An Early Report on ENCOMPASS

Robert B. Terwilliger*
Roy H. Campbell**

CU-CS-380-87 October 1987

*Department of Computer Science, Campus Box 430, University of Colorado, Boulder, Colorado, 80309. This research supported by NASA grant NAG 1-138.

**Department of Computer Science, University of Illinois, Urbana, IL 61801. This research supported by NASA grant NAG 1-138.

An Early Report on ENCOMPASS

Robert B. Terwilliger†

Department of Computer Science,
University of Colorado,
Boulder, CO 80309-0430

Roy H. Campbell

Department of Computer Science
University of Illinois
Urbana, IL 61801

Abstract

ENCOMPASS is an environment to support the incremental construction of Ada® programs using executable specifications and formal techniques similar to the Vienna Development Method. ENCOMPASS supports the *rigorous* development of software: parts of a project may use completely formal methods, while other, less critical parts use less expensive techniques. ENCOMPASS provides automated support for all aspects of the development process including specification, prototyping, testing, formal verification, documentation, configuration control and project management. In ENCOMPASS, software can be specified using PLEASE, an Ada-based executable specification language which can be automatically translated into Prolog. A prototype implementation of ENCOMPASS has been constructed. In this paper, we give an overview of ENCOMPASS, describe the decisions made in the design of the prototype, and discuss the lessons learned in the process.

1. Introduction

The production of software is both difficult and expensive. One of the largest problems is *quality*; many of the systems produced do not satisfy their purchasers in either functionality, performance or reliability. The software quality problem can be subdivided in a number of ways, depending on the model of software development used. 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 [32]. 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 [32]. The specification need not be executable; in general, it must be translated into an implementation. Depending on the method used for translation, the exact relationship between the specification and implementation may be unknown. An implementation is *verified* when it is shown to satisfy its specification [32].

---

† This work was performed while the author was a Ph.D. candidate at the University of Illinois

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

® Ada ® is a trademark of the US Government, Ada Joint Program Office.

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 [50, 101]; providing prototypes for experimentation and evaluation should enhance the validation process.

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 [33, 60]; 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 [31, 95]; 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 [49, 54, 98]; 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 [27, 29]; therefore, methods which combine a number of techniques have been proposed [74, 100].

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 [11, 13, 14, 25, 48, 49, 67, 78]. In this method, components are first written using a combination of conventional programming languages and predicate logic. A procedure or function may be specified using pre- and post-conditions: the *pre-condition* states the properties that the inputs must satisfy, while the *post-condition* states the relationship of inputs to outputs. Similarly, an abstract data type may be specified with an *invariant* defining the acceptable states and pre- and post-conditions for the operations. To increase the expressive power of specifications, the high-level types *set*, *list*, and *map* are added to the language.

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. 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.

VDM is used in industrial environments to enhance the development process [14, 67, 78]. 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. In these situations, formal specifications serve mostly as a tool for precise communication, and the major impact on methodology is that more time is spent on specification and design. However, the methods do prove useful in practice. VDM could prove even more useful if it was applied more formally and supported by automated tools. Many feel the cost is justified, and environments to support VDM are being constructed [12]. We feel 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.

1.2. ENCOMPASS

The SAGA project is investigating both the formal and practical aspects of providing automated support for the full range of software engineering activities [15-19, 44, 51, 52, 85-89]. The group has constructed ENCOMPASS [85, 86], an integrated environment to support incremental software development in a manner similar to VDM. ENCOMPASS extends VDM with the use of executable specifications and testing-based verification methods. It automates these techniques and integrates them as smoothly as possible into the traditional life-cycle.

In ENCOMPASS, software is specified using a combination of natural language and PLEASE [87-89], an Ada-based, wide-spectrum, executable specification language. PLEASE specifications may be used in proofs of correctness; they also may be transformed into prototypes which use Prolog [23] to execute pre- and post-conditions. These prototypes can enhance both the validation and verification processes. 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 [51] and project management [18].

ENCOMPASS is an environment for the *rigorous* [49] 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. 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.

