) vol. 25, pp. 323–373.

17. Fagan, Michael E. *Advances in Software Inspections.* **IEEE Transactions on Software Engineering** (July 1986) vol. SE-12, no. 7, pp. 744–751.

18. Fairley, Richard. **Software Engineering Concepts**. McGraw-Hill, New York, 1985.

19. Goldberg, Allen T. *Knowledge-Based Programming: A Survey of Program Design and Construction Techniques.* **IEEE Transactions on Software Engineering** (July, 1986) vol. SE-12, no. 7, pp. 752–768.

20. Green, Cordell. *Application of Theorem Proving to Problem Solving.* **Proceedings of the First IJCAI** (1969) pp. 219–239.

21. 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).

22. Henderson, Peter B. (ed.). **Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.**, 1986.

23. Hoare, C. A. R. *An Axiomatic Basis for Computer Programming.* **Communications of the ACM** (October 1969) vol. 12, no. 10, pp. 576–580.

24. Hogger, C. J. *Derivation of Logic Programs.* **Journal of the Association for Computing Machinery** (April 1981) vol. 28, no. 2, pp. 372–392.

25. Jones, Cliff B. *Constructing a Theory of a Data Structure as an Aid to Program Development.* **Acta Informatica** (1979) vol. 11, pp. 119–137.

26. ——. **Software Development: A Rigorous Approach**. Prentice-Hall International, Engelwood Cliffs, N.J., 1980.

27. ——. *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.

28. 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.

29. Knuth, D. E. and P. E. Bendix. *Simple Word Problems in Universal Algebra.* In: **Computational Problems in Abstract Algebra**, J. Leech, ed. Pergamon, New York, 1970, pp. 263–297.

30. 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.

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

32. Lubars, Mitchell D. and Mehdi T. Harandi. *Knowledge-Based Software Design Using Design Schemas.* **Proceedings of the 9th International Conference on Software Engineering** (1987) pp. 253–262.

33. Luckham, David C. and Friedrich W. von Henke. *An Overview of Anna, a Specification Language for Ada.* **IEEE Software** (March, 1985) vol. 2, no. 2, pp. 9–22.

34. Manna, Zohar and Richard Waldinger. *A Deductive Approach to Program Synthesis.* **ACM Transactions on Programming**

Languages and Systems (January 1980) vol. 2, no. 1, pp. 90–121.

35. Martin, Alain J. *A General Proof Rule for Procedures in Predicate Transformer Semantics.* **Acta Informatica** (1983) vol. 20, pp. 301–313.

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

37. 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.

38. 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.

39. Seviora, Rudolf E. *Knowledge-Based Program Debugging Systems.* **IEEE Software** (May 1987) vol. 4, no. 3, pp. 20–32.

40. 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.

41. 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.

42. Sommerville, Ian (ed.). **Software Engineering Environments.** Peter Perigrinus Ltd., 1986.

43. 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.

44. 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.

45. ——. "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 (also to appear in the **Journal of Systems and Software**), 1986.

46. ——. "PLEASE: Executable Specifications for Incremental Software Development", Report No. UIUCDCS–R–86–1295, Dept. of Computer Science, University of Illinois at Urbana–Champaign (also to appear in the **Journal of Systems and Software**), 1986.

47. ——. *PLEASE: Predicate Logic based ExecutAble SpEcifications.* **Proceedings of the 1986 ACM Computer Science Conference** (February, 1986) pp. 349–358.

48. ——. *PLEASE: a Language for Incremental Software Development.* **Proceedings of the 4th International Workshop on Software Specification and Design** (April 1987) pp. 249–256.

49. Waters, Richard C. *The Programmer's Apprentice: A Session with KBEmacs.* **IEEE Transactions on Software Engineering** (November 1985) vol. SE–11, no. 11, pp. 1296–1320.

50. Wegner, Peter. **Programming with Ada: an Introduction by Means of Graduated Examples.** Prentice–Hall, Englewood Cliffs, New Jersey, 1980.

51. 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.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-87-1334	2.	3. Recipient's Accession No.

4. Title and Subtitle

KNOWLEDGE-BASED DEVELOPMENT IN ENCOMPASS

5. Report Date July 1987

6.

7. Author(s)

Robert Barden Terwilliger

8. Performing Organization Rept. No. R-87-1334

9. Performing Organization Name and Address

Department of Computer Science
University of Illinois
1304 W. Springfield, 240 Digital Computer Lab
Urbana, IL 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.

NASA NAG 1-138
1-5-20691

12. Sponsoring Organization Name and Address

NASA Langley Research Center
Hampton, VA 23665

13. Type of Report & Period Covered

14.

15. Supplementary Notes

16. Abstracts

PLEASE is an executable specification language which supports incremental software construction in a manner similar to the Vienna Development Method. PLEASE is part of the ENCOMPASS environment which provides automated support for all aspects of the development process. In ENCOMPASS, PLEASE specifications are incrementally refined into Ada implementations. ENCOMPASS is now being extended with a knowledge-based assistant which uses deductive synthesis techniques. During the refinement process, the assistant can provide advice on design and implementation decisions. The assistant also contains a library of program schemas which can be instantiated during development. In this paper, we give an overview of ENCOMPASS and present an example of development using the environment.

17. Key Words and Document Analysis. 17a. Descriptors

software engineering, artificial intelligence, deductive synthesis, automatic programming, software development environments

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
unlimited	UNCLASSIFIED	25
	20. Security Class (This Page) UNCLASSIFIED	22. Price

FORM NTIS-35 (10-70)

USCOMM-DC 40329-P71

APPENDIX G

PLEASE:
Executable Specifications
for
Incremental Software Development

Robert B. Terwilliger
Roy H. Campbell

PLEASE: A LANGUAGE FOR INCREMENTAL SOFTWARE DEVELOPMENT

(This research is supported by NASA grant NAG 1-138)

Robert B. Terwilliger and Roy H. Campbell

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

ABSTRACT: PLEASE is an executable specification language which supports program development by incremental refinement. In this paper, we present the PLEASE specification for a small library data base. 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 allows a procedure or function to be specified with pre- and post-conditions written using Horn clauses. 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.

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[12,21]. 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[5,6]. PLEASE is a language being developed by the SAGA group to support the specification, prototyping, and incremental development of software components[30,31]. PLEASE is part of the ENCOMPASS environment which provides support for all aspects of the software development process[28,29]. In this paper we briefly describe the development methodology for which PLEASE was created, present an example specification written in PLEASE, and outline the methods used to produce prototypes from PLEASE specifications.

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 that it correctly reflects the users' desires[10]. 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. *Prototyping*[13,20] and the use of executable specification languages[17,32] have been suggested as partial solutions to these problems. Providing the customers with prototypes for experimentation and evaluation early in the development process can 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[10]. Many methodologies for the design and development of implementations have been proposed[1,2,16,24]. 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.

The Vienna Development Method (VDM) supports the top-down development of programs specified in a notation suitable for mathematical verification[3,4,16]. In this method, programs are first written in a language combining elements from conventional programming languages and mathematics; a procedure or function can be specified using *pre-* and *post-conditions* written in predicate logic. 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.

The ENCOMPASS environment is being developed by the SAGA project to provide automated support for all aspects of a software development process similar to VDM. We believe that neither testing[11,23], technical review[9], or formal verification[21] alone can guarantee program correctness; therefore, ENCOMPASS provides a framework in which all three methods can be used as

needed. ENCOMPASS incorporates a number of different tools including: a structure editor which develops programs and their verification conditions simultaneously; a test harness; and a simple configuration control and project management system. ENCOMPASS is in the early stages of development; an initial prototype has been constructed and used to develop small programs.

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 can be specified with pre- and post-conditions and an implementation can be completely annotated. At present, all our implementation efforts involve Ada[1] as the base language. PLEASE specifications can be used in proofs of correctness; they can also be transformed into prototypes which use Prolog[8] to "execute" pre- and post-conditions, and can 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 the ENCOMPASS environment of which it is a part. In section three, we give an example specification in PLEASE and in section four we discuss how an executable prototype is constructed from this specification. In section five we summarize the use of PLEASE specifications in software development.

2. Software Development in ENCOMPASS

ENCOMPASS is based on a *traditional* or *phased*[10] life-cycle model extended to support executable specifications and formal verification. 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, 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 should 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 incremental construction of a proof in the Hoare calculus. In ENCOMPASS, the refinement process is supported by a language oriented tool similar to[26]. As the specification is transformed into an implementation (and the proof is constructed) the syntax and semantics are 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[15]. In our experience, it is too expensive to mechanically certify all of the verification conditions; therefore, the implementor can simply inspect the verification conditions for a refinement and continue. The verification conditions are recorded by ENCOMPASS for use in project monitoring 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 using the ENCOMPASS test harness. 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*[16] 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.

[1]Ada is a trademark of the US Government, Ada Joint Program Office

3. Specifying Software using PLEASE

To better understand the strengths and weaknesses of our approach, we will consider a PLEASE specification of Kemmerer's library example[18]. The example concerns a small library data base which supports the following transactions:

1 - Check out or return a book.
2 - Add or remove a book from the library.
3 - Get all the books by a particular author.
4 - Get the books checked out to a borrower.
5 - Find out who has a book checked out.

Users of the system are divided into *staff* and *non-staff* categories. Only staff users can perform transactions one, two, four or five, except that any one can perform transaction four to find the books they currently have checked out. The data base must satisfy the following integrity constraints:

1 - All books must be available or checked out.
2 - No book may be both available and checked out.
3 - No user may have more than a set number of books checked out at any time.

Figure 1 shows the PLEASE specification of the data structures for the library data base[2]. In PLEASE, several high level data types are added to the base language to enhance the expressivity of specifications. For example, the specification of the library data base uses the PLEASE type *list* to define the type *BOOK_LIST* as *list of BOOK*. 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 Ada is retained and all the elements of a list must have the same type. For example, each element of a *BOOK_LIST* has type *BOOK*. The data base consists of four data structures. *SHELF_LIST* is a list of all the books owned by the library, while *AVAILABLE* is a list of all the books currently available for check out. *CHECKED_OUT* contains a record of each book currently checked out, while *BORROWERS* records the number of books currently checked out by each borrower.

In PLEASE, the integrity constraints on the data base are expressed as an *invariant*; the invariant must be true both before and after any transaction is performed. Figure 2 shows the PLEASE specification of the invariant for the library data base. In PLEASE, *assertions* state logical properties which must be satisfied by the software being specified; each line of assertions begins with the symbol --|. *Virtual program text* defines constructs which are used only in assertions, not in the actual program being constructed; each line of virtual program text begins with the symbol --:. For example, Figure 2 contains an assertion which specifies the invariant for the data base, as well as virtual program text which defines a number of predicates used in the invariant.

