ENCOMPASS: A SAGA BASED ENVIRONMENT FOR THE COMPOSITION OF PROGRAMS AND SPECIFICATIONS

Robert B. Terwilliger

Roy H. Campbell

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana,Illinois

July, 1985

Submitted to the 19th Annual

Hawaii International Conference on System Sciences

July, 1985

ENCOMPASS: a SAGA Based Environment for the Composition of Programs and Specifications

Robert B. Terwilliger
Roy H. Campbell

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

Submitted to the 19th Annual
Hawaii International Conference on System Sciences

ENCOMPASS: a SAGA Based Environment for the Composition of Programs and Specifications

Robert B. Terwilliger
Roy H. Campbell

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

Abstract

ENCOMPASS is an example integrated software engineering environment being constructed by the SAGA project. ENCOMPASS supports the specification, design, construction and maintenance of efficient, validated, and verified programs in a modular programming language. In this paper, we present the life-cycle paradigm, schema of software configurations, and hierarchical library structure used by ENCOMPASS. In ENCOMPASS, the software life-cycle is viewed as a sequence of developments, each of which reuses components from the previous ones. Each development proceeds through the phases planning, requirements definition, validation, design, implementation, and system integration. The components in a software system are modeled as *entities* which have *relationships* between them. An entity may have different *versions* and different *views* of the same project are allowed. The simple entities supported by ENCOMPASS may be combined into *modules* which may be collected into *projects*. ENCOMPASS supports multiple programmers and projects using a hierarchical library system containing a *workspace* for each programmer; a *project library* for each project, and a *global library* common to all projects. A prototype implementation of ENCOMPASS is being constructed on the UNIX[1] operating system using an existing revision control system and many tools developed by the SAGA project.

1. Introduction

It is widely acknowledged that software is both difficult and expensive to produce and maintain. 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[15,34,42,43]. The SAGA project is investigating both the formal and practical aspects of providing automated support for the full range of software engineering activities[2,5,7,21]. A SAGA-based software tool or environment is created by combining *standard components* which are generated by

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

[1]UNIX is a trademark of Bell Laboratories

meta-tools. ENCOMPASS is an example software engineering environment being developed by the SAGA group. In this paper we describe the life-cycle paradigm, schema of software configurations, and hierarchical library structure used by ENCOMPASS.

It has been suggested that *modular programming*[35] and the *top-down development* of programs[48] can help reduce the difficulty of program development 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[9,26,28] and environments to support modular programming have been designed[4] and constructed[41,50]. Methods to support the top-down development of programs have been devised[19,36] and put into use[37].

A *life-cycle model* describes the sequence of distinct stages through which a software product passes during its life-time[11]. There is no single, universally accepted model of the software life-cycle[3,51]. The stages of the life-cycle generate *software components* such as specifications of various forms, code written in programming languages, and many types of documentation. *Configuration management* is concerned with the identification, control, auditing, and accounting of components produced and used in software development and maintenance[1]. Configuration control systems[10,23,38] and models of software configurations[24,33] have been suggested as aids to configuration management. Life-cycle and configuration models that are understood and accepted by everyone involved can enhance communication, aid project management and increase product quality.

ENCOMPASS is a software engineering environment concerned with the construction and maintenance of efficient, validated, and verified programs in a modular programming language. The software life-cycle is viewed as a sequence of developments, each of which reuses components from the previous ones. Each development passes through the stages planning, requirements definition, validation, design, implementation, and system integration. An executable specification language is used to produce pro-

grams for experimentation, evaluation, and validation as early as possible in the development process. The components in a software project are modeled as entities which have relationships between them, and different views of the same project are allowed. The simple entities supported by ENCOMPASS may be combined into modules which may be collected into projects. ENCOMPASS supports multiple programmers and projects using a hierarchical library system containing a workspace for each programmer; a project library for each project, and a global library common to all projects.

