Cost Effective Use of Formal Methods in Verification and Validation

D. Richard Kuhn   Ramaswamy Chandramouli
National Institute of Standards and Technology
Gaithersburg, MD 20899

Ricky W. Butler
NASA Langley Research Center
Hampton, VA

Abstract

Formal methods offer the promise of significant improvements in verification and validation, and may be the only approach capable of demonstrating the absence of undesirable system behavior. But it is widely recognized that these methods are expensive, and their use has been limited largely to high-risk areas such as security and safety. This paper focuses on cost-effective applications of formal techniques in V&V, particularly recent developments such as automatic test generation and use of formal methods for analyzing requirements and conceptual models without a full-blown formal verification. We also discuss experience with requiring the use of formal techniques in standards for commercial software.

Contents

1 Introduction

According to the Department of Defense Verification, Validation, and Accreditation (VV&A) Recommended Practices Guide [DMSO, 2001], formal methods "are based on formal mathematical proofs of correctness and are the most thorough means of model V&V." The Recommended Practices Guide goes on to state that

> *If attainable, a formal proof of correctness is the most effective means of model V&V. Unfortunately, "if attainable" is the sticking point. Current formal proof of correctness techniques cannot even be applied to a reasonably complex simulation; however, formal techniques can serve as the foundation for other V&V techniques* [DMSO, 2001]

In this paper we describe applications of formal techniques both for software verification as well as for other practices useful in V&V, in particular generation of complete system tests from specifications Because there is a recognized need for certification and accreditation within the field of modeling and simulation (M&S), we discuss historical experience with the inclusion of formal methods in government standards. In all areas, our focus is on cost effectiveness, and on using formal techniques as an aid in reasoning about systems and software, rather than absolute "proof of correctness", which may be unattainable. From a cost standpoint, formal methods are not always the best methods to apply in V&V. The challenge for both developers and certifiers is to determine when formal techniques can be applied effectively, and what processes make their application profitable in terms of cost reduction and prevention of failure.

We draw on two fields that have long established experience with the application of formal techniques to software requiring government certification: computer security and software safety. Formal methods have taken hold, although not extensively, in these fields partially because the cost of failure can be high, but also because in both fields there is a need to prove the absence of undesirable behavior, in addition to the presence of designed-in features and behavior. Modeling and simulation presents unique challenges, such as the need to emulate the real world and to extrapolate into unknown regions. But M&S also has much in common with other high-integrity software. Notably, M&S systems are often very large, made up of many components whose correct interoperation is difficult to predict. In addition, there is a need to demonstrate the absence of undesired behavior, not simply verify the existence of particular features, and above all a need to conduct extensive system testing regardless of what other assurance techniques have been applied. Formal techniques have been designed to assist in all of these tasks. If used properly, they can be cost effective. Engineering judgment is required to determine when and how to use formal methods. This paper surveys the available techniques, suggests how they can be used effectively, and provides some lessons learned from a wide variety of projects that have applied formal methods.

2 Formal Methods and Software Assurance

What are formal methods and how can they be useful for VV&A? To frame discussion of this question, consider the (somewhat idealized) diagram in Figure 1of verification that must be done for any software system. In addition, the requirements, specifications, and the completed system must be validated against real world needs, a problem that is particularly acute for M&S systems, which must emulate the behavior of physical entities rather than interact with them.

In the idealized structure shown in Figure 1, formal techniques can be used in the following places:

- Requirements policy – For a secure system, these may be the major security properties that must be preserved by the system (called the formal security policy model), such as confidentiality or integrity of data; for a system requiring high dependability, essential properties may include freedom from deadlock.

- Specifications – The formal specification is typically a mathematically based description of system behavior, using state tables or mathematical logic. It will not normally describe lowest level software, such as mathematical subroutine packages or data structure manipulation, but will describe the response of the system to events and inputs to a degree necessary to establish critical properties. Engineering judgment is required to determine the proper level of depth in the specification.

- Proof of correspondence between specification and requirements – It must be shown that the system, as described by the specification, establishes and preserves the properties in the requirements policy. If both are in a formal notation, rigorous proofs can be constructed, either manually or with machine assistance.

