ISLET: a Program/Proof Editor to Support the Vienna Development Method

Robert B. Terwilliger

CU-CS-401-88 June 1988

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

ISLET: a Program/Proof Editor to Support the Vienna Development Method

Robert B. Terwilliger

Department of Computer Science
University of Colorado at Boulder
ECOT7-7, Campus Box 430
Boulder, CO 80309-0430
(303) 492-7514

email: terwilli@boulder.colorado.edu

Technical Report CU-CS-401-88 (June 1988)

Preprint June 14, 1988

ISLET: a Program/Proof Editor to Support the Vienna Development Method

Robert B. Terwilliger

Department of Computer Science
University of Colorado
ECOT7-7, Campus Box 430
Boulder, Colorado 80309-0430
(303) 492-3903
email: terwilli@boulder.colorado.edu

Abstract

Traditional methods do not ensure the production of correct software. VDM (the Vienna Development Method) is a technique which has been used in industrial settings to enhance the development process. ENCOMPASS is an environment to support the incremental construction of Ada® programs using executable specifications, testing based verification methods, and formal techniques similar to VDM. One of the most important tools in ENCOMPASS is ISLET, a program/proof editor which supports the construction of formal specifications and their incremental refinement into verified implementations. As the specifications are entered and refined the syntax and static semantics are constantly checked. In ISLET, the refinement process can be viewed as the construction of a proof in the Hoare calculus. Some steps in the proof generate verification conditions in the underlying first-order logic. These verification conditions are first subjected to a number of simple (and inexpensive) proof tactics, which in our experience can certify a large percentage of the verification conditions generated at a very low cost. If a set of verification conditions can not be proved using these methods alone, then they can be submitted to a peer review process, a series of tests, or a more powerful (and expensive) mechanical proof system. The combined use of these methods supports the *rigorous* development of programs. Most of the verification conditions will be certified using simple methods; however, more expensive techniques are available on demand. Parts of a system may be developed using completely mechanized formal methods, while other, less critical parts may use less expensive techniques. In this paper, we give an overview of ENCOMPASS and ISLET and present an example of development using the editor.

1. Introduction

The production of software is both difficult and expensive. The rising cost of software relative to hardware in complex systems has led some to speak of the "software crisis" [18]. One of the largest problems is *quality*; many of the systems produced do not satisfy their purchasers in either functionality, performance or reliability. ENCOMPASS [61, 62, 66, 67, 69] is an environment which addresses the software quality problem using a combination of executable specifications, peer review, testing, and formal techniques similar to the Vienna Development Method. One of the most important tools in ENCOMPASS is ISLET, a language-oriented program/proof editor which supports the construction of formal specifications and their incremental refinement into verified implementations. In ISLET, the refinement process can be viewed in two ways: as the development of a program, or as the construction of a proof of correctness. Form the proof view, some refinements generate *verification conditions* which must be

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

This research was supported in part by NASA Grant NAG 1-138 and NSF Grant CCR-8809418.

true for the step to be correct. ISLET encorporates a number of simple methods which can inexpensively certify a large percentage of the verification conditions generated. In this paper we give an overview of ENCOMPASS and ISLET and present an example of development using the editor.

Depending on the model of software development used, the software quality problem can be subdivided in a number of ways. At first, a system exists only as an idea in the minds of its users or purchasers. In many models, 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 [20]. 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 [20]. Creating a valid specification is a difficult task; the users of the system may not really know what they want, and they may be unable to communicate their desires to the development team. Formal specifications may be an ineffective medium for communication between customers and developers, but natural language specifications are notoriously ambiguous and incomplete. It has been suggested that *prototyping* and the use of *executable specification languages* can enhance the communication between customers and developers [1, 14, 21, 30, 73, 75]; providing prototypes for experimentation and evaluation should enhance the validation process.

The specification may not be executable, or it may not be acceptably efficient; 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 [20]. Many different techniques can be used to determine if an implementation satisfies a specification [74]. For example, *testing* can be used to check the operation of an implementation on a representative set of input data [34, 47]; however, in general, a program cannot be tested on all possible inputs. In a *peer review* process, the specification and implementation are inspected, discussed and compared by a group of knowledgeable personnel [19, 72]; 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 [33, 42, 48]; however, with the current state of verification technology, many widely used languages are not completely verifiable. Many feel that no one technique alone can solve the verification problem [16, 17, 53]; therefore, methods which combine a number of techniques have been proposed [51, 57, 58, 76].

One solution to the verification problem is VDM (the Vienna Development Method). VDM supports the top-down development of software specified in a notation suitable for formal verification [6, 8, 9, 13, 35-37, 54, 59]. In this method, components are first written using a combination of conventional programming languages and predicate logic. These *abstract components* are then incrementally refined into components in an implementation language. The refinements are performed one at a time, and each is verified before another is applied; therefore, the final components produced by the development satisfy the original specifications. Since each refinement step is small, design and implementation errors can be detected and corrected sooner and at lower cost. VDM is used in industrial environments to enhance the development process [9, 54, 59]. In this type of environment, the method is not typically applied in all its formality; 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 of these tools is justified, and environments to support VDM are being constructed [7].

ENCOMPASS [61, 62, 66, 67, 69], is an integrated environment to support incremental software development in a manner similar to VDM. The general approach in ENCOMPASS is to address the validation problem using rapid prototyping with executable specifications and to attack the verification problem using a combination of peer review, testing, and formal techniques similar to VDM. We can view ENCOMPASS as supporting a development paradigm which extends VDM with the use of executable specifications and testing-based verification. ENCOMPASS 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 [63-65, 68], a wide spectrum executable specification and design language. PLEASE specifications may be used in proofs of correctness; they also may be transformed into prototypes which use logic programming techniques to "execute" pre- and post-conditions. These prototypes can enhance both the validation and verification processes.