[2]To increase readability and understandability, the syntax of PLEASE/Ada is similar to Anna[22].

```
type BOOK is record
        ID     : BOOK_ID ;
        TITLE  : STRNG ;
        AUTHOR : STRNG ;
    end record ;

type BORROWER is record
        NAME            : USER ;
        NUM_CHECKED_OUT : NATURAL ;
    end record ;

type CHECK_OUT_REC is record
        U  : USER ;
        BK : BOOK ;
    end record ;

type BOOK_LIST      is list of BOOK ;
type BORROWER_LIST  is list of BORROWER ;
type CHECK_OUT_LIST is list of CHECK_OUT_REC ;

SHELF_LIST  : BOOK_LIST ;
AVAILABLE   : BOOK_LIST ;
CHECKED_OUT : CHECK_OUT_LIST ;
BORROWERS   : BORROWER_LIST ;
```

Figure 1. Specification of the library data base

In PLEASE, a predicate syntactically resembles a procedure and may contain local type, variable, function or predicate definitions. For example, the predicate *UNDER_LIMIT* states that a borrower is under his limit if the number of books he has checked out is less then the pre-defined *BORROW_LIMIT*. At present, predicates are specified using Horn clauses: a subset of predicate logic which is also the basis for Prolog. This approach allows a simple translation from predicate definitions into Prolog procedures; however, there are drawbacks.

In pure Horn clause programming, there is no way to specify the falsehood of formulae; for example, the fact that *UNDER_LIMIT* is not true if the number of books checked out by the borrower is greater than the borrow limit. 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[25]. At present, the best solution using PLEASE is to define a new predicate which is understood to be the negation of the predicate in question; unfortunately, this relationship can not be recorded in a formal manner. Eventually, we plan to extend PLEASE to support a more powerful logic.

Figure 3 shows the specification of procedures to perform the check in and check out transactions. In

```
--: predicate ALL_AVAILABLE_OR_CHECKED_OUT
--:     is true if
--: begin
--:     permutation( SHELF_LIST,
--:         AVAILABLE || BOOKS_CHECKED_OUT)
--: end ;

--: predicate NONE_AVAILABLE_AND_CHECKED_OUT
--:     is true if
--: begin
--:     member_both(AVAILABLE,BOOKS_CHECKED_OUT)
--:             = []
--: end ;

--: predicate UNDER_LIMIT(
--:     BORROWER : in out BORROWER
--: ) is true if
--: begin
--:     BORROWER.NUM_CHECKED_OUT
--:             <= BORROW_LIMIT
--: end ;

--: predicate ALL_UNDER_LIMIT(
--:     BORROWERS : in out BORROWER_LIST
--: ) is true if
--: begin
--:     BORROWERS = []
--:         or
--:     UNDER_LIMIT(hd(BORROWERS)) and
--:     ALL_UNDER_LIMIT(tl(BORROWERS))
--: end ;

--| where ALL_AVAILABLE_OR_CHECKED_OUT,
--|       NONE_AVAILABLE_AND_CHECKED_OUT,
--|       ALL_UNDER_LIMIT(BORROWERS) ;
```

Figure 2. Invariant for the library data base

PLEASE, a procedure is specified with a pre-condition, which states the conditions which must hold before execution of the procedure begins, and a post-condition, which states the conditions which must hold after execution has terminated. The state before execution begins is denoted by *in(...)*, while the state after execution is complete is denoted by *out(...)*.

For example, the procedure *CHECK_OUT* implements the check out operation. It is called with the identity of the user performing the operation, as well as the identity of the borrower and book in question. The pre-condition for *CHECK_OUT*, states that the user performing the transaction must be a staff member and that the book to be checked out must be available. The post-condition for *CHECK_OUT*, states that the book must be checked out to the borrower, that the book must not be on the available list, that the borrower's record is updated to reflect the new book checked out, and that the borrower is still under the limit for number of books checked out.

Figure 4 shows a number of user-defined predicates which are used in the specification of *CHECK_OUT*. The specification also uses a number of operations on the type *list*. In PLEASE, as in Prolog, a list literal is denoted by a comma separated list of elements surrounded by [and] and the empty list is denoted by []. The functions *hd*, *tl*, and *cons* have their usual meanings and L_1 || L_2 denotes the concatenation of the elements of L_1 and L_2. The function *extract(list,member)* returns a list with all instances of *member* removed.

When a book is checked out of the library, the state of the library data base is changed; the specification of *CHECK_OUT* refers to the state of the data base both

```
procedure CHECK_OUT(
                U  : in USER ;
                B  : in BORROWER ;
                BK : in BOOK ) ;
--| where in( IS_STAFF(U) and
--|           IS_AVAILABLE(BK) ),
--|       out( BOOK_IS_CHECKED_OUT(
--|                    B.NAME,
--|                    BK,
--|                    in(CHECKED_OUT) ) and
--|            AVAILABLE =
--|                extract(in(AVAILABLE),BK) and
--|            BORROWER_IS_UPDATED(
--|                    B,
--|                    1,
--|                    in(BORROWERS)) and
--|            UNDER_LIMIT(BORROWER) ) ;

procedure CHECK_IN(
                U  : in USER ;
                B  : in BORROWER ;
                BK : in BOOK ) ;
--| where in( IS_STAFF(U) and
--|           IS_CHECKED_OUT(BK) ),
--|       out( AVAILABLE =
--|                cons(BK,in(AVAILABLE)) and
--|            BOOK_IS_NOT_CHECKED_OUT(
--|                    B.NAME,
--|                    BK,
--|                    in(CHECKED_OUT)) and
--|            BORROWER_IS_UPDATED(
--|                    B,
--|                    -1,
--|                    in(BORROWERS)) ) ;
```

Figure 3. Procedures to check books in and out

```
--: predicate BOOK_IS_CHECKED_OUT(
--:     B              : in out USER ;
--:     BK             : in out BOOK ;
--:     CHECKED_OUT_O  : in out CHECK_OUT_LIST
--: ) is true if
--:     NEW_RECORD : CHECK_OUT_REC ;
--: begin
--:     NEW_RECORD.U = B and
--:     NEW_RECORD.BK = BK and
--:     CHECKED_OUT =
--:         cons(NEW_RECORD,CHECKED_OUT_O)
--: end ;

--: predicate BOOK_IS_NOT_CHECKED_OUT(
--:     B              : in out USER ;
--:     BK             : in out BOOK ;
--:     CHECKED_OUT_O  : in out CHECK_OUT_LIST
--: ) is true if
--:     NEW_RECORD : CHECK_OUT_REC ;
--: begin
--:     NEW_RECORD.U = B and
--:     NEW_RECORD.BK = BK and
--:     CHECKED_OUT =
--:         extract(CHECKED_OUT_O,NEW_RECORD)
--: end ;

--: predicate BORROWER_IS_UPDATED(
--:     B            : in out BORROWER ;
--:     INC          : in out integer ;
--:     BORROWERS_O  : in out BORROWER_LIST
--: ) is true if
--:     NEW_B : BORROWER ;
--:     BORROWERS_TAIL : BORROWER_LIST ;
--: begin
--:     BORROWERS_TAIL = extract(BORROWERS_O,B)
--:     and NEW_B.NAME = B.NAME and
--:     NEW_B.NUM_CHECKED_OUT =
--:         B.NUM_CHECKED_OUT + INC and
--:     BORROWERS = cons(NEW_B,BORROWERS_TAIL)
--: end ;
```

Figure 4. Predicates to support check in operation

before and after the operation is performed. A predicate is evaluated in a single state; therefore, in order to refer to both the initial and final states, the value of one of the states must be passed as a parameter. For example, in the post-condition for *CHECK_OUT* the initial value of *CHECKED_OUT*, denoted by *in(CHECKED_OUT)*, is used as an argument to the predicate *BOOK_IS_CHECKED_OUT*. This allows the predicate to reference both the initial and final values of the list.

Figure 5 shows the specification of procedures which add books to or remove books from the library. Figure 6

```
procedure ADD_BOOK(
                U  : in USER ;
                BK : in BOOK ) ;
--| where in( IS_STAFF(U) ),
--|       out( AVAILABLE =
--|                cons(BK,in(AVAILABLE)) and
--|            SHELF_LIST =
--|                cons(BK,in(SHELF_LIST)) ) ;

procedure REMOVE_BOOK(
                U  : in USER ;
                BK : in BOOK ) ;
--| where in( IS_STAFF(U) and IS_AVAILABLE(BK) ),
--|       out( AVAILABLE =
--|                extract(in(AVAILABLE),BK) and
--|            SHELF_LIST =
--|                extract(in(SHELF_LIST),BK) ) ;
```

Figure 5. Procedures to add and remove books

shows the PLEASE specification of a function which returns a list of all the books by a particular author. There is no pre-condition specified for this function; therefore, it is assumed to be *true*. For this function, the type declarations for the parameters and the invariant for the data base give all the requirements for the input. The post-condition for the function specifies that any list returned must satisfy the predicate *ALL_BY_AUTHOR*. Figure 7 shows the specification of functions which return the borrower to whom a book is checked out, as well as the list of all books checked out to a borrower.

4. Prototyping the Specification

The specification given in section three can be automatically translated into a prototype written in a combination of Prolog and Ada. The user-defined predicates and pre- and post-conditions for functions and procedures are translated into Prolog, which is executed by an interpreter. When a procedure or function is called, the *in* parameters are converted to the Prolog representation and the call is passed to the interpreter. When the Prolog procedure completes, the *out* parameters are 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 and other implementation level details.

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, the pre-condition for *ADD_BOOK*[3] simply

[3] Figure 5 shows the specification of *ADD_BOOK*.

```
--: predicate ALL_BY_AUTHOR(
--:         SHELF_LIST : in out BOOK_LIST ;
--:         AUTHOR     : in out STRNG ;'
--:         LIST       : in out BOOK_LIST
--: ) is true if
--:     TAIL : BOOK_LIST ;
--: begin
--:     SHELF_LIST = [] and LIST = []
--:         or
--:     hd(SHELF_LIST).AUTHOR = AUTHOR and
--:     ALL_BY_AUTHOR(
--:             tl(SHELF_LIST),AUTHOR,TAIL) and
--:     LIST = cons(hd(SHELF_LIST),TAIL)
--:         or
--:     ALL_BY_AUTHOR(tl(SHELF_LIST),LIST)
--: end ;

function BOOKS_BY_AUTHOR(
                U      : in USER ;
                AUTHOR : in STRNG
) return BOOK_LIST ;
--| where return LIST : BOOK_LIST =>
--|     ALL_BY_AUTHOR(SHELF_LIST,AUTHOR,LIST) ;
```

Figure 6. Function to return all books by an author

checks that the procedure is being invoked by a staff user and that the invariant is satisfied. Executing a post-condition means finding data that satisfies a logical expression. Tor example, the post-condition for *ADD_BOOK* must find a value for *AVAILABLE* that is equal to a list with *BOOK* as the head and the initial value of *AVAILABLE* as the tail.

Although many implementations show significant deviations[27], a "pure" Prolog interpreter can be viewed as a resolution theorem prover for Horn clauses[7,8]. 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.

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[14,19]). 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. 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.