In the remainder of this paper, we present ENCOMPASS, PLEASE and IDEAL in more detail, describe the decisions made in the design of the preliminary implementation, and discuss the lessons learned in the process. In section two we describe the software development paradigm ENCOMPASS is designed to support, and in section three we describe the PLEASE executable specification language. In section four we describe IDEAL, the programming-in-the-small environment for PLEASE, and in section five we present the programming-in-the-large tools provided by ENCOMPASS. In section six we describe the system status and in section seven we compare our work with that of other researchers. Finally, in section eight we summarize and draw some conclusions from our work so far.

2. Development Paradigm

ENCOMPASS is designed to support a particular software development paradigm; this is basically Fairley's *phased* or *waterfall* life-cycle [32], 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. A development passes through the phases planning, requirements definition, validation, refinement and system integration.

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. The *validation phase* attempts to show that any system which satisfies the software requirements specification will also satisfy the customers. If not, then the requirements specification should be corrected before the development proceeds. To aid in the validation process, the PLEASE components in the specification may be transformed into executable prototypes which can be used in interactions with the customers. These prototypes 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*. 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. The design transformation may produce 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.

PLEASE prototypes can be used to verify the correctness of refinements using testing techniques. Most simply, the prototype produced from a PLEASE specification can be used as a test oracle against which implementations can be compared. In a more complex situation, 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 [46, 54].

ENCOMPASS provides support for all aspects of this development paradigm. Many of the tools in ENCOMPASS are independent of the language used for development, but others are specific to PLEASE.

3. PLEASE

The first step in the program of research described in this paper is the design of a wide-spectrum, executable specification language called PLEASE. 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. At present, we are using Ada [92] 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 [57]; 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 [64]; 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 validity problem for first-order logic is undecidable [54,58]. Therefore, if we allow full first-order logic to be used for the specifications, in some cases we will be unable to construct prototypes which are guaranteed to terminate [91].

A resolution theorem prover for first-order logic could be used to construct prototypes[1]; however, the performance of these prototypes would be very poor. The axioms for types such as sets would result in a further decrease in performance[2]. The emergence of logic programming as a technology, most notably Prolog [23], 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 [71]. By restricting the types used to those with efficient Prolog implementations, reasonable specification power is combined with

[1] which are not guaranteed to terminate.

[2] For sets in their full generality. Restricted forms can be implemented efficiently.

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) [92] states that *in out* parameters may be implemented using either a copy/restore or pointer strategy. Most of the work on program verification assumes one implementation or the other. Although PLEASE uses Ada syntax, the constructs do not necessarily have the Ada semantics; the semantics of PLEASE are defined using Hoare calculus proof rules [91]. 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 base language.

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

3.1. Specifying a Procedure

For example, Figure 7 shows the PLEASE specification of a package, *sort_pkg*, which provides a procedure called *sort*[3]. 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*[4]. 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 the specification, the state before execution begins is denoted by *in(...)*, while the state after execution has completed is denoted by *out(...)*. 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.

---

[3] To increase readability and understandability, the syntax of PLEASE is similar to Anna [57].

