focus review

WINTER 2001

A Review of Some Rigorous Software Design and Analysis Tools

The increasing maturity of formal methods cannot be attributed to only the formal notations and methodologies that are accessible to system designers. The development of powerful software tools that apply and facilitate the use of these notations and methodologies effectively has been crucial. In this paper, **FREDERICK SHELDON**, **GAOYAN XIE**, **OREST PILSKALNS** and **ZHIHE ZHOU** survey some well-known software tools that have been deployed and used by both academic and industrial sections for rigorous design and analysis. The software tools are categorized by both the notations and methodologies, upon which they are based. We mainly discuss the tools' underlying formal methods, achievements and scope of applicability. We finish with the future trend of the development of such software tools.

Frederick Sheldon[1]
Software Engineering for Dependable Systems Laboratory
School of EECS, Washington State University, Pullman, Washington 99164-2752, USA

[1] Currently on leave at Daimler Chrysler Research Information and Community/ System Safety (RIC/AS) in Stuttgart.

Introduction

Rigorous development generally means the application of some sort of formal method(s) in software development. A formal method [1][2] in software development is a method that usually provides: (1) mathematically based notations (such notations are mostly either textual or graphical, but tabular representations can also be seen in [3][4]), (2) a process for describing how a software artifact (specifications, designs, source code, etc.) is produced, and (3) software tools for formally deducing or checking about properties of the artifact so expressed. Although the cost of applying formal methods could be high at early phases of the Software Life-Cycle (SLC), the overall cost decreases and the overall productivity increases [5]. Furthermore, formal methods usually make it easy to find errors in the requirements, specification and/or design, which is crucial for the development of mission-critical and safety-critical systems where extensive tests are generally impossible [6]. The value of such methods has already been witnessed in handling several industrial-sized examples [7][8]; in some cases, these methods are already being used on a regular basis in industry.

In any formal method, powerful software tools are usually indispensable for the effective application of formal notations and for reasoning about various software artifacts. In this paper, we survey some popular software tools for applying formal methods in rigorous software design and analysis. They are PVS, Spin, UltraSAN, Statemate and Z/EVES, which are broadly recognized by both the academic and industrial sectors. Generally, formal methods and their supporting tools can be applied in any phase of the SLC from requirements analysis to software maintenance and there are numerous methods and tools available. However, in this paper we focus on the tools that facilitate the application of formal methods to the upstream activities of the SLC, namely requirements, specification and design. This does not mean to say that formal methods are not important to the other aspects/phases of the SLC. However, the simple reason is that obtaining consistent and complete (unambiguous) requirements specification and design is of overwhelming importance to safety-critical and mission-critical systems with which we are primarily concerned. Because these tools are built so closely upon some basic underpinnings, we cannot proceed without providing some general characterizations (next section). Later, each of five tools is described with some established criteria like history, underlying notation, advantages and disadvantages, applicability and representative examples. We summarize this survey in the closing section.

Formal Methods

Formal methods can be applied in various phases of the software development process, but they are mainly used for system specification, verification and validation. Therefore, in this section, we will describe some background on specification languages, verification

techniques, program refinement techniques and validation techniques respectively.

Specification Languages

Formal specification languages usually provide mathematically based notations to describe a system's structure, behavior and desired properties. The kinds of properties might include functional behaviors, timing constraints, performance characteristics, etc. In general, there are two kinds of formal specification languages that have been used to write detailed specifications for non-trivial software systems [9]:

- ***Model-based*. With model-based specification languages, a model of the system is created using mathematical constructs such as sets and sequences and the system operations are defined by how they modify the system state. Typical model-based specification languages are CSP [10] and Petri Net [11], while Z [12] is a general specification language that is normally used in a modeling style.**
- ***Algebra-based*. With algebra-based specification languages, a system is described in terms of operations and their relationships. Typical algebra based specification languages are Larch [13] and Lotos [14].**

Algebraic approaches are suitable to describe systems where the object operations are independent of the object state, such as the interface specification. While model-based specification exposes the system state and defines the operations in terms of changes to that state. Additionally, some specification languages like Z and Larch focus on specifying behaviors of sequential systems, while specification languages like CSP and Petri Net are generally used to specify concurrent systems.

Verification Techniques

Verification techniques go one step beyond specification languages; they are used to analyze whether a given specification meets its properties. Simply, verification techniques can be grouped into two categories:

- ***Deductive Verification.* Deductive Verification normally refers to the use of axioms and proof rules to prove the correctness of a system, such as the partial correctness proof of programs with the Hoare logic [15] and theorem proving techniques [16]. Typical deductive verification systems are HOL [17] and PVS [18].**
- ***Model Checking.* Model Checking is a technique that relies on building a finite-model of a system and uses an exhaustive search of the state space of the model to check that a property holds in that model. The typical model checkers are SMV [19] and Spin [20].**

Deductive verification is a time-consuming process and it also requires much expertise in logic reasoning. The proof of a single protocol can last days or months. Consequently, it is applied primarily to highly sensitive systems such as security protocols. Another disadvantage of deductive verification is that it cannot be completely automated, due to the fact that correct termination of the program is not decidable. However, an advantage of deductive verification is that it can be used for reasoning about infinite state systems. Though the model checking technique [21] has the advantage that the verification can be performed automatically, it has the drawback that it currently can only be applied to the verification of the finite-state systems.

Program Refinement Techniques

Program refinement is a technique of developing computer programs in a stepwise way. The idea of program refinement originates from [22], where the author argues that the '*Program should be gradually developed in a sequence of refinement steps. In each step, a given task is broken up into a number of subtasks. Each refinement in the description of a task may be accompanied by a refinement of the description of the data that constitutes the means of communication between the subtasks.*' Generally, refinement is used to refer to the process of refining an abstract specification into an imperative program through a series of transforming steps while preserving all properties that the original specification holds.

A classic formalism that applies this idea is the Refinement Calculus, described by Ralph-Johan Back [23] in his Ph.D. thesis in 1978. This formalism provides a general mathematical theory for the stepwise refinement approach to program construction. The refinement calculus extends the weakest precondition technique of Dijkstra [24] to procedural and data refinement, and can also be used for stepwise refinement of parallel and distributed programs. Algorithms are derived by a series of correctness preserving refinements and program transformations from very high-level specifications. The derivation is carried on until a program that meets the stated criteria of efficiency and implementability has been constructed. A group at Åbo Akademi University in Finland has developed a prototype tool called the Refinement Calculator [25], which supports this type of refinement.

One of the earliest approaches to couple formal methods with stepwise refinement was the VDM (Vienna Development Method) [26]. VDM encompasses the entire program development process from specification to implementation. Operations are specified in the now common precondition, post-condition style. After specification, abstract data-types are replaced by concrete data-types in a process called data reification. The next stage of the development, called operational decomposition, is the stepwise derivation of an implementation. Each step of the decomposition involves introducing a programming language construct. There is a small collection of decomposition laws that are used to justify these steps (one per program construct). It is not expected that users of VDM will want to develop their own decomposition rules. Indeed, the semantics of the programming language and the proofs of the decomposition laws are not included in the usual expositions of VDM. VDM also has an object-oriented extension, called VDM++. VDM++ has been used in various projects for requirement specification and validation [27].