The prototypes produced by this translation process are *partially correct*[21] 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*[21] if it was also guaranteed to terminate normally. The set of all logically

```
--: predicate OUT_TO_BORROWER(
--:         CHECKED_OUT : in out CHECK_OUT_LIST ;
--:         B           : in out USER ;
--:         LIST        : in out BOOK_LIST
--: ) is true if
--:     TAIL : BOOK_LIST ;
--: begin
--:     CHECKED_OUT = [] and LIST = []
--:         or
--:     hd(CHECKED_OUT).U = B and
--:     OUT_TO_BORROWER(
--:             tl(CHECKED_OUT),B,TAIL) and
--:     LIST = cons(hd(CHECKED_OUT).BK,TAIL)
--:         or
--:     OUT_TO_BORROWER(tl(CHECKED_OUT),LIST)
--: end ;

function WHAT_CHECKED_OUT(
                U : in USER ;
                B : in USER
) return BOOK_LIST ;
--| where in( IS_STAFF(U) or U = B ),
--|       return LIST : BOOK_LIST =>
--|           OUT_TO_BORROWER(
--|                   CHECKED_OUT,B,LIST) ;

--: predicate HAS_BOOK(
--:             B  : in out USER ;
--:             BK : in out BOOK
--: ) is true if
--:     TEMP : CHECK_OUT_REC ;
--: begin
--:     member(CHECKED_OUT,TEMP) and
--:     TEMP.BK = BK and TEMP.U = B
--: end ;

function WHO_HAS(
            U  : in USER ;
            BK : in BOOK ;
            B  : in USER
) return USER ;
--| where in( IS_STAFF(U) ),
--|       return B : USER =>
--|           HAS_BOOK(B,BK) or B = NONE ;
```

Figure 7. Functions to examine check out status

valid formulae of predicate logic is not decidable[21]; 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 practice, this is not always a problem. For example, the library specification given in section three produces a Prolog prototype which always terminates in normal use.

5. 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 programs 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 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 prototypes are based on existing Prolog technology, and their performance will improve as the speed of Prolog implementations increases. 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.

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. Bjorner, D. and Cliff B. Jones. **Formal Specification and Software Development.** Prentice-Hall, Englewood Cliffs, N.J., 1982.

4. 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.

5. 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.

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. Chang, Chin-Liang and Richard Char-Tung Lee. **Symbolic Logic and Mechanical Theorem Proving.** Academic Press, New York, 1973.

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

9. Fagan, Michael E. *Advances in Software Inspections.* **IEEE Transactions on Software Engineering** (July 1986) vol. SE-12, no. 7, pp. 744–751.

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. Gehani, Narain and Andrew D. McGettrick (eds.). **Software Specification Techniques.** Addison Wesley, Reading, Massachusetts, 1986.

13. 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.

14. 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.

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

17. 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).

18. 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.

19. Kornfeld, William A. *Equality for Prolog.* **Prpceedings of the International Joint Conference on Artificial Intelligence** (1983).

20. 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.

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

22. Luckham, David C. and Friedrich W. von Henke. *An Overview of Anna, a Specification Language for Ada.* **IEEE Software** (March, 1985) vol. 2, no. 2, pp. 9–22.

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

24. 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.

25. Reiter, Raymond. *On Closed World Data Bases.* In: **Logic and Data Bases**, H. Gallaire and J. Minker, ed. Plenum Press, 1978.

26. 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.

27. Stickel, Mark E. *A Prolog Technology Theorem Prover.* **Proceedings of the International Symposium on Logic Programming** (February 1984) pp. 211–217.

28. 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.

29. ——. "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.

30. ——. "PLEASE: Executable Specifications for Incremental Software Development", Report No. UIUCDCS-R-86-1295, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1986.

31. ——. *PLEASE: Predicate Logic based ExecutAble SpEcifications.* **Proceedings of the 1986 ACM Computer Science Conference** (February, 1986) pp. 349–358.

32. 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.

ORIGINAL PAGE IS
OF POOR QUALITY

APPENDIX H

ENCOMPASS:
an Environment
for
the Incremental Development of Software

Robert B. Terwilliger
Roy H. Campbell

Also to appear in the Journal of Systems and Software.

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 October 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.

It may be difficult to determine if an implementation is correct with respect to a specification. 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 ensure 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 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 pro-

cedure 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 accessed through different "views". The configuration management system stores and structures the components developed and used in a project, as well as providing an access control mechanism. The project management system uses a milestone-based policy implemented using the mechanisms provided by the configuration management system. 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 primitive 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 *milestone* which

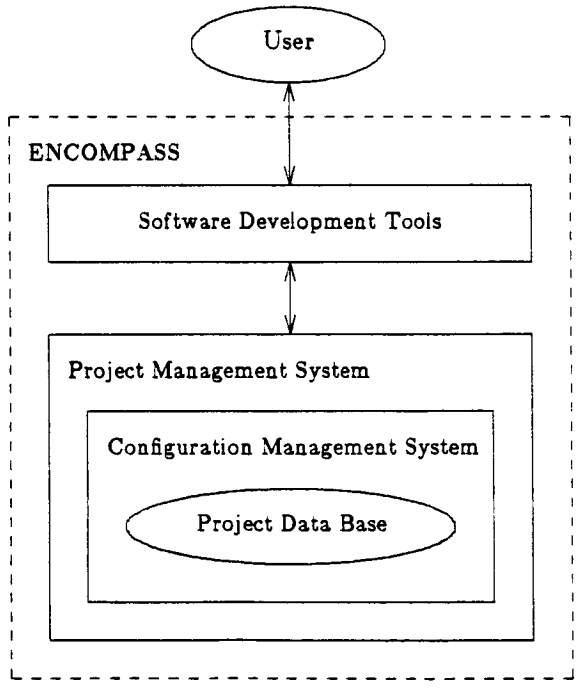


Figure 1. Architecture of ENCOMPASS

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]. A number of different configuration control systems and models of software configurations have been used as aids to configuration management[4,35,40,53,55,67,68,71,78,89,102,114]. In ENCOMPASS, software configurations are modeled using a 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 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*. The products produced by a project are stored in a task called the project library. 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

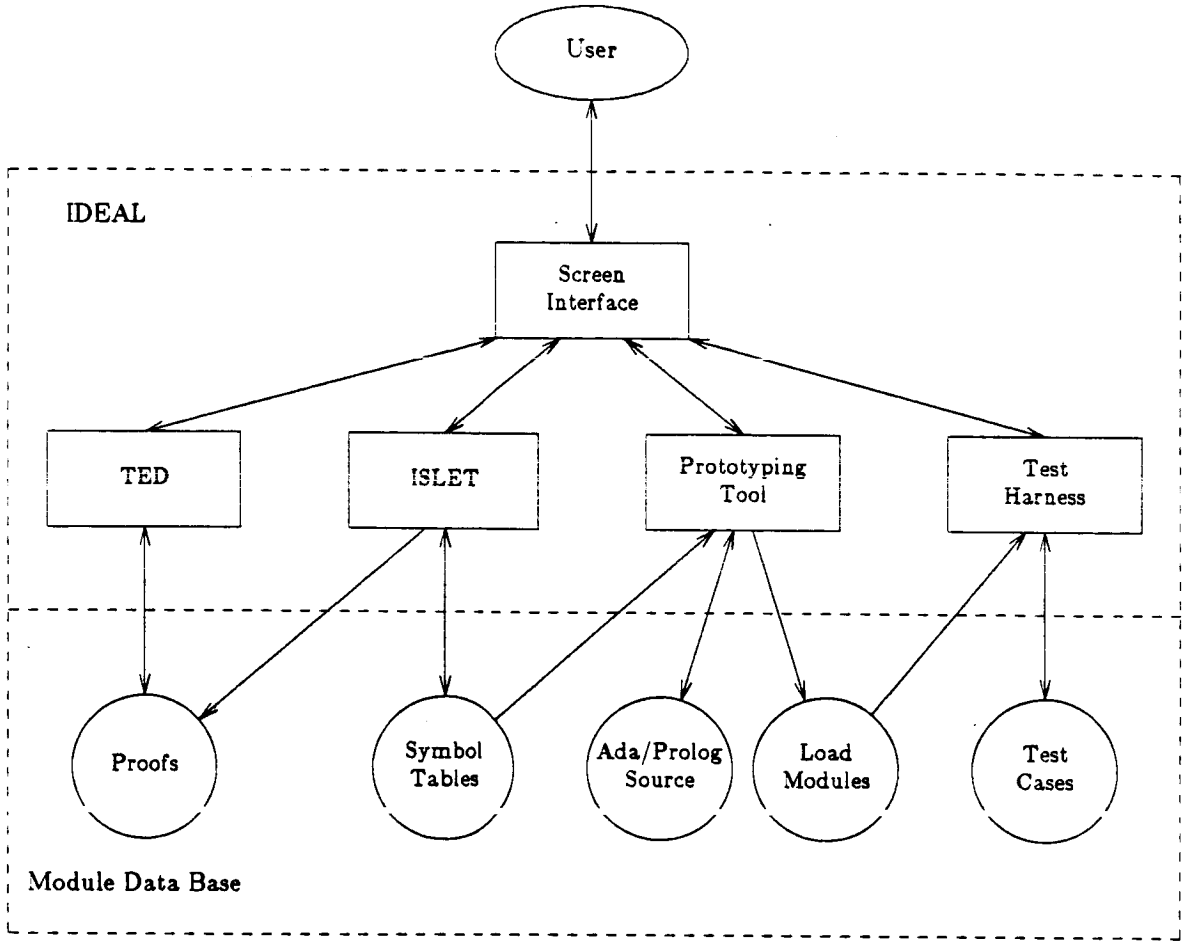


Figure 2. Architecture of IDEAL

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*, which can be grouped into *design transformations*. An atomic transformation cannot be decomposed. 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 expen-

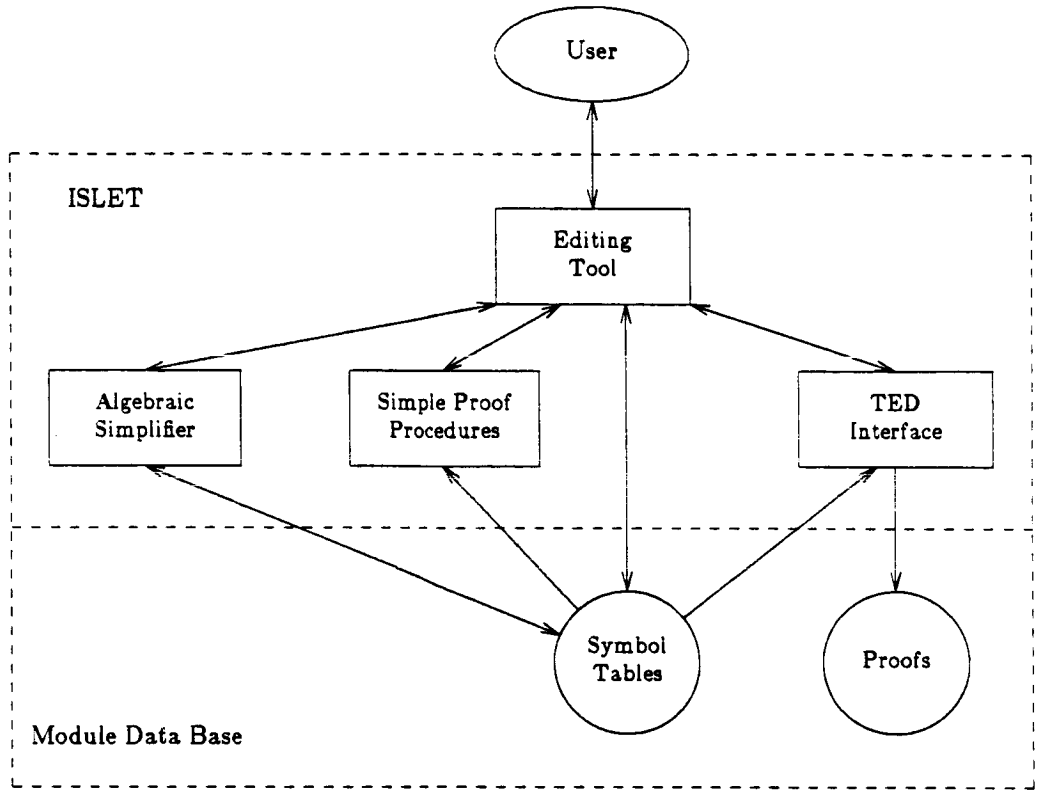


Figure 3. Architecture of ISLET

sive techniques.