[4] In PLEASE, as in Prolog, the empty list is denoted by [ ], and a list literal is denoted by [*l*], where *l* is a comma separated list of elements. The functions *hd*, *tl*, and *cons* have their usual meanings and $L_1 \mid\mid L_2$ denotes the concatenation of the elements of L_1 and L_2.

```
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 7. Specification of *sort* procedure

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 [20, 23]. This approach allows a simple translation from predicate definitions into Prolog procedures.

3.2. Prototyping the Specification

The specification in Figure 7 can be automatically translated into a prototype written in a combination of Prolog and Ada[5]. The predicates *permutation* and *sorted* and the pre- and post-conditions for *sort* are translated into

[5] For a more complete treatment of this example see [88] or [91].

Prolog procedures, which are executed by an interpreter in the current implementation. When the *sort* procedure is called, the *in* parameter is converted to the Prolog representation and the call is passed to the interpreter. When the Prolog procedure for *sort* completes, the *out* parameter is converted to the Ada representation and the original call returns. Tools in the ENCOMPASS environment perform the translation and generate code to handle I/O and other implementation level details. The Prolog procedure for *sort* simply "executes" the pre- and post-conditions. The notion of execution is quite different for pre- and post-conditions. Executing a pre-condition involves checking that given data satisfies a logical expression. Executing a post-condition means finding data that satisfies a logical expression. For example, the post-condition for *sort* must find a value for the output list such that the input and output are permutations of each other and the output is sorted.

Although many implementations show significant deviations [82], a "pure" Prolog interpreter can be viewed as a resolution theorem prover [20, 23]. In order to dramatically increase Prolog's efficiency, several concessions were made. First, Prolog is based on *Horn clauses* [20, 23]. 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. 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 *sorted([2,1])* is false. 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 [72]. We do not believe the closed world assumption is suitable for software engineering applications. We view software development as an incremental process in which each step adds more information about the system to be built to the knowledge-base. Therefore, the knowledge-base will almost always be incomplete. Also, at times it is more convenient to state what is false than what is true.

Viewing Prolog as a theorem prover, execution can be seen as proving a formula of the form $\exists \bar{X}\ p(\bar{X})$ by finding an example $\bar{a}$ such that $p(\bar{a})$ is true. Using this model, the translation from PLEASE predicates to Prolog code is simply a sequence of transformations between equivalent formulae. The only complication is that terms are *unraveled* into conjunctions of relations to provide a reasonable notion of equality[6]. Briefly, we assume that for each function $f(\bar{x})$, there exists a relation $F(\bar{x},y)$ such that $f(\bar{x})=y$ iff $F(\bar{x},y)$. We unravel the formula $P(..f(\bar{x})..)$ into

[6] For a more complete explanation of the translation process see [88, 91].

the equivalent formula $\exists t\,(F(\bar{x},t) \wedge P(..t..))$[7]. The prototypes produced by this translation process are *partially correct* [54, 58] with respect to the specifications. In other words, if a prototype terminates normally then the value returned will satisfy the post-condition. A prototype would be *totally correct* [54, 58] if it was also guaranteed to terminate normally. In general it is not possible to extend our approach to total correctness [91]. Although the Prolog procedures produced by our translation process have the proper logical properties, there is no guarantee that they will terminate. In the last step of the translation process, a number of heuristic transformations are used on the Prolog procedures to increase the chances of termination.

Prolog is implemented using a depth-first search strategy and clauses are considered in the order they appear textually. This allows a very efficient implementation and enables the programmer to control the search process in a simple manner. At present, PLEASE specifications are purely declarative; unlike Prolog, the evaluation order can not be controlled by the programmer. At the beginning of our work, we felt the addition of an operational semantics to PLEASE would simply complicate matters. Our experience leads us to believe this is not practical; it is too difficult for a translator to construct a prototype with reasonable chances of termination from a purely declarative specification. Future versions of PLEASE will allow the programmer the control the order in which clauses are evaluated in a manner similar to Prolog. We believe that we can allow the programmer to force evaluation and "cut" branches from the search without compromising the declarative semantics of the specification.

4. IDEAL

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 is an environment that supports these activities. IDEAL contains four tools: TED [44], a proof management system which is interfaced to a number of theorem provers; ISLET, 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*.