IDEAL is an environment for programming-in-the-small within ENCOMPASS. The central tool in IDEAL is ISLET, a program/proof editor which supports the creation of PLEASE specifications and their incremental refinement into Ada implementations. As the specifications are entered and refined the syntax and static semantics are constantly checked. In ISLET, the refinement process can be viewed as the construction of a proof in the Hoare calculus [32, 42]. Some steps in the proof 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. IDEAL also provides facilities to verify the correctness of a refinement using peer review or testing techniques.

IDEAL and ISLET support the *rigorous* [36] 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. Development using PLEASE is not limited to the subset of the implementation language for which formal descriptions have been created. Systems can be constructed using all the implementation language features; however, only the parts of the system consisting entirely of formally described constructs can be mathematically verified or prototyped using logic programming techniques. We are continuely extending the set of constructs formally defined. We feel this is a reasonable approach to introducing formal methods into practical development. We believe that formal understanding of components will prove useful even if entire systems do not yield to such methods.

In the remainder of this paper we describe ENCOMPASS and ISLET in more detail and give an example of development using the editor. In section two we give a more thorough overview of ENCOMPASS including the PLEASE executable specification language, the development paradigm ENCOMPASS is designed to support, and the architecture of IDEAL. In section three we discuss ISLET in more detail including its major sub-components, an algebraic simplifier and a set of simple proof methods. In section four we present the development of a small procedure using the editor; we follow the development from the creation of a formal specification through its refinement into an implementation. We discuss in detail the commands used by the programmer to control this process, the verification conditions generated by the editor, and the ability of the simple proof methods to certify these VCs. In section five we describe the status of the system and in section six we summarize and draw some conclusions from our experience.

2. ENCOMPASS

ENCOMPASS owes much to the work of previous researchers in programming environments [31,49,77,78] and systems for the formal verification of programs [25,28,43,56]. As far as we know, ENCOMPASS is unique in its combination of an incremental verification system and executable specifications based on resolution theorem proving with a test harness and an environment for programming-in-the-large. The ENCOMPASS development paradigm draws on many previous methods [26,36,50], including transformational programming [2,55] and "proofs as programs" [3,46]. In fact, ENCOMPASS can be seen as a transformational approach [4]. ENCOMPASS is best viewed as an attempt to integrate executable specifications and incremental verification into the traditional life-cycle. ENCOMPASS 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 specifically designed to support such a methodology.

The foundation of ENCOMPASS is the PLEASE [63-65,68] executable specification language. The design of PLEASE was greatly influenced by previous work on software specification [23,79], most notably that involving formal techniques [1,5,24,27,30,52]. One approach to formal specification involves annotating a program with formulae written in predicate logic. *Annotations* formally state conditions that must hold at different points in the program's execution. For example, annotations can specify properties that must hold for all values of a type, invariants for loops, or the pre- and post-conditions for procedures. At the beginning of our work we decided that PLEASE would use predicate logic annotations to specify software. We also decided that it must be possible to automatically construct executable prototypes from these specifications.

Unfortunately, there is a conflict between specification power and prototype efficiency. A fairly powerful system would use full first-order, predicate logic for annotations which would be executed using a resolution theorem prover; however, the performance of these prototypes would be very poor. The emergence of logic programming as a technology [12,15,22,41,60], most notably Prolog [11], suggests that these techniques may provide a good compromise. Although in one sense not as powerful as full first-order logic, Prolog allows much more efficient implementation techniques to be used. By restricting the annotations to a logic with an efficient, Prolog-style implementation, reasonable specification power is combined with implementation efficiency.

We have designed and implemented a number of logic-based executable specification languages during the course of our research, and our approach has undergone significant modification. We developed the initial version

of PLEASE using a Pascal derivative [63]. However, the bulk of our work (including that reported in this paper) has been performed using Ada [70] as the base language [64, 65]. The choice of Ada proved successful for a number of reasons, the most significant being the existence of a large body of work on the Ada-based specification language ANNA [44, 45]. PLEASE/Ada may be considered a subset of ANNA specifically chosen to support VDM. PLEASE/Ada restricts the logic used in annotations to Horn clauses so that pre- and post-conditions can be translated into Prolog procedures [64, 67].

The latest member of the PLEASE family is PK/C++ [68] (Please Kernel on C++). Our major goal for the language is to produce an implementation practical enough that we can develop significant software using the methods we have described. PK/C++ differs from its predecessor by being based on C++ rather than Ada, having an operational as well as declarative semantics, and being based on flat (unification but no backtracking) rather than standard Prolog. We feel these changes will help us make significant progress towards our goal.

We can better understand PLEASE and ENCOMPASS after considering the development paradigm they were designed to support.

2.1. Incremental Software Development

ENCOMPASS is designed to support a *phased* or *waterfall* life-cycle [20], 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. In ENCOMPASS, a development passes through the phases planning, requirements definition, validation, refinement and system integration.

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, peer 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.

We have developed a number of tools to support this development paradigm.