In section two, we describe the life-cycle paradigm on which ENCOMPASS is based and in section three, we present its schema of software configurations. In section four, we describe the hierarchical library structure used by ENCOMPASS and in section five, we discuss a prototype implementation of ENCOMPASS which is being constructed on the UNIX operating system. In section six, we describe our plans for extending ENCOMPASS and in section seven, we summarize and draw some conclusions from our experience.

2. The Software Life-Cycle

ENCOMPASS is used by a programming team to construct and/or maintain a *system*, which may contain *programs* written in different languages. Modular programming techniques may be supported directly by the languages[9,26,28] or by coding conventions and/or a pre-processor[46]. A system must usually satisfy both *performance constraints*, such as speed or storage requirements, and *design constraints*, such as proper modularization and documentation. *Verification* guarantees that software components are correct and complete relative to each other, while *validation* shows that a system performs the functions desired by the *customers*[11].

It has been suggested that the *reuse* of software can significantly reduce the cost of program development[17], and systems which contain libraries of previously coded modules and/or a number of standard designs for program have been proposed[25,29]. In ENCOMPASS, any software component or group of components can be saved for later reuse in a central library. The library supports a number of concurrent projects, both accepting and supplying components for reuse in all phases of the life-cycle. ENCOMPASS supports the reuse of all the components produced in the development of a system. In

addition to source and object code, documentation, formal specifications, proofs of correctness, test data and test results can all be stored in the central library for reuse.

Figure 1 shows the proposed software life-cycle which consists of a sequence of developments. These developments might produce a series of prototypes which are used in the production of a system. In this case, each prototype would be evaluated and the results incorporated in the next stage of production. During the next stage, all the materials from the development of the prototype would be available for reuse. A sequence of developments might also produce a *family* of systems for use in different operating environments or with different optional features. In this case, all the materials from the development of the family would be available for reuse in the development of new family members. A sequence of developments might also represent what is traditionally called the *maintenance* phase of a development. A system, which has been constructed and installed, may have to be modified, corrected, or enhanced. In ENCOMPASS, this is seen as a new development, but with all the products of the previous development available for reuse. In this way ENCOMPASS supports both development and maintenance with the same methods and tools.

ENCOMPASS supports program development by successive refinement using the Vienna Development Method[19,37]. In this method, programs are first written in a language combining elements from conventional programming languages and mathematics. These *abstract programs* are then incrementally refined into programs in an implementation language. The refinements are performed one at a time and each is verified before another is applied. Therefore, the final program produced by the development and the original abstract program are equivalent. In ENCOMPASS, abstract programs may be written in the executable specification language PLEASE[44], which is an extension to the language Path Pascal[6] allowing routines and data types to be specified using predicate logic. A procedure or function may be specified using *pre* and *post-conditions* and an *invariant* for a data type may be specified.

It has been proposed that software development may be viewed as a sequence of transformations between specifications written at different *linguistic levels*[27] and systems to support similar development methodologies have been constructed[32]. ENCOMPASS supports this view of software develop-

