FORMAL SPECIFICATION OF SOFTWARE

A tremendous increase in the variety of fields in which computers are used has brought about an immense increase in the size and concurrency in software and hardware designs that form with a computing device. As the size and amount of concurrency increases, it becomes increasingly difficult to raise the level of confidence in the correctness of the design. The size of the design makes it tedious and time-consuming to check its correctness, while concurrency makes it difficult and too many cases to consider for correctness. Furthermore, ensuring correctness becomes even more difficult if the software is used in applications that are subject to real-time constraints. "Correctness" means that the sequence of behaviors allowed by the implementation is a subsequence of the behaviors permitted by the specification. Trivial implementations that allow an empty sequence of behaviors can be ruled out either by showing that at least one behavior is allowed by the implementation, or by showing that the implementation is equivalent to its specification with respect to behavior. There are two main schools of thought in formal software and hardware development:

- The transformational design methodology, which entails beginning with a validated high-level specification of the design and applying a sequence of correctness-preserving transformations on the specification obtaining a correct design.
- A method that entails obtaining a design independent of the high-level specification and validating the design with respect to the high-level specification.

The rest of this article is organized as follows. The section entitled "Transformational Software Design" discusses software development based on correctness-preserving transformations of formal specifications. In the section entitled "Formal Methods: Overview" we give an overview of formal methods, followed by a discussion of the Prototype Verification System (PVS) in the section entitled "PVS." The difficulty in developing a formal specification is discussed in the section entitled "From Informal to Formal Specifications," followed by a brief discussion on abstraction in the section entitled "Abstraction." In the section entitled "Formal Specification Languages" a variety of formal notations and specification formalisms are discussed. Finally some conclusions are presented in the section entitled "Conclusions."

TRANSFORMATIONAL SOFTWARE DESIGN

There have been several efforts made with regard to the specification and verification of refinements used in program development from high-level specifications. Most of the efforts involve selecting a specification formalism and then developing a notion of correctness and an associated set of transformations based on the semantics of the formalism.

The refinement calculus (1) for specifications based on Dijkstra's guarded command language and weakest precondition semantics has been formalized in HOL (2). Transformations such as data refinement and superposition have been verified to be correct. A formalization of incremental development of programs from specifications for distributed real-time systems has been worked out in PVS (3). In this formalism, an assertional method based on a compositional framework of classical Hoare triples is developed for stepwise refinement of specifications into programs.

The KIDS (4) system is a program derivation system. High-level specifications written in a language called Refine are transformed by data-type refinements and optimization transformations (such as partial evaluation and finite differencing) into a Refine program. The disadvantage of this method is the quality of the design: size of code and performance.

FORMAL METHODS: OVERVIEW

Formal methods could be divided into two main categories: property-oriented methods and model-oriented methods (5). In a property-oriented method, the system under consideration is specified by asserting properties of the system, minimizing the details of how the system is constructed. In a model-oriented method, the specification describes the construction of the system from its components. An axiomatic approach is a property-oriented method. Typically, a small set of properties, called *axioms,* are asserted to be true, while other properties, called *theorems,* are derived.

In model-checking (6), a typical implementation specification is a state machine. The verification that the implementation satisfies a property is carried out by reachability analysis. The relationship that a model I satisfies a property S is written as

$$I \models S$$

In generic theorem-proving, the specification could be of any form belonging to the logical language of the theorem-prover (a typical logical language is based on typed higher-order logic). The verification of a property proceeds by a series of application of deduction rules such as induction. The relationship whereby an implementation I satisfies a property specification S is written as

$$\vdash I \Rightarrow S$$

PVS

The Prototype Verification System (PVS) (7,8) is an environment for specifying entities such as hardware and software models and algorithms and for verifying properties associated with the entities. An entity is usually specified by asserting a small number of general properties that are known to be true. These known properties are then used to derive other desired properties. The process of verification involves checking relationships that are supposed to hold among entities. The checking is done by comparing the specified properties of the entities. For example, one can compare if a register-transfer-level implementation of hardware satisfies the properties expressed by its high-level specification.

PVS has been used for reasoning in many domains, such as in hardware verification (9,10), protocol verification, algorithm verification (11,12), and multimedia (13).