2.2. The IDEAL Environment

The tools in ENCOMPASS can be divided into two main groups: support for programming-in-the- large, including the configuration and project management systems [10,38]; and IDEAL (Incremental Development Environment for Annotated Languages), an environment for programming-in-the-small using PLEASE. IDEAL is concerned with the development of single modules. It provides facilities to create PLEASE specifications, construct prototypes from these specifications, validate the specifications using the prototypes produced, refine the validated specifications into implementations, and verify the correctness of the refinement process.

Figure 1 shows the top-level architecture of IDEAL, which contains four tools: TED [29], a proof management system that is interfaced to a number of theorem provers; ISLET (Incredibly Simple Language-oriented Editing Tool), a prototype program/proof editor; a tool to support the construction of executable prototypes from PLEASE specifications; and a test harness. The user interacts with these tools through a common interface. The tools in IDEAL operate on components that are stored in a *module data base*. The module data base contains five types of components: symbol tables, proofs, source code, load modules and test cases.

A set of *symbol tables* represent the PLEASE specifications and Ada implementations being developed. These symbol tables are displayed and manipulated by ISLET, which can be used to create PLEASE specifications and incrementally refine them into Ada implementations. Some refinments may generate verification conditions; these can be reformated as *proofs* which serve as input for TED. Using TED, the user can structure the proof into a

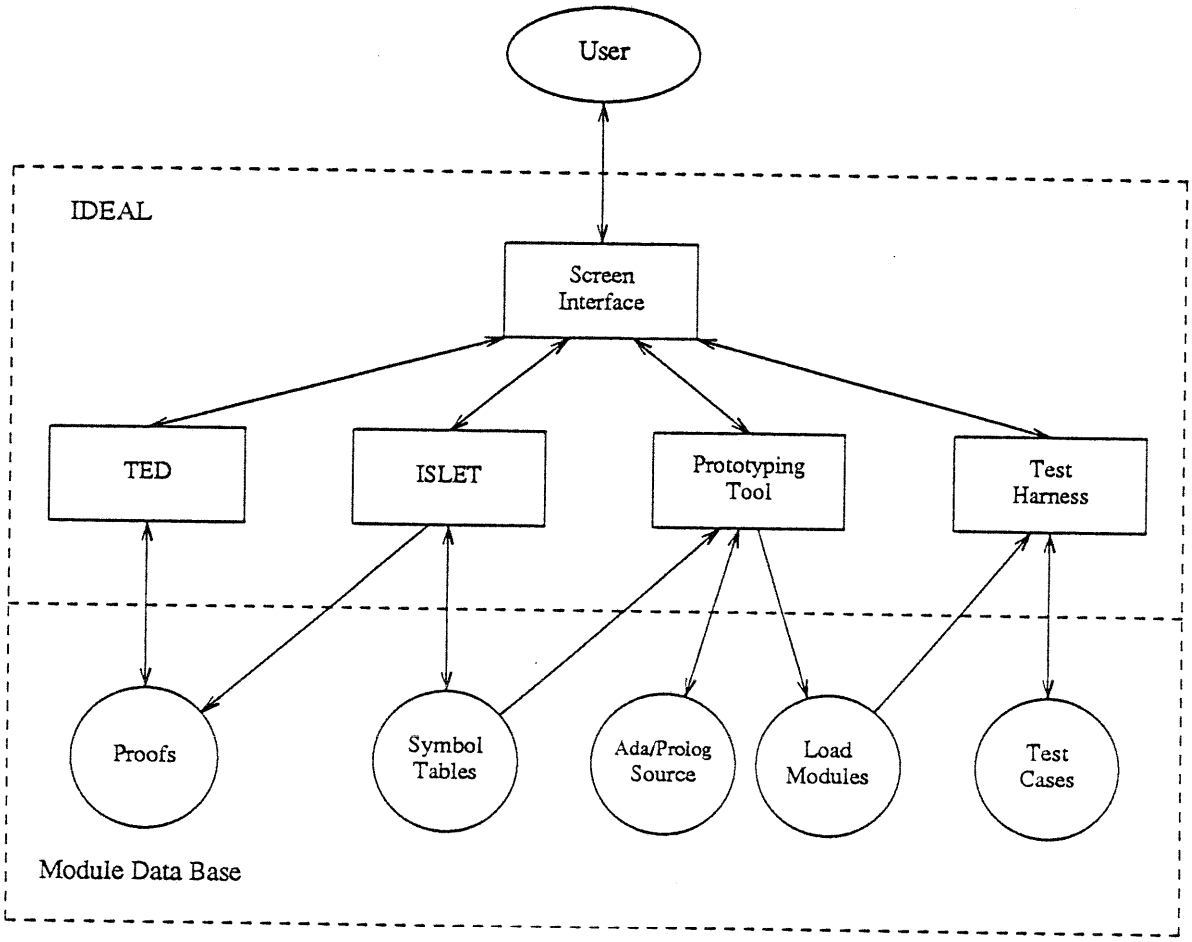

Figure 1. Architecture of IDEAL

number of lemmas and bring in pre-existing theories. The symbol tables also serve as input for the prototyping tool, which uses them to produce executable prototypes from PLEASE specifications. The *source code* for the prototypes is written in a combination of Prolog and Ada and utilizes a number of run-time support routines in both languages. The *load modules* produced from both prototypes and final implementations are used by the test harness. From the test harness, the user can invoke commands to manipulate *test cases*. Commands are available to: edit or browse the input for a test case; generate output for a test case; or run a program and compare the results with output that has been previously checked for correctness.

The central tool in IDEAL is ISLET. It not only manipulates the symbol tables representing specifications and implementations, but provides a user interface and, in a sense, controls the entire development process.