One of the most widely used formal methods in Europe that incorporate the program refinement idea is the B-Method, created by Jean-Raymond Abrial [28]. The B-method makes use of a notation based on the mathematical concepts of set theory. Generally, the initial requirement is written in natural language, or by juxtaposition of several descriptions: graphs, automatons, logics tables, Petri Nets, etc. A B model is constructed by reusing the requirements description. The B model is thereafter refined until a complete implementation of the software system is obtained. Several refinements can fulfill a specification and the choice of

solutions rests on various criteria such as the processing speed, the precision of calculation or demonstrable simplicity. The consistency of the model and the conformity of the final program in relation to this model are guaranteed by mathematical proofs. The demonstration of these proofs in a concrete case can only be considered with the use of automatic proof tools, such as those provided by Atelier B [29]. A commercially available software tool that is based on the B-Methods is the B-Toolkit [30].

Validation Techniques

Initially, a system model is created at an abstract level (i.e., abstract model). To do an analysis of the system we need measures of the system using some type of measurement technique (i.e., we start at the *Real System*). The *Proposed System* is detailed later, but without a real system to compare to, a great deal of work is needed to explore the parameter space. The data collected from system measurements are used to parameterize the abstract model. However, usually the system model will still contain too many details that prevent an efficient system analysis. In a second abstraction step the computational model is created which allows an easier and more efficient system analysis. The computational model can be considered to be the highest level of model abstraction. The process of refining the computational model is a matter of building confidence in the model. Thus, the process of operational validation is performed which results in a modified system and with modified system input parameters/data. This step can be repeated until the computed performance measures fulfill the requirements.

If there exists no way to gather system measurements for the purpose of parameterizing the system model then there may be a great deal of more work to perform. In addition, the stopping rule for accomplishing operational validation is now based on relative comparisons from one iteration to the next. Different model parameterizations are used to compare the different design (or implementation) candidates with the goal of making architectural design decisions. In simple terms, using a technique commonly known as sensitivity analysis to optimize the model's structure toward achieving critical functional and non-functional requirements.

There are two basic methods used to solve the system model: mathematical and system simulation. The mathematical solution method may be further classified into analytical (non-state-space-based) and numerical (state-space-based). The mathematical method works by solving a system (or set) of linear or differential equations while a simulation is differentiated into discrete event simulation and continuous simulation. Combining mathematical techniques with simulation is called hybrid simulation and when possible, independent components are solved separately using either technique and are then combined in a stepwise fashion (aggregation/disaggregation). One very well known problem when using the numerical method is the high computational cost due to huge state space.[1]

The process of validation demonstrates that the model accurately represents the desired system behavior with enough detail. In general, there are three important steps:

- **verify the model assumptions against the real system (reality check),**
- **analyze the model structure and the logical relations among its components,**
- **compare the behavior of the model under different experimental conditions (e.g., different load models).**

We distinguish between *operational* and *conceptual* validation because there may not be a way to parameterize the model (i.e., extracting system measures from an existing system or its analog). In the conceptual validation phase the computational model is compared to the system model (i.e., its more detailed and less abstract predecessor). This comparison must be done to test the assumptions of the higher-level abstraction to determine if the computational model's assumptions are correct or at least reasonable. In fact, this may be a process of adjusting things down to a tolerable level of simplification.

Conversely, during operational validation, the computed performance results (i.e., predictions and/or estimates) are compared to the system measures. This is a very important step because it determines how the system model (and input parameters) may be modified to more closely describe the actual system behavior. This process of refinement gives a computational model that more accurately predicts the real system behavior under different (or new) conditions.

The validation phase will result in a validated final model that may have been modified with respect to its structure and/or its parameterization (i.e., relationships among variables, initialization and initial state etc.). For example, modifications to the model can be carried out with the goal of predicting the behavior for the system under study (SUS). When used in this sense, parts of the model are removed or changed in an effort to investigate the cause and effects of proposed enhancements or adaptations. Furthermore, once a model is validated it may be used to perform sensitivity analysis, which can be used to support or discredit the modeling assumptions and analysis conclusion(s). The two most common forms of sensitivity analysis include:

- **Testing the robustness of the computational results against the model assumptions. This requires that the model be analyzed some number of times to allow comparing the results from one run to the next.**
- **Obtaining bounds on the expected performance measures by evaluating the model under worst and/or best case assumptions.**

In effect, the modeling cycle yields many insights into the SUS. These insights result from the different steps in the modeling cycle and are used to improve the system in some desired aspect. Thus, given a formal model and its external constraints, we can explore what mechanisms are available to optimize the system behavior. For example, consider such factors as the SUS topology, fault tolerance, timeliness, resource allocation, communications etc. How do such mechanisms impact the behavioral aspects such as reliability and performability? Refining the system model can reveal trade-offs in design alternatives such as

[1] Taking a pragmatic view, there are two ways to cope with this. First we may somehow tolerate or modify the large state spaces, or second, we may prevent the origination of such large state spaces. Depending on what type of analysis is desired, either transient or steady state, different efficient solution techniques are available.

deciding what features of the system should be changed to improve the system's reliability or validating certain assumptions with respect to various performance goals. More detail on this can be seen in [31].

Software Tools Used for Rigorous Development

This section elaborates on five typical software tools for rigorous development, and the comparisons between them are summarized in Table 1.

The PVS system

PVS [32] (Prototype Verification System) is a deductive verification system developed by SRI and is mainly intended for the formalization of requirements and design-level specifications, and for the analysis of intricate and difficult problems. It has been chiefly applied to algorithms and architectures for fault-tolerant flight control systems, and to problems in hardware and real-time system design. Several examples are described in [18][33][34]. Projects involving PVS are ongoing with NASA and several aerospace companies including a microprocessor for aircraft flight-control, diagnosis and scheduling algorithms for fault-tolerant architectures, and requirements specification for portions of the Space Shuttle flight-control system.

Basically, PVS is based on a theorem proving technique. The PVS system consists of a specification language, a number of predefined theories and a theorem prover. PVS exploits the synergy between a highly expressive specification language and powerful automated deduction; for example, some elements of the specification language are made possible because the type-checker can use theorem proving. This distinguishing feature of PVS has allowed clear and efficient treatment of many examples that are considered difficult for other verification systems.

The specification language of PVS is based on classical, typed higher-order logic. PVS specifications are organized into parameterized theories that may contain assumptions, definitions, axioms, and theorems. An extensive prelude of built-in theories provides hundreds of useful definitions and lemmas and user-contributed libraries provide many more.

The PVS theorem prover provides a collection of powerful primitive inference procedures that are applied interactively under user guidance within a sequent calculus framework. The primitive inferences include propositional and quantifier rules, induction, rewriting, and decision procedures for linear arithmetic.

Tool	Underlying Methods	Advantages	Disadvantages	Applicable Domain
PVS	Theorem proving	Can be applied to infinite-state systems	Cannot be automated, need human guide	Highly-sensitive systems, security protocols
Spin/ PROMELA	Model checking	Verification can be fully automated	Can be applied only to finite-state systems	Concurrent/ distributed systems and protocols
UltraSAN	Markov model and Simulation	Analytic approach and discrete event simulation	Constructed models are not easy to modify	Real-time systems, performability evaluation
STATEMATE	Model-based specification languages	Executable specification	Generated code can be large and unwieldy	Reactive embedded systems
Z/EVES	Model-based specification language and theorem proving	Precise specification and preconditions can be deduced	Notations of Z are too abstract, proof needs human guide	Sequential software systems

Table 1 Comparisons of five rigorous software tools

The implementations of these primitive inferences are optimized for large proofs: for example, propositional simplification uses binary decision diagrams, and auto-rewrites are cached for efficiency. User-defined procedures combine these primitive inferences to yield higher-level proof strategies. Proofs yield scripts that can be edited, attached to additional formulas, and rerun. This allows many similar theorems to be proved efficiently, permits proofs to be adjusted economically to follow changes in requirements or design, and encourages the development of readable proofs. Figure 1 from John Rushby's slides [35] gives an overview of the GUI of PVS using Greatest Common Divisor (GCD) as an example. PVS includes a BDD (binary decision diagram)–based decision procedure for the relational mu-calculus and thereby provides an experimental integration between theorem proving and CTL (computing temporal logic) model checking [36]. As a theorem prover, PVS has the inherent disadvantages of a deductive verification system we mentioned in section 1.1.2. But it is still worthwhile to use PVS to treat those highly sensitive and dependable systems, such as nuclear power plants [37] and security protocols [38].