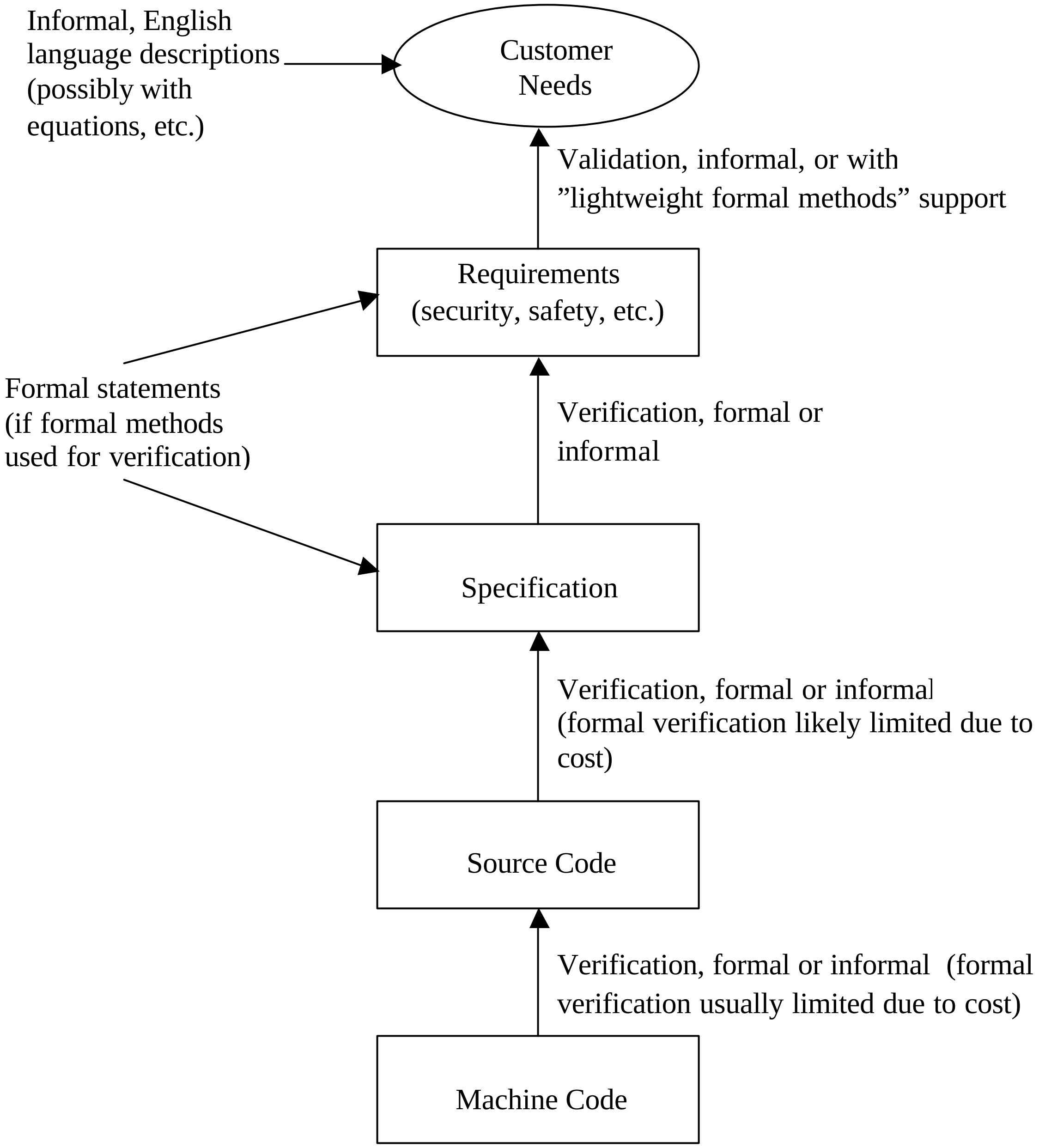
- Proof of correspondence between source code and specifications – Although many formal techniques were initially created to provide proof of correctness of code, this is rarely done because of the time and expense involved, but may be done for particularly critical portions of the system.

- Proof of correspondence between machine code and source code – This type of proof is rare, both because of the expense involved and because modern compilers are very reliable.

In Figure 1, arrows point upward, intended to convey the property that must be shown: an implication from a lower component to a higher component of the diagram. Formally, if S is the specification and R represents requirements, verification means proving $S \Rightarrow R$. In essence, S is more specific than R. This is as we would expect, because there are multiple ways to implement most requirements.

In addition to defining properties and proving that the properties are preserved, formal specifications make it possible to generate test cases that can be used to exercise the system and demonstrate correct functions. In this paper we review the varieties of formal techniques and tools that are available for system and software verification, and suggest ways that these methods may be useful for M&S systems. In addition, we consider some possibilities for use of formal methods in the validation process, an application of formal techniques that is comparatively unexplored.

Figure 1. Software Assurance Processes


Informal, English language descriptions (possibly with equations, etc.)
Customer Needs
Validation, informal, or with ”lightweight formal methods” support
Requirements (security, safety, etc.)
Formal statements (if formal methods used for verification)
Verification, formal or informal
Specification
Verification, formal or informal (formal verification likely limited due to cost)
Source Code
Verification, formal or informal (formal verification usually limited due to cost)
Machine Code

2.1 Improving Precision in Specifications

One of the most widely recognized problems in software development is the difficulty of clearly specifying expected software behavior. This problem is particularly acute with the trend toward component-based development. Frequently a developer has only a textual description of what a callable procedure does, its allowed inputs and expected outputs.

Ensuring that component-based software is dependable is a difficult problem, not only because of the size and complexity of the software, but also because source code is often not available for components that are acquired rather than developed in-house. The trend toward use of acquired components is occurring in all fields of software development, and writers have argued that modeling and simulation software could move toward this mode of development as well, because of the cost reduction advantages of component based software.

But the initial cost of constructing software is only one part of the picture. Cost effective methods of assurance are also essential for complex component-based systems. The fundamental ingredient in providing rigorous assurance is a *precise definition of the software's expected behavior*. Formal specifications provide the greatest precision for software, and formal methods will be increasingly important as M&S development moves toward greater use of standardized, off-the-shelf components.

Many observers have noted that the process of developing a formal specification is often as effective for finding errors as the verification effort in which the specification is to be used. Developing a formal specification requires a detailed and precise understanding of the system, which helps to expose errors and omissions, so much of the benefit of using formal methods comes from the process of developing the specification [Clarke and Wing, 1996, Johnson et al., 1999]. In formalizing the system description, ambiguities and omissions are detected, and the formal specification can improve communication between developers and customers [Brilliant, Knight and Elder, 1996].

2.2 Analyzing and Proving Properties of Systems and Specifications

Above all else, the field of formal methods was developed to allow reasoning about systems and software. Once a formal specification has been constructed, it is possible to analyze, manipulate, and reason about it just like any other mathematical expression. A significant difference between formal specifications of software and the more familiar mathematical expressions of algebra or calculus, however, is that specifications are typically much larger - often hundreds or thousands of lines. To deal with the size and complexity of these expressions, many varieties of software tools have been constructed. These tools can be grouped broadly into two categories: *theorem proving tools* and *model checkers*.