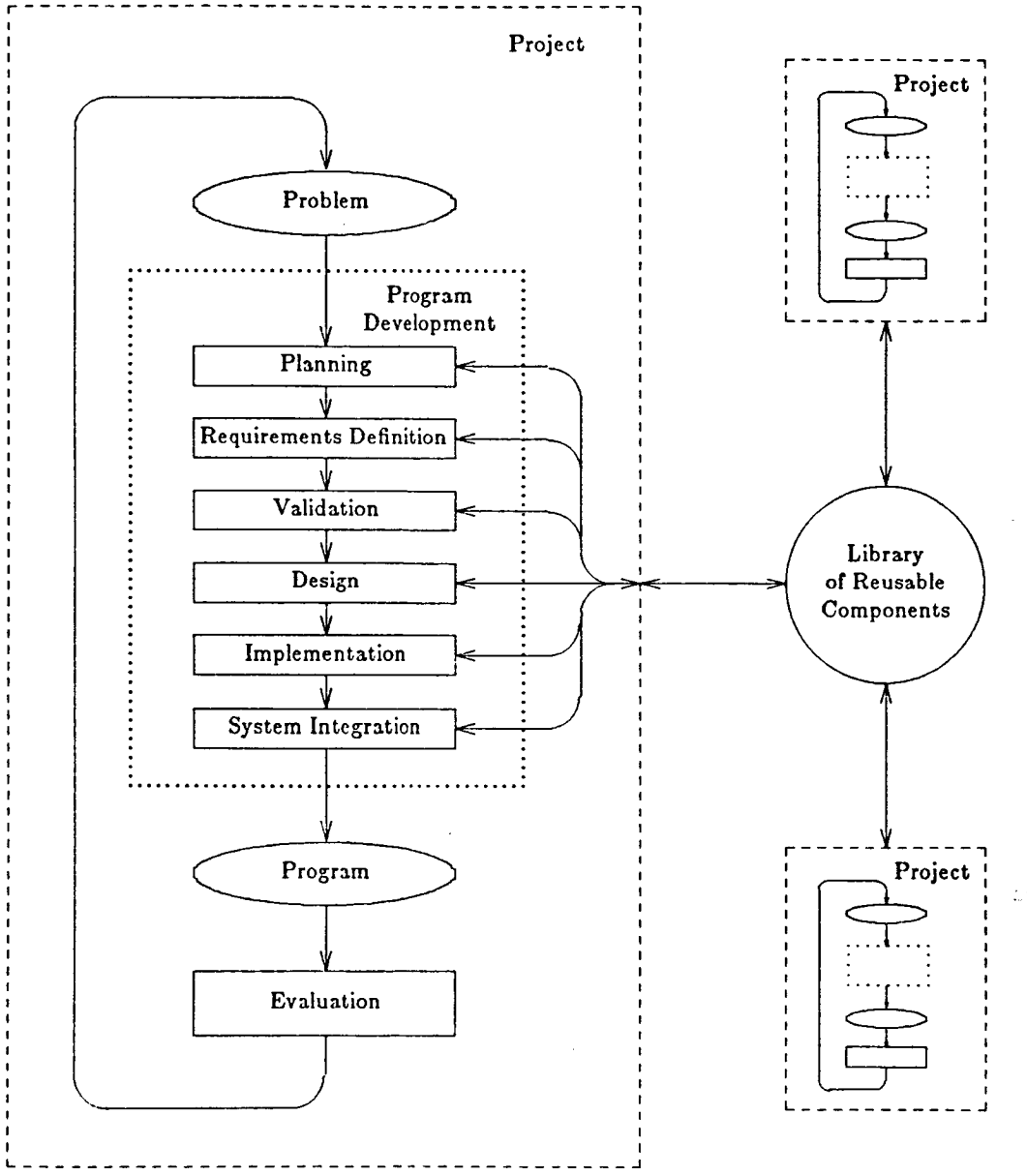

Figure 1. The Software Life-Cycle

ment by allowing abstract, predicate logic based definitions of data types or routines to be transformed into successively more concrete realizations. The use of executable specifications allows two or more linguistic levels to be run in parallel and compared for the purposes of verification or debugging.

The development steps in ENCOMPASS may be much smaller than in the traditional software life-cycle. For example, a system might go through a very large number of prototypes before delivery to the customers. Developments may also be composed hierarchically. For example, if a system is very large and complex, the production of an executable specification for the system may in itself be a complete development. If the system is composed of several major components, the production of each component might also be a complete development. By dividing the life-cycle into small steps using the mechanisms of sequential and 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. Software Development

Each development passes through the phases: planning, in which the problem is defined and it is determined if a computer solution is feasible and cost effective; requirements definition, which produces a high-level specification of the system to be produced; validation, which determines that the system described by the specification will satisfy the customers; design, in which the basic structure of the system is described; implementation, in which components of the system are constructed; and system integration, in which the components are integrated into a complete system, acceptance tests are performed, and the product is delivered. This structure is Fairley's *phased* life-cycle model[11], extended to support the Vienna Development Method and the use of an executable specification language.