The Spin model checker

Developed at Bell Labs in the formal methods and verification group starting in 1980, Spin [20] has been a widely distributed software package that supports the formal verification of distributed systems. Spin is mainly used to trace logical design errors in distributed systems design (e.g., operating systems, data communications protocols, switching systems, concurrent algorithms). It checks the logical consistency of a specification and reports on deadlocks, unspecified receptions, flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes.

Spin's underlying mechanism is model checking. However Spin can be distinguished from conventional model checking tools because it doesn't require the construction of a global state graph [39] (or a Kripke structure), that is why Spin is called an *on-the-fly* model-checker. As a model checker, Spin uses a high level language called Promela to specify the system model, and supports all correctness properties expressed as system or process invariants or as general

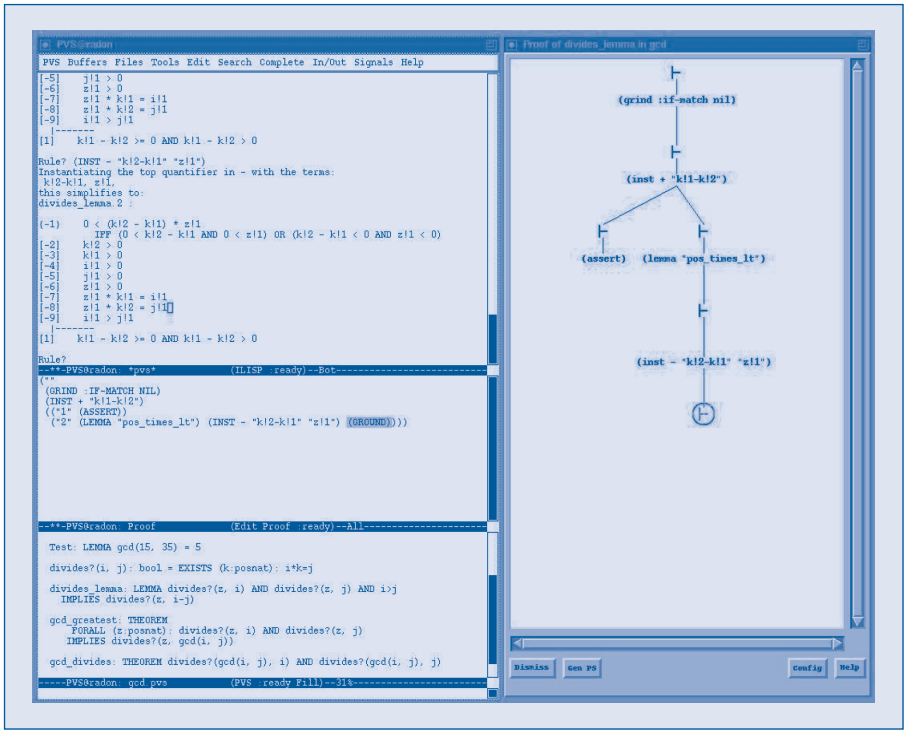

Figure 1 GUI of PVS

LTL(linear temporal logic) [40] requirements, either directly in the syntax of LTL or indirectly as Büchi Automata [41]. Figure 2 shows the Promela and Büchi Automata representations of the LTL formula '[](pUq)' (left) and '[] > p' (right) respectively.

Spin can be used in three basic modes:

- **As a simulator, allowing for rapid prototyping with a random, guided, or interactive simulation.**
- **As an exhaustive state space analyzer, capable of rigorously proving the validity of user specified correctness requirements (using partial order reduction theory to optimize the search).**
- **As a bit-state space analyzer that can validate even very large protocol systems with maximal coverage of the state space (a proof approximation technique).**

Figure 3 depicts the structure of and workings of Spin. Besides all the benefits of model checking, Spin has other advantages:

- **Spin supports dynamically growing and shrinking numbers of processes, using a rubber state vector technique.**
- **Spin supports both rendezvous and buffered message passing, and communication through shared memory. Mixed systems, using both synchronous and asynchronous communications, are also supported.**
- **Spin supports random, interactive and guided simulation, and both exhaustive and partial proof techniques.**
- **Spin exploits efficient partial order reduction techniques, and BDD-like storage techniques to optimize the verification runs.**

However, due to the inherent restrictions of model checking mentioned in Section 1.1.2, Spin can only be used to verify the finite-state system. Models that can be specified in PROMELA are, therefore, required to be bounded, and have only countably many distinct behaviors. Spin is by far one of the most successful software tools for formal verification and it is especially useful for developing communication protocols.

Applications of Spin to real-life problems also span a broad range of fields. The obvious applications are to prove the correctness of generic distributed algorithms, such as the leader election algorithm, nonstandard mutual exclusion algorithms [42], distributed process-scheduling algorithms [43], and

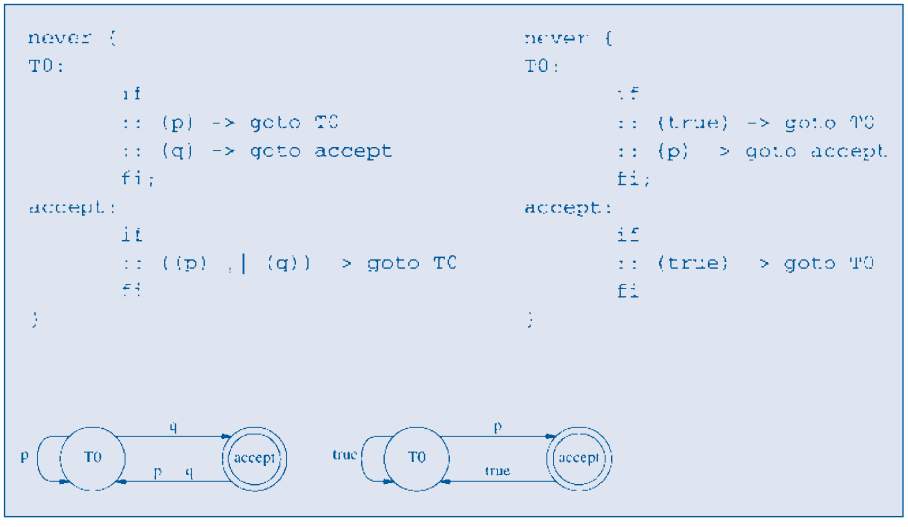

Figure 2 Automaton representation of LTL formulas

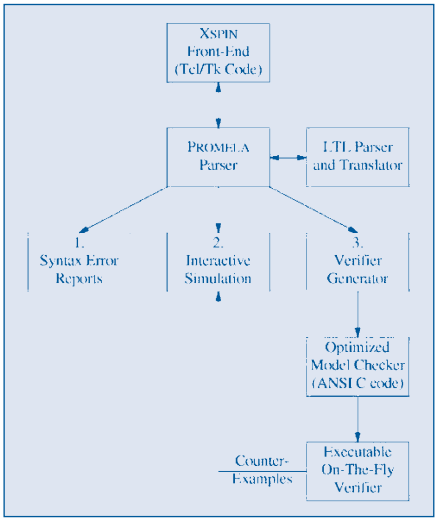

Figure 3 Structure and process of spin

rendezvous algorithms. Spin has also been applied to the verification of data transfer protocols [44], error control protocols, controllers for reactive systems [45], railway signaling protocols and circuitry, security protocols [46], flood surge control systems, Ethernet collision avoidance techniques [47], etc. Holzmann [48] gives a detailed treatment on how to design and validate communication protocols with Spin. Recently, Spin was used successfully to analyze modules of a space-craft control system to identify undiscovered mission critical errors [49][2].