3. 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 [32, 42]. The refinement process is a sequence of atomic transformations, which can be grouped into design transformations. A *design transformation* implements a choice of data structure or algorithm; for example, whether to use a hash table or B-tree to implement a data base. *Atomic transformations* are the smallest distinguishable changes to the system; in ISLET, editor commands are atomic transformations. From the program view, an atomic transformation changes an unknown statement into a particular language construct; from the proof view, an atomic transformation adds more steps to an incomplete proof. From the program view, defining a predicate adds a new construct to the program; from the proof view, defining a predicate adds new axioms to the first-order theory on which the proof is based.

Figure 2 shows the architecture of ISLET. The user interacts with ISLET through a simple language-oriented editor similar to [56]. The editor provides commands to add, delete, and refine constructs; as the program/proof is incrementally constructed, the syntax and semantics are constantly checked. The editor also controls the other components: an algebraic simplifier, a number of simple proof procedures, and an interface to TED. Many steps in the refinement process generate verification conditions in the underlying first-order logic. These verification conditions are first simplified algebraically and then subjected to a number of simple proof tactics. These methods can handle a large percentage of the verification conditions generated. If a set of verification conditions can not be proved using these methods alone, the TED interface is invoked to create a proof in the proper format.

TED can then be invoked in an attempt to prove the verification conditions. Using TED is very expensive, both in system resources and user time; however, many complex theorems can be proved with its aid. The algebraic simplification and simple proof tactics used in ISLET are very inexpensive; however, they are not very powerful. The combined use of these two methods supports the *rigorous* [36] development of programs. Most of the verification conditions will be proven using inexpensive methods; those that are expensive to formally verify may be proven immediately, deferred until a later time, or certified using peer review or testing techniques. Parts of a sys-

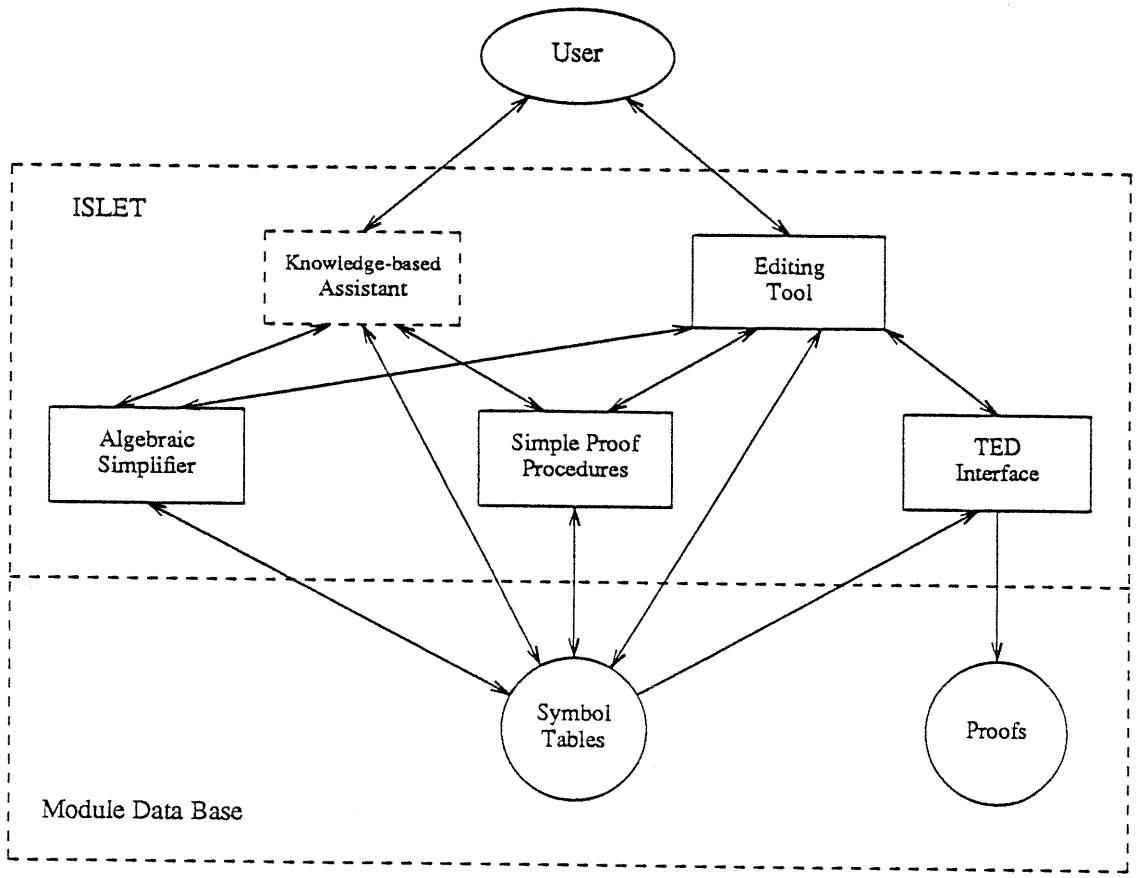

Figure 2. Architecture of ISLET

tem may be developed using completely mechanized formal methods, while other, less critical parts may use less expensive techniques.