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 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 assignment by electronic mail, he creates a project library called *combinatorics* 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 *combinatorics* 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.

[2]The number of k-combinations is equal to $n!/(k!(n-k)!)$

ORIGINAL PAGE IS
OF POOR QUALITY

ORIGINAL PAGE IS
OF POOR QUALITY

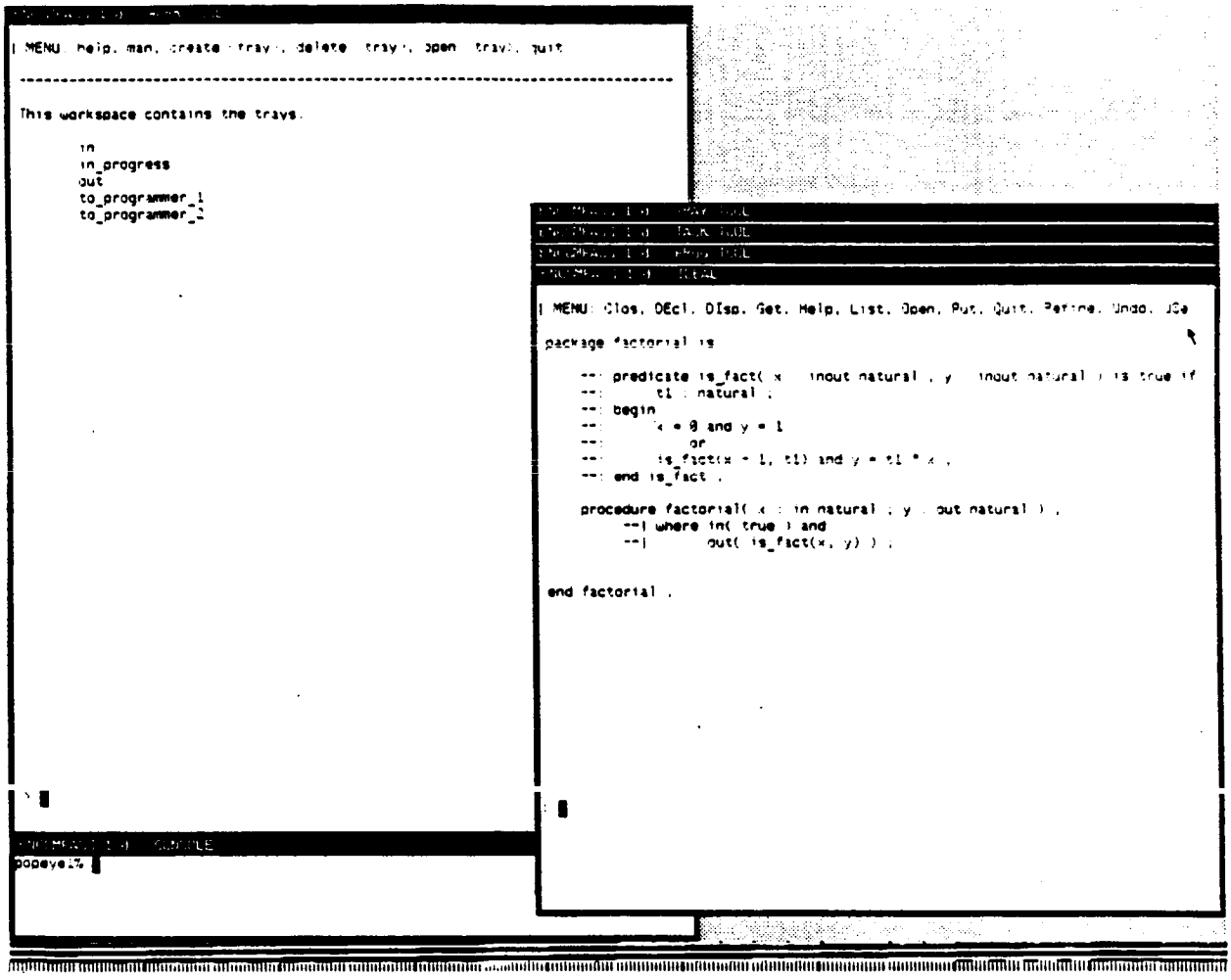


Figure 4. Team Leader's Screen After Specifying *factorial*

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 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.

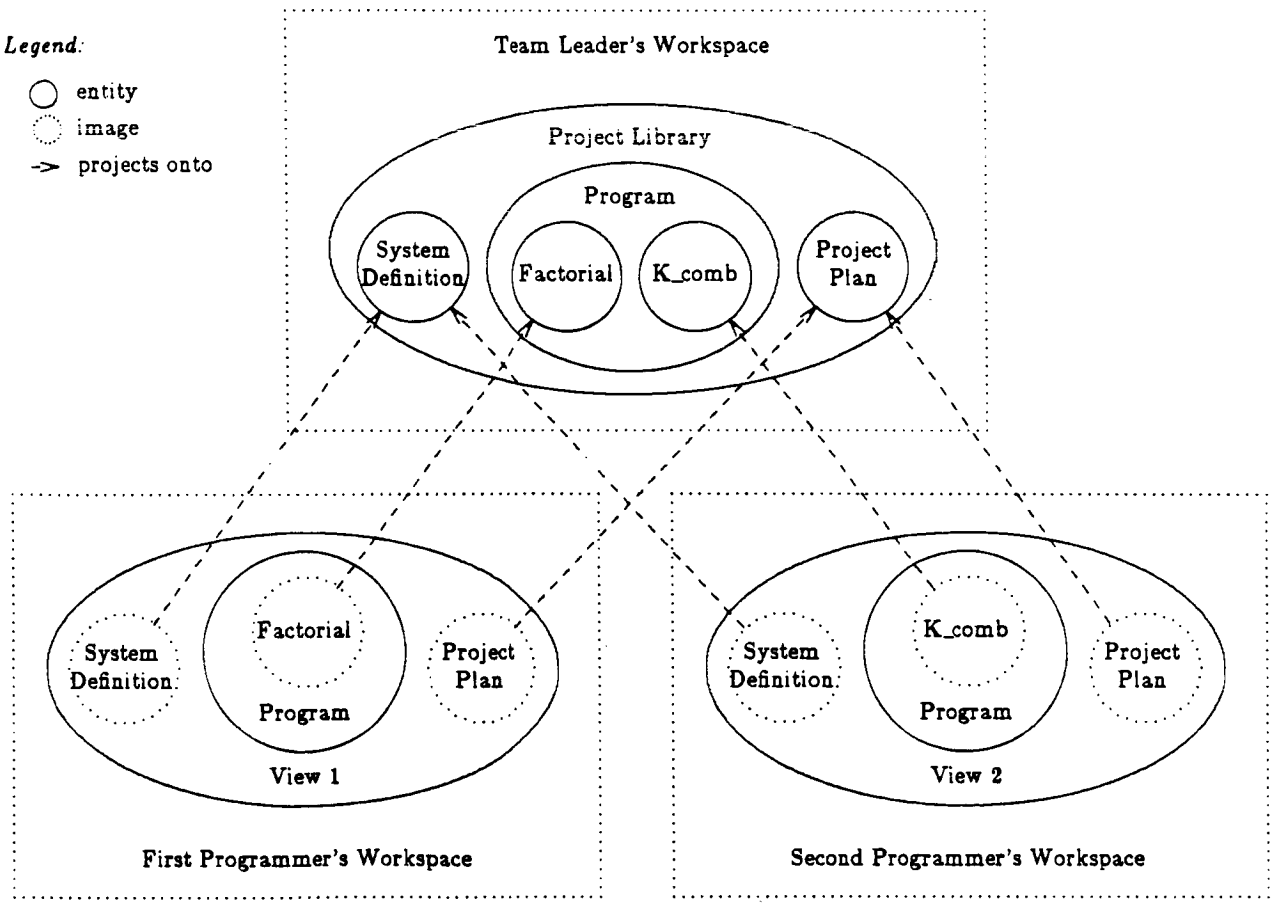


Figure 5. Different Views of the Project Library

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*. 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 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 simple implementations of the project management and configuration control systems, as well as IDEAL. At present, the implementation of ENCOMPASS is not complete. The software capabilities used by the configuration control system and the automatic recognition of completed milestones by the project management system are currently under development. 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. Our experience so far leads us to believe that the complete, mechanical verification of large programs will be prohibitively expensive; however, inexpensive methods can eliminate a large percentage of the verification conditions generated during a development. By eliminating these "trivial" verification conditions, the total number is reduced so that the verification conditions remaining can be more carefully considered by the development personnel.

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 can be accessed through different views. The configuration management system stores and structures the components developed and used in a project, as well as providing an access control mechanism. The project management system uses a milestone-based policy implemented using the access control mechanism provided by the configuration control system.

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. Although the prototype does not implement all the features of the environment, it is substantial enough to demonstrate that the construction of a complete prototype is feasible. We believe the use of future environments similar to 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.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-86-1296	2.	3. Recipient's Accession No.
4. Title and Subtitle ENCOMPASS: an Environment for the Incremental Development of Software			5. Report Date October 1986 6.
7. Author(s) Robert B. Terwilliger and Roy H. Campbell			8. Performing Organization Rept. No. R-86-1296
9. Performing Organization Name and Address Department of Computer Science 1304 W. Springfield 240 Digital Computer Lab 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, Virginia 23665			13. Type of Report & Period Covered 14.

15. Supplementary Notes

16. Abstracts

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.