UltraSAN

UltraSAN is a software tool developed by the UltraSAN group in the Center of Reliable and High-Performance Computing at the University of Illinois at Urbana-Champaign. This tool is very suitable for mode-based performance, dependability and performability evaluation of computer, communication and other systems [50]. The UltraSAN tool provides high-level modeling constructs and offers hierarchical modeling by means of composed models. Both analytic solvers and simulators are provided, and the tool also has a report generator, which facilitates the generation of graphs and tables from the obtained results.

A system is modeled as a stochastic activity network (SAN), which is an extension of a Petri net. A SAN extends Petri nets by allowing transitions to be associated with random times and to have multiple output paths with each having certain probability, and thus transitions are referred to as activities in a SAN. Gates, a new type of model construct, are introduced, which act as predicates to define which activities are allowed to fire at any given time. Reward functions are given for activities to collect data for the reward variables that are defined according to the desired performance measurements.

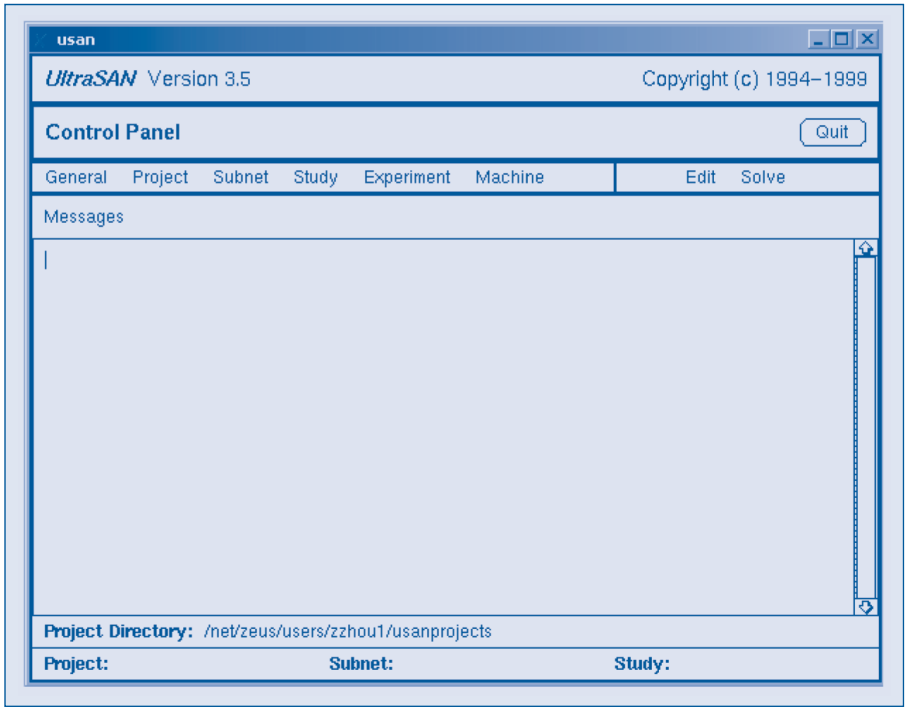

Figure 4 The UltraSAN control panel

The UltraSAN GUI, shown in Figure 4, provides the main control center when UltraSAN starts up. It contains menus and buttons that will bring up other utility windows for editing and solving SANs. Figure 5 shows the user interface of the SAN sub-net editor. The entire system model is divided into several subnets, and each of them is created using the SAN sub-net editor. High-level model constructs are provided in the sub-net editor. The SAN model shown in Figure 5 gives an example of a CPU module in a multi-processor system.

The process to build and solve a model for a given system using UltraSAN includes the following steps:

- **Build subnets: The system is dissected into subsystems and each subsystem is modeled as a SAN subnet.**
- **Create the composed model: Sub-nets and/or replicas of subnets are joined together according to their shared places and result in a composed model.**
- **Create a solvable model: Reward variables are defined and the model is now solvable.**
- **Study editor: values are assigned to certain global variables, and ranges and increments are defined for other global variables where we want to know how the change of their values will impact the performance of the system.**
- **Choose an appropriate solver and apply it to solving the system model.**
- **Generate reports and plot graphs using the Report Generator.**

The UltraSAN tool is based on stochastic process theory. When certain criteria are met, for example, all activities are associated with exponentially distributed random times, a SAN can be converted to a Markov model because the underlying stochastic process is actually a Markov process. Both the analytic solver and discrete event simulators are applicable to solving Markov models. To handle the state space explosion problem a Reduced Base Model Generator is provided as a tool that generates the state space and the state transitions of the stochastic process. This tool incorporates a technique based on subnet replication to reduce the state

[2] The Remote Agent (RA) embedded software (for the DEEP SPACE 1 space-craft) was model checked using in the PROMELA language of SPIN and then two properties, that were later formulated by the RA programmers, were verified. Interestingly both properties were broken and a total of five flawed code fragments were identified (a very successful result). According to the RA programming team, the effort has had a major impact, locating errors that they believe would not have been located otherwise and in identifying a major design flaw. As an interesting aftermath, one of the five error patterns identified was mistakenly reintroduced in a different part of the plan execution module (not examined using SPIN) that caused a deadlock *during* flight in space. The problem was repaired with the help of SPIN which avoided a concurrency bug that could have caused a mission critical failure.

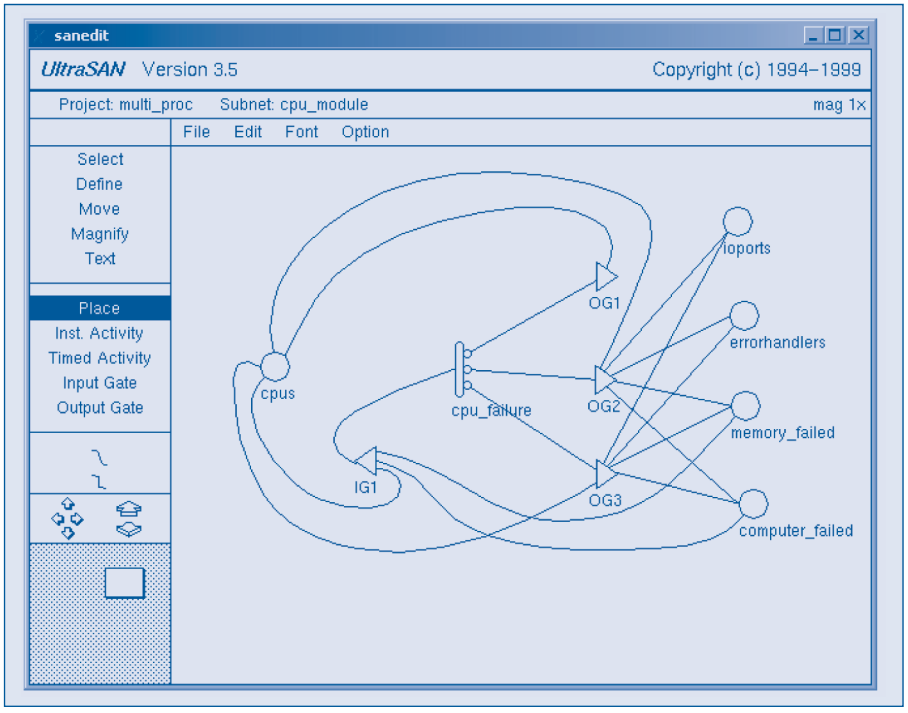

Figure 5 CPU module in subnet editor

space of the Markov model, which would otherwise quickly become unmanageable for even reasonably sized SANs. For those SANs that cannot be converted to Markov models, discrete event simulators are the only solvers applicable.