Theorem proving tools assist the user in constructing proofs, generally to show that the specification has desired properties such as absence of deadlock or various security properties. These tools require a degree of skill to use, but they can handle very large specifications with complex properties. A more recent development in formal methods has been the introduction of model checkers, which explore enormous state spaces that cover, to some degree, all possible executions of the specified program. They can demonstrate that the specified program has a desired property by exploring all possible executions, or produce counterexamples where the property does not hold. Although these tools can be made fully automatic, they cannot handle problems as large or varied as theorem proving tools. The most sophisticated tools combine aspects of both model checkers and theorem provers, doing model checking on some parts of the specification, but relying on user guidance to prove difficult properties. This section surveys the types of tools that are available, the tradeoffs involved, and appropriate problem domains for the different tools.

2.2.1 Theorem Proving Tools

Since both a computer program and its specification are mathematical entities, it is reasonable to ask whether a computer program can be constructed that will decide whether a given program meets its specification. Unfortunately this problem is *undecidable.* Early in the 20th century Kurt Gödel, Alan Turing, Alonzo Church, and other mathematicians demonstrated that it is fundamentally impossible to develop algorithms that solve this problem and other similar ones. In fact, these mathematicians showed that any logical system expressive enough for mathematics is inherently undecidable as well.[1] So we are left with a basic choice: restrict our systems of interest to a decidable domain or rely upon human intelligence. Model checking is an example of the former and theorem proving is and example of the latter.

Theorem proving tools have been developed to aid the human in demonstrating that a program meets its specification. There are many theorem proving tools available including: PVS, ACL2, HOL, Isabelle, Nuprl, Z/Eves, SCR, and the B-Tool (See www.afm.sbu.ac.uk for a fuller list.) Instead of examining these tools in detail, this section will seek to answer the question of why there is such a wide variety of tools.

Cause 1: There are a lot of different logics out there. Probably the most basic of divisions is whether the theorem prover is based upon a set theory such as ZFC or higher-order logic. Both of these provide a foundation rich enough to support all of mathematics. Z/Eves is an example of a theorem prover based upon set theory, whereas PVS is an example of a theorem prover based upon type theory. Within these broad categories are

[1] At first this is puzzling since mathematicians prove theorems every day. But unless one is a naturalist who assumes that man is 100% machine, this is not really a problem. If man is assumed to be spiritual in addition to bio-chemistry, the problem disappears. As Gödel himself put it, "Either mathematics is too big for the human mind, or the human mind is more than a machine." See Roger Penrose's excellent book, Shadows of the Mind for a full discussion of this issue.

many subdivisions arising from fundamental difficulties: such as (1) how does one handle division by zero or other undefined results or (2) how are you going to deal with time.

Cause 2: Degree of Customization to problem domain. The ultimate goal of the theorem prover developers is to provide a proof environment that is powerful and easy to use. There are different ideas about how to make a theorem prover powerful and at the same time cost-effective with respect to a user's time. Some developers believe that the key is to provide a tool tailored to a particular problem domain. In this approach the developers deliberately restrict the problem domain so that notations and proof commands can be tailored to that particular problem domain. A good example of this is the Naval Research Lab's SCR (Software Cost Reduction) toolkit. The SCR tool provides a table-oriented specification language tailored to the needs of a designer of safety-critical applications rich in mode logic. On the other end of the spectrum are the developers of fully general-purpose tools that make no assumptions about the problem domain.

Cause 3: Degree of automation. Even though a theorem prover is working in an undecidable domain, subgoals often fall completely into a fragment of mathematics that is fully automatible such as linear arithmetic, pure propositional formulas, extensional arrays, bit-vectors and many others. Fully automated *decision procedures* have been developed for these fragments. But what about formulas that involve constructs from more than one of these fragments at the same time? The integration of multiple decision procedures in an efficient and useful way has been one of the fundamental pursuits of many of the theorem prover developers for many years. Without decision procedures the user of a theorem prover is forced to carry out even the most trivial of proof steps in a manual way. The integration of decision procedures with the other capabilities of a theorem prover such as automatic rewriting is an enormously challenging endeavor yet can lead to a much more powerful theorem prover. The PVS theorem prover is an excellent example of a theorem prover where the tight integration of decision procedures into a general-purpose proof environment has resulted in a powerful proof tool but with a complex user interface.

Cause 4: Use of heuristic search. Given an undecidable domain another option is available, namely heuristic search. Although not guaranteed to terminate or find a solution, a heuristic search can sometimes find a deduction, and save the user a lot of time. Of course there is a trade off between how long one is willing to wait around for heuristic search and the probability of success. Theorem provers can supply commands that perform such searches or provide mechanisms such as tactics that enable the user to create his own search methods.

Cause 5: Degree of certainty in the soundness of the theorem prover. The integration of decision procedures, automatic rewriting, heuristic search, and other techniques in a theorem prover can lead to an extremely complex piece of software. If there is a bug in the theorem prover, the soundness of the prover may be compromised. A soundness bug (if not discovered) could deceive a user into thinking he has a legitimate proof when the proposition is actually false. Needless to say, this is a primary concern of all theorem prover developers. However, they differ greatly on their approaches to solving this problem. Some have sought to develop small deduction kernels on which the soundness depends. The tool is engineered in a manner so that software external to that kernel cannot compromise soundness. The ACL2 and HOL theorem provers were developed this way. Unfortunately this approach makes the integration of decision procedures into the tool much more difficult.

2.2.2 Model Checkers

One of the challenges faced by formal methods researchers is the need to make formal techniques available to the broadest possible audience of developers. Model checkers attempt to make formal techniques easier to use by providing a high degree of automation at the expense of generality. Inputs to a model checker are typically a finite state model of a system, along with a set of properties that are expected to be maintained by the system. Properties to be verified can usually be categorized as one of the following [Janssen et al., 1999]:

1. Correct sequences of events
2. Proper consequences of activities
3. Simultaneous occurrence of particular events
4. Mutual exclusion of particular events
5. Required precedence of activities

The model checker effectively explores all possible event sequences of the finite state model to determine if properties, typically expressed in temporal logic, always hold. Because the model is finite, the state space search is guaranteed to terminate. If properties hold, the model checker outputs a confirmation. If a property fails to hold for some possible event sequences, the tool produces counterexamples, i.e., traces of event sequences that lead to failure of the property in question. A useful trick that can be applied in model checking is to specify that the *negation* of a desired property that should hold. In this case, the model checker produces an event trace that results in a failure of the negated property. This event sequence, therefore, is actually a valid trace of the system, which can be postprocessed to generate complete test cases (i.e., both test input and expected system result) [Ammann, Black, Majurski, 1998].