---

[7] For other solutions to the equality problem for Prolog see [36, 53].

In the module database, a set of *symbol tables* represent the PLEASE specifications and Ada programs being developed. These symbol tables are displayed and manipulated by ISLET. ISLET can be used to create PLEASE specifications and incrementally refine them into Ada programs. Steps in the refinement process 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 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: 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.

4.1. 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 [46, 54]. The user interacts with ISLET through a simple language-oriented editor similar to [73]. 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, TED can 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* [49] 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.

5. ENCOMPASS

IDEAL provides an integrated environment for programming-in-the-small using PLEASE; while this alone can enhance the development process, more can be gained with the addition of an environment for programming-in-the-large. ENCOMPASS is such an environment; it provides support for all aspects of software development. 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.

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 configuration management system also provides a primitive form of software capabilities to control access to components. The *project management system* uses these capabilities to control both access to data and interactions between developers.

5.1. Configuration and Project Management

Configuration management is concerned with the identification, control, auditing, and accounting of the components produced and used in software development and maintenance [10]. A number of different configuration management systems have been proposed, developed and/or used [30, 65, 70]. In ENCOMPASS, the configuration management system is responsible for maintaining the consistency of, integrity of, and relationships between the products of software development. In ENCOMPASS, software configurations are modeled using a variant of the *entity-relationship model* [22] which incorporates the concept of *aggregation* [79]. At present, most databases do not provide the features necessary to support integrated software development environments [8]; our model provides us with a natural way to describe software and also has a convenient implementation on conventional computer systems [51].

In ENCOMPASS, a *view* is a mapping from names to components; access to components is controlled through the use of views. A project under development has a unique *base view* or *project library* which contains all the components of the system being developed and all the relationships between them. Other views can be created from the project library by restricting access; a derived view can only contain entities or relationships present in the base view. The granularity of the present implementation is somewhat coarse: locking is at the module level. While this notion of view is simpler than that typically taken in the database community [26], we feel the elimination of the view-update problem justifies the restriction. A software development database stores objects which undergo frequent changes of state; the underlying database should be able to process these changes without difficulty.

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 [77]. The ENCOMPASS project management system is based on a *management by objectives* approach [39]: each step in the development process satisfies an objective by producing a milestone. For example, the objective of the requirements definition phase is to describe the properties that the software system to be constructed must satisfy; the milestone for this phase is a specification which lists these properties. In ENCOMPASS, PLEASE 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. We feel the existence of these milestones significantly enhances automation of the approach.

The ENCOMPASS project management system is based on a hierarchical project organization: the team members form a DAG. The system is organized around *work trays* [18], which provide a mechanism to allocate, store, track and transfer the components produced and used 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. Work trays may be used for communication within a project: tasks may be transferred between workspaces. Any object stored in the project database can be transferred using the tray system without special "packing" or preparation. We feel this greatly increases the utility of the system.

6. 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, 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 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. Research is continuing at both the University of Illinois and the University of Colorado, including construction of second generation configuration and project management systems [19].

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 initial implementation of PLEASE is based on the UNSW Prolog interpreter [76] and the Verdix Ada Development System [93]; it runs under Berkeley Unix® on a Sun 2/170. The Prolog interpreter and Ada program run as a separate processes and communicate through pipes[8]. 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 this paper 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. Current research on PLEASE involves the addition of more pre-defined types and an algebraic type definition facility. We are also constructing an improved implementation in which the Prolog interpreter and base language share run-time data structures, thus reducing the procedure call overhead.