17. Key Words and Document Analysis. 17a. Descriptors

software engineering, programming environments, program verification, executable specifications

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
unlimited	UNCLASSIFIED	26
	20. Security Class (This Page) UNCLASSIFIED	22. Price

FORM NTIS-35 (10-70)

USCOMM-DC 40329-P71

APPENDIX I

Supporting the Software Development Process with Attributed NLC Graph Grammars

S. Kaplan
S. Goering
R. H. Campbell

Supporting the Software Development Process with Attributed NLC Graph Grammars

Simon M. Kaplan
Steven K. Goering
Roy H. Campbell

Department of Computer Science
University of Illinois
Urbana, Illinois 61801, USA

Abstract

An important research problem in software engineering is to find appropriate formalisms and tools to support the software development process. Efforts to build program development support tools have developed schemes that employ an internal data structure based on trees. Trees are inherently limited and also create long path lengths along which semantic information is forced to flow. We propose ANLC graph grammars as a formalism which can be used to generate tools based on graphs rather than tree structures. This paper defines ANLC grammars, illustrates the use of the formalism with an example and discusses the advantages of the use of graphs rather than trees in building program development tools.

1 Introduction

The development of tools that will assist in all aspects of the software development process is a major software engineering research problem. This paper introduces *attributed NLC* (ANLC) graph grammars, a formalism that we believe is suitable for the specification of tools that support many different aspects of the software development process.

Traditional program development tools that are generated from formal specifications, such as the Cornell Synthesizer Generator [23], Pecan [21], Mentor [7], POE [9] or the SAGA editor [3] all use trees as the internal structure to represent the object (usually a program) that a user is editing. The tree corresponds to the derivation tree of the string representing the program. An important feature of such tools is that they *incrementally* (i.e. after each edit) check the semantic consistency of the structure, thereby providing the user with a programming environment in which errors in his program are reported immediately. This incremental checking is performed by passing attribute values that represent semantic information along arcs of the tree. For example, in a Pascal programming environment the semantic restriction that "identifiers must be declared before use" could be checked by building a symbol table attribute holding all identifiers and then propagating that table attribute up to the root of the tree and then down to each use of an identifier in order to check that the identifier has been declared. After an edit on the tree, the appropriate set of attributes is reevaluated so that the semantic consistency of the program can be reestablished. [22] discusses how this is done. From the viewpoint of the software researcher building such environments, the derivation tree representing a program is far more important than the string itself, and most research into programming environments has concentrated on manipulating this tree, with the string representing the program as a byproduct of this manipulation.

In order to generate such tools automatically, attribute grammars [16] are most commonly used. An attribute grammar is a context-free grammar – which describes the structure of legal derivation trees – augmented with attribution rules – which describe the semantic restrictions on the grammar. Algorithms to incrementally perform attribute updating after an edit in optimal time have been developed by Reps [22].

Many interesting structures – such as module interdependence structures, manager/programmer relations or critical path networks – for which one might want to develop editing systems are *graphs* rather than trees. We believe that in order to automate the software development process, it must

be possible to build environments that manipulate graph structures rather than trees. We must also retain the utility of editing on trees, specifically the ability to perform incremental semantic consistency checking. Further, we do not want to build tools for individual applications by hand, as this would be far too expensive; rather we wish to build a tool generating system that can take as input a specification of a tool and generate a corresponding tool automatically. We plan to use graph grammars as the formalism in which to specify the graph structures. Because we want to permit semantic analysis of programs represented by these structures, we will augment the graph grammars with attribution rules to obtain attributed graph grammars. In order to manage the complexities inherent in graph rewriting systems, we use node-label controlled (NLC) graph grammars [12].

This paper overviews attributed NLC graph grammars and illustrates their use through a simple example. The body of the paper is structured as follows. Section 2 introduces attributed NLC grammars and overviews some results concerning properties of the graphs. Section 3 discusses editing graphs constructed from the grammars, and section 4 illustrates the use of the formalism through an example. Section 5 elaborates on some practical applications of the formalism. The paper concludes with a discussion of related work.

2 Attributed Graph Grammars

This section of the paper introduces the reader to attributed graph grammars. It is broken into three subsections: section 2.1 introduces NLC grammars, section 2.2 extends NLC grammars to attributed NLC (or ANLC) grammars, and section 2.3 summarises some results concerning the graphs constructable from the formalism. We do not give proofs of results in this paper; readers are referred to [14] for proofs as well as more details concerning our attributed graph grammar formalism. First, we informally overview graph grammars.

Graph grammars are similar in structure to string grammars. Productions have a nonterminal symbol as the goal, and graphs (called *body-graphs*) rather

than strings on the right-hand sides of the productions. Each vertex in the body-graph is labelled by a terminal or nonterminal symbol. In the particular graph grammar formalism we are using the rewriting action on a graph is the *replacement* of a vertex labelled with a nonterminal by the bodygraph of a production for which that nonterminal is the goal. We call the graph in which the rewriting is performed the *host* graph. Performing the rewriting action in this manner requires that each production must be augmented by an *embedding rule* that describes how to link the bodygraph into the host graph.

Recall that in traditional programming environments we are more interested in the derivation tree of a program than its representation as a string because we want to use the tree as a vehicle for semantic analysis. In the graph case, however, we use the graph itself for attribute propagation rather than a derivation structure. This creates a counter-intutitive correspondence between the tree case and graph case; in the former we build a derivation tree from a string and then use that tree, but in the latter we build a graph and use that directly. Therefore the derivation tree in the string case corresponds to the graph in the graph case.

Because the rewriting action involves the replacement of a node (vertex) by a body-graph, and the symbol labelling the node controls the set of body-graphs which can be used, this form of graph grammar is called *node-label controlled* (NLC). We should also stress that the graphs constructed from NLC grammars are undirected.

In NLC grammars the embedding rules are restricted in that when a vertex v is rewritten, only vertices that are in the *neighbourhood* of v – those connected to v by a path of unit length – can be connected to the vertices in the bodygraph that replaces v. This restriction greatly reduces the complexity of the embedding process.

2.1 NLC Graph Grammars

This section of the paper formally defines NLC grammars. For more information on NLC grammars see [12]. We begin with some preliminary definitions:

Definition 1 *For any graph G, let V_G denote the vertices in G and E_G the edges of G. Edges are denote $[v, w]$, and are undirected unless otherwise indicated.*

Definition 2 *For any vertex v in a graph G, the neighbourhood of v $\mathcal{N}_v = \{w \mid [v, w] \in E_G\}$*

Because it is unwieldy to maintain a continual distinction between the symbol labelling a vertex in the structure graph and the vertex itself, we will generally refer only to the vertex. By this we will mean either the vertex or the symbol associated with it, depending on the context.

We can now define *NLC* grammars:

Definition 3 *An NLC graph grammar is a tuple NLC$= (N, T, P, Z)$, where*

- *N is a finite set of vertex labels called the nonterminals of the grammar; and*
- *T is a finite set of vertex labels called the terminals of the grammar, such that $T \cap N = \emptyset$;*
- *P is a set of productions, where productions are defined in definition 4 below; and*
- *Z is a unique distinguished nonterminal known as the axiom of the grammar.*

Definition 4 *A production in a NLC graph grammar is defined as: $p : L_p \rightarrow B_p, F_p$ where*

- *p is a unique label;*
- *$L_p \in N$ is called the goal of the production;*
- *B_p is an arbitrary graph (called the bodygraph of the production), where each vertex is labeled by an element of $T \cup N$;*
- *F_p is the embedding rule of the production: a set of symbol pairs $< v, w >$ where $v \in V_{B_p}$ and $w \in T \cup N$. When rewriting L_p by p, for each symbol pair $< v, w >$ an undirected edge is placed between v and each w in $\mathcal{N}_{L_p}$.*

Note that the same symbol may appear several times in a bodygraph; this is resolved using a standard convention of subscripting the symbol with an index value to allow them to be distinguished [22] [26]. For example, multiple occurences of a symbol X would be distinguished as $X\$1$, $X\$2$, *etc.*

Definition 5 *The rewriting (or refinement) of a vertex v in a graph G constructed from a NLC grammar by a production p for which v is the goal is performed in the following steps:*

- *The neighbourhood $\mathcal{N}_v$ is identified.*
- *The vertex v and all edges incident on it are removed from G.*
- *The bodygraph B_p is instantiated to form a* **daughter-graph** *which is inserted into G.*
- *The daughter graph is embedded as described in definition 4.*

We call the graph constructed in this way the *structure graph.*

Note the unpredictable nature of the embedding F_p, in that some subset of the symbol pairs $< v, w >$ in F_p may *not* result in edges being generated because there is no w in $\mathcal{N}_z$ (if z is the vertex being rewritten). Conversely, sometimes there may be multiple vertices w in the neighbourhood and in this case edges to each will be placed.

Requirement 6 *The axiom Z is restricted in that*

- *There must be exactly one production with Z as the goal;*
- *Z may not appear in any bodygraph.*

This requirement is not a restriction in practice as one can always augment the grammar with a distinguished production that satisfies this requirement.

2.2 Attributed NLC Grammars

Attributes are a practical system used by software tools to annotate a structure with semantic information. Each node in the structure has a (possibly empty) set of annotations, called attribute instances. Relations among the annotations are given by attribution rules and induce a directed graph overlaying the structure. Informally, an attributed NLC graph grammar is a NLC graph grammar with attributes attached to each symbol and attribution rules attached to each production.

Definition 7 *An attribute is just an identifier[20]. Each symbol in the grammar has a (possibly empty) set of attributes associated with it; when necessary we will qualify an attribute by its associated symbol to avoid ambiguity. For attribute α and symbol X we write this as $X.\alpha$.*

Definition 8 *An attribute instance is a value. Each vertex in the structure graph has a set of attribute instances associated with it for each attribute associated with the vertex's symbol, there is an attribute instance associated with the vertex.*

Attribute instances are assigned values as the result of the evaluation of attribution rules (definition 9). We will usually ignore the distinction between attribute and attribute instance.

Definition 9 *An attribution rule has the form $X.\alpha \leftarrow f(\overline{\beta})$ where X is a vertex, $\overline{\beta}$ is a vector of attributes such that each attribute is associated with a vertex in $\mathcal{N}_X \cup \{X\}$ and f is a total function.*

Definition 10 *A completing rule is an attribution rule where the function f is a constant function.*

Definition 11 *An attribute embedding pair associated with a production p is a pair $< v, l : g(\overline{\alpha}) >$, where $v \in V_{B_p}$, g is a predicate on a vector of attributes $\overline{\alpha}$ and l is a unique label. An attribute embedding pair set is a finite set of attribute embedding pairs.*

Definition 12 *An attributed NLC graph grammar is a tuple ANLC=* (G, A, R, C, E), *where*

- $G = (N, T, P, Z)$ *is a context-free graph grammar as defined above;*
- $A = \bigcup_{X \in T \cup N} A_X$ *is a finite set of attributes;*
- $R = \bigcup_{p \in P} R_p$ *is a finite set of attribution rules;*
- $C = \bigcup_{p \in P} C_p$ *is a finite set of attribute completing rules.*
- $E = \bigcup_{p \in P} E_p$ *is a finite set of attribute embedding pair sets.*

Productions in ANLC grammars are productions for NLC grammars with each production p augmented by a set R_p of attribution rules, C_p of attribute completing rules and E_p of attribute embedding pair sets.