Unlike Z/EVES or Spin, UltraSAN takes into consideration time information for a given system modeled as a stochastic process. Hence, UltraSAN is well suited for evaluating the system performance. These measures are hard to obtain without a stochastic model [51]. UltraSAN can also be used for dependability analysis [52]. The advantages of UltraSAN can be summarized as follows:

- **UltraSAN supports composed model construction. Replicating and joining small models can obtain large models. Model replication facilitates reusing small models.**
- **UltraSAN provides both analytic solvers and simulators.**
- **UltraSAN has techniques to reduce state space.**
- **UltraSAN can solver both Markov models and non-Markov models.**

Although UltraSAN provides plenty of solvers, there are certain limitations on using these solvers. Analytic solvers can only be used for solving Markov or semi-Markov models. These models must have a finite state space. For a semi-Markov model, analytic solvers are applicable only when the model has at most one enabled deterministic activity at the time. Simulators can handle all models, but an inherent drawback of simulation is the long times necessary to obtain reasonable results for rare event simulation. A SAN for a large system tends to be quit complicated. There are many global variables and reward variables. Determining the values for those variables and optimizing these parameters is difficult. One should have a thorough knowledge of SANs and the tool to handle such models. Moreover, UltraSAN models are not easily modified if the interconnect topology changes.

STATEMATE

I-Logix's STATEMATE is a complex graphical modeling tool with code generation and simulation capabilities. It is often used in the requirements, specification, and design phases of reactive embedded systems. Through the use of graphical formalisms, it allows the user to create complete specifications that describe a system's performance. The specification is executable, which allows for simulations that can test the systems capabilities and check for correct behavior at the earliest stages of design. In addition, the executable specification allows the customer and engineers to easily proceed through the iterative process of verifying that all the requirements have been satisfied.

STATEMATE uses modeling languages, which include Statecharts, Sequence Diagrams, Activity-Charts, and Module-Charts. These languages enable the user to develop various behavioral, functional, and structural views of a system. Statecharts and Sequence Diagrams examine the 'behavior' of systems by exploring conditions and events that cause transitions between states. A Statechart can be considered a generalized state-transition diagram with multi-level decomposition. UML statecharts specified by the OMG (Object Management Group) are very similar, but differ in semantics when compared to STATEMATE's statecharts. The OMG defines implementation-level semantics whereas STATEMATE uses requirements-level semantics [53]. As an example, an event will live arbitrarily long and actions take time with implementation-level semantics as opposed to STATEMATE, where events exist for one step and actions are instantaneous. The 'functional' view of the system is embodied in *activity-charts*, which determine the processes, functions, and the flow of information by using graphical diagrams. Module-charts are useful in analyzing the actual 'structure' of the system from a hierarchical sub-system perspective.

Let us inspect a simple example of an automobile's gear controller that demonstrates the diagrammatic capabilities of STATEMATE. Figure 6, adapted from Ecosoft's project 10407 [54], contains the top-level design of the controller, including the major sub-activities and the interfaces between these activities and the external environment: the driver motor, mechanical gearbox, and clutch. In this example, the activities are obtained by completing a structural analysis of a completed system. Notice that only the data flow and not the control flow is present in order to reflect the real system, since control flow is an abstract entity. A hierarchy of sub-activities can quickly be formed allowing for a more understandable model by transferring the unnecessary details to other diagrams.