Unix® is a trademark of AT&T.

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

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, [88] presents a design transformation on the specification in this paper. The transformation consists of twenty-six steps, only two of which generated verification conditions which could not be certified by the simple methods. The example presented in [85] also consists of twenty six steps, only four of which generated verification conditions which did not 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 complex verification condition in [88] can be certified in about five CPU seconds simply by invoking the theorem prover on the file produced by ISLET. The second verification condition can not be proved in this manner; it requires a considerable investment of user time to decompose it into a number of lemmas. Current work on IDEAL involves the use of more complex proof tactics, such as induction over the pre-defined types and bounded-depth search. We are also extending IDEAL with a deductive synthesis component [90].

7. Comparison with Related Work

The work presented in this paper is similar to a number of previous or on-going research projects. For example, many specification methods have been previously proposed, designed or put into use [34,99]; they can be formally-based [1,9,35,41,45,63], or encorporate natural language and graphics [75]. The formally-based methods may be roughly divided into model-oriented [1,9,45] and axiomatic approaches [35,40,63], although languages which combine the two methods have also been proposed [42,96]. 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.

Many different development methods have been proposed [3,38,47,49,62,97]; for example, in *transformational programming* a very-high level specification is used to produce an efficient implementation by a series of correctness preserving modifications [2,4,21,66,69]. Artificial intelligence techniques can be applied to the software development problem in a number of ways [3,5,37,55,80,94]; for example, in *deductive synthesis*, theorem proving techniques are used create verified code from specifications [28,37,59]. 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 [6,24]. 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 [7]. 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 encorporate artificial intelligence techniques [90].

Many different tools have been proposed to support the software development process [61]. These include systems for the verification of programs [43,56], environments for programming-in-the-small [84], environments for the construction of a program and its proof simultaneously [73], and environments for programming-in-the-large [68,81,83]. 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. ENCOMPASS also allows some modules of a project to be developed using PLEASE and formal methods, while other modules are developed using conventional techniques; we know of no other environment designed to support such a methodology.

8. Summary and Conclusions

Traditional methods do not ensure the production of correct software. VDM [13,49,67] is a method which has been used in industrial settings to enhance software development. In VDM, software is first specified using a combination of programming languages and predicate logic. These abstract components are then refined into components in an implementation language. 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. ENCOMPASS [85,86] is an integrated environment which supports software development using executable specifications and formal techniques similar to VDM. ENCOMPASS enhances VDM by reducing both the effort involved and the chance of errors.

ENCOMPASS is an environment for the *rigorous* [49] 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.

In ENCOMPASS, software is first specified using a combination of natural language and PLEASE [87-89], an Ada-based, wide spectrum, executable specification and design language. PLEASE specifications can be used in proofs of correctness; they can also be transformed into prototypes which use Prolog to "execute" pre and post-conditions. We feel the early production of prototypes will increase customer/developer communication and enhance the software development process. We believe that specifications combining conventional languages and logic programming are a promising basis for a development methodology. As the specifications are both executable and formally based, the equivalence between specification and implementation can be determined using either testing or proof techniques.

At present, PLEASE specifications are based on Horn clauses in a manner similar to pure Prolog. Our experience leads us to believe that this is not sufficient for software specification. Using this method, it is not possible to state that something is false. The solution used in Prolog is the closed world assumption: it is assumed that all truths are known. If something is not provably true, then it is assumed to be false. We do not believe this is suitable for software engineering applications. We view software development as an incremental process in which each step adds more information about the system to be built to the knowledge-base. Therefore, the knowledge-base will almost always be incomplete. Also, at times it will be more convenient to state what is false than what is true.