PVS Specification Language

The specification language (7) features common programming language constructs such as arrays, functions, and records. It has built-in types for reals, integers, naturals, and lists. A type is interpreted as a set of values. One can introduce new types by explicitly defining the set of values, or indicating the set of values, by providing properties that have to be satisfied by the values. The language also allows hierarchical structuring of specifications. Besides other features, it permits overloading of operators, as in some programming languages.

PVS Verification Features

The PVS verifier (8) is used to determine if the desired properties hold in the specification of the model. The user interacts with the verifier by way of a small set of commands. The verifier contains procedures for Boolean reasoning, arithmetic, and (conditional) rewriting. In particular, model checking (6) based on binary decision diagram (BDD) (14,15) simplification may be invoked for Boolean reasoning. It also features a variety of general induction schemes to tackle large-scale verification. Moreover, different verification schemes can be combined into general-purpose strategies for similar classes of problems, such as verification of microprocessors (9,10).

A PVS specification is first parsed and type-checked. At this stage, the type of every term in the specification is unambiguously known. The verification is done in the following style: We start with the property to be checked and repeatedly apply rules on the property. Every such rule application is meant to obtain another property that is simpler to check. The property holds if such a series of applications of rules eventually leads to a property that is already known to hold. Examples illustrating the specification and verification in PVS are described in the section entitled "Specification and Verification Examples in PVS."

Notes on Specification Notation

In PVS specifications (shown displayed in monospace font), an object followed by a colon and a type indicates that the object is a constant belonging to that type. If the colon is followed by the key word *VAR* and a type, then the object is a variable belonging to that type. For example,

```
x: integer
y: VAR integer
```

describes *x* as a constant of type integer and describes *y* as a variable of type integer (in C, they would be declared as *const int x*; int *y*).

Sets are denoted by {. . .}: They can be introduced by explicitly defining the elements of the set, or implicitly by a characteristic function. For example,

```
{0, 1, 2}
{x: integer | even(x) AND x /= 2}
```

The symbol | is read as *such that,* and the symbol `/=` stands for *not equal* to in general. Thus, the latter example above should be read as "set of all integers x, such that x is an even number and x is not equal to 2."

New types are introduced by a key word *TYPE* followed by its description as a set of values. If the key word *TYPE* is not followed by any description, then it is taken as an uninterpreted type.

Some illustrations are:

```
even_time: TYPE = {x: natural| even(x)}
unspecified_type: TYPE
```

One type that is used widely in this work is the *record type.* A record type is like the *struct* type in the C programming language. It is used to package objects of different types in one type. We can then treat an object of such a type as one single object externally, but with an internal structure corresponding to the various fields in the record.

The following operators have their corresponding meanings:

```
FORALL x: p(x)
```

means *for every* x, predicate $p(x)$ is *true* (a predicate is a function returning a Boolean type: {*true, false*}).

```
EXISTS x: p(x)
```

means *for at least a single x*, predicate $p(x)$ is *true.*

We can impose constraints on the set of values for variables inside FORALL and EXISTS as in the following example:

```
FORALL x, (y| y = 3*x): p(x,y)
```

which should be read as *for every x and y such that y is 3 times x, p*(x, y) is *true.*

A property that is already known to hold without checking is labeled by a name followed by a colon and the keyword AXIOM. A property that is checked using the rules available in the verifier is labeled by a name followed by a colon and the keyword THEOREM. The text followed by a % in any line is a comment in PVS. We illustrate the syntax as follows:

```
ax1: AXIOM % This is a simple axiom
FORALL (x:nat): even(x) = x divisible_by 2


th1: THEOREM % This is a simple theorem
FORALL (x:nat): prime(x) AND x /= 2 IMPLIES NOT
  even(x)
```

We also use the terms *axiom* and *theorem* in our own explanation with the same meanings. A *proof* is a sequence of deduction steps that leads us from a set of axioms or theorems to a theorem.