Because attribution rules are total functions, it is imperative that all their argument attributes can always be assigned a value. In the applications for which we will use ANLC grammars we need to be able to attribute a graph in which there are nonterminals; therefore we use the completing rules to give the attributes of nonterminals "placeholder" values where appropriate. We should note that completing rules are a syntactic artifice; we could achieve the same effect with regular attribution rules but introduce the distinction to clarify specifications.

Definition 13 *The attribute embedding pairs* $< v, l : g(\overline{\alpha}) >$ *in the attribute embedding pair set* E_p *of a production* p *are used to embed the daughter-graph during the rewriting of* L_p *in the following manner:*

- *Identify the set of vertices in* $\mathcal{N}_{L_p}$ *that have all of the attributes in* $\overline{\alpha}$ *and for which the predicate* g *holds.*
- *Place an edge from* v *to each element of that set.*

In this paper g will always be the trivial predicate (meaning that we will only be interested in identifying vertices by virtue of their having the attributes $\overline{\alpha}$ and not in any relation between the attributes. The role of the label l is to

stand in for the symbols qualifying the attributes in $\overline{\alpha}$, because it may not be possible to identify these symbols *a priori*.

In NLC grammars, rewritings of the structure graph can remove and introduce vertices and edges in a fairly arbitrary way (constrained by the structure of the grammar). In order to maintain consistent relations among attributes it is important that an attribute β which is required for the evaluation of some other attribute α not suddenly vanish from the structure graph because the vertex to which it is attached is removed due to a rewriting. The following requirements and theorem guarantee that the rewritings will introduce new vertices and edges sufficient to reestablish the consistency of attribute relations.

Requirement 14 (Law of guaranteed attribute flow through embedding) *Given an (undirected) edge $[v, w]$ in the structure graph such that v qualifies an attribute α, w qualifies an attribute β, and there is a dependency between α and β, any refinement of v must introduce a vertex z such that z qualifies an attribute α. Further, the embedding for the production being used to refine v must place an edge between z and w.*

Requirement 15 (Law of initial embedding) *For all $v \in V_{B_p}$ such that p is the unique production with the axiom Z of the grammar as goal, the neighbourhood $\mathcal{N}_v$ must be such that the refinement of v by any production p' with v as the goal and embedding $E_{p'}$ allows the embedding of the daughter-graph of p' in such a way that requirement 14 is not violated.*

2.3 Properties of Graphs

We give several theorems about the structure of ANLC grammars that obey these requirements. These theorems allow reasoning about the correct behaviour of the structure graph and the attribute flows, and a definition of editing on a structure graph constructed from an ANLC grammar. For proofs and detailed discussion about ANLC grammar analysis and incremental attribute evaluation on graphs constructed from ANLC grammars, see [14].

Theorem 16 *Whenever a vertex v in a structure graph G constructed from an ANLC grammar which obeys requirements 14 and 15 is rewritten using some production p, the neighbourhood $\mathcal{N}_v$ of v will be such that all the edges necessary for meaningful attribute flow will be placed by the embedding E_p.*

From this point on, by ANLC grammar we mean an attributed graph grammar that obeys requirements 14 and 15 (and therefore theorem 16). The theorem leads naturally to these two corollaries:

Corollary 17 *It is a decidable problem to determine whether an ANLC grammar satisfies requirements 14 and 15.*

Corollary 18 *Given a graph G constructed from an ANLC grammar, the vertices in G may be rewritten in any order.*

Definition 19 *By* recursive rewriting *of a vertex v we mean possibly rewriting v to some graph – the instantiation of the bodygraph B_p of some rule p for which v is the goal – and then rewriting recursively the vertices in that graph.*

Definition 20 *For any vertex v in a graph G, let*

- *$\mathcal{N}_v^*$ denote the universe of possible neighbourhoods of v that could arise by rewriting (recursively) the vertices of $\mathcal{N}_v$;*
- *let G_v^* denote the universe of graphs obtainable by all possible recursive rewritings of v;*
- *let $S_v^* = G_v^* - (G - \{v\})$[1], i.e. S_v^* is just the set of subgraphs constructable from v in the recursive rewriting*

Theorem 21 (Theorem of bounded extent) *Given a vertex v in a graph G. Any (recursive) rewriting of v will not introduce edges from the vertices of the daughter graph of v (or any daughter graph recursively introduced into that daughter graph) to any vertex that is not in $\mathcal{N}_v^* \cup S_v^*$.*

[1]Note this is set difference so the "-" does not distribute.

These results indicate that ANLC grammars have a property that comes close to the *Church-Rosser* (or confluence) property, *viz.* that the graph may be rewritten in parallel. Because of neighbourhood changes, we cannot rewrite *any* vertices in parallel. However, given two vertices v and w, where neither vertex is in the neighbourhood of the other, v and w can be rewritten in parallel[2].

3 Editing Structure Graphs

Questions such as "is this graph a member of some class of graphs" are in general very difficult to answer (often NP-complete, at least). A graph editor cannot afford this time overhead so we introduce the concept of a *embedding-tree* that holds a "revisionist" refinement history for a graph, and reduces the cost of identifying a subgraph to a linear time operation. The theorems we will prove about graph structure are vital to the attribute evaluation engine's provably optimal behaviour.

The embedding tree uses multicolored edges to achieve easy identification of neighbourhoods and refinement histories in the graph. When editing a graph, initially:

- The embedding-tree consists of the axiom of the grammar;
- The structure graph consists of the body-graph of the production with axiom as the goal; and
- there is an edge colored *yellow* between each vertex in the graph and the axiom.

When a vertex v in the graph is rewritten by a production p, it is replaced by a daughter-graph: an instantiation of the bodygraph B_p. The algorithm in figure 1 performs this process.

[2]Another way of saying the same thing is to say that rewriting in our system is an atomic action, and all the vertices in the neighbourhood of the vertex being rewritten cannot themselves be rewritten until the rewriting is complete.

- Identify the production p, which has v as a goal, which is being used to refine v.
- Identify the neighbourhood $\mathcal{N}_v$.
- Identify the "parent" of the vertex being refined. This is the vertex in the embedding-tree such that there is a *yellow* edge from it to the vertex v being refined.
- Remove v and all edges incident on it (regardless of their color) from G.
- Place v in the embedding-tree by placing an edge colored *red* from it to its parent.
- Place an edge colored *green* from v (now in the embedding-tree) to each vertex in $\mathcal{N}_v$. (This neighbourhood information is needed if the refinement is ever reversed by a deletion operation; this will be discussed further below).
- Instantiate the bodygraph B_p to create a daughter-graph.
- Place an edge colored *yellow* from v to each vertex in the daughter-graph.
- Embed the daughter-graph using the embedding rule F_p and the embedding strategy defined in definition 5. If any vertex b in $\mathcal{N}_v$ that gets an edge added to it in the embedding process has a *yellow* edge $[b, c]$ incident on it, then place a *green* edge $[a, c]$ from the a that was introduced in the instantiation of B_p to the vertex c (which must be in the embedding-tree).
- Complete the embedding of the daughter-graph into G using the attribute embedding rule: For each attribute embedding pair $< a, l : \overline{\alpha} >$ in E_p, identify the set of vertices $\{b \mid \overline{\alpha} \subseteq A_p(b)\}$ and place an edge to each vertex in this set. If any vertex b in $\mathcal{N}_v$ that gets an edge added to it in the embedding process has a *yellow* edge $[b, c]$ incident on it, then place a *green* edge $[a, c]$ from the a that was introduced in the instantiation of B_p to the vertex c (which must be in the embedding-tree).

Figure 1: Algorithm to refine vertex v by production p

The embedding-tree creates a tree that shows the refinement relation between vertices in the tree. There are red edges between nodes of the tree. Each leaf of the tree has a set of yellow edges incident on it that indicates the location in the structure graph of the unrefined vertices of that leaf's daughter-graph. Green edges link the vertices in the embedding-tree to the neighbourhood they would have had if they had not yet been refined (this information is used to relink them when deleting subgraphs).

A deletion of a subgraph that corresponds to the daughter-graph of some production (regardless of how the vertices therein have been refined) can be acomplished by reference to the embedding-tree in time proportional to the number of vertices in the embedding-tree and graph reachable from the "parent" of the daughter-graph in the embedding-tree. The algorithm in figure 2 accomplishes the deletion.

An immediate implication of the above strategies is:

Lemma 22 *Each vertex in the structure graph has exactly one yellow edge incident on it.*

Lemma 23 *When deleting a subgraph and replacing it by its "parent" vertex, which is found in the embedding-tree, the neighbourhood into which it is embedded (determined by the* green *edges incident on the vertex whilst in the embedding-tree) is the neighbourhood the vertex would have had, had it never been rewritten.*

The *auxiliary* nature of the embedding-tree should be strongly emphasized; the embedding-tree is used to prevent a combinatorial explosion in the time needed for the administrative effort of maintaining the structure graph. Attribute propagations through the embedding-tree are *not* allowed.

4 Example

This section of the paper illustrates the use of the ANLC formalism by considering a simple desk-calculator language. An expression in the language

- Identify, in the embedding-tree, the "parent" of the graph to be deleted. This is the goal symbol of the production, which will be inserted into the structure graph to replace the deleted graph. Call this the replacement vertex (r).
- Identify all the vertices in the structure graph that can be reached via the tree rooted in r using *red* and *yellow* edges.
- Remove all the vertices identified in the previous step from the structure graph, and all edges incident on them. Delete all children of r (and all edges incident on them) from the embedding-tree.
- Identify the new neighbourhood of the replacement vertex $\mathcal{N}_r$. These are all the vertices at the other end of the *green* edges incident on r.
- Remove all *green* edges incident on r.
- Remove r from the embedding-tree, and place it in the structure graph. Place a *yellow* edge from it to the vertex in the embedding-tree that was at the other end of the *red* edge incident on r.
- Embed r into the structure graph by placing an edge from it to every vertex in $\mathcal{N}_r$. If any vertex b in $\mathcal{N}_r$ that gets an edge added to it in the embedding process has a *yellow* edge $[b, c]$ incident on it, then place a *green* edge $[r, c]$, where c must be in the embedding-tree.

Figure 2: Deletion of Subgraph

consists of a single identifier-value binding (via a `let` clause) followed by an expression, for example:

```
let a = 10 in a + 10
```

The grammar has the form shown in figure 3. Issues such as syntactic sugar are ignored in the grammar. Because of space limitations we have allowed only one variable binding clause and one operator (+); however the grammar is easily extensible. Also, because we only ever use the trivial *true* predicate we have omitted the predicate symbols in the embedding clauses. The structure graph for the expression is shown in figure 4(a). A fully attributed version of this figure is found in figure 5. The attribution is not shown on the other examples of structure graphs, but is similar to that shown in this figure. The nonterminals `id` and `int` are *intrinsic*; we do not rewrite them further, rather they have fixed meaning, and pre-supplied attributes *idval* and *intval* respectively. By corollary 18 it makes no difference in which order the `binding` and `expr` nonterminals are refined[3].