The Vienna Development Method can aid in the production of correct software by allowing a system to be produced by a sequence of refinements, each of which is shown correct before proceeding further in the development. The use of an executable specification language allows each refinement to be verified by testing techniques as well as by mathematical proof. Abstract programs can also enhance the design phase by allowing experiments to be performed which influence design decisions, and the validation phase by allowing the customers to evaluate a running system early in the development process. We believe the the early validation will aid in lowering the cost of correcting errors made during requirements definition. Each phase of the development produces certain components which may be used and/or updated during the rest of the life-cycle[11].

2.1.1. Planning

In the planning phase the problem to be solved is defined and it is determined if a computer solution is feasible and cost effective[11]. Alternative solutions to the problem are considered and compared for cost effectiveness and preliminary plans and schedules for the project are created. In ENCOMPASS, these processes can be enhanced by the use of abstract programs as prototypes for experimentation and evaluation. This phase produces the two natural language documents[11]: the *system definition*, and the preliminary *project plan*. The *system definition* describes the original problem, gives justifications for the proposed computer system as a solution, and contains *acceptance criteria* which describe the standards and procedures to be used for evaluating the system. The *project plan* describes the milestones and specific products to be produced as well as the organizational structure to be used by the project. Once the problem has been defined and it is clear that a computer solution will be cost effective, a more detailed description of the system requirements is needed.

2.1.2. Requirements Definition

Requirements definition determines the functions and qualities of the software to be produced by the development[11]. This phase 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. This phase produces[11] the *software requirement specification*, and preliminary versions of the *users manual*, and the *software verification plan*. The *software requirement specification* precisely describes each requirement of of the software to be produced. It contains a functional specification of the system, descriptions of the external interfaces, and performance and design constraints. The *users manual* is documentation for the customers. It contains an overview of the system, tutorials on various system functions, and detailed users documentation on all system commands. The *software verification plan* describes the methods to be used in verifying that the system produced by the development satisfies the software requirement specification. Although the requirement 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.

2.1.3. Validation

The validation phase attempts to show that a 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 design, implementation, and system integration. In the validation phase, the developers interact with the customers and the *system validation summary* is produced. This document describes the customers evaluation of the software requirements specification. It lists any problems encountered and the solutions agreed upon.

Traditionally, producing a correct 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*[14,22], and the use of executable specification languages[20,31,52] have been suggested as partial solutions to this problems. Providing the customers with prototypes for experimentation and evaluation may increase customer/developer communication and enhance the validation process.

In ENCOMPASS, we extend Fairley's model to include software requirements specifications which are a combination of natural language and abstract programs written in PLEASE. PLEASE programs are prototypes which can be used for experimentation and evaluation, and a formal specification of a part of the system to be produced which can be used throughout the rest of the life-cycle. By providing executable programs early in the development process, errors in the requirements specification may be discovered and corrected before the internal structure of the system has been defined.

2.1.4. Design

In the design phase, the structure of the software system is defined[11]. The components of the system; their interfaces; the flow of control and data between components; and global data abstractions, structures and formats are all designed and documented. This phase produces the *software design specification*[11], which provides both a record of the design decisions made and a blueprint for the

implementation phase. This document is created in two steps: first the *architectural design specification*, and then the *detailed design specification*. In ENCOMPASS, the software design specification may contain PLEASE programs which describe the modular structure, and possibly the function, of parts of the system. These programs may be used as prototypes in experiments performed to guide the design process. They may also be used to verify parts of the design using techniques from the Vienna Development Method[19]. During the implementation phase, these PLEASE programs can be refined into programs in the implementation language Path Pascal.

2.1.5. Implementation

In the implementation phase, programming language code for the system is produced[11]. Each separately constructed module must be written, compiled, debugged, and documented. Each module must also be shown to satisfy the requirements and design specifications. In ENCOMPASS, this may be accomplished using mathematical reasoning[16,49], testing[13,18,30], technical review[47], or inspection. The use of executable specifications enhances the verification of system components using either testing or proof techniques. The executable 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. PLEASE provides a framework for the *rigorous*[19] development of programs. Although detailed formal proofs are not required at every step, the framework is present so that they can be constructed if necessary. Parts of a project may use detailed formal verification while other, less critical parts may be handled using less expensive techniques. Once the separate components have been constructed and verified, they must be integrated and verified as a system.