Specification and Verification Examples in PVS

We illustrate here three examples from arithmetic. The first two examples are taken from the tutorial (16). The last example illustrates the use of a general purpose strategy to automatically prove a theorem of arithmetic. The first example is the sum of natural numbers up to some arbitrary finite number n is equal to $n*(n + 1)/2$. The specification is encapsulated in the sum THEORY. Following introduction of n as a natural number nat, sum(n) is defined as a recursive function with a termination MEASURE as an identity function on n. Finally, the THEOREM labeled closed_form is stated to be proved.

```
sum: THEORY
  BEGIN

  n: VAR nat

  sum(n): RECURSIVE nat =
    (IF n = 0 THEN 0 ELSE n + sum(n − 1) ENDIF)
    MEASURE (LAMBDA n: n)

  closed_form: THEOREM sum(n) = (n * (n + 1))/2

  END sum
```

The THEORY is first parsed and type checked, and then the prover is invoked on the closed_form THEOREM. The proof is automatic by applying induction and rewriting. The proof session is as follows:

```
closed_form :

  |-------
{1} (FORALL (n: nat): (sum(n) = (n * (n + 1)) /
  2))

Running step: (INDUCT ''n'')
```

Inducting on n, this yields 2 subgoals:

```
closed_form.1 :

  |-------
{1} sum(0) = (0 * (0 + 1)) / 2

Running step: (EXPAND ''sum'')
```

Expanding the definition of sum, this simplifies to:

```
closed_form.1 :

  |-------
{1} 0 = 0 / 2

Rerunning step: (ASSERT)
```

Invoking decision procedures, this completes the proof of closed_form.1.

```
closed_form.2 :

  |-------
{1} (FORALL (j: nat):
   (sum(j) = (j * (j + 1)) / 2
   IMPLIES sum(j + 1) = ((j + 1) * (j + 1 +
  1)) / 2))

Running step: (SKOLEM 1 (''j!1''))
```

For the top quantifier in 1, we introduce Skolem constants: (j!1), the simplifies to:

```
closed_form.2:

  |-------
{1} sum(j!1) = (j!1 * (j!1 + 1)) / 2
   IMPLIES sum((j!1 + 1)) = ((j!1 + 1) * (jj!1 +
  1) + 1)) / 2

Running step: (FLATTEN)
```

Applying disjunctive simplification to flatten sequent, this simplifies to:

```
closed_form.2 :

{−1} sum(j!1) = (j!1 * (j!1 + 1)) / 2
  |-------
{1} sum((j!1 + 1)) = ((j!1 + 1) * ((j!1 + 1) +
  1)) / 2

Running step: (EXPAND ''sum'' +)
```

Expanding the definition of sum, this simplifies to:

```
closed_form.2 :

[−1] sum(j!1) = (j!1 * (j!1 + 1)) / 2
  |-------
{1} (j!1 + 1) + sum(j!1) = (j!1 * j!1 + 2 * j!1
  + (j!1 + 2)) / 2

Running step: (ASSERT)
```

Invoking decision procedures, this completes the proof of closed_form.2.
Q.E.D.

```
Run time = 8.09 s.
Real time = 9.89 s.
NIL
>
```

FROM INFORMAL TO FORMAL SPECIFICATIONS

The most difficult and error-prone part in formal methods is developing a proper formal specification from informal specifications. Even though the informal specifications were well-documented, creating a formal specification requires expressing informal ideas such as *behavior* and *mutual exclusiveness* in mathematically precise terms.

One of the first tasks that aids the specification process is the choice of abstraction level: How much of the detail present in the informal document should the specification represent? The choice could be based on how the formal specification has to be verified.

Another important issue in developing a formal specification from an informal document is deciding on data structures to represent entities specified informally. It is desirable to have a formal specification that very closely resembles the informal document. This is essential to map a formal specification back to its informal document. It is essential also for understanding a formal specification and for tracing errors that have been found in the specification back to its informal representation.

ABSTRACTION

A typical design would be too large for current formal verification methods to efficiently validate the design. Therefore it is necessary to remove details from the design description that do not alter the property of the original concrete design. Such a process of removing portions of the design redundant for verification is called *abstraction.* Abstraction is termed *conservative* if we can conclude that a property holds on the original concrete design if the property holds on the abstracted design.

FORMAL SPECIFICATION LANGUAGES

A number of methods have been developed to specify the requirements that needs to be satisfied by a software design. In this section we describe some of more often used notations and methods.