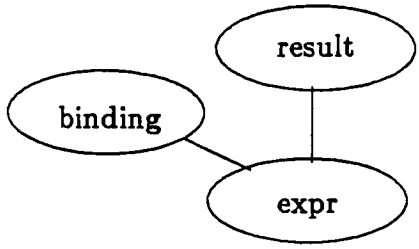
Dependencies among the attributes are induced from the attribution rules to form a directed dependency graph which overlays the structure graph. It is possible to determine attribute values in optimal time, the details are beyond the scope of the paper. The interested reader should refer to [14].

Consider now the effect of replacing the literal `11` in the expression by the variable `a`. The effect on the structure graph is shown in figure 4(b). The `int` `11` is pruned out and replaced by the `id` `a`. The embedding for `a` connects the vertex holding the variable to the vertex defining the *bind* attribute. The *val* attribute associated with this `id` is given the value 10 (determined from the binding pair) which is then propagated to the `plus` vertex and then onward to the `result` vertex.

Now suppose the identifier-value binding is edited to 15. The *val* attributes of both `id` clauses will now be incorrect. These are reevaluated, which in turn triggers the reevaluation of the *val* attribute of the `plus` ver-

[3]The value `nil` which is assigned to attributes through the completing rules is a predefined value of appropriate type.

program →

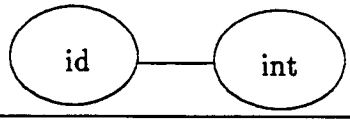


EMBEDDING

COMPLETIONS
binding.bind = <nil, 0>
expr.val = 0

attribution rules
result.value = expr.val

binding →

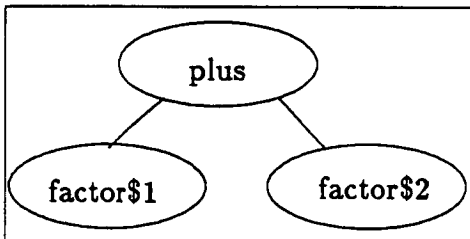


EMBEDDING
(id – m:val)

COMPLETIONS
int.intval = 0
id.idval = nil

attribution rules:
id.bind = <id.idval, int.intval>

expr →

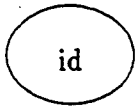


EMBEDDING
(plus – l:value)
(factor$1 – m:bind)
(factor$2 – m:bind)

COMPLETIONS
factor$1.val = 0
factor$2.val = 0

attribution rules
plus.val = factor$1.val + factor$2.val

factor →

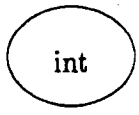


EMBEDDING
(id – l:bind)
(id – m:val)

COMPLETIONS
id.val=0
id.idval=nil

attribution rule
id.val=if id.idval=projection1(id.bind) then projection2(id.bind) else 0

factor →



EMBEDDING
(int – l:bind)
(int – m:val)

COMPLETIONS
int.intval=0

attribution rules
int.val=int.intval

Figure 3: Grammar for Module Language

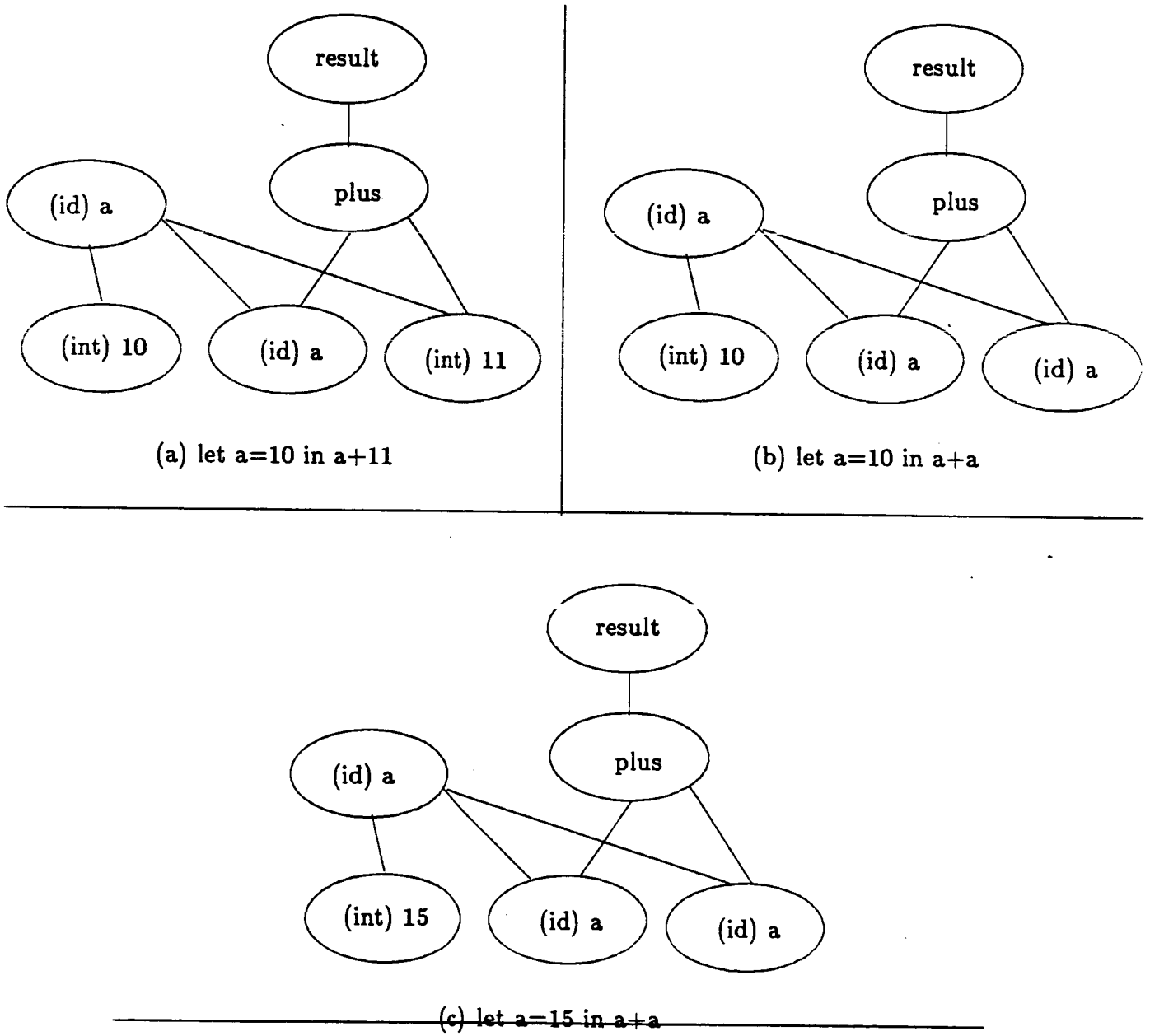


Figure 4: Some Refinements of a Structure Graph

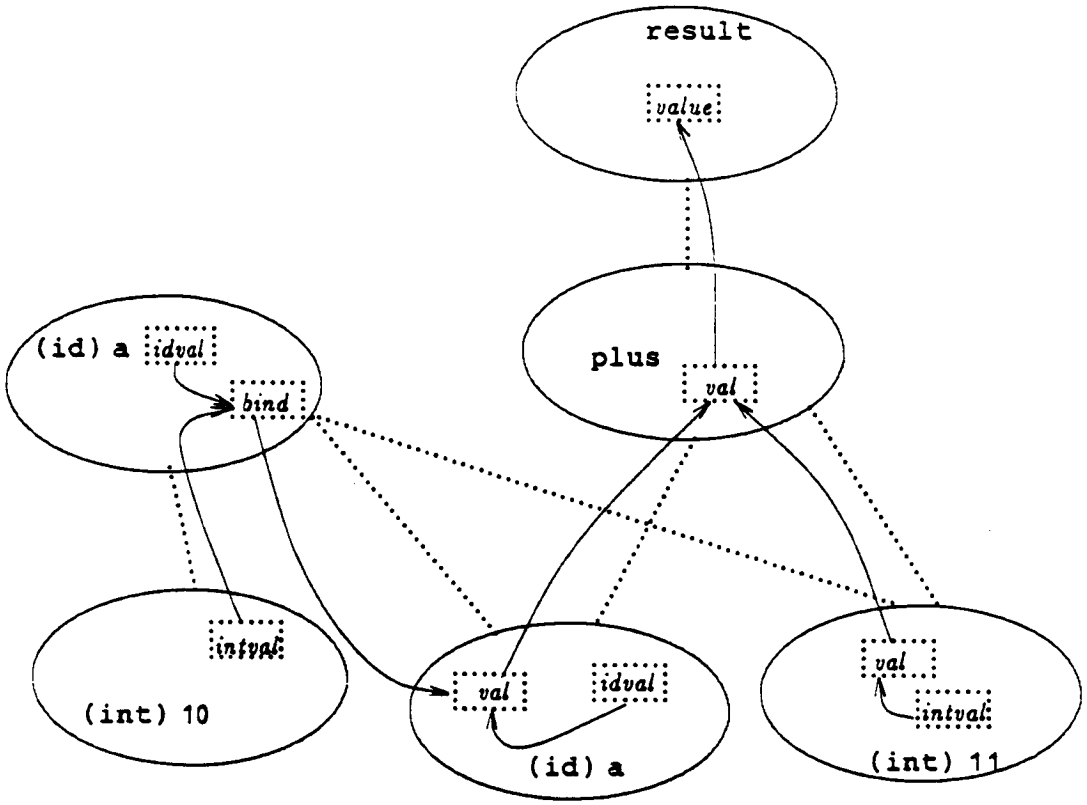


Figure 5: Attributed Structure Graph

tex, which becomes 30. This value is then passed to the `result` vertex. Figure 4(c) shows the structure graph after this edit.

5 Supporting the Software Development Process

It is widely believed that both software production costs and quality can be improved by the use of appropriate tools during the process of designing and building the software. Schofeld argues that editing is a paradigm for building all interfaces to a computer system [24] and many projects including the Cornell Synthesizer, the Gandalf, and MENTOR projects have used such ideas in building program coding tools. It is claimed that such context sensitive editors can help the user by identifying errors early in the coding phases, improving software organization by constraining the software that is built to a set of standard building blocks, and documenting the development process. A problem that underlies the building of such tools is managing the modification of complex information. Much of the information involved in semantic analysis of software can be represented graphically [19]. However, the modification of such graphical structures while maintaining their consistency is difficult. We believe that the ANLC approach can be used to reduce the complexity of building such tools by introducing a notation from which the tools can be manufactured automatically.