2.1.6. System Integration

In the system integration phase, separately implemented modules are integrated into larger and larger units, each of which is shown to satisfy the specifications[11]. 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. This phase produces the *software verification summary*[11] which describes the results of all reviews, inspections, tests, and formal verifications which have been performed. ENCOMPASS provides tools to aid in the hierarchical integration and testing of programs. When using these tools, all modules which are used by a particular module 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.

After the development has been completed a *development legacy*[11] is written. The legacy summarizes the development and provides a permanent record of what problems and solutions were encountered. This document provides both an aid to management in evaluating the effectiveness of the tools and methods used on the project, and an index to the development to be used by other developers wishing to reuse the components produced. The evaluation and reuse of components is further enhanced by the use of a configuration model to describe software components and their relationships.

3. A Model for Software Configurations

The ENCOMPASS model of software configurations is a refinement of the model presented in[21]. It is similar to the *entity-relationship* model[8] and uses the concepts of *aggregation* and *generalization*[39,40]. The model provides us with a natural way to describe software and also has a convenient representation on conventional computer systems which can be used as the basis for software engineering environments.

3.1. Entities and Relationships

An *entity* is a distinct, uniquely named component. An example of an entity is a file, which could contain the source code for a program, some test data, or an executable program. 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 para-

graphs 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 specification[2], body, object code and load module for a group of routines might be abstracted into a single entity called a "code module". An *aggregation hierarchy* describes the way components are combined to form more and more complex structures.

A *generalization* is an abstraction which allows a number of distinct components to be grouped together into a single named component. A *generalization hierarchy* shows the way components with similar attributes are grouped into more and more general components. In our model, the set of entities which share certain attributes may be viewed as a *generic* entity. For example, the specification and body for a module might share the attributes "module name" and "type" (for example, source code, object code, test data or text). These two entities might then be grouped together into a generic component representing the source code for the module.

An entity has an internal state which may change with time. A *version* represents the state of an entity at a particular point in time. A version of an aggregate entity denotes the versions of all the entities of which it is composed. The same version of an entity may be used in many different composite entities or versions of the same aggregate entity.

3.2. Components Supported by ENCOMPASS

The aggregation hierarchy for ENCOMPASS contains three levels: *simple entities* may be combined into aggregates called *modules*, which may be collected into aggregates called *projects*. An entity

[2] In PLEASE a separately compiled module may have a specification, which describes the interface and function of the module, and a body, which contains the implementation of the module. The two are compiled as a unit to produce a single piece of object code which may be linked with other separately compiled modules to form an executable load module.

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 with the attributes "language", "modify time", and "size". A *module* is an aggregate entity composed of other entities which are closely related or have some common property. For example, a *code module* could contain the specification, body, object code, and load module for a program. The module would have attributes specification, body, object and load with the appropriate entities as values. A *project* is an aggregate entity composed of modules. For example all the modules used in developing a program might be be grouped together into a project.

The generalization hierarchy for ENCOMPASS includes several sub-classes for both modules and simple entities. A module may be: a *code module*, which contains entities associated with the production and debugging of code; a *test module*, which contains materials for the testing of other modules such as sets of test data and test drivers or harnesses; a *proof module*, which contains entities used in the proof of a refinement; a *document module*, which contains entities used in the production of documentation; or a *history module*, which contains components used to track the history of a project. Simple entities may be: *code components*, including source code, object code, load modules and include files; *makefiles*[12], which contain instructions for compilation, linking, and testing; *test data*, such as the input or correct output from a program; *proof data*, which might be input for a mechanical theorem prover; and *document data*, such as input to text processing programs.