The two greatest advantages of model checkers are their fully automatic analysis and their ability to produce counterexamples that can be used in testing or other analysis. Their disadvantage results from the tradeoffs made to make automation possible. Because they effectively explore a complete state space, the complexity of analysis normally grows exponentially with the size of the model. To reduce the size of the finite model, abstractions must be used, such as reducing the range of discrete variables, or using a simple predicate to represent a more complex condition. As a result, model checkers can only be used to analyze an abstraction of the system, rather than a complete model, so they are sometimes regarded as debugging tools (for specifications) rather than verification tools [Merz, 2001]. Despite their limitations, model checkers have been applied to significant real-world systems, particularly integrated circuit design and cryptographic protocols. Abstraction techniques make it possible to model systems with 200 variables, and up to 10^{120} reachable states [Burch et al., 1994]. Table 1 summarizes differences between model checkers and theorem proving tools.

	Model checkers	**Theorem provers**
Typical notation	FSMs, temporal logic	First order, higher order logics
Method of operation	Automatic	Semi-automatic
Output	Confirmation or counterexample	Proof (if successful)
Range of applicability	Bounded, finite models	Essentially unlimited

Table 1. Summary of model checker and theorem prover properties

2.3 Generating Test Cases from Formal Specifications

Formal verification techniques depend on mathematically precise specifications. But developing rigorous system tests also requires a precise, complete description of system functions, and practical system assurance always requires testing, even when formal methods are used. One of the most interesting applications of formal methods has been the development of tools that can generate complete test cases from formal specifications. Although a large number of "automated testing" tools are available on the market, most of these tools automate the more mundane aspects of software testing: generating test data, passing input data to the system under test, and recording results. Defining the correct system response for a given set of input data is the hard task that most tools cannot accomplish when system behavior is defined only with natural language specifications. Because the expected system response can only be determined by reading the specification, programmers are expected to provide this critical missing link in most automated testing systems. The great advantage of test generation tools based on formal methods is that a formal specification describes system behavior mathematically, so expected system responses for particular inputs can be generated; i.e., the tool can generate complete test cases, not just test data input or test scaffolding.

From a cost-benefit standpoint, generating tests from specifications can be one of the most productive uses of formal methods. Approximately half the staff time in a typical commercial software development effort is spent on testing. As computer users have recognized, even this level of effort only removes the most obvious flaws. Much of the software industry operates under a marketing strategy that gives feature richness and time to market a higher priority than quality, because users have demonstrated a willingness to accept bugs in return for more features. Some of the newer test generation tools hold the promise of improving quality while simultaneously reducing time to market, because less developer time is spent on programming test cases. These tools can also provide benefits for custom software, such as most M&S systems, by reducing the time spent on test development and thus allowing more time for other tasks. Some empirical measurements have shown that tests generated by these tools provide test coverage as good or better than that achieved by manually generated tests, so developers can choose between producing more tests in the same number of staff hours, or reducing the number of hours required for testing.

2.4 Using Formal Techniques in Validation

While the verifications shown in Figure 1 can be conducted semi-automatically, and proofs checked mechanically, *validation* is a different problem. A succinct distinction between verification and validation is that verification is “building the system right”, while validation is “building the right system”. If we have a set of requirements, we can verify, formally or informally, that the system implements the requirements. But validation is necessarily an informal process. Only human judgment can determine if the system that was specified and built is the right one for the job.

Despite the necessity of using human judgment in the validation process, formal methods do have a place in validation, particularly in large complex applications such as M&S. One of the most promising applications of formal techniques is the “lightweight” application of formal methods for requirements modeling. By stating requirements formally, theorem proving tools can be used to explore properties, often detecting conflicts between different requirements or missing assumptions. This approach does not replace human judgment, but can aid in determining if the “right system” has been specified by making it easier to determine if desired properties hold.

A significant difference between the validation problem for M&S systems and for software designed for control or calculation is that M&S systems have two types of validation requirements. By definition, the M&S system must model and predict behavior of some real world entity. This problem has been called “operational validation”. A second aspect to the validation problem for M&S systems is “conceptual model validation”, which is concerned with ensuring that the assumptions underlying the conceptual model are correct and that the logic and structure of the model are suitable for the model’s intended purpose [Sargent, 1999]. Where formal methods have been applied to validation, their use corresponds most closely to the problem of conceptual model validation. Figure 2 shows a view of the modeling process [Sargent, 1999; Stevenson, 2002], annotated to illustrate the applicability of formal techniques.

Because the conceptual model describes what is to be represented by the simulation, it must include assumptions about the system and its environment, equations and algorithms, data, and relationships between model entities. Although algorithms and equations are necessarily formal statements, the assumptions and relationships are most often described using natural language, which introduces the potential for ambiguities and misunderstandings between developers, users, and subject matter experts. A relatively recent trend in formal techniques, often called “lightweight formal methods” [Jackson, 2001], has shown a potential for detecting major errors in requirements statements, without the expense of a formal design verification, by applying formal analysis to earlier products of the system design process.