For example, an incomplete program contains declarations, control statements, and data structures that are interdependent. As new control statements are added to the program, declarations of procedures or data structures may be added to the program. Existing variables that are used in new control statements can be checked for consistency of use. Executable code could be generated for completed fragments of the program. The interdependencies within such a program involve syntactic and semantic knowledge of the underlying programming language. While various methods of specifying the semantics of programming languages exist (for example, denotational seman-

tics [25]), creating an editor to assist in the construction of a program requires a new notation; one that allows the definition of partially complete programs and allows that definition to be refined.

However, the context sensitive editor is but one example of a range of tools that would enhance software production. Each of these tools depends upon the consistent manipulation of a complex structure under the guidance of a user. Other examples include configuration management, change control, project management and scheduling. Configuration management is concerned with the identification, control, auditing, and accounting of the components produced and used in software development and maintenance [1]. Project management controls the software development process: setting objectives, coordinating development activities, creating schedules, allocating resources, monitoring milestones and reporting on progress [11]. To be effective, the mechanisms and policies involved in configuration and project management must be integrated with the methodology used to develop and maintain the software [4]. In a large project producing many components, automating management should have a major impact on quality and productivity.

Software configurations can be modeled using a variant of the *Entity-relationship model* [6] [5]. The elements of a software project are entities; for example, items (modules, object code, documents), versions and users are all entities. An entity may have attributes; for example, an item has attributes name, type, owner, and identification number. Entities may have relationships between them; for example, version_of, contains, derived_from, manager_of, and user_of are all relations used in CLEMMA [4]. Versions of items may be grouped into aggregate items called views, which have many applications. Views allow abstractions of a project's components to be constructed, manipulated and maintained (for example, a software release.) They may also represent a selected subset of the components, chosen by a functional abstraction of the development process. For example, a test view of a module may contain a specification of the module, the binary object files, test data and results and a test harness. In existing systems, a configuration management system is often implemented as a relational database. However, the relational

database is a monolithic approach to representing the information and, although it provides operations to examine the entities and relations, it does not easily support the consistent and interactive manipulation of software configurations required in a software development environment. Following Scholfeld's paradigm, it is desirable to interact with the configuration control system by editing configurations. The ANLC approach would allow graphs of entities and relations to be manipulated by productions. The analysis supported by the ANLC formalism would ensure the consistency of these edits.

An on-line project management system should track, audit, and control a software development project. The data dependencies in the management system may also be described using an entity relationship model. As a project progresses, the system manipulates the entities and relations to reflect the current structure and status of the project. A task is created and assigned according to the project's work breakdown structure. The management system monitors task dependencies to ensure that inputs are available and sequencing constraints are obeyed. For example, the initiation of a programming task may depend on a successful design document walkthrough; therefore, the system will not allow the programming task to proceed before the walkthrough is complete.

Once again, it is desirable to interact with project management system by editing, for example, tasks, resource allocations, and schedules. Again the project structure is complex but the ANLC approach provides a method of formalizing and specifying the system. Vertices of the ANLC graphs represent development and management tasks as well as resources like specifications, documents, code, test cases and reports. Relations exist between these entities, examples include the work breakdown structure and scheduling dependencies. The various components have attributes like cost and completion deadlines. Productions would allow manipulation of the project structure; permitting creation of new tasks and allocation of resources.

In summary for the potential applications of the ANLC approach, we conclude that an ANLC representation of many aspects of the software de-

velopment process could have a significant advantage over current methods.

6 Related Work

Generating environments from attributed string grammars has already been widely investigated [22] [13] [15]. The principles used to generate environments based on graph grammars are similar to those described in the references. The major advantages of the ANLC approach over the attributed string approaches of the references are that (1) graphs allow short path lengths over which attributes must flow, thereby reducing semantic evaluation and checking overheads; and (2) because the ANLC approach uses graphs it is useful in a larger domain of applications. Further, the optimal attribute evaluation behaviour which is a feature of the cited approaches is retained in the ANLC approach [14].

One project to use graph grammars to build programming environments is IPSEN [17] [8]. In this project, the environments are constructed by hand-translating the graph grammars, rather than generating environments automatically as we propose. Further, the grammars only specify the structures to be constructed in the environments; semantic checking of the kind we perform with attribute grammars is also hand-coded using a style very similar to the action routines approach taken in Gandalf [18]. Because this style of specifying semantics is non-declarative, the writer is forced to provide many different kinds of action routines, for example, one for insertion of a subgraph and another for its deletion. The attributed graph grammar approach, in contrast, allows declarative specification of the semantics, thereby eliminating the need to provide more than one semantics specification.

Attributes have been used with various flavors of graph grammars in the past [2] [10]. Attributes in these cases appear to have purely *local* values (rather than being propagated around the graph) and are used to indicate static information such as how to lay out a subgraph on the screen when displaying graphs. We use attributes in a more general way. Also, we are interested in incremental attribute propagation on structure graphs.

7 Conclusions

A major research problem in software engineering is to find appropriate technologies for the construction of tools that automate and support the software development process. This paper has proposed *attributed NLC* grammars as a formalism that is appropriate for such a purpose. There are several reasons why the ANLC approach is attractive:

- ANLC is a *formalism*. This means that we can reason about specifications of tools.
- Attribute evaluation on graphs constructed from ANLC grammars is time-optimal. This implies that tools generated from ANLC grammars will be efficient.
- Tools may be generated automatically from ANLC specifications, rather than having to be hand-coded.
- Because ANLC grammars generate graphs, rather than trees, it is possible to manipulate and reason about structure graphs with very different semantics, as opposed to traditional work using trees, where the scope of application of the work was inherently limited by the fact that the structures are trees. Examples of structure graphs range from compact program representations with very short path lengths (which reduce attribute propagation overhead) to module interdependence hierarchies and organizational structure charts.
- ANLC grammars have application in many areas outside that of software development environments. Applications that we have investigated include generating schedulers for operating systems and modeling distributed systems. This wide-ranging domain of application of the formalism encourages us in our belief of the importance of graph grammars to computer science in general.

We are currently building a prototype tool generator based on ANLC grammars. The status of this tool generator is that it can take a grammar

as input, analyze it and produce a rudimentary tool as output. We are currently incrementally improving the generated tools; particular emphasis is being given to the hard problem of the user interface to the system.

References

[1] Wayne A. Babich. *Software Configuration Management.* Addison-Wesley, Reading, Mass., 1986.

[2] Horst Bunke. Graph grammars as a generative tool in image understanding. In Hartmut Ehrig, Manfred Nagl, and Grzegorz Rozenberg, editors, *Proceedings of the second International Workshop on Graph Grammars and their Application to Computer Science, LNCS 153*, pages 8–19, Springer-Verlag, 1982.

[3] Roy H. Campbell and Peter A. Kirslis. The saga project: a system for software development. In *Proceedings of the ACM SIGPLAN/SIGSOFT Symposium on Practical Software Development Environments*, pages 73–80, Pittsburgh, Pa, May 1984.

[4] Roy H. Campbell, Hal Render, Robert N. Sum Jr., and Robert Terwilliger. *Automating the Software Development Process.* Technical Report UIUCDCS-R-87-1333, University of Illinois at Urbana-Champaign, Department of Computer Science, May 1987.

[5] Roy H. Campbell and Robert B. Terwilliger. The saga approach to automated project management. In Lynn R. Carter, editor, *International Workshop on Advanced Programming Environments*, pages 145–159, Springer-Verlag, New York, 1986.

[6] Peter Pin-Shan Chen. The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[7] Veronique Donzeau-Gouge, Gilles Kahn, Bernard Lang, and Bertrand Melese. Documents structure and modularity in mentor. In *Proceedings*

of the ACM SIGPLAN/SIGSOFT Symposium on Practical Software Development Environments, pages 141–148, Pittsburgh, Pa, May 1984.

[8] Gregor Engels and Wilhelm Schafer. Graph grammar engineering: a method used for the development of an integrated programming environment. In Hartmut Ehrig, Christiane Floyd, Maurice Nivat, and James Thatcher, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT), LNCS 186*, pages 179–193, Springer-Verlag, 1985.

[9] Charles N. Fisher, Anil Pal, Daniel L Stock, Gregory F. Johnson, and Jon Mauney. The poe language-based editor project. In *Proceedings of the ACM SIGPLAN/SIGSOFT Symposium on Practical Software Development Environments*, pages 21–29, Pittsburgh, Pa, May 1984.

[10] Herbert Gottler. Attributed graph grammars for graphics. In Hartmut Ehrig, Manfred Nagl, and Grzegorz Rozenberg, editors, *Proceedings of the second International Workshop on Graph Grammars and their Application to Computer Science, LNCS 153*, pages 130–142, Springer-Verlag, 1982.

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

[12] Dirk Janssens and Grzegorz Rozenberg. Graph grammars with node-label control and rewriting. In Hartmut Ehrig, Manfred Nagl, and Grzegorz Rozenberg, editors, *Proceedings of the second International Workshop on Graph Grammars and their Application to Computer Science, LNCS 153*, pages 186–205, Springer-Verlag, 1982.

[13] Gregory F. Johnson and Charles N. Fischer. Non-syntactic attribute flow in language based editors. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages (POPL)*, January 1982.

[14] Simon M. Kaplan. *Incremental Attribute Evaluation on Graphs (Revised Version).* Technical Report UIUC-DCS-86-1309, University of Illinois at Urbana-Champaign, December 1986.

[15] Simon M. Kaplan and Gail E. Kaiser. Incremental attribute evaluation in distributed language- based environments. In *5th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 121–130, Calgary, Alberta, Canada, August 1986.

[16] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968.

[17] M. Nagl, G. Engels, R. Gall, and W. Schafer. Software specification by graph grammars. In Hartmut Ehrig, Manfred Nagl, and Grzegorz Rozenberg, editors, *Proceedings of the second International Workshop on Graph Grammars and their Application to Computer Science, LNCS 153*, pages 267–287, Springer-Verlag, 1982.

[18] David Notkin. The gandalf project. *Journal of Systems and Software*, 5(2):91–106, May 1985.

[19] Harold L. Ossher. Grids: a new program structuring mechanism based on layered graphs. In *Proceedings of the 11th ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 11–22, January 1983.

[20] Frank G. Pagan. *Formal Specification of Programming Languages.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[21] Steven P. Reiss. Graphical program development with pecan program development systems. In *Proceedings of the ACM SIGPLAN/SIGSOFT Symposium on Practical Software Development Environments*, pages 30–41, Pittsburgh, Pa, May 1984.

[22] Thomas Reps. *Generating Language-Based Environments.* MIT Press, 1984.

[23] Thomas Reps and Tim Teitelbaum. The synthesizer generator. In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA, April 1984.

[24] J. Schofield. *Editing as a Paradigm for User Interaction - A Thesis Proposal.* Technical Report 81-11-01, Dept. of Computer Science, FR-35, University of Washington, November 1981.

[25] Joseph Stoy. *Denotational Semantics.* MIT Press, 1977.

[26] William M. Waite and Gerhard Goos. *Compiler Construction.* Springer-Verlag, New York, 1984.