PLEASE: a Language Combining Imperative and Logic Programming

Robert B. Terwilliger

CU-CS-381-87 November 1987

*Department of Computer Science, Campus Box 430, University of Colorado, Boulder, Colorado, 80309. This research supported in part by NASA grant NAG 1-138 and NSF grant CCR-8705162.

PLEASE: a Language Combining Imperative and Logic Programming

Robert B. Terwilliger

Department of Computer Science
University of Colorado
ECOT7-7, Campus Box 430
Boulder, Colorado 80309-0430
(303) 492-3903

Abstract

PLEASE is an Ada-based, executable specification language which supports a formal development method similar to VDM. Like ANNA , PLEASE allows software to be annotated with formulae written in predicate logic; annotations can be used in proofs of correctness and to generate run-time assertion checks. PLEASE differs from ANNA by restricting the logic used in annotations to Horn clauses; therefore, PLEASE specifications can also be transformed into prototypes which use Prolog to "execute" pre- and post-conditions. We believe the early production of executable prototypes will enhance the development process. We also believe the restriction to Horn clauses will allow us to build on the significant research being performed on logic-based programming languages and databases. In this paper we give an overview of PLEASE, present an example specification, describe the translation of PLEASE specifications into Prolog procedures, and discuss some of the techniques used to obtain acceptable efficiency in the resultant prototypes.

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. In many development methods, the first step is the creation of a *specification* which precisely describes the properties and qualities of the software to be constructed [13]. Many specification methods have been previously proposed, designed or put into use [1,3,15,16,18,19,32,47,48]. Unfortunately, with current methods there is no guarantee that the specification correctly or completely describes the customers desires; 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. It has been suggested that *prototyping* and the use of *executable specification languages* can enhance the communication between customers and developers [12,20,23,24,49]; providing prototypes for experimentation and evaluation should enhance the software development process.

One approach to 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, ANNA

This research was supported in part by NASA grant NAG 1-138 and NSF grant CCR-8705162.

[28, 29] is an annotated version of Ada. In ANNA, assertions can specify properties that must hold for all values of a type, invariants for loops, or the pre- and post-conditions for procedures. PLEASE [38, 41, 45] is similar to ANNA, but restricts the logic used in annotations to Horn clauses: a subset of predicate logic which is also the basis for Prolog [9, 10]. This approach allows pre- and post-conditions to be translated into Prolog procedures. We feel this approach will allow us to build upon the significant research being performed on logic-based programming languages [2, 8, 17, 26] and data bases [14, 31, 35]. We also feel that the combination of Horn clause and equational theories will support the construction of powerful knowledge-based tools [44]. PLEASE is part of ENCOMPASS, an environment which supports formal software development using methods similar to VDM [6, 22].

In section two of this paper we give an overview of this ENCOMPASS, and in section three we describe the design of PLEASE in more detail. In section four we present an example PLEASE specification, and in section five we discuss the translation from PLEASE specifications to Prolog procedures. In section six, we summarize, admit the actual status of the implementation, and draw some conclusions from our work so far.

2. VDM and ENCOMPASS

The Vienna Development Method (VDM) supports the top-down development of software specified in a notation suitable for formal verification [4, 6, 7, 11, 21, 22, 33, 36]. In this method, components are first written using a combination of conventional programming languages and predicate logic. 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. VDM is used in industrial environments to enhance the development process [7, 33, 36]. The method is in wide enough use that considerable resources are being devoted to the construction of environments for its support [5].

ENCOMPASS [39, 40, 42] is 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, the traditional life-cycle is extended to include a separate phase for user validation;

the design and implementation processes are also combined into a single refinement phase. In ENCOMPASS, a development passes through the phases planning, requirements definition, validation, refinement and system integration. In ENCOMPASS, software is specified using a combination of natural language and PLEASE [38,41,45]. PLEASE specifications may be used in proofs of correctness; they also may be transformed into prototypes which use Prolog [10] to "execute" pre- and post-conditions. These prototypes have many uses. To enhance validation, they may be subjected to a series of tests, or be delivered to the customers for experimentation and evaluation. Prototypes can be used to verify the correctness of refinements using testing techniques. Also, PLEASE prototypes can be used in experiments performed to guide the design process.

3. PLEASE

PLEASE is designed to satisfy a number of sometimes conflicting goals. First, it allows the implementation of software using a conventional (in other words imperative) programming language. At present, we are using Ada [46] as the base language. Second, PLEASE allows the specification of software using pre- and post-conditions written in predicate logic. Third, the language allows the rapid, automatic construction of executable prototypes from these specifications. Unfortunately, there is a conflict between the second and third goals. In one sense, the more powerful the logic used in the specifications, the less effort required to specify systems. However, the more powerful the logic, the greater the effort required to construct prototypes with acceptable efficiency.