The algebraic simplifier is implemented as a term rewriting system [40, 52]; it contains a knowledge-base of rules which are assumed to be convergent. The simple proof procedures rely on a knowledge base containing information such as: if the formulae F_1 and F_2 are equivalent under renaming of variables, then the formula $F_1 \supset F_2$ is always true. Other rules implement simple knowledge of equality; for example, if F(X) and X=c are both true then so is F(c). At present, it is difficult to examine, analyze or change the contents of these knowledge bases; for example, algorithms exist to determine if a set of rewrite rules are convergent, but they are not implemented in ISLET.

We are developing tools to correct these deficiencies.

The user can also interact with ISLET through a *knowledge-based assistant* [66] (currently under development) based on deductive synthesis techniques. During the refinement process, the user can ask the assistant for advice on how to implement an undefined construct. The assistant attempts to solve this problem by first searching for values that satisfy the pre- and post-conditions for the construct and then synthesizing Ada code to set these values. The assistant can access the information stored in the symbol table and invoke the algebraic simplifier and simple proof procedures. The assistant also contains a library of *program schemas* which can be instantiated to produce code fragments. The user can browse this library and instantiate schemas with the aid of the assistant; he can then use the instantiated schema in a refinement.

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

4. An Example of Software Development

Assume that a programmer must implement a procedure that takes a natural number as input and produces its factorial as output. The programmer first creates an empty module data base and then invokes IDEAL, which produces a display showing the empty module. The programmer then invokes ISLET to specify the procedure. Figure 3 shows the completed specification, which includes both the pre- and post-conditions for the procedure and the definition of the *is_fact* predicate. This figure, and the others in this section, show the actual output from the current implementation; therefore, they do not always exactly follow Ada syntax.

The specification defines a package *factorial*, which provides a procedure by the same name. In PLEASE, procedures are defined using pre- and post-conditions which are designated by *in(...)* and *out(...)* respectively. The pre-condition for a procedure specifies the conditions the input data must satisfy before procedure execution begins. The pre-condition for *factorial* is *true*; the type declarations for the parameters give all the requirements for the input. The post-condition for a procedure states the relationship the input and output data must satisfy after procedure execution has completed. The post-condition for *factorial* is *is_fact(x,y)*; the predicate *is_fact* must be true of the parameters to *factorial* after execution is complete. The predicate *is_fact* is not pre-defined; it was developed as part of the *factorial* specification. The definition of *is_fact* states that x factorial is equal to y if x equals zero and

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

package factorial is

    --: predicate is_fact( x : inout natural ; y : inout natural ) is true if
    --:       t1 : natural ;
    --: begin
    --:       x = 0 and y = 1
    --:           or
    --:       is_fact(x - 1, t1) and y = t1 * x ;
    --: end is_fact ;

    procedure factorial( x : in natural ; y : out natural ) ;
         --| where in( true ) and
         --|       out( is_fact(x, y) ) ;


end factorial ;