At present, PLEASE specifications are purely declarative; unlike Prolog, the evaluation order can not be controlled by the programmer. At the beginning of our work, we felt the addition of a operational semantics to PLEASE would simply complicate matters. Our experience leads us to believe this is not practical; it is too difficult for a translator to construct a prototype with reasonable chances of termination from a purely declarative specification. Future versions of PLEASE will allow the programmer the control the order in which clauses are evaluated in a manner similar to Prolog. We believe that we can allow the programmer to force evaluation and "cut" branches from the search without compromising the declarative semantics of the specification.

In ENCOMPASS, components specified in PLEASE are incrementally refined into Ada components using IDEAL, an environment for programming in the small which supports verification using any combination of testing, peer review, or formal methods. From a formal viewpoint, many steps in the refinement process generate

verification conditions in the underlying first-order logic. These verification conditions are first subjected to a number of simple (and inexpensive) 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, then a more powerful (and expensive) system can be invoked. In our experience, most of the verification conditions will be proven using the inexpensive methods. Those that are expensive to verify may be proven immediately, or deferred until a later time.

The work described in this paper has just completed the "proof of concept" stage. A prototype implementation has been constructed and demonstrated on a number of small examples. Even at this stage, we feel we have shown that logic programming and conventional languages can be combined into executable specifications and that automated environments can provide significant support for development methods such as VDM. We believe a large class of tools can be constructed which provide fifty to ninety percent solutions to the generally unsolvable problems involved. We feel that the use of future environments similar to ENCOMPASS will greatly enhance the specification, design and development of software.

9. References

1. Auernheimer, B. and R. A. Kemmerer, "RT-ASLAN: A Specification Language for Real-Time Systems", *IEEE Transactions on Software Engineering SE-12, 9* (September 1986), 879-889.
2. Balzer, R., T. E. Cheatham and C. Green, "Software Technology in the 1990's: Using a New Paradigm", *IEEE Computer 16, 11* (November 1983), 39-45.
3. Balzer, R., "A 15 Year Perspective on Automatic Programming", *IEEE Transactions on Software Engineering SE-11, 11* (November 1985), 1257-1268.
4. Barstow, D. R., "On Convergence Toward a Database of Program Transformations", *ACM Transactions on Programming Languages and Systems 7, 1* (January 1985), 1-9.
5. Barstow, D. R., "Artificial Intelligence and Software Engineering", *Proceedings of the 9th International Conference on Software Engineering*, 1987, 200-211.
6. Bates, J. L. and R. L. Constable, "Proofs as Programs", *ACM Transactions on Programming Languages and Systems 7, 1* (January 1985), 113-136.
7. Benzinger, L. A., "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, May 1987.
8. Bernstein, P. A., "Database System Support for Software Engineering", *Proceedings of the 9th International Conference on Software Engineering*, 1987, 166-178.
9. Berry, D. M., "Towards a Formal Basis for the Formal Development Method and the Ina Jo Specification Language", *IEEE Transactions on Software Engineering SE-13, 2* (February 1987), 184-201.
10. Bersoff, E. H., "Elements of Software Configuration Management", *IEEE Transactions on Software Engineering SE-10, 1* (January 1984), 79-87.
11. Bjorner, D. and C. B. Jones, *Formal Specification and Software Development*, Prentice-Hall, Englewood Cliffs, N.J., 1982.
12. 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.
13. Bjorner, D., "On The Use of Formal Methods in Software Development", *Proceedings of the 9th International Conference on Software Engineering*, 1987, 17-29.
14. Bloomfield, R. E. and P. K. D. Froome, "The Application of Formal Methods to the Assessment of High Integrity Software", *IEEE Transactions on Software Engineering SE-12, 9* (September 1986), 988-993.

15. Campbell, R. H. and P. G. Richards, "SAGA: A system to automate the management of software production", *Proceedings of the National Computer Conference*, May 1981, 231-234.

16. Campbell, R. H. and P. 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, 73-80.

17. Campbell, R. H., "SAGA: A Project to Automate the Management of Software Production Systems", in *Software Engineering Environments*, Sommerville, I. (editor), Peter Perigrinus Ltd, 1986.

18. Campbell, R. H. and R. B. Terwilliger, "The SAGA Approach to Automated Project Management", in *International Workshop on Advanced Programming Environments*, Carter, L. R. (editor), Springer-Verlag Lecture Notes in Computer Science, New York, 1986, 145-159.

19. 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, May 1987.

20. Chang, C. and R. C. Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.

21. Cheatham, T. E., G. H. Holloway and J. A. Townley, "Program Refinement By Transformation", *Proceedings of the 5th International Conference on Software Engineering*, 1981, 430-437.