The basic premise of this approach is to use formal techniques in analyzing the assumptions, relationships, and properties of requirements stated in a requirements statement or conceptual model. An advantage of this approach is that it can be applied to

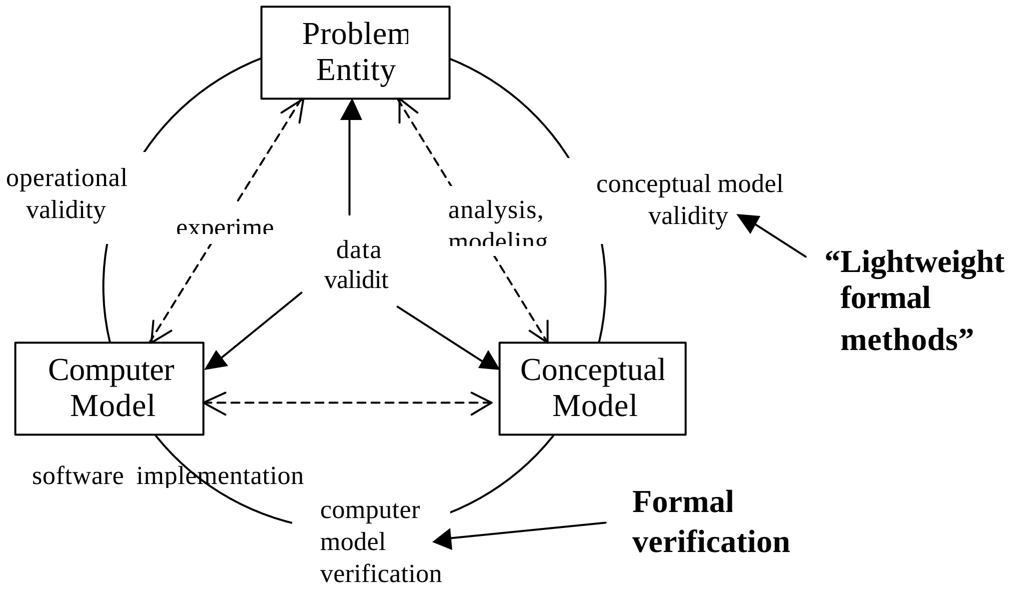


Figure 2. Applicability of formal techniques to M&S validation and verification.

partial specifications, or to a limited segment of a complete specification. The analysis is done in three phases [Easterbrook and Callahan, 1998]:

1. Restate the requirements and conceptual model in a formal (or semi-formal) notation, typically a state table description.
2. Identify and correct ambiguities, conflicts, and inconsistencies.
3. Use a model checker or theorem prover to study system behavior, demonstrate properties, and produce traces of system behavior. Developers, users, and subject matter experts can then use these results to improve the conceptual model.

The representational abstraction phase of conceptual model design uses a variety of modeling methods to represent simulation elements and their relationships [Pace, 2000; DMSO, 2000]. Notations such as those provided in the Unified Modeling Language are often effective. While standard UML does not contain sufficient formality to map directly to formal representations used by model checkers, some more recent additions to UML may make this possible. The Object Constraint Language [Warmer and Kleppe, 1998] includes elements of first order logic that, when combined with some of the state machine representations of UML, can provide a rigorous system specification. Because popular model checkers use state machine representations as input, a conceptual model defined with OCL would appear to have the potential for efficient translation into the input notations of either theorem proving tools or model checkers such as the popular SMV and SPIN tools.

A particularly interesting aspect of the "lightweight" approach to formal methods is that it has been used to model and analyze the behavior of software, hardware, and humans acting together in systems. Agerholm and Larsen [1997] describe the application of formal modeling to a NASA extravehicular activity system. Using the PVS theorem proving tool, the authors were able to model the EVA system and study its properties using a model that is essentially executable. Lutz [1997] describes requirements validation of onboard fault monitors for a spacecraft. An interesting aspect of this project is that developers were able to reuse the requirements model for a second project that evolved from the first in a series of builds. Janssen et al. [1999] describe the application of model checking to the analysis of automated business processes, such as insurance claim processing. The formal model is used to ensure that processes maintain desired properties, such as ensuring that the proper sequence of processing is maintained, that two mutually exclusive outcomes are prevented by the system, or that particular events always lead to the correct outcome. The formal analysis helps to prevent unexpected failures that can occur in large distributed systems where processes occur with partial human intervention. By modeling processes at the requirements stage, developers can identify problems that might require major rework if not detected until the system is built and tested.

3 Formal Methods in Real World V&V

To provide a better understanding of the use of formal methods in practice, this section surveys the use of formal techniques across a wide range of applications. Two excellent surveys of industrial applications of formal methods are Craigen, Gerhart and Ralston [1993], and Clarke and Wing [1996]. This section presents primarily recent work not described in these earlier surveys.

3.1 Examples From Aerospace Software Safety

As one of the few ways to attack the problem of ensuring that software does not produce undesirable results, formal methods have a natural application in safety-critical areas such as commercial aviation. This section describes the current work being pursued by the formal methods team at NASA Langley Research Center, which leads NASA initiatives in aviation safety. These project descriptions highlight participants, basic objectives of the work, and difficulties and challenges associated with this work.

Aviation Safety Program (AvSP)
(http://www.aero-space.nasa.gov/goals/safety.htm)

The Aviation Safety Program supports the practical application of formal methods to improve safety in commercial aviation. Among these are cooperative agreements with:

- Rockwell Collins Advanced Technology Center - to develop extensions to existing methods and commercial off-the-shelf tools that enable (1) requirements modeling and analysis, (2) safety analysis and partitioning, (3) mode confusion detection, and (4) auto-generation of code. The project is seeking to develop an automatic translator from RSML-e to PVS that can be used to verify safety properties and absence of features that have historically lead to mode confusion in flight guidance systems and flight management systems.