Requirements State Machine Language

Requirements State Machine Language (RSML) (17) is based on an underlying mealy machine and adopts some of the features introduced in statecharts (18), including hierarchical abstraction into superstates and communicating parallel state machines. Components communicate only through point-to-point messages over defined channels. Messages are received asynchronously and queued upon arrival. Each component contains a state hierarchy, transitions between states, a set of input and output interfaces, a set of variables and constants, and a set of events to order the transitions. Internal events are broadcast only within a component. The interfaces are connected to specific communication channels where the receipt of a message on a channel can set variable values and trigger events. Each channel is connected to one input interface and one output interface, and each interface is connected to exactly one channel. Each transition between states has a source, destination, trigger event, and events that it triggers along with a guarding condition that must be true for the transition to be taken. RSML provides a rich language for guarding conditions: A guarding condition may be either a simple Boolean true or false, and AND/OR table, or an existential or universal quantifier of a variable over another condition. SVC (19) has been used to check RSML specifications of an Air Traffic Alert and Collision Avoidance System (TCAS II).

Z

The formal specification notation Z (20) (pronounced “zed”) is based on Zermelo–Fraenkel set theory and first order predicate logic. Z has been developed primarily by the Programming Research Group (PRG) at the Oxford University Computing Laboratory (OUCL). A host of tools to check Z specifications have been developed. Z is mostly used as a formal notation for specification rather than as a verification framework.

B-Method

The B-method (21) is a collection of mathematically based techniques for the specification, design, and implementation of software components. Systems are modeled as a collection of interdependent abstract machines, for which an object-based approach is employed at all stages of development.

An abstract machine is described using the Abstract Machine Notation (AMN). A uniform notation is used at levels of description, from specification, through design, to implementation. AMN is a state-based formal specification language in the same school as VDM and Z. An abstract machine comprises a state together with operations on that state. In a specification and a design of an abstract machine the state is modeled using notions like sets, relations, functions, sequences, and so on. The operations are modeled using pre- and post-conditions using AMN.

In an implementation of an abstract machine the state is again modeled using a set-theoretical model, but this time we already have an implementation for the model. The operations are described using a pseudo-programming notation that is a subset of AMN.

The B-method prescribes how to check the specification for consistency (preservation of invariant) and how to check designs and implementations for correctness (correctness of data refinement and correctness of algorithmic refinement).

The B-method further prescribes how to structure large design and large developments, and it promotes the reuse of specification models and software modules, with object orientation central to specification construction and implementation design.

Protocol Verification Using SPIN

SPIN from Lucent Bell Labs supports the formal verification of distributed systems. SPIN has been used to trace logical design errors in distributed systems design, such as operating systems, data communications protocols, switching systems, concurrent algorithms, railway signaling protocols, and so on. The tool checks the logical consistency of a specification. It reports on deadlocks, unspecified receptions, flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes. SPIN works on-the-fly, which means that it avoids the need to construct of a global state graph, as a prerequisite for the verification of any system properties. Correctness properties can be specified as system or process invariants (using assertions) or as general linear temporal logic (LTL) requirements, either directly in the syntax of next-time free LTL requirements or indirectly as Bœ,A-œ (Bchi Automata). SPIN supports both rendezvous and buffered message passing, and communication through shared memory. Mixed systems, using both synchronous and asynchronous communications, are also supported. Message channel identifiers for both rendezvous and buffered channels can be passed from one process to another in messages. SPIN supports random, interactive, and guided simulation and both exhaustive and partial proof techniques. To optimize the verification runs, the tool exploits efficient partial-order reduction techniques and efficient Boolean representation techniques.

A major experiment with the SPIN modelchecker has been used in the identification of five classical concurrency errors in the operating system of NASA's autonomy AI software of the Deep Space-1 spacecraft. This work demonstrates an application of the finite-state model checker SPIN to formally verify a multithreaded plan execution programming language. The plan execution language is one component of NASA's New Millennium Remote Agent, an artificial-intelligence-based spacecraft control system architecture that is scheduled to launch in October 1998 as part of the Deep Space-1 mission to Mars. The language is concretely named ESL (Executive Support Language) and is basically a language designed to support the construction of reactive control mechanisms for autonomous robots and spacecrafts. It offers advanced control constructs for managing interacting parallel goal-and-event driven processes and is currently implemented as an extension to a multithreaded Common Lisp. A total of five errors were identified. According to the Remote Agent programming team the effort has had a major impact, locating errors that would probably not have been located otherwise and identifying a major design flaw not easily resolvable.