22. Chen, P. P., "The Entity-Relationship Model - Toward a Unified View of Data", *ACM Transactions on Database Systems 1, 1* (March 1976), 9-36.

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

24. Constable, R. L. and M. J. O'Donnell, *A Programming Logic*, Winthrop Publishers, Cambridge, Massachusetts, 1978.

25. Cottam, I. D., "The Rigorous Development of a System Version Control Program", *IEEE Transactions on Software Engineering SE-10, 3* (March 1984), 143-154.

26. Date, C. J., *An Introduction to Database Systems*, Addison-Wesley, Reading, Massachusetts, 1986.

27. DeMillo, R. A., R. J. Lipton and A. J. Perlis, "Social Processes and Proofs of Theorems", *Communications of the ACM 22, 5* (May 1979), 271-280.

28. Dershowitz, N., "Synthetic Programming", *Artificial Intelligence 25* (1985), 323-373.

29. Dijkstra, E. W., "Structured Programming", in *Software Engineering Principles*, Buxton, J. N. and B. Randall (editor), NATO Science Committee, Brussels, Belgium, 1970.

30. Estublier, J. and N. Belkhatir, "Experience with a Data Base of Programs", *Proceedings of the Second ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, December 1986, 27-33.

31. Fagan, M. E., "Advances in Software Inspections", *IEEE Transactions on Software Engineering SE-12, 7* (July 1986), 744-751.

32. Fairley, R., *Software Engineering Concepts*, McGraw-Hill, New York, 1985.

33. Gannon, J., P. McMullin and R. Hamlet, "Data-Abstraction Implementation, Specification, and Testing", *ACM Transactions on Programming Languages and Systems 3, 3* (July 1981), 211-223.

34. Gehani, N. and A. D. McGettrick, eds., *Software Specification Techniques*, Addison Wesley, Reading, Massachusetts, 1986.

35. Goguen, J., J. Thatcher and E. Wagner, "An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types.", in *Current Trends in Programming Methodology, IV*, Yeh, R. (editor), Prentice-Hall, London, 1978, 80-149.

36. Goguen, J. A. and J. Meseguer, "Equality, Types, Modules and (why not?) Generics for Logic Programming", *Logic Programming 1, 2* (1984), 179-210.

37. Goldberg, A. T., "Knowledge-Based Programming: A Survey of Program Design and Construction Techniques", *IEEE Transactions on Software Engineering SE-12, 7* (July 1986), 752-768.

38. Gries, D., *The Science of Programming*, Springer-Verlag, New York, 1981.

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

40. Guttag, J. V. and J. J. Horning, "The Algebraic Specification of Abstract Data Types", *Acta Informatica 10* (1978), 27-52.

41. Guttag, J. V. and J. J. Horning, "Formal Specification as a Design Tool", *Proceedings of the 7th ACM Symposium on the Principles of Programming Languages*, 1980, 251-261.

42. Guttag, J. V., J. J. Horning and J. M. Wing, "The Larch Family of Specification Languages", *IEEE Software 2, 5* (September 1985), 24-36.

43. Halpern, J. D., S. Owre, N. Proctor and W. F. Wilson, "Muse - A Computer Assisted Verification System", *IEEE Transactions on Software Engineering SE-13, 2* (February 1987), 151-156.

44. Hammerslag, D. H., S. N. Kamin and R. 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.

45. Henderson, P., "Functional Programming, Formal Specification, and Rapid Prototyping", *IEEE Transactions on Software Engineering SE-12, 2* (February 1986), 241-250.

46. Hoare, C. A. R., "An Axiomatic Basis for Computer Programming", *Communications of the ACM 12, 10* (October 1969), 576-580.

47. Jackson, M., *System Development*, Prentice-Hall, Englewood Cliffs, N.J., 1983.

48. Jackson, M. I., "Developing Ada Programs Using the Vienna Development Method (VDM)", *Software - Practice and Experience 15, 3* (March 1985), 305-318.

49. Jones, C. B., *Software Development: A Rigorous Approach*, Prentice-Hall International, Engelwood Cliffs, N.J., 1980.

50. 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.

51. Kirslis, P. A., R. B. Terwilliger and R. 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, 44-53.

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

53. Kornfeld, W. A., "Equality for Prolog", *Proceedings of the International Joint Conference on Artificial Intelligence*, 1983.

54. Loeckx, J. and K. Sieber, *The Foundations of Program Verification*, John Wiley & Sons, New York, 1984.

55. Lubars, M. D. and M. T. Harandi, "Knowledge-Based Software Design Using Design Schemas", *Proceedings of the 9th International Conference on Software Engineering*, 1987, 253-262.

56. Luckham, D. C., S. M. German, F. W. 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, March 1979.

57. Luckham, D. C. and F. W. Henke, "An Overview of Anna, a Specification Language for Ada", *IEEE Software 2, 2* (March 1985), 9-22.

58. Manna, Z., *Mathematical Theory of Computation*, McGraw-Hill, New York, 1974.

59. Manna, Z. and R. Waldinger, "A Deductive Approach to Program Synthesis", *ACM Transactions on Programming Languages and Systems 2, 1* (January 1980), 90-121.

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

61. Miller, E., ed., *Tutorial: Automated Tools for Software Engineering*, IEEE Computer Society, New York, 1979.

62. Mills, H. D. and R. C. Linger, "Data Structured Programming: Program Design without Arrays and Pointers", *IEEE Transactions on Software Engineering SE-12, 2* (February 1986), 192-197.

63. Musser, D. R., "Abstract Data Type Specification in the AFFIRM System", *IEEE Transactions on Software Engineering SE-6, 1* (January 1980), 24-32.

64. Myers, W., "Ada: First users - pleased; prospective users - still hesitant", *IEEE Computer 20, 3* (March 1987), 68-73.