For example, a fairly powerful specification method could use pre- and post-conditions written in the full first-order, predicate logic. Unfortunately, the validity problem for first-order logic is undecidable [27,30]. 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 [43]. A general purpose, 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 emergence of logic programming as a technology, most notably Prolog [10], suggests a good compromise. Although many implementations show significant deviations [37], a "pure" Prolog interpreter can be viewed as a resolution theorem prover [9,10]. By restricting the logic used to allow efficient Prolog implementations, reasonable specification power is combined with implementation efficiency.

[1] which are not guaranteed to terminate.

Prolog is much more efficient than general purpose theorem provers; to achieve this, several concessions were made. First, Prolog is based on *Horn clauses* [9, 10]. Horn clauses allow a much more efficient implementation, but represent only a subset of first-order logic. A limitation of Prolog is that there is no way to state that a predicate is not true for a particular value. 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 [35]. 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 being built to an incomplete knowledge-base. We believe it is necessary to distinguish between formulae that are provably true, provably false, or not provably either.

Prolog is also 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 the beginning of our work, PLEASE specifications were purely declarative; 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 to automatically construct a prototype from a purely declarative specification. PLEASE now allows 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 destroying the declarative semantics of the specification.

Another feature is that PLEASE supports the *rigorous* [22] development of programs: parts of a system can be developed using completely formal methods, while other, less critical parts are constructed using less expensive techniques. Development using PLEASE is not limited to the subset for which formal descriptions have been created[2]. Systems can be constructed using all the features of Ada, including task types and pointers; however, only the parts of the system consisting entirely of formally described constructs can be verified or prototyped using the Prolog translation 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 order to further clarify the concepts and design of PLEASE, we will present an example specification.

[2] for a more complete discussion of this issue see [43].

4. An Example Specification

For example, Figure 1 shows a simplified version of a PLEASE specification developed in cooperation with the Aracadia Project, a research effort into environments to support the notion of *process programming* [34]. The

```
type object_rec is record
    Name   : name_type   ;
    Object : obj_handle  ;
    Info   : info_type   ;
end record ;

task object_manager is

    --: property Objects : set of object_rec ;
    --|          where for all Obj1,Obj2 : object_rec =>
    --|                member(Obj1,Objects) and member(Obj2,Objects) and
    --|                ( Obj1.Name = Obj2.Name or Obj1.Object = Obj2.Object )
    --|                    -> Obj1 = Obj2  ;

    --: Obj0 : constant object_rec ;

    entry create( User : in user_type ; Name : in name_type ) ;
        --| where in ( cnot member((Name,_,_),Objects) and can_create(User,Name) ),
        --|       out( is_okinfo(no_info,User,create,(Info:info_type))
        --|            and is_initial_object((Obj:object_handle)) and
        --|            Objects = insert((Name,Obj,Info),in(Objects)) ) ;

    entry destroy( User : in user_type ; Name : in name_type ) ;
        --| where in ( Obj0.Name = Name and member(Obj0,Objects) and
        --|            can_destroy(User,Obj0) ),
        --|       out( Objects = delete(Obj0,in(Objects)) ) ;

    entry open( User : in user_type ; Name : in name_type ; Handle : out obj_handle ) ;
        --| where in ( Obj0.Name = Name and member(Obj0,Objects) and
        --|            can_open(User,Obj0) ),
        --|       out( (Obj:object_rec).Name = Name and
        --|            Obj.Object = Obj0.Object = Handle and
        --|            is_okinfo(Obj0.Info,User,open,Obj.Info) and
        --|            Objects = insert(Obj,(delete(Obj0,in(Objects)))) ) ;

    entry close( User : in user_type ; Name : in name_type ) ;
        --| where in ( Obj0.Name = Name and member(Obj0,Objects) and
        --|            can_close(User,Obj0) ),
        --|       out( (Obj:object_rec).Name = Name and Obj.Object = Obj0.Object and
        --|            is_okinfo(Obj0.Info,User,close,Obj.Info) and
        --|            Objects = insert(Obj,(delete(Obj0,in(Objects)))) );

end object_manager ;
```

Figure 1. PLEASE specification of object manager task

specification defines a task *object_manager* which supports the operations *create*, *destroy*, *open*, and *close*. For the purpose of specification, the task has a *virtual state* consisting of a set of object records. Each item of type *object_rec* has three fields: the name of the object, a pointer to the object, and some information on the objects history being kept by the manager. The set of object records has an *invariant* stating that no two names refer to the same object, and no two objects have the same name.