3.3. Views

A *view* is a mapping from names to components. A project under development has a distinguished *base view* which describes the entities of the system being designed and the primitive relationships between these entities. Other views of the project are produced from this base view by selecting, and possibly renaming, certain entities with particular attributes. For example, the development and quality assurance teams may have different views of the software system being developed by the project. The development team may use a view of the system which includes all the specifications and software being developed. However, the quality assurance team may have a different view which contains the

specifications, executable code and, in addition, the test cases. Views may be used to abstract the phases of the project corresponding to planning, requirements definition, validation, design, implementation, and system integration. Views may be used to identify a slice of the software being developed, for example, in order to restrict the activities of a programmer to a particular group of modules. Views may also be constructed to represent the effect of a modification on the rest of a system. In ENCOMPASS, access to components is controlled through the use of views and a hierarchical library structure.

4. Library Structure

Figure 2 shows the library structure used by ENCOMPASS which contains a *workspace* for each programmer, a *project library* for each project, and a *global library* common to all projects. Each programmer controls his own workspace while each *project leader* controls the library for his project and the *librarian* controls the global library. All components which are accessed by more than one programmer reside in the project or global libraries where they are controlled by either the project leader 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 or global libraries through a view. A workspace may reference the *working copy* of an components or a *version* fixed at some earlier point in time. The project library contains components that must be available to all the personnel on a particular project, and can aid the project leader in controlling and monitoring the development. The project leader controls the components in the project library by controlling access and the views into the library.

For example, a component containing the specification and body for a module might reside in the project library. Assume two programmers are working on the module. Programmer A is assigned the task of writing a specification for the module. Therefore he may access the working copy of the specification from his workspace, but he has no access to the body for the module. Programmer B is assigned the task of writing the body from the completed specification. Therefore his workspace contains references to a fixed version of the specification and the working copy of the body.