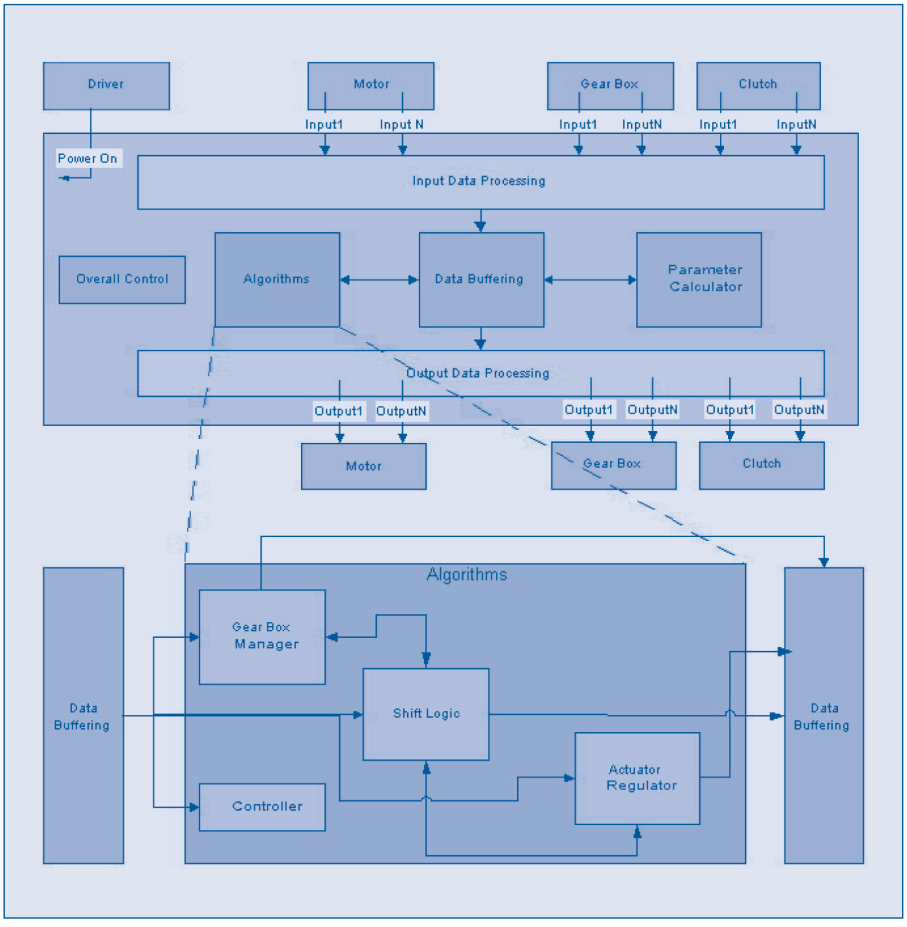

Figure 6 Activity chart for a gear controller

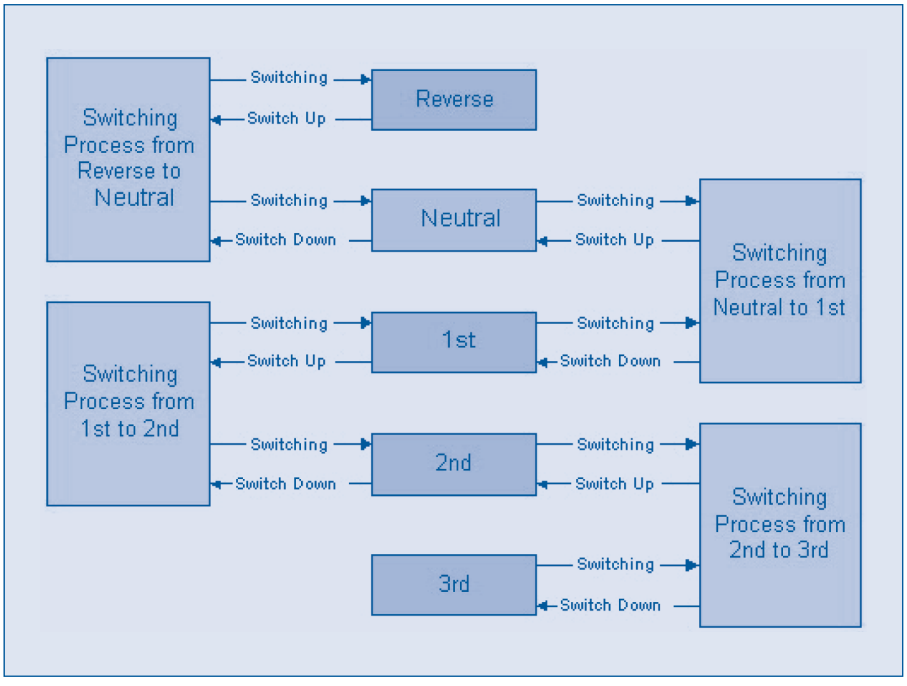

Figure 7 Statechart for transitions between gears

The 'Algorithms' activity in Figure 6 is broken out as a separate activity chart permitting the user to create a detailed specification without the additional clutter.

Figure 7 models the transition between gears and demonstrates how a Statechart can quickly be created for an activity in an activity diagram. Once the activity charts and Statecharts have been completed STATEMATE can take the specification for the gear controller and generate C or ADA code that will allow the user to run simulations. For example STATEMATE would allow the user to enter an input to the motor, such as the amount of gas pedal depression. We would then see the results of our input, hopefully through a successful gear change.

To achieve a better understanding of the power of STATEMATE let us briefly look at the theoretical underpinnings by examining one of the diagrammatic languages in more detail. Statecharts are similar to finite state machines and are represented by the tuple <S, T, E, V> where S is a set of states, T is a set of transitions, E is a set of events and V is a set of variables. There are three possible states: BASIC, OR, AND. BASIC states have no sub-states, OR and AND states have sub-states, known as orthogonal components, that are related relatively by the *exclusive-or* relation plus the *and* relation. Being in an OR state means being in only one of its sub-states, dissimilarly, being in an AND state implies being in all of its orthogonal components. Changes among states are represented by a transition (i.e., event [condition] / action). An event is an instantaneous occurrence of a stimulus (trigger), a condition is a predicate that must be satisfied for a transition to occur and an action may generate other events or perform computations. Thus, Statecharts = finite state machines + depth + orthogonality + broadcast. The depth is achieved by OR states and orthogonality is achieved by AND states. Broadcast is used to communicate among states and is achieved by the action of a transition. In other words, when a transition is triggered, an action generates an event, which we assume is to be globally broadcast [55]. Statecharts (with STATEMATE tool-support) provide a natural way to specify complex reactive systems both in terms of how objects communicate and collaborate and how they carry out their own internal

behavior. Together Activity-charts, Sequence Diagrams and Statecharts are used to describe the system functional building blocks, activities and the data that flows between them. The module charts are most useful at gaining knowledge of the physical structure of the system. These languages are highly diagrammatic in nature, constituting full-fledged visual formalisms, complete with rigorous semantics providing an intuitive and concrete representation for inspecting and checking for conflicts [56].

In conclusion, STATEMATE provides an excellent array of tools that can aid in the requirements, specifications, and design phases of a project [57]. Due to its visual nature STATEMATE allows customer and engineering staff to minimize design miscommunication [54]. One of the only drawbacks of STATEMATE is that the generated code can be large and unwieldy, however, once running, it does provide a venue for quality control at the earliest stages of development.

Z/EVES

Z/EVES [58][59] is an interactive software tool developed by Odyssey Research Associates (ORA), Canada, which integrates the Z [12][60] specification language with EVES [61] — an automated deduction system of ORA. The Z's notations are based on some mathematic structures like sets, sequences, relations and functions. It is mainly used to write a system's specification by defining all states of a system [62], and the ways in which states may change. It can also be used to describe system properties by defining some theorems about the system, and to reason about possible refinements of a system design. Though Z is flexible for specifying a system's function and structure, it is not intended for the description of non-functional properties, such as usability, performance, size, and reliability, neither is it intended for the description of timed or concurrent behaviors.

The Z/EVES has a GUI, which can facilitate the development of a specification with almost all Z notations. Figure 8 provides you with the specification window of Z/EVES. Furthermore, Z/EVES can enable you to analyze or explore a specification in several ways: (1) Syntax and type checking, (2) Schema expansion, (3) Precondition calculation, (4) Domain checking, and (5) General theorem proving.

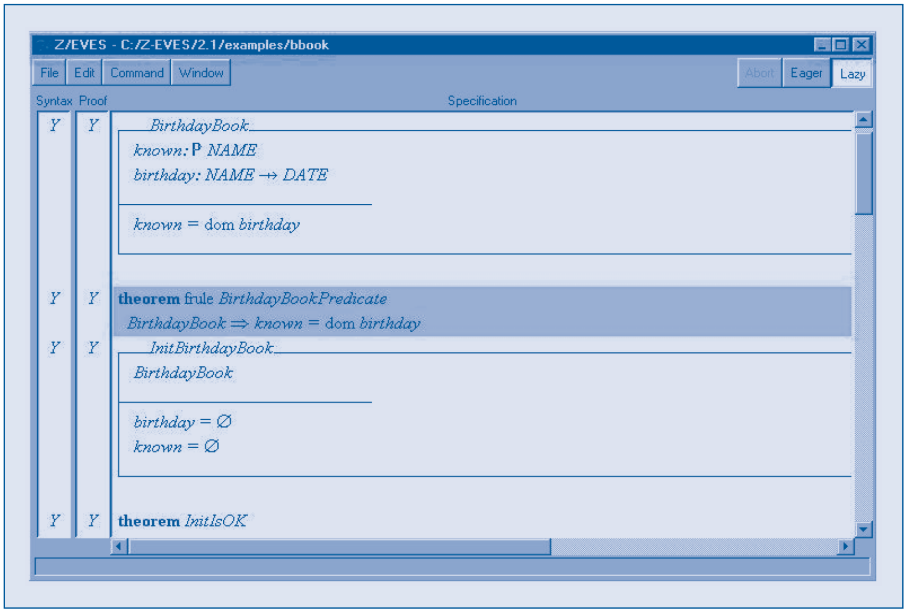

Figure 8 The specification window of Z/EVES

Though Z/EVES is a useful tool to precisely write down a system's specification and to reason about the specification, two main drawbacks prevent it from being a handy tool for many system designers. The first drawback is that it is inconvenient to input the various irregular mathematic symbols of Z when composing a specification. Although Z/EVES provides a soft-keyboard where you can directly select with mouse whatever symbols you need, it is still a tedious task. The second drawback is related to its ability to reason about a specification's properties defined in terms of theorems. Although Z/EVES provides some high-level proof commands, which can automatically carry out part of the proof work, the inherent hardness of theorem proving still makes exploration of a specification nontrivial and challenging.