UPPAAL

UPPAAL (22) is developed in collaboration between the Design and Analysis of Embedded Systems group at Uppsala University, Sweden and Basic Research in Computer Science at Aalborg University, Sweden. UPPAAL real-time model-checker has been applied to a couple of real-life Audio/Video protocols (23) for the Audio/Video company Bang & Olufsen (B&O). In the first application a 10-year-old error was located; B&O was aware of its existence, but had never been able to locate via normal testing. Both protocols were highly dependent on real time. UPPAAL is a tool suite for validation and verification of a real-time system modeled as networks of time automata extended with (arrays of) data variables. The tools in UPPAAL have WYSIWYG (what you see is what you get) interfaces and feature: graphical editing, graphical symbolic simulation, and symbolic verification of safety and liveness properties.

Other Notations

A notation (24) for specifying requirements in a very abstract and succinct form. They have developed a scheme for checking properties of NP specifications that exploits symmetry in the mathematical structure of the property being checked. A tool called Nitpick has been built that works completely automatically for analyzing specifications.

Many different kinds of problem can be specified in NP, so Nitpick (25) can be used to analyze not only requirements, but also specifications and abstract designs. Using Nitpick/NP, they found some interesting problems with the paragraph style mechanism of Microsoft Word. They have also analyzed an air-traffic control handoff protocol, a basic telephone switch, and, with Jeannette Wing and Dave Johnson, a mobile internet protocol.

In reverse engineering, structural information is extracted from large programs. A tool called Chopshop has been developed that calculates program slices for large C programs in a modular fashion; and it can display the results not only as code highlighted in an editor buffer, but also as graphs showing the semantic relationships between procedures. Lackwit, an improvement of Chopshop, produced information about global use of data structures that was not easily obtainable by any other method, and it exposed a variety of flaws such as a storage leak in a loop.

Another simple method of requirements specification is based on tables (26) for specifying software. It supports the production of software documentation through an integrated set of tools which manipulate multidimensional tabular expressions. This tabular representation of mathematical expressions improves the readability of complex design documentation. The table cells may contain conventional logic expressions, or even other tables.

There has been a lot of work on verification of clock synchronization algorithms in safety-critical fault-tolerant systems (27). There have been mistakes found using formal methods in published clock synchronization algorithms (28).

CONCLUSIONS

In this article we have presented a spectrum of formal methods for software development. Formal methods have matured to a point where they can be applied to small industrial designs. However, further research in abstraction and efficient software code generation from formal specifications is needed to apply formal methods on a large scale.