- Honeywell Engines and Systems - to develop a fault-tolerant integrated modular avionics architecture for a Full Authority Digital Engine Control (FADEC). This architecture is based on the Time Triggered Architecture (TTA) that has been developed over the past fifteen years at Vienna University of Technology. The challenge of this work is to develop formal proofs of the TTA architecture and to establish a basis for certification based on formal proof.

- Barron Associates/Goodrich - to develop formal verification methods that can serve as a basis for certifying non-adaptive neural nets. The goal is to develop methods and tools to guarantee that the neural net output will be bounded throughout its operating regime (with respect to look-up tables that they replace.) Barron/Goodrich are working with the FAA to get their tools and methods qualified as a method for certifying neural nets. This work is not addressing closed-loop behavior/stability of a NN within a controller

- Grant with the University of Virginia to develop tools that facilitate the integration of formal verification methods into the software development lifecycle within an aerospace company using a combination of natural language and formal languages.

Information Technology Strategic Research Program (ITSR) (http://www.nas.nasa.gov/IT/)

This project supports both external and in-house research, including:

- Cooperative agreement with Honeywell Technology Center (Minneapolis) and SRI International to develop and apply formal methods technology to DEOS, a partitioning real-time operating system used in the Honeywell's Primus Epic Flight deck that is being developed to DO-178B Level A certification standards. The Digital Engine Operating System™ supports time & space partitioning, dynamic thread creation, and slack reclamation. The task execution budgets are managed using rate monotonic scheduling. Earlier versions of DEOS did not support slack-time reclamation (i.e. when a task completes before its deadline, the re-allocation of this time to other tasks). The SPIN model checker was used to successfully analyze the system. However, the slack-time reclamation led to an enormous increase in the size of the state-space, far beyond the reaches of SPIN. Current work is focusing on the use of theorem proving to attack the large state space.

- Inhouse work to develop efficient means for reasoning about non-linear algebraic formulas in the PVS theorem prover. Previously when the PVS user encountered propositions that contained non-linear algebraic terms, he was forced to directly invoke the field axioms of the reals or related lemmas. The PVS strategy mechanism is being exploited to develop commands the mimic the way reasoning was actually done in high-school algebra. Commands include DIV-BY, MULT-BY, FACTOR, CROSS-MULT..

Aerospace Vehicle Systems Technology Program (AVST) (http://www.aero-space.nasa.gov/programs/vst.htm)

This project is investigating the use of formal methods in certification, including:

- SPIDER: Scalable Processor-Independent Design for Electromagnetic Resilience. Inhouse project to (1) Develop fault-tolerant computer architecture in accordance with RTCA SC-180 (DO-254) guidelines, (2) demonstrate the feasibility of formal methods as means of certification, (3) develop training materials for FAA, and (4) provide advanced fault-tolerant computer architecture platform for inhouse analysis and experimentation. The SPIDER architecture provides a fault-tolerance middleware for critical applications. It is based upon three key protocols: fault-tolerant clock synchronization, group membership, and interactive consistency. These protocols are mutually dependent and pose difficult challenges to the formal analyst. A working prototype is resident at NASA Langley. The formal verification of the key protocols in PVS is near completion. Future work will focus on the design and verification of a real-time operating system for SPIDER.

- Contract with Odyssey Research Associates to develop model checking approaches to real-time system analysis. The work centers around a subset of Ada called the Ravenscar profile. The goal is to develop rigorous methods for assuring that Ada programs written in this Ada subset meet all of its timing requirements.

Engineering of Complex Systems Program (ECS)

Within this program, SafeWare is extending capabilities of SpecTRM-RL with expanded visualizations (both static and dynamic) of SpecTRM models and increase the analysis capabilities of SpecTRM with model checking and theorem proving. The enhanced tools will be demonstrate by application to Space Station rendezvous and docking

SRI International is applying formal methods to Mathworks tools. In principle one would like to model check the Stateflow diagrams which are at the heart of a Simulink design, but it is impossible establish anything useful without some knowledge of the constraints imposed by the differential equations in the other blocks. The SRI approach is to automatically construct sound discrete abstractions of the differential equations using automated theorem proving over the reals. A new decision procedure called QEPCAD, which performs quantifier elimination using cylindrical algebraic decomposition is used to find the sign-invariant regions. The end result is a completely discrete system you can model check.

Advanced Air Transportation Technologies Program (AATT)
(http://www.asc.nasa.gov/aatt/)

This is a project to develop formal methods to analyze conflict detection algorithms used for future free flight concepts and self-spacing and merging algorithms needed in the terminal area. Air Traffic Management algorithms are difficult to analyze because they are inherently hybrid, i.e. discrete logic is used to control the trajectories of aircraft in 3D-space. Automatic methods such as model checking cannot directly handle the continuous trajectories, and discretization leads to unacceptable errors.

Small Aircraft Transportation System (SATS) (http://sats.nasa.gov/)

This is a project to develop and formally verify algorithms that can generate arrival trajectories for SATS-configured small aircraft for regional airports that do not have towered ATC services.

3.2 Examples from Computer Security and Electronic Commerce

The earliest uses of formal methods were in computer security. One of the best-known examples was the 1973 Provably Secure Operating System (PSOS) [Neumann et al., 1980] project, which used a layered architecture with formal proofs of the properties of each layer and the relationship between layers. Tools, methods, and even sophisticated logics have been designed specifically for security applications. This section describes some of the more recent uses of formal methods in computer security.

Model based intrusion detection. One of the most challenging problems in computer security is intrusion detection in networks. Most commercial intrusion detection systems use a “signature based” approach, which scans network traffic for recognizable segments or hashes of known malicious software (e.g., viruses). The inherent limitation of this approach is that it cannot detect novel attacks for which no signature exists. A number of experimental intrusion detection systems attempt to solve this problem by defining a formal model of expected system behavior, then checking network traffic for message sequences that deviate from normal.