65. Narayanaswamy, K. and W. Sacchi, "Maintaining Configurations of Evolving Software Systems", *IEEE Transactions on Software Engineering SE-13, 3* (March 1987), 324-334.

66. Neighbors, J. M., "The Draco Approach to Constructing Software from Reusable Components", *IEEE Transactions on Software Engineering SE-10, 5* (September 1984), 564-574.

67. Oest, O. N., "VDM From Research to Practice", *Information Processing*, 1986, 527-533.

68. Osterweil, L. J., "Toolpack - An Experimental Software Development Environment Research Project", *IEEE Transactions on Software Engineering SE-9, 6* (November 1983), 673-685.

69. Partsch, H. and R. Steinbruggen, "Program Transformation Systems", *Computing Surveys 15, 3* (September 1983), 199-236.

70. Perry, D. E., "Version Control in the Inscape Environment", *Proceedings of the 9th International Conference on Software Engineering*, 1987, 142-149.

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

72. Reiter, R., "On Closed World Data Bases", in *Logic and Data Bases*, Gallaire, H. and J. Minker (editor), Plenum Press, 1978.

73. Reps, T. and B. Alpern, "Interactive Proof Checking", *Proceedings of the 11th ACM Symposium on the Principles of Programming Languages*, January 1984, 36-45.

74. Richardson, D. J. and L. A. Clarke, "Partition Analysis: A Method Combining Testing and Verification", *IEEE Transactions on Software Engineering SE-11, 12* (December 1985), 1477-1490.

75. Ross, D. T. and K. E. Schoman, Jr., "Structured Analysis for Requirements Definition", *IEEE Transactions on Software Engineering SE-3, 1* (January 1977), 6-15.

76. Sammut, C. A. and R. A. Sammut, "The Implementation of UNSW-Prolog", *The Australian Computer Journal 15, 2* (May 1983), 58-64.

77. Schwartz, D. P., "Software Evolution Mangement: An Integrated Discipline for Managing Software", *Proceedings of the 9th International Conference on Software Engineering*, 1987, 388-397.

78. 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 9, 2* (April 1984), 54-79.

79. Smith, J. M. and D. C. P. Smith, "Database Abstractions: Aggregation", *Communications of the ACM 20, 6* (June 1977), 405-413.

80. Smith, D. R., G. B. Kotik and S. J. Westfold, "Research on Knowledge-Based Software Environments at Kestrel Institute", *IEEE Transactions on Software Engineering SE-11, 11* (November 1985), 1278-1295.

81. Standish, T. A. and R. 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, 57-64.

82. Stickel, M. E., "A Prolog Technology Theorem Prover", *Proceedings of the International Symposium on Logic Programming*, February 1984, 211-217.

83. Swinehart, D. C., P. T. Zellweger and R. B. Hagmann, "The Structure of Cedar", *ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, June 1985, 230-244.

84. Teitelbaum, T. and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", *Communications of the ACM 24, 9* (September 1981), 563-573.

85. Terwilliger, R. B. and R. 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 (also to appear in the *Journal of Systems and Software*), September 1986.

86. Terwilliger, R. B. and R. 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, 436-447.

87. Terwilliger, R. B. and R. H. Campbell, "PLEASE: Predicate Logic based ExecutAble SpEcifications", *Proceedings of the 1986 ACM Computer Science Conference*, February 1986, 349-358.

88. Terwilliger, R. B. and R. 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 (also to appear in the *Journal of Systems and Software*), September 1986.

89. Terwilliger, R. B. and R. H. Campbell, "PLEASE: a Language for Incremental Software Development", *Proceedings of the 4th International Workshop on Software Specification and Design*, April 1987, 249-256.

90. Terwilliger, R. B., "An Example of Knowledge-Based Development in ENCOMPASS", *Proceedings of the Third Annual Conference on Artificial Intelligence & Ada*, George Mason University, October 1987, 40-55.

91. Terwilliger, R. B., "ENCOMPASS: an Environment for Incremental Software Development using Executable, Logic-Based Specifications", Report No. UIUCDCS-R-87-1356 (Ph.D. Dissertation), Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1987.

92. "Reference Manual for the ADA Programming Language", American National Standards Institute/MIL-STD-1815A-1983, U.S. Dept. Defense, 1983.

93. *VADS Reference Manual*, Verdix Corporation, Chantilly, Virginia, 1986.

94. Waters, R. C., "The Programmer's Apprentice: A Session with KBEmacs", *IEEE Transactions on Software Engineering SE-11, 11* (November 1985), 1296-1320.

95. Weinberg, G. M. and D. P. Freedman, "Reviews, Walkthroughs, and Inspections", *IEEE Transactions on Software Engineering SE-10, 1* (January 1984), 68-72.

96. Wing, J. M., "Writing Larch Interface Language Specifications", *ACM Transactions on Programming Languages and Systems 9, 1* (January 1987), 1-24.

97. 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.

98. "Special Issue: Proceedings of VERkshop III -- A Formal Verification Workshop", *Software Engineering Notes 10, 4* (February 1984).

99. *Proceedings of the 4th International Workshop on Software Specification and Design*, April 1987.

100. "Proceedings of the NRL Invitational Workshop on Testing and Proving: Two Approaches to Assurance", *ACM Software Engineering Notes 11, 5* (October 1986), 63-85.

101. "Special Issue on Rapid Prototyping: Working Papers from the ACM SIGSOFT Rapid Prototyping Workshop", *Software Engineering Notes 7, 5* (December 1982).