One of the notable examples of Z's application is the success of using Z to formalize part of IBM's Customer Information Control System [60]. Measurements taken by IBM throughout the development process indicated an overall improvement in the quality of the product, a reduction in the number of errors discovered, and earlier detection of errors found in the process. IBM also estimated a 9% reduction in the total development cost of the new release. In [63], Z proof is compared with various types of testing approach, and *"the most striking result is that the Z proof appears to be substantially more efficient at finding faults than the most efficient testing phase"*. However, Z's notations being too abstract has the flexibility to describe complex systems but it makes it not an easy task to work out such an abstract description. Though Z/EVES's is useful to reason about some desired properties about a system specification, it requires too much expertise on theorem proving. And these two aspects still hinder them from being generally accepted by system designers.

Summary

We have introduced some basics of formal methods and five typical software tools that can be used for rigorous design and analysis. It is indisputable that these software tools, together with other tools that are not fully discussed here, such as the Alloy Constraint Analyzer [64], Möbious [65] etc., have greatly facilitated the application of their underlying formal methods. However, the effectiveness and applicability of these software tools are also restricted by the limitations of their underlying formal methods. An exciting research trend [66] is to combine several formal methods into one software tool to eliminate some restrictions. For an example, the PVS system has adopted a

model-checking technique during the theorem proof process and thus, the degree of automation of proof can be improved. Another trend is that some traditional design tools [67] and modeling languages [68][69] are including some kinds of formal notations and reasoning functionalities, which can be seen as a good sign of the realization and acceptance of the importance of formal methods by general software practitioners.

The authors would like to express thanks to the reviewers for their kind and knowledgeable critique. They provided practical and concrete suggestions. We wish also to thank Dr Stefan Greiner, Markus Degen and Dr Juergen Schwarz at DaimlerChrysler Research Information and Communication for their discussion of *operational* and *conceptual* validation.

In addition, we thank two WSU Graduate Students: Geir Bjune for his contribution regarding Büchi Automata and Hye Yeon Kim for her contribution regarding Statemate. Finally, we thank C. Michael Holloway at NASA LaRC for pointing out the oxymoron viz. specification is sometimes (appropriately) misspelled *specifiction*!

References

[1] M. Clarke and J.M. Wing, *Formal Methods: State of the Art and Future Directions.* ACM Computing Surveys, 1996. **28**(4): p. 626–643.

[2.] D. Craigen, S. Gerhart, and T. Ralston, *Formal Methods Reality Check: Industrial Usage.* IEEE Transactions on Software Engineering, 1995. **21**(2): p. 90–98.

[3.] D.L. Parnas and D.K. Peters. *An Easily Extensible Toolset for Tabular Mathematical Expressions*. in *Proceedings of the Fifth International Conference on Tools And Algorithms For The Construction And Analysis Of Systems (TACAS '99)*. 1999. Amsterdam, Netherlands.

[4.] J. Kirby, J. Archer, and C. Heitmeyer. *SCR: A Practical Approach to Building a High Assurance COMSEC System*. in *Proceedings of the 15th Annual Computer Security Applications Conference (ACSAC '99)*. 1999: IEEE Computer Society Press.

[5.] A. Hall, *Seven Myths of Formal Methods.* IEEE Software, 1990. **7**(5): p. 11–19.

[6.] C. Heitmeyer, J. James Kirby, B. Labaw, M. Archer, and R. Bharadwaj, *Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications.* IEEE Trans. on Software Engineering, 1998. **24**(11): p. 927–948.

[7.] J. Crow and B.D. Vito, *Formalizing space shuttle software requirements: four case studies.* ACM Transactions on Software Engineering and Methodology, 1998. **7**(3): p. 296–332.

[8.] M. Hinchey and J. Bowen, *Industrial-Strength Formal Methods in Practice*. Formal Approaches to Computing and Information Technology series (FACIT), ed. S.A. Schuman. 1999, London: Springer-Verlag.

[9.] I. Sommerville, *Software Engineering*. 6th ed. 2001: Addison Wesley.

[10.] C.A.R. Hoare, *Communicating Sequential Processes*. 1985, London: Pretice-Hall.

[11.] J.L. Peterson, *Petri Net Theory and the Modeling of Systems*. 1981, New York: McGraw-Hill.

[12.] J. Jacky, *The Way of Z: Practical Programming with Formal Methods*. 1997: Cambridge University Press.

[13.] J.V. Guttag and J.J. Horning, *Larch:Languages and Tools for Formal Specification*. 1994, Heidelberg: Springer-Verlag.

[14.] E. Brinksma and T. Bolognesi, *Introduction to the ISO specification language LOTOS.* Computer Networks and ISDN Systems, 1987. **14**(1): p. 25–59.

[15.] K.R. Apt, *Ten Years of Hoare's Logic: A Survey — Part I.* ACM Transactions on Programming Languages and Systems, 1981. **3**(4): p. 431—483.

[16.] D.A. Cyrluk and M.K. Srivas. *Theorem Proving: Not an Esoteric Diversion, but the Unifying Framework for Industrial Verification*. in *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors (ICCD '95)*. 1995. Austin, TX.

[17.] S. Agerholm. *Mechanizing Program Verification in HOL*. in *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*. 1991. Davis, California: IEEE Computer Society Press.

[18.] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. *PVS: Combining Specification, Proof Checking, and Model Checking*. in *Proceedings of the Computer-Aided Verification, CAV '96*. 1996: Springer-Verlag.

[19.] K.L. McMillan, *Symbolic Model Checking — an Approach to the State Explosion Problem*, in *SCS*. 1992, Carnegie Mellon University: Pittsburgh.

[20.] G.J. Holzmann, *The Model Checker Spin.* IEEE Trans. on Software Engineering, 1997. **23**(5): p. 279–295.

[21.] E.M. Clarke, O. Grumberg, and D.A. Pled, *Model Checking*. 1999, Cambridge, Massachuesetts: The MIT Press.

[22.] N. Wirth, *Program Development by Stepwise Refinement.* Communications of the ACM, 1971. **14**(4): p. 221–227.

[23.] R.J. Back and J.v. Wright, *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. 1998: Springer-Verlag.

[24.] E.W. Dijkstra, *A Discipline of Programming*. 1976: Prentice-Hall.

[25.] M. Butler, J. Grundy, T. Långbacka, R. Ruksenas, and J.v. Wright. *The Refinement Calculator: Proof Support for Program Refinement*. in *Formal Methods Pacific'97:Proceedings of FMP'97*. 1997. Wellington, New Zealand: Springer-Verlag.

[26.] C.B. Jones, *Systematic Software Development Using VDM*. 2 ed. Prentice Hall International Series in Computer Science. 1990, London: Prentice Hall International.

[27.] J. Hörl and B.K. Aichernig, *Validating Voice Communication Requirements Using Lightweight Formal Methods.* IEEE Software Magazine, 2000. **17**(3): p. 21–27.

[28.] J.R. Abrial, *The B Book*. 1996: Cambridge University Press.

[29.] Atelier B http://www.atelierb.societe.com/PAGE_B//uk/atb-01.htm,

[30.] B-core, http://www.b-core.com/,

[31.] F.T. Sheldon and S.A. Greiner, *Composing, Analyzing and Validating Software Models to Assess the Performability of Competing Design Candidates.* Annals of Software Engineering –Special Volume on Software Reliability, Testing and Maturity, 1999. 8: p. 239–287.