Access control policy composition. Although formal methods have been applied to analyzing access control policies for more than 30 years, early efforts assumed a single, monolithic system. During the past two decades, a need for analyzing compositions of security policies has been recognized, due to the demands of distributed systems, “virtual enterprise” joint ventures in commerce, and coalition security in military systems. Bonatti, Vimercati, and Samarati [Bonatti et al, 2000] describe an approach to analyzing compositions of complex, independent access control policies. Security policy statements are translated into logic programs, which can then be executed to analyze the effects of policy composition.

Mondex electronic cash system. In the early 1990s, the National Westminster Bank in the United Kingdom initiated development of a smartcard based electronic cash system [Stepney, 2001]. With this system, customers would be able to store monetary amounts on smartcards, and spend the stored electronic cash without any third-party approval or authentication. Because of the risk involved in this design, the company committed to having the system evaluated to the ITSEC E6 level, the highest level included in the European ITSEC standard. E6 certification requires proof of correspondence between a high level abstract security policy and lower level design. The Z notation was used to develop 80 pages of formal specifications, with machine assisted proofs of approximately 200 pages. Mondex became the first product to achieve the E6 level certification, and is being successfully marketed.

Secure Electronic Transaction Protocols – SET [Visa, 1997] is a very large collection of protocols designed by Visa and MasterCard to protect the confidentiality and authenticity of electronic commerce transactions. Many parts, though not all, of the SET specification have been subjected to formal verification, revealing several vulnerabilities [Meadows and Syverson, 1998], [Bella et al., 2000], [Bella, Massaci, Paulson, 2001].

Multimedia Messaging System on a Private Branch Exchange – Formal description techniques were used to specify the system and derive test cases for the operations, Administration and Maintenance software of the messaging system [Wong and Chechik, 2001]. Errors were detected in the requirements, the development cycle was shortened, and software quality improved. The authors recorded time spent on formal techniques, concluding that the application of these methods is cost effective if a "light-weight" approach is used, i.e., formalizing specifications with some automated support, but not necessarily proving properties of the system.

Universal Electronic Payment System – The UEPS was designed to be implemented using smart cards to store funds, for use in developing countries without significant banking infrastructure. The formal verification showed the protocol to be sound, and no losses due to fraud were encountered in operation [Anderson, 1997].

ANSI X9.17 Message Authentication Protocol – The X9.17 standard was designed to provide authentication of the origin of financial transactions. The formal verification revealed a specification ambiguity that could lead to a vulnerability, depending on how it was interpreted by developers [Kuhn and Dray, 1990].

4 Formal Techniques from Specification through Testing Phases

Formal methods can be used effectively at all stages of software development, from initial design and specification through testing and assurance. This section describes a well-known methodology for formally specifying and analyzing real-time and control software, the SCR model [Faulk & Clements, 1987]. The name "SCR" is an acronym for "Software Cost Reduction", which was the US Naval Research Laboratory project under which the methodology was first developed. SCR uses a state machine formalism, with tables defining the state transitions and outputs for all possible inputs and events. It is particularly useful for embedded software such as that used in largely event-driven devices such as computer peripherals, various weapons systems, or automated teller machines. SCR also lends itself well to automated generation of test cases, thus spanning the software development process. In this section we describe the methodology, how it can be used in verification, and how tests can be generated from SCR specifications.

4.1 SCR Modeling, Consistency Checking & Simulation

In the SCR formal model, the behavior of a software system is modeled by first identifying all outputs that the software must produce, and then expressing the value of each output as a mathematical function of state and history of the environment. SCR uses a special tabular notation to represent these functions precisely and compactly. A function is made up of variables and a particular type of functional construct. SCR uses two main types of variables – *output or controlled variables* and *input or monitored variables*. Since a specification for a complex real-world system may involve many monitored and controlled variables, the SCR uses two other additional types of variables – the *terms* and *mode classes* to facilitate ease of modeling and representation. A term is an auxiliary variable (that may be a combination of monitored variables or other terms used for simplifying the model or for representing some intermediate concepts). A term thus helps keep the specification concise. A mode class is a special case of a term whose values are modes. A mode stands for an equivalence class of system states useful in specifying the required system behavior.

As regards functional constructs, the two most important constructs used in SCR are *conditions* and *events.* A condition is defined as a predicate defined on one or more state variables (a state variable is a monitored or controlled variable, a mode class or a term). A condition in SCR is thus a predicate on a single system state. An event is said to occur when a state variable changes value. An event in SCR is thus a predicate on two system states that is true if the states differ in the value of at least one state variable. Since a condition represents a predicate using state variables in a given state, an event is most often represented using conditions (instead of state variables directly), and represent situations where the value of conditions change from true to false or vice versa. The tables that are used in SCR to represent conditions, events and mode transitions are called Condition Function Tables, Event Function Tables and Mode Transition Tables respectively. More specifically mode transition tables provide a list of valid states (modes) and the events that result in transition from one given state to another.
After an SCR specification is developed it has to be first checked for model-related (as opposed to domain or application-related) errors. Hence the SCR modeling tool [Heitmeyer et al, 1998] provides a consistency checker that exposes syntax and type errors, variable name discrepancies (or duplicates), missing cases, disjointness (which may result in nondeterminism), ambiguity and circular definitions.

Next an SCR specification has to be checked for domain or application-related errors. For this purpose the SCR modeling tool provides a simulator to symbolically execute the specification to validate it using the following approaches:

(a) The specification verifier may run scenarios (a scenario is a list of monitored or input variable name and value pairs which describes a sequence of input events) and let the simulator describe the outcome of the specification by displaying the values of dependent variables (e.g. controlled variables, mode classes, and terms). The outcome can then be analyzed to ensure that the specification captures the intended behavior.