:
```

Figure 3. ISLET display showing *factorial* specification

y equals one, or if *x* minus one factorial is equal to *t1* and *y* equals *t1* times *x* (in other words, *is_fact(x,y)* is true if $(x = 0 \wedge y = 1) \vee ((x\text{-}1)! = t1 \wedge y = t1 * x)$).

The top line of the display gives a menu of ISLET commands. ISLET has a *focus of attention* which is always on a particular symbol table scope; for example, in Figure 3 ISLET's focus of attention is on the *factorial* package. The visible objects are the *factorial* procedure and the *is_fact* predicate. The *open* command changes the focus of attention to an inner scope; for example, the command "open! procedure factorial." would shift the focus of attention to the body of the *factorial* procedure[1]. The *clos* command shifts the focus to the containing scope, while the *display* command presents the focus of attention on the screen. The *decl* command allows declarations, and the

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

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

As the programmer enters the specification, the syntax and static semantics are checked for correctness; unfortunately, at present the granularity is somewhat coarse. For example, the programmer must enter the definition of *is_fact* as a unit; if a mistake is made the entire definition must be re-entered. Also, the definition of *is_fact* must be entered before the pre- and post-conditions for *factorial*. This is because *is_fact* is referenced in the post-condition; an undeclared identifier error would result if the order of entry were reversed. The utility of ISLET would be dramatically increased by a finer grained implementation; for example, one based on an editor, such as Epos [39], that allows an arbitrary number of text-oriented commands to be performed before the syntax and semantics are checked.

Figure 4 shows the ISLET display as the programmer opens the factorial procedure to begin the refinement process. At this point, the procedure consists of an unknown statement sequence, denoted by *unknown_0*, surrounded by assertions *true* and *is_fact(x,y)*. In PLEASE, assertions are described by lines beginning with the symbol "--|". Assertions are formulae which must be true whenever execution reaches that point in the program. For example, *true* must hold whenever execution reaches the point immediately preceding *unknown_0* (which will later be refined into a program construct) and *is_fact(x,y)* must be true whenever execution reaches the point immediately following *unknown_0*. This is equivalent to saying that *unknown_0* has a pre-condition of *true* and a post-condition of *is_fact(x,y)*. The goal of the refinement process is to produce an implementation of *unknown_0* which is correct with respect to these pre- and post-conditions. This problem is simplified by the fact that the factorial calculation can be divided into two cases: if the input is 0 then the result is 1, otherwise more computation is needed.

With this in mind, the programmer refines the unknown statement sequence into an *if-then-else*; he types the command "refine! 0 if x=0.", which can be read as: refine unknown zero into an *if-then-else* on condition *x* equal to zero. In ISLET, each refinement step corresponds to at most two proof steps in the Hoare calculus: a step using the rule for the appropriate language construct, and possibly a step using the consequence rule. For example, the current refinement uses the rule for the *if-then-else* construct, but does not make use of the consequence rule;

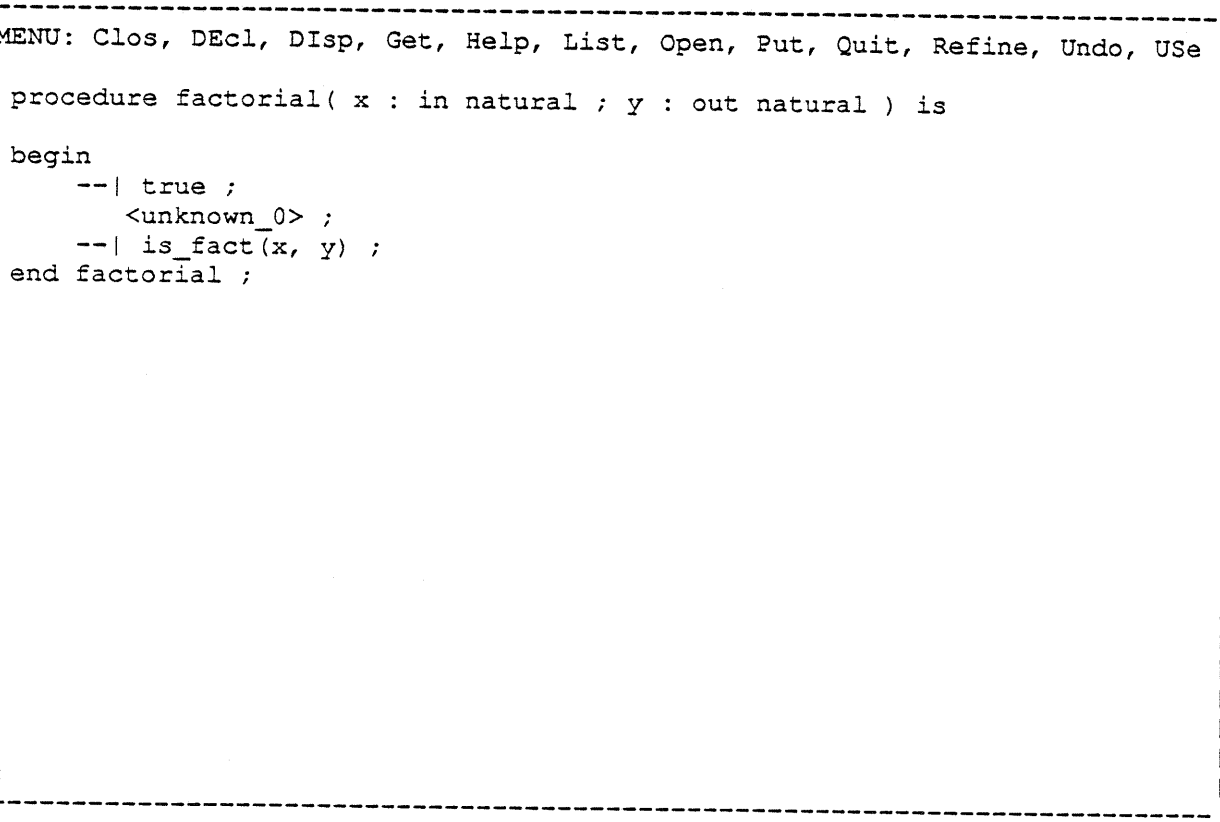

Figure 4. ISLET display at beginning of refinement process

therefore, no verification conditions are generated.

Figure 5 shows the ISLET display after the refinement is compete; *unknown_0* has been transformed into an *if-then-else* with an unknown at each branch. ILSET generates all the new assertions necessary to annotate the refinement. The refinement process now has two sub-goals. An implementation of *unknown_1* must be found which is correct with respect to pre-condition *true and* $x = 0$ and post-condition *is_fact(x,y)*. Similarly, *unknown_2* must be refined into a statement sequence which is correct with respect to *true and not* $x = 0$ and *is_fact(x,y)*. These goals can be pursued in any order: the programmer can perform a number of refinements on *unknown_1*, switch his attention to *unknown_2*, and then return to finish *unknown_1*. Fortunately, this is not necessary: *unknown_1* has a simple implementation.

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

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

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

:
```

Figure 5. ISLET display after initial refinement of *factorial*

Knowing that the factorial of zero is one, the programmer refines *unknown_1* into a statement assigning one to *y*; Figure 6 is produced by ISLET after the command "refine! 1 y := 1.". The proof of this refinement step makes use of both the Hoare axiom for assignment statements and the consequence rule; therefore, verification conditions are generated. The algebraic simplifier is able to reduce the verification conditions by realizing that for any formula F, true and F is equivalent to F. Although the verification conditions are true, the simple proof procedures are not able to certify this. The problem is that the current implementation does not add user definitions to the rule base; the proof methods have no knowledge of *is_fact*. We plan to correct this problem in the future.

At this point the programmer examines the verification conditions; ISLET has created input for TED, which can be invoked if the programmer desires. In this case the programmer is convinced that the verification conditions

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

Verification conditions are:

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

Simplified verification conditions are:

     x = 0 =>
          is_fact(x, 1)

Trying simple proof procedure ...

Simple proof procedure failed - generating ted file ...


Should I invoke TED (y/n) ? n

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

Figure 6. ISLET display showing verification conditions for first assignment

are correct; he decides to wait until after the refinement process is complete to formally certify them. ISLET displays the completed refinement and the development process continues. The programmer decides to implement the *else* branch using a sequence consisting of a *while* loop and its initialization. He realizes he needs a loop counter and declares a variable *i*.