The specification makes use of *predicates* which are defined elsewhere. In PLEASE, a *predicate* definition syntactically resembles a procedure and may contain local type, variable, function or predicate definitions. Predicate definitions describe acceptable tuples in a format which has an simple translation into Prolog procedures. For example, the predicate *can_create(User,Name)* is true only if *User* is allowed to create an object called *Name*. Similarly, the predicate *is_okinfo(Oldinfo,User,Op,Newinfo)* is true only if *Newinfo* is the correct history information for an object which had history *Oldinfo* before *User* performed *Op*, and *is_initial_object* is true only for the "empty" object.

In Ada, inter-task communication is accomplished with *rendezvous* at entry points. *Object_manager* has an entry for each operation it provides; other tasks will call these entries to initiate a rendezvous. Each entry n *object_manager* has a pre-condition, which states the conditions necessary for a rendezvous to begin, and a post-condition, which states the conditions which must hold when the rendezvous is complete. In the specification, the state before execution begins is denoted by *in(...)*, while the state after execution has completed is denoted by *out(...)*. For example, the pre-condition for *create* states that there is no record for an object called *Name*[3] and that *User* has the proper permissions. The post condition for *create* states that an initialized object called *Name* has been created with the proper history information. Similarly, the pre-condition for *destroy* states that there is an object with the specified name and *User* is authorized to destroy it. The post-condition for *destroy* states that the object has been deleted from the set of object records[4].

5. Producing Prototypes from PLEASE Specifications

PLEASE specifications can be automatically translated into prototypes written in a combination of Prolog and Ada. The predicates and pre- and post-conditions are translated into Prolog procedures, which are executed by an

[3] *cnot* is a closed world not, which in this case is acceptable.

[4] Since *Obj0* is a constant, it must be the same in the input and output states.

interpreter in the current implementation. When a procedure is called, the *in* parameters are converted to the Prolog representation and the call is passed to the interpreter. When the Prolog procedure completes, the *out* parameters are converted to the Ada representation and the original call returns. Tools in the ENCOMPASS environment perform the translation and generate code to handle I/O and other implementation level details. The Prolog procedure 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.

Viewing Prolog as a theorem prover, execution can be seen as proving a formula of the form $\exists \overline{X}\ p(\overline{X})$ by finding an example $\overline{a}$ such that $p(\overline{a})$ is true. 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[5]. Briefly, we assume that for each function $f(\overline{x})$, there exists a relation $F(\overline{x},y)$ such that $f(\overline{x})=y$ iff $F(\overline{x},y)$. We unravel the formula $P(..f(\overline{x})..)$ into the equivalent formula $\exists t\,(F(\overline{x},t) \wedge P(..t..))$[6]. The prototypes produced by this translation process are *partially correct* [27,30] with respect to the specifications. In general it is not possible to extend our approach to total correctness [43]. 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.

The translation process assumes the existence of procedures for pre-defined operations such as addition and multiplication or simple manipulation of lists. It also is assumed that these procedures will run "forward" or "backward"; in other words, it is assumed that the procedures will find an acceptable tuple of the relation for any combination of instantiated or uninstantiated parameters. The efficiency of the prototypes produced is dependent on the efficiency of these procedures. For example, a naive approach would define the natural numbers in a manner similar to Peano's axioms [43]. This simple approach results in procedures which run forward and backward[7], but the time complexity of addition is $O(n)$[8] and multiplication is $O(n^2)$. To achieve acceptable efficiency we must utilize the $O(1)$ machine operations available to perform these operations. For natural numbers, this can be achieved by the

[5] For a more complete explanation of the translation process see [41,43].

[6] for other solutions to the equality problem for Prolog see [17,25].

[7] although they are not guaranteed to terminate.

[8] where n is the size of the numbers to be added.

hand coding of Prolog procedures. We feel that one of the keys to efficiency in our approach is the use of predefined operations which have been hand optimized for performance.

6. Summary, Status and Conclusions

Traditional methods do not ensure the production of correct software. VDM [6,22,33] is a method which is 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. ENCOMPASS [39,40,42] is an integrated environment which supports software development using executable specifications and formal techniques similar to VDM. In ENCOMPASS, software is first specified using a combination of natural language and PLEASE [38,41,45], 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.