[32.] S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert, *PVS: An Experience Report*, in *Applied Formal Methods — FM-Trends 98*, D. Hutter, W. Stephan, P. Traverso, and M. Ullman, Editors. 1998, Springer-Verlag: Boppard, Germany. p. 338—345.

[33.] J. Rushby, *Specification, Proof Checking, and Model Checking for Protocols and Distributed Systems with PVS*. 1997, SRI International Computer Science Laboratory.

[34.] D.W.J. Stringer-Calvert, S. Stepney, and I. Wand. *Using PVS to Prove a Z Refinement: A Case Study*. in *Proceedings of the FME '97: Formal Methods: Their Industrial Application and Strengthened Foundations*. 1997: Springer-Verlag.

[35.] J. Rushby, *A Brief Overview of PVS.*, SRI International Computer Science Laboratory.

[36.] H. Saïdi and N. Shankar. *Abstract and Model Check while you Prove*. in *Proceedings of the Computer-Aided Verification (CAV'99)*. 1999. Trento, Italy: Springer-Verlag.

[37.] S. Koo, H. Son, and P. Seong, *Mathematical Verification of a Nuclear Power Plant Protection System Function with Combined CPN and PVS.* Journal of the Korean Nuclear Society, 1999. 31: p. 157–171.

[38.] D. Monniaux. *Decision Procedures for the Analysis of Cryptographic Protocols by Logics of Belief*. in *Proceedings of the 12th Computer Security Foundations Workshop*. 1999. Mordano, Italy: IEEE Computer Society.

[39.] G.J. Holzmann and A. Puri, *A Minimized automaton representation of reachable states.* Software Tools for Technology Transfer, 1999. **3**(1).

[40.] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. *Simple On-the-fly Automatic Verification of Linear Temporal logic*. in *Proceedings of the PSTV 1995 Conference*. 1995. Warsaw, Poland.

[41.] M.Y. Vardi and P. Wolper. *An Automata-theoretic Approach to Automatic Program Verification*. in *Proceedings of the First IEEE Symposium on Logic in Computer Science*. 1986.

[42.] G.J. Holzmann, *Protocol Design: Redefining The State of the Art.* IEEE Software, 1992. **9**(1): p. 17–22.

[43.] R. Pike, D. Presotto, K. Thompson, and G.J. Holzmann. *Process Sleep and Wakeup on a Shared-Memory Multiprocessor*. in *Proceedings of the Spring EurOpen Conference*. 1991. Tromso, Norway.

[44.] R. Bharadwaj and C. Hemeyer. *Verifying SCR Requirements Specifications Using State Exploration*. in *Proceedings of the First ACM/SIGPLAN Workshop Automatic Analysis of Software*. 1997. Paris, France.

[45.] T. Cattel. *Using Concurrency and Formal Methods for the Design of Safe Process Control*. in *Proceedings of PDSE/ICSE018 Workshop*. 1996. Berlin, Germany.

[46.] A. Joesang. *Security Protocol Verification Using SPIN*. in *Proceedings of the First SPIN Workshop*. 1995. INRS Quebec, Canada.

[47.] H.E. Jensen, K. Larsen, and A. Skou. *Modeling and Analysis of a Collision Avoidance Protocol Using SPIN and UPPAAL*. in *Proceedings of the Second SPIN Workshop*. 1996. Rutgers Univ., New Brunswick, N.J.: American Mathematical Society.

[48.] G.J. Holzmann, *Design and Validation of Computer Protocols*. 1991, Englewood Cliffs, N.J.: Prentice Hall.

[49.] K. Havelund, M. Lowry, and J. Penix, *Formal Analysis of a Space-Craft Controller Using SPIN*. IEEE Trans. on Software Engineering, 2001. **27**(8): p. 749–765.

[50.] *UltraSAN User's Manual: Version 3.0*. 1995, Center for Reliable and High-Performance Computing Coordinated Science Laboratory, University of Illinois at Urbana-Champaign.

[51.] K. Goseva-Popstojanova and K. Trivedi, *Stochastic Modeling Formalisms for Dependability, Performance and Performability*. Lecture Notes in Computer Science, 2000(1769): p. 403–422.

[52.] J. Couvillion, R. Freire, R. Johnson, W.D. Obal, M.A. Qureshi, M. Rai, W. Sanders, and J. Tvedt, *Performability Modeling with UltraSAN*. IEEE Software, 1991. **8**(5): p. 69–80.

[53.] R. Eshuis and R. Wieringa, *Requirements-level Semantics for UML Statecharts*, in *Formal Methods for Open Object-Based Distributed Systems IV — Proc. FMOODS'2000, September, 2000, Stanford, California, USA*. 2000, Kluwer Academic Publishers.

[54.] W. Gerst, A. Imminger, and R. Pfister, *Improvements in Specification, Design and Test of Software for Industrial and Automotive Controllers*. 1996, Getreg.

[55.] S.D. Cha and H.S. Hong. *Specification and Analysis of Real-Time Systems in Statecharts*. in *Proceedings of the 2nd International Workshop on Object-Oriented Real-Time Systems '96*. 1996.

[56.] D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts*. 1998: McGraw-Hill.

[57.] D. Harel, *From Play-In Scenarios to Code: An Achievable Dream*. IEEE Computer Magazine, 2001. **34**(1): p. 53–60.

[58.] I. Meisels and M. Saaltink, *The Z/EVES Reference Manual*. 1997.

[59.] M. Saaltink. *The Z/EVES System*. in *Proceedings of the10th International Conference of Z Users*. 1997. Reading, UK: Springer-Verlag.

[60.] J. Davies and J. Woodcock, *Using Z: Specification, Refinement and Proof*. 1996: Prentice Hall International Series in Computer Science.

[61.] D. Craigen, S. Kromodimoeljo, I. Meisels, B. Pase, and M. Saaltink. *Eves System Description*. in *Proceedings of the Conference on Automated Deduction*. 1992.

[62.] D. Craigen, I. Meisels, and M. Saaltink, *Analysing Z Specifications with Z/EVES*, in *Industrial-Strength Formal Methods in Practice*, J.P. Bowen and M.G. Hinchey, Editors. 1999, Springer-Verlag: London. p. 255–285.

[63.] S. King, J. Hammond, R. Chapman, and A. Pryor, *Is Proof More Cost-Effective Than Testing?* IEEE Trans. on Software Engineering, 2000. **26**(8): p. 675–686.

[64.] D. Jackson. *Automating first-Order Relational Logic*. in *Proceedings of the ACM SIGSOFT Eighth International Symposium on the Foundations of Software Engineering*. 2000. San Diego CA.

[65.] D. Daly, D.D. Deavours, J.M. Doyle, P.G. Webster, and W.H. Sanders. *Möbius: An Extensible Tool for Performance and Dependability Modeling*. in *Computer Performance Evaluation: Modelling Techniques and Tools: Proceedings of the 11th International Conference*. 2000. Schaumburg, IL: Springer-Verlag.

[66.] S. Rajan, N. Shankar, and M.K. Srivas. *An integration of model checking with automated proof checking*. in *Proceedings of the 1995 Workshop on Computer-Aided Verification*. 1995: Springer-Verlag.

[67.] T. Quatrani, *Visual Modeling with Rational Rose 2000 and UML*. 2000: Addison Wesley.

[68.] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modeling with UML*. Object Technology Series. 1999: Addison Wesley.

[69.] OMG, Unified Modeling Language (UML) Specification, 2001. http://www.omg.org/technology/documents/formal/uml.htm