In ISLET, a loop has both an *invariant*, which is maintained by the body, and a *condition*, which controls termination. If the invariant is true before the loop begins execution, then both the invariant and the negation of the condition will be true when the loop terminates. The programmer's strategy involves the invariant *is_fact(i,y)* and loop condition $i \neq x$. The program will first initialize the invariant for an low value of i and then maintain the invariant as i is counted up to x. On loop termination i will be equal to x and *is_fact(x,y)* will be true.

First, the programmer enters the code to initialize the invariant. Initialization consists of two statements: one assigning 1 to i and one assigning 1 to y. The verification conditions for the first assignment can be certified using

simple methods; the second assignment generates verification conditions:

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

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

The programmer must now refine an unknown with pre-condition *is_fact(i,y)* and post-condition *is_fact(x,y)* into the *while* loop. He types the command "refine! 4 while i /= x.", to refine *unknown_4* into a *while* loop with condition *i* not equal to *x*; Figure 7 shows the ISLET display. ISLET generates verification conditions for this refinement based on both the rule for *while* loops and the consequence rule; the current ISLET implementation only verifies partial correctness. The verification conditions can be algebraically simplified, and the result can be proved using basic knowledge of equality: if two terms are equal, then we can substitute one for the other in any formula without changing its meaning.

The programmer realizes that the body of the loop must both increment the loop counter and update the calculated quantity. Figure 8 shows the ISLET display after he refines the body into a statement sequence. The

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

Verification conditions are:

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

Simplified verification conditions are:

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

Trying simple proof procedure ...

Simple proof procedure successful
```

Figure 7. ISLET display showing verification conditions for loop creation

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

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

    i : natural ;

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

:
```

Figure 8. ISLET display showing decomposition of loop body

programmer then refines *unknown_8* into an assignment which increments *i* and produces verification conditions:

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

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

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

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

The implementation of *factorial* is now complete; Figure 9 shows the completed procedure. The body of *factorial* is completely annotated; in other words, there is an assertion both before and after each executable statement.

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

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

    i : natural ;