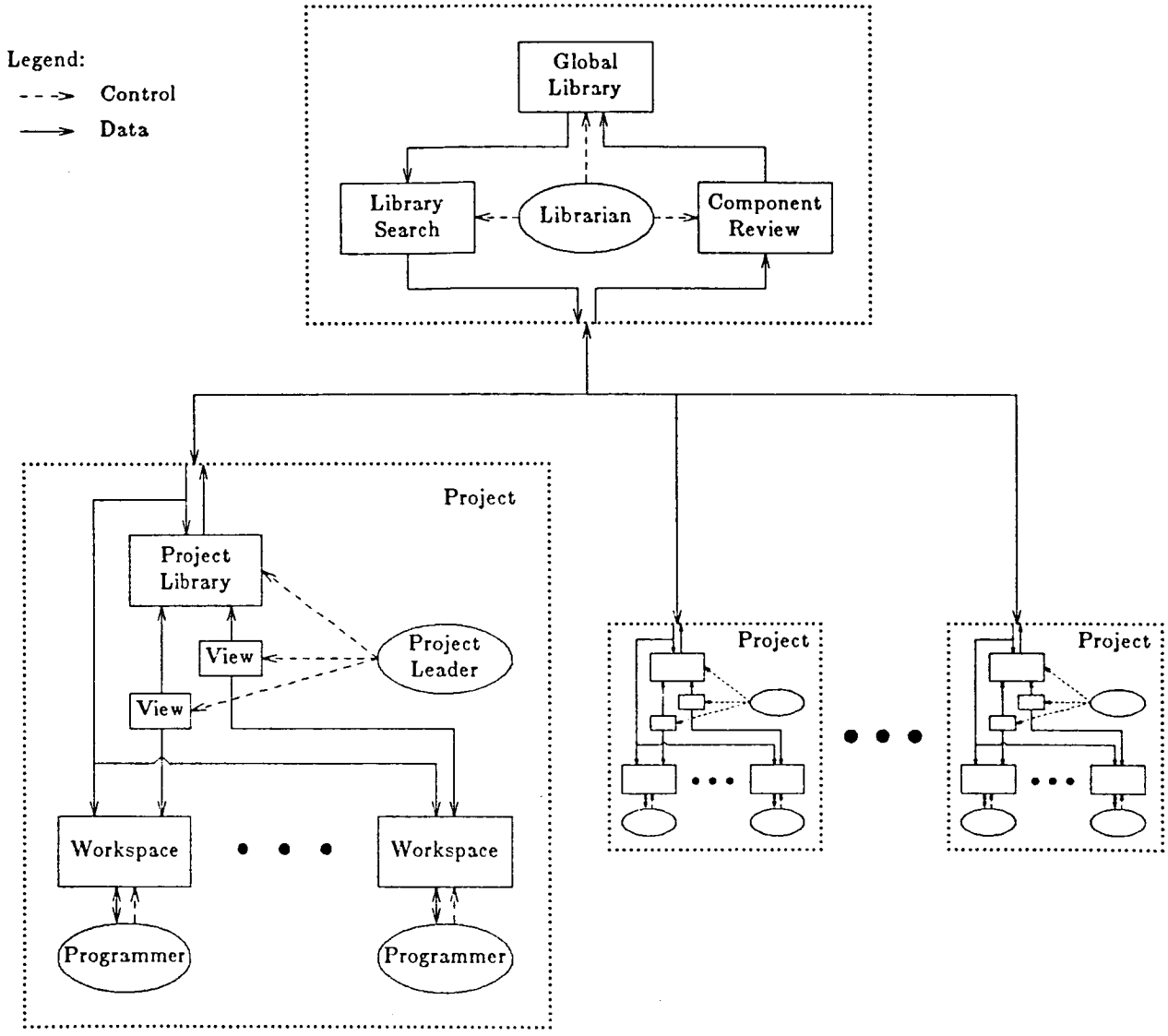

Figure 2. ENCOMPASS Library Structure

The global library 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]. Components in the library may be accessed either individually or in groups. 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 a reference to 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 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. The proof data and any associated documentation could also be retrieved, modified, and reused in the new development.

5. Implementation

A prototype implementation of ENCOMPASS is being constructed on a Vax running BSD 4.2 UNIX. ENCOMPASS is designed to be an extension of the UNIX environment, so standard software tools can be used. ENCOMPASS currently encorporates standard editors, text processors, compilers, linkers and many other tools. Language-oriented tools for PLEASE are being constructed with the SAGA meta-tools. For example, a language-oriented editor for PLEASE is created from a BNF description of the language. Other language-oriented tools being constructed include an interactive tool to

[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.

transform PLEASE programs into executable form and a verification condition generator.

The configuration control tools and the hierarchical library structure are implemented using a representation of our configuration model on the UNIX file system[21]. The representation uses files to represent simple entities, directories to represent modules and projects, and symbolic links[4] to represent complex relationships. For example, a directory representing a module may contain files representing simple entities such as the specification of the module, the body of the module, the object code, and possibly the load module. A number of tools have been written which use the underlying directory structures. For example, complex entities can be moved and copied as single units. A version of any entity can be saved using the RCS revision control system[45]. For complex entities a table containing the versions of all the sub-components is stored.

The use of symbolic links simplifies the interaction of the configuration tools and existing systems components. By implementing references between modules by symbolic links, tools such as a compiler can directly access the required source needed for the compilation and existing compilers can be used in our environment without alteration. Another benefit of the use of symbolic links is that the makefile for a module only needs to search the current directory for source dependencies. Therefore, the makefile can use pattern matching techniques to access all the relevant files in a module and does not have to be rewritten every time the modularization of the program is changed.

The workspaces and libraries are implemented as directories, which are owned by the person who controls them. These directories contain sub-directories, files and symbolic links with the meanings given above. Views are implemented as directories containing symbolic links. References from workspaces, through views, to components in the project and global libraries are implemented as chains of symbolic links. Views are created and modified by csh[5] scripts which are saved and run by project leaders. If a view references a particular version of an entity, rather than the working copy, the version is checked out of RCS into a special area of the library when the view is created. This structure has

[4] 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.