BIBLIOGRAPHY

1. R. J. R. Back, A calculus of refinements for program derivations, *Acta Inf.,* **25**: 593–624, 1988.
2. J. von Wright and K. Sere, Program transformations and refinements in hol, in M. Archer et al. (eds.), *Higher Order Logic Theorem Proving and Its Applications (4th International Workshop, HUG '91),* Los Alamitos, CA: IEEE Computer Society Press, 1991.
3. J. Hooman, Correctness of real time systems by construction, in H. Langmaack, W.-P. de Roever, and J. Vytopil (eds.), *Formal Techniques in Real-Time and Fault-Tolerant Systems, Lecture Notes in Computer Science,* New York: Springer-Verlag, 1994, vol. 863, pp. 19–40.
4. D. R. Smith, Kids: A semi-automatic program development system, *IEEE Trans. Softw. Eng.,* **SE-16**: 1990.
5. J. M. Wing, A specifier's introduction to formal methods, *IEEE Comput.,* **23** (9): 8–22, 1990.
6. K. L. McMillan, *Symbolic Model Checking,* Norwell, MA: Kluwer, 1993.
7. S. Owre, N. Shankar, and J. M. Rushby, *User Guide for the PVS Specification and Verification System (Beta Release),* Menlo Park, CA: Computer Science Laboratory, SRI Int., 1993.
8. N. Shankar, S. Owre, and J. M. Rushby, *The PVS Proof Checker: A Reference Manual (Beta Release),* Menlo Park, CA, Computer Science Laboratory, SRI Int., 1993.
9. D. Cyrluk, *Microprocessor verification in PVS: A methodology and simple example,* Tech. Rep. CSL-93-12, Menlo Park, CA: SRI Int., 1993.
10. R. Kumar and T. Kropf (eds.), *Proc. 2nd Conf. Theorem Provers Circuit Design,* Bad Herrenalb (Blackforest), Germany, Forschungszentrum Informatik an der Universität Karlsruhe, FZI Publication,1994.
11. P. Lincoln et al., *Eight papers on formal verification,* Tech. Rep. SRI-CSL-93-4, Menlo Park, CA: Computer Science Laboratory, SRI Int., 1993.
12. S. Owre et al., Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS, *IEEE Trans. Softw. Eng.,* **21**: 107–125, 1995.
13. S. P. Rajan, P. V. Rangan, and H. M. Vin, A formal basis for structured multimedia collaborations, *Proc. 2nd IEEE Int. Conf. Multimedia Comput. Syst.,* Los Alamitos, CA: IEEE Computer Society, 1995, pp. 194–201.
14. K. S. Brace, R. L. Rudell, and R. E. Bryant, Efficient implementation of a BDD package, *Proc. 27th ACM/IEEE Des. Autom. Conf.,* Assoc. Comput. Mach., 1990, pp. 40–45.
15. G. L. J. M. Janssen, *ROBDD Software,* Department Electr. Eng., Eindhoven Univ. Technol., 1993.
16. N. Shankar, S. Owre, and J. M. Rushby, *PVS Tutorial,* Menlo Park, CA: Computer Science Laboratory, SRI Int., 1993. Also appears in Tutorial Notes, *Formal Methods Europe '93: Industrial-Strength Formal Methods,* Odense, Denmark, 1993, pp. 357–406.
17. N. G. Leveson et al., Requirements specification for process-control systems, *IEEE Trans. on Softw. Eng.,* **20**: 1994.
18. D. Harel, Statecharts: A visual formalism for complex systems, *Sci. Comput. Program.,* (8): 231–274, 1987.
19. D. Y. W. Park et al., Checking properties of safety critical specifications using efficient decision procedures, in M. Ardis (ed.), *Second Workshop on Formal Methods in Software Practice (FMSP '98),* Clearwater Beach, FL: Assoc. Comput. Mach., 1998, pp. 34–43.
20. J. M. Spivey, *Understanding Z: A Specification Language and Its Formal Semantics,* Cambridge Tracts in Theoretical Computer Science 3, Cambridge, UK: Cambridge Univ. Press, 1988.
21. J.-R. Abrial et al., The B-method, in S. Prehn and W. J. Toetenel (eds.), *VDM '91: Formal Software Development Methods, Lecture Notes in Computer Science,* New York: Springer-Verlag, 1991, vol. 552, pp. 398–405; *Tutorials,* vol. 2.
22. K. G. Larsen, P. Pettersson, and W. Yi, *UPPAAL: Status & Developments,* pp. 22–25.
23. K. Havelund et al., Formal modelling and analysis of an audio/video protocol: An industrial case study using UPPAAL, *Real Time Syst. Symp.,* San Francisco, IEEE Computer Society, 1997, pp. 2–13.
24. D. Jackson and C. A. Damon, Elements of style: Analyzing a software design feature with a counterexample detector, *IEEE Trans. Softw. Eng.,* **22**: 484–495, 1996.
25. C. A. Damon and D. Jackson, Efficient search as a means of executing specifications, *Tools and Algorithms for the Construction and Analysis of Systems TACAS '96, Lecture Notes in Computer Science,* New York: Springer-Verlag, 1996, vol. 1055, pp. 70–86.
26. D. L. Parnas, *Tabular representation of relations,* Tech. Rep. CRL Report 260, Telecommun. Res. Inst. Ontario (TRIO), Faculty Eng., McMaster Univ., Hamilton, Ontario, Canada, 1992.
27. J. Rushby, *Formal methods and certification of critical systems,* Tech. Rep. SRI-CSL-93-7, Menlo Park, CA: Computer Science Laboratory, SRI Int., 1993. Also available as NASA Contractor Report 4551, 1993.
28. J. Rushby and F. von Henke, *Formal verification of the Interactive Convergence clock synchronization algorithm using EHDM,* Tech. Rep. SRI-CSL-89-3R, Menlo Park, CA: 1989 (revised 1991). Computer Science Laboratory, SRI Int.. Original version also available as NASA Contractor Report 4239, 1989.

SREERANGA P. RAJAN
Fujitsu Laboratories of America

FORMAL SPECIFICATIONS. See VIENNA DEVELOPMENT METHOD.

FORMAL VERIFICATION. See KNOWLEDGE VERIFICATION.