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

:
```

Figure 9. ISLET display showing completed implementation of *factorial*

The assertions plus the executable statements form a proof in the Hoare calculus. Before the proof is really complete, the programmer must certify the verification conditions which did not yield to the simple methods implemented in ISLET; these are:

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

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

In IDEAL, the programmer can verify these conditions using any combination of peer review, testing, or formal techniques. The formal techniques used can range from a "proof sketch" done with pencil and paper to a complete, mechanized proof performed with the aid of TED. In this case, the programmer correctly assumes that TED can certify the verification conditions and invokes the tool on the appropriate files. TED certifies the verification conditions and the development is complete.

5. System Status

The ENCOMPASS environment has been under development since 1984; a prototype implementation has been running under Berkeley UNIX® on Sun workstations since 1986. The prototype is written in a combination of C, Csh, Prolog and Ada and supports the construction of software using the Verdix Ada Development System [71]. This prototype includes all the tools necessary to support software development using PLEASE: an initial version of ISLET; software which automatically translates PLEASE specifications into Prolog procedures and generates the support code necessary to call these procedures from Ada; the run-time support routines and axiom sets for a number of pre-defined types; and interfaces to the ENCOMPASS test harness and TED. PLEASE and ENCOMPASS have been used to develop a number of small programs, including specification, prototyping, and mechanical verification. The prototype implementation of ISLET is written in Prolog and performs exactly as described in section four of this paper. An experimental implementation of the knowledge-based assistant has been written in Prolog. It is not a completely general tool, but does perform the deductions described in [66]. Work is now underway

UNIX® is a trademark of AT&T Bell Laboratories.

to integrate the assistant into the full ISLET implementation.

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 percent of the verification conditions generated during refinement. For example, [64] presents a design transformation consisting of twenty-six steps, only two of which generated verification conditions which could not be certified by the simple methods. The example presented in [62] 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 [64] 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.

6. Summary and Conclusions

Traditional methods do not ensure the production of correct software. VDM [8, 36, 54] 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 [61, 62] 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.

In ENCOMPASS, software is first specified using a combination of natural language and PLEASE, [63-65, 68] 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. As PLEASE specifications are both executable and formally based, the equivalence between specification and implementation can be determined using either testing or proof techniques.

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. The most important tool in IDEAL is ISLET, a language-oriented program/proof editor which views the refinement process as the construction of a proof in the Hoare calculus [32, 42]. Many steps in the refinement process generate verification conditions in the underlying first-order logic. In ISLET, these verification conditions are first simplified algebraically and then 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.

ISLET supports the *rigorous* [36] 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.

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. Although the current implementation of ISLET is quite limited, it demonstrates much of the potential of such tools. 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 tools similar to ISLET will greatly enhance the specification, design and development of software.

7. 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. Bates, J. L. and R. L. Constable, "Proofs as Programs", *ACM Transactions on Programming Languages and Systems 7, 1* (January 1985), 113-136.
4. 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.
5. 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.
6. Bjorner, D. and C. B. Jones, *Formal Specification and Software Development*, Prentice-Hall, Englewood Cliffs, N.J., 1982.
7. 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.
8. Bjorner, D., "On The Use of Formal Methods in Software Development", *Proceedings of the 9th International Conference on Software Engineering*, 1987, 17-29.
9. 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.
10. 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.
11. Clocksin, W. F. and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.
12. Cohen, J., "A View of the Origins and Development of Prolog", *Communications of the ACM 31*, 1 (January 1988), 26-36.
13. Cottam, I. D., "The Rigorous Development of a System Version Control Program", *IEEE Transactions on Software Engineering SE-10, 3* (March 1984), 143-154.
14. Davis, R. E., "Runnable Specification as a Design Tool", in *Logic Programming*, Clark, K. L. and S. A. Tarnlund (editor), Academic Press, London, 1982, 141-149.
15. Davis, R. E., "Logic Programming and Prolog: A Tutorial", *IEEE Software 2, 5* (September 1985), 53-62.
16. 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.
17. Dijkstra, E. W., "Structured Programming", in *Software Engineering Principles*, Buxton, J. N. and B. Randall (editor), NATO Science Committee, Brussels, Belgium, 1970.
18. Dijkstra, E. W., "The Humble Programmer (Turing Award Lecture)", *Communications of the ACM 15*, 10 (October 1972), 859-866.
19. Fagan, M. E., "Advances in Software Inspections", *IEEE Transactions on Software Engineering SE-12, 7* (July 1986), 744-751.
20. Fairley, R., *Software Engineering Concepts*, McGraw-Hill, New York, 1985.
21. Futatsugi, K., J. Goguen, J. Meseguer and K. Okada, "Parameterized Programming in OBJ2", *Proceedings of the 9th International Conference on Software Engineering*, 1987, 51-60.
22. Gallaire, H., J. Minker and J. Nicolas, "Logic and Databases: A Deductive Approach", *ACM Computing Surveys 16*, 2 (June 1984), 153-185.
23. Gehani, N. and A. D. McGettrick, eds., *Software Specification Techniques*, Addison Wesley, Reading, Massachusetts, 1986.
24. 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.
25. Goguen, J. and M. Moriconi, "Formalization in Programming Environments", *IEEE Computer 20*, 11 (November 1987), 55-64.
26. Gries, D., *The Science of Programming*, Springer-Verlag, New York, 1981.
27. Guttag, J. V., J. J. Horning and J. M. Wing, "The Larch Family of Specification Languages", *IEEE Software 2, 5* (September 1985), 24-36.
28. 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.
29. 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.
30. Henderson, P., "Functional Programming, Formal Specification, and Rapid Prototyping", *IEEE Transactions on Software Engineering SE-12, 2* (February 1986), 241-250.
31. Henderson, P. B., ed., *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ACM, December 1986.
32. Hoare, C. A. R., "An Axiomatic Basis for Computer Programming", *Communications of the ACM 12, 10* (October 1969), 576-580.

33. Hoare, C. A. R., "An Overview of Some Formal Methods for Program Design", *IEEE Computer 20*, 9 (September 1987), 85-91.

34. Howden, W. E., "The Theory and Practice of Functional Testing", *IEEE Software 2, 5* (September 1985), 6-17.

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

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

37. Jones, C. B., *Systematic Software Development Using VDM*, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

38. 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.

39. 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.

40. Knuth, D. E. and P. E. Bendix, "Simple Word Problems in Universal Algebra", in *Computational Problems in Abstract Algebra*, Leech, J. (editor), Pergamon, New York, 1970, 263-297.

41. Krall, A., "Implementation of a High-Speed Prolog Interpreter", *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, June 1987, 125-131.

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

43. 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.

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

45. Luckham, D. C., D. P. Helmbold, S. Meldal, D. L. Bryan and M. A. Haberler, "Task Sequencing Language for Specifying Distributed Ada Systems, TSL-1", Report No. CSL-TR-87-334, Computer Systems Laboratory, Stanford University, July 1987.

46. Martin-Lof, P., "Constructive Mathematics and Computer Programming", *Proceedings of the 6th International Congress for Logic, Method, and Philosophy of Science*, 1982, 153-175.

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

48. Mili, A., J. Desharnais and J. R. Gagne, "Formal Models of Stepwise Refinement of Programs", *ACM Computing Surveys 18*, 3 (September 1986), 231-276.

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

50. 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.

51. Mills, H. D., M. Dyer and R. Linger, "Cleanroom Software Engineering", *IEEE Software 4*, 5 (September 1987), 19-25.

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

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

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

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

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

57. 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.

58. Selby, R. W., V. R. Basili and F. T. Baker, "Cleanroom Software Development: An Empirical Evaluation", *IEEE Transactions on Software Engineering SE-13*, 9 (September 1987), 1027-1037.

59. 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.

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

61. 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.

62. 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.

63. 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.

64. 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.

65. 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.

66. 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.

67. 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.

68. Terwilliger, R. B. and P. A. Kirslis, "PK/C++: an Object-Oriented, Logic-Based, Executable Specification Language", Technical Report CU-CS-400-88, Department of Computer Science, University of Colorado at Boulder, June 1988.

69. Terwilliger, R. B. and R. H. Campbell, "An Early Report on ENCOMPASS", *Proceedings of the 10th International Conference on Software Engineering*, April 1988, 344-354.

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

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

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

73. Zave, P., "The Operational Versus the Conventional Approach to Software Development", *Communications of the ACM 27, 2* (February 1984), 104-118.

74. "Theme Issue: Software Quality Assurance", *IEEE Software 4*, 5 (September 1987).

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

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

77. "Special Issue on Integrated Environments", *IEEE Software 4*, 6 (November 1987).

78. "Special Issue on Integrated Environments", *IEEE Computer 20*, 11 (November 1987).

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