PLEASE and ENCOMPASS have been under development since 1984; a prototype implementation has been operational since 1986. PLEASE and ENCOMPASS have been used to develop a number of small programs, including specification, prototyping, and mechanical verification. The subset of PLEASE now formally described includes: the *if*, *while*, and assignment statements; procedure calls with *in*, *out* or *in out* parameters; and a small set of types including natural numbers, lists and characters. The initial implementation of PLEASE runs under Berkeley Unix® on Sun workstations. The Prolog interpreter and Ada program run as separate processes and communicate through pipes[9]. This implementation is expensive; for example, a procedure call from Ada to Prolog costs about forty milliseconds (excluding parameter conversion, which is O(n) in the size of the parameters). We are constructing an improved implementation in which the Prolog interpreter and Ada program share run-time data structures (thus reducing the procedure call overhead to O(1)).

The work described in this paper has just completed the "proof of concept" stage. Even at this early point, we feel we have shown that specifications combining imperative languages and logic programming are a promising basis for a development methodology. As the specifications are both executable and formally based, the

Unix® is a trademark of AT&T.

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

equivalence between specification and implementation can be determined using either testing or proof techniques. We believe we can build upon the significant research being performed on logic-based programming languages [8,17] and data bases [14,31], and construct powerful knowledge-based tools [44]. We have drawn some conclusions from our experience. We believe that pure Horn clauses are not sufficient for software specification; we do not believe the closed world assumption is suitable for software engineering applications. We feel that the automatic construction of prototypes from purely declarative specifications is not practical; we feel we can add an operational semantics without destroying the declarative semantics of specifications. We believe that the use of future languages similar to PLEASE will 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. Barklund, J., "Efficient Interpretation of Prolog Programs", *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, June 1987, 132-137.
3. 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.
4. Bjorner, D. and C. B. Jones, *Formal Specification and Software Development*, Prentice-Hall, Englewood Cliffs, N.J., 1982.
5. 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.
6. Bjorner, D., "On The Use of Formal Methods in Software Development", *Proceedings of the 9th International Conference on Software Engineering*, 1987, 17-29.
7. 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.
8. Bowen, K. A., "New Directions in Logic Programming", *Proceedings of the ACM Computer Science Conference*, February 1986, 19-27.
9. Chang, C. and R. C. Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.
10. Clocksin, W. F. and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.
11. Cottam, I. D., "The Rigorous Development of a System Version Control Program", *IEEE Transactions on Software Engineering SE-10, 3* (March 1984), 143-154.
12. 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.
13. Fairley, R., *Software Engineering Concepts*, McGraw-Hill, New York, 1985.
14. Gallaire, H., J. Minker and J. Nicolas, "Logic and Databases: A Deductive Approach", *ACM Computing Surveys 16*, 2 (June 1984), 153-185.
15. Gehani, N. and A. D. McGettrick, eds., *Software Specification Techniques*, Addison Wesley, Reading, Massachusetts, 1986.
16. 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.
17. Goguen, J. A. and J. Meseguer, "Equality, Types, Modules and (why not?) Generics for Logic Programming", *Logic Programming 1, 2* (1984), 179-210.
18. 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.
19. Guttag, J. V., J. J. Horning and J. M. Wing, "The Larch Family of Specification Languages", *IEEE Software 2, 5* (September 1985), 24-36.
20. Henderson, P., "Functional Programming, Formal Specification, and Rapid Prototyping", *IEEE Transactions on Software Engineering SE-12, 2* (February 1986), 241-250.
21. Jackson, M. I., "Developing Ada Programs Using the Vienna Development Method (VDM)", *Software - Practice and Experience 15, 3* (March 1985), 305-318.
22. Jones, C. B., *Software Development: A Rigorous Approach*, Prentice-Hall International, Engelwood Cliffs, N.J., 1980.

23. 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.

24. Komorowski, H. J. and J. Maluszynski, "Logic Programming and Rapid Prototyping", Report Tech. Rep.-01-86, Center for Research in Computing Technology, Harvard University (also to appear in the **Science of Computer Programming**), Cambridge, MA, 1986.

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

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

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

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

29. 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.

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

31. Minker, J., "An Experimental Relational Data Base System Based on Logic (or Clause Encounters of a Logical Kind)", in *Logic and Data Bases*, Gallaire, H. (editor), Plenum Press, New York, 1978.

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

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

34. Osterweil, L. J., "Software Processes Are Software Too", *Proceedings of the 9th International Conference on Software Engineering*, 1987, 2-13.

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

36. 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.

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

38. 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.

39. 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.

40. 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.

41. 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.

42. Terwilliger, R. B. and R. H. Campbell, "An Early Report on ENCOMPASS", Report No. CU-CS-380-87 , Department of Computer Science, University of Colorado at Boulder, October 1987.

43. 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.

44. 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.

45. 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.

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

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

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

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