[5] 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.

been used to support PLEASE, Path Pascal, C, Pascal and csh programs.

6. Future Work

Although ENCOMPASS is independent of the language used for development, currently all the language-oriented tools are being constructed for PLEASE and Path Pascal. We plan to apply our executable specification method to ADA and create the language-oriented tools to support it. We plan to extend the notion of versions used in ENCOMPASS to differentiate between sequential *revisions* and parallel *alternatives*. A revision supercedes the component from which it was created, while an alternative provides a choice between component. For example, different alternatives of a program can be maintained for use with different operating systems. Each alternative passes through a series of revisions as it evolves.

Presently the configuration control tools in ENCOMPASS can only be used on projects which follow certain conventions for directory structure. We would like to extend the implementation of ENCOMPASS to allow its use with any pre-existing directory structure on UNIX. We would also like to extend ENCOMPASS to support aggregation hierarchies of arbitrary complexity and a generalized hierarchical library structure. We plan to use ENCOMPASS to maintain itself, and to develop several new software tools. We hope that this experience will give us new insights which will be incorporated in future versions of ENCOMPASS.

7. Summary and Conclusions

ENCOMPASS is an example software engineering environment being constructed by the SAGA project to support a particular model of the software life-cycle and software configurations. In ENCOMPASS, the software life-cycle is viewed as a sequence of developments, each of which reuses components from the previous ones. An executable specification language is used so that programs are available for experimentation, evaluation, and validation as early as possible in the development process. ENCOMPASS supports the Vienna Development Method, in which a system is constructed by first producing a specification in an executable specification language and then incrementally refining it into a program in

an implementation language. Each refinement produces an executable program which may be used as a prototype system. By producing a running system early and often in the development process, design and specification errors can be detected and corrected earlier and at lower cost.

The components in a software system are modeled as entities which have relationships between them. An entity may have different versions and different views of the same project are allowed. ENCOMPASS supports multiple programmers and projects using a hierarchical library system containing a workspace for each programmer; a project library for each project, and a global library common to all projects. By dividing the life-cycle into a sequence of small steps, using a rigorous model for the components produced and used, and incorporating a hierarchical library structure, ENCOMPASS should enhance the tracking, evaluation and management of software projects.

8. References

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

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

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

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

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

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

7. 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.

8. Chen, Peter Pin-Shan. *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.

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

10. 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.

11. Fairley, Richard. Software Engineering Concepts. McGraw-Hill, New York, 1985.

12. Feldman, Stuart I. *Make - A Program for Maintaining Computer Programs.* Software - Practice and

Experience (1979) vol. 9, pp. 255-265.

13. 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.

14. 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.

15. Goldberg, A. Smalltalk-80: The Interactive Programming Environment. Addison-Wesley, Reading, MA, 1984.

16. 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.

17. 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.

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

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

20. Kamin, 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).

21. 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).

22. 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.

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

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

25. 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.

26. 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.

27. 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.

28. 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.

29. 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.

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

31. 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.

32. 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.

33. 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.

34. 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.

35. 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.

36. 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.

37. 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.

38. 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.

39. Smith, John M. and Diane C. P. Smith. *Database Abstractions: Aggregation.* Communications of the ACM (June, 1977) vol. 20, no. 6, pp. 405-413.

40. 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.

41. 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.

42. 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.

43. Teitelman, W. and L. Masinter. *The Interlisp Programming Environment.* Computer (April 1981) vol. 14, no. 4, pp. 25-33.

44. Terwilliger, Robert B. and Roy H. Campbell. *A Preliminary Look at PLEASE: an Executable Specification Language for Concurrent Programs.* Technical Report in Preparation, Dept. of Computer Science, University of Illinois at Urbana-Champaign (1985).

45. Tichy, Walter F. *Design, Implementation, and Evaluation of a Revision Control System.* Proceedings of the 6th IEEE International Conference on Software Engineering (September 1982) pp. 58-67.

46. 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.

47. 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.

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

49. 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.

50. 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.

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

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