(b) In addition, the specification verifier can define application properties believed to be true of the required behavior and, using simulation, execute a series of scenarios to determine if any violate the properties. More specifically the SCR tool provides a mechanism to define statements (called assertions) describing what must be true of any state, or what must be true of any two consecutive states. These assertions are stored by the SCR tool in an Assertion Dictionary. During execution, the Simulator will determine whether assertions in the Assertion Dictionary are violated and report the violations through log files.

4.2 Detailed Case Study

Independent Security Functional Testing on a commercial software product is very rarely performed in traditional security evaluations except in cases of high-assurance products used in mission-critical applications. The reasons for this scenario are the high cost (not sufficient number of evaluations to recover the initial investment) and technical complexity (development of proper specifications and satisfaction of test coverage requirements). In this case study we outline an approach and an associated toolkit that NIST has funded to develop, that has the potential to improve the economics of security functional testing as well as meet the technical requirements.

We have organized our presentation of the case study as follows. In section 4.2.1 we discuss the characteristics of security functional testing and contrast this type of testing with the other type of security testing – i.e. the security vulnerability testing. Section 4.2.2 provides a road map of the approach we have used for security functional testing and the tools used in the various phases of the approach. Sections X.44.2.3 through 4.2.7 describe in detail each of the processing steps in our approach with particular emphasis on the security function modeling and test vector generation aspects. The processing steps are explained with respect to an application involving security functional testing of a commercial DBMS product (Oracle 8.0.5). Section 4.2.8 outlines the major lessons learnt.

4.2.1 Characteristics of Security Functional Testing

There are subtle differences between traditional software conformance testing and security testing in general, in terms of purpose, scope, emphasis, error implications and strategy [Jansen, 1998]. The main purpose of software conformance testing is verification of *correctness* of implementation with respect to specifications. The market largely determines the *effectiveness* of the implementation. However security testing is concerned with both correctness and effectiveness since measures of effectiveness like strength of functions and robustness are very much an integral part of any security specifications. In traditional conformance testing, the emphasis is on testing the implementation for conformance to functional specifications while in security testing the product must be tested not only for conformance to security function specifications but also for compliance with mandatory features of the underlying security model. For example testing an access control function in a DBMS product will involve not only verification of specified behavior (correct access denials and clearances for a particular user) but also conformance to the underlying Discretionary Access Control Model (DAC) that provides the logic governing denials and clearances depending upon certain user attributes and state variables. In traditional conformance testing, verification using test cases that satisfy some statistical coverage measures can provide the assurance that certain defects will seldom occur. However in security testing, complete test coverage is required since obscure flaws can be exploited individually and collectively to subvert the behavior of other correctly implemented functions. The requirement for complete coverage therefore results in the number of test cases for security testing being order of magnitude much more than for traditional conformance testing.

Security Testing itself can be generally classified as security functional testing and security vulnerability testing. **Security functional testing** involves testing the product or implementation for conformance to the security function specifications as well as for the underlying security model. The conformance criteria state the conditions necessary for the product to exhibit the desired security behavior or satisfy a security property. In other words security functional testing involves what the product *should do*. **Security vulnerability testing** on the other hand is concerned with identification of flaws in design or implementation that may be exploited to subvert the security behavior which has been made possible by the correct implementation of the security functions. In other words security vulnerability testing involves testing the product for what it *should not do*.

4.2.2 Roadmap of Approach to Verification and Functional Testing

Our approach to Security Functional Testing makes use of suitable tools to accomplish the following objectives:

(a) Develop a machine-readable formal specification of security functions of a product.
(b) Automatically generate test vectors and executable test code using the machine-readable specification of security functions.

Based on the above objectives it should be clear that ours is an automated approach to security functional testing. The underlying framework of our approach is called the Test Automation Framework (TAF) [Safford, 2000]. Hence we named our toolkit TAF-SFT (Test Automation Framework-Security Functional Testing). The activities performed in each of the process steps using TAF-SFT and the associated tool module involved is given below:

(a) Step1 – Develop a machine-readable specification of the various security functions of the product under test using the formal model SCR (language) - model development tool TTM.

(b) Step 2 – Validate the SCR security specification model by verifying certain application-independent properties – TTM tool.

(c) Step 3 – Generate test vectors using the SCR security specification model. This may require some transformation of the model into a form amenable for test vector generation – T-VEC test generation tool.

(d) Step 4 – Generate the following artifacts to augment the SCR security specification model (transformed) with the following information – A Text editor:

 (i) Information about interfaces involved in establishing communication with the product and extracting its information contents (APIs).

 (ii) AA general strategy for exercising the product with actual tests using the generated test vectors

(e) Step 5 – Generate executable test code (called test driver) using the validated (and transformed if need be) SCR security specification model and the artifacts developed in Step 4 – T-VEC test driver generator tool.

(f) Step 6 - Execute the tests against the product, compare expected and actual outputs and generate a test report – T-VEC test comparison tool.

Details of the above process steps in the context of security functional testing for a commercial DBMS product (Oracle version 8.0.5) are discussed in the following sections. The process flow diagram of the TAF-SFT toolkit application for security functional testing of Oracle version 8.0.5 is given in Figure 4.1.

4.2.3 Develop a machine-readable specification of security functions

For our case study involving testing the security functions of a DBMS product, we obtained the text-based specification of the security functions for Oracle 8.0.5 from the Oracle 8.0.5 Security Target (ST) Document [Oracle, 2000]. The Security Target is a structured specification of security functional requirements as well as specification of security functions that meet those requirements expressed using pre-defined catalog of requirements and function representations in the international security criteria ISO/IS 15408 [ISO/IS 15408, 2000]. The next step after obtaining the text-based security functions specifications is to develop an SCR model of these specifications. The specification for a security function that stipulates the conditions under which an Oracle database user can grant an object privilege to another user as stated in the Oracle ST document is:

Granting Object Privilege Capability (GOP) **-** A normal user (the grantor) can grant an object privilege to another user, role or PUBLIC (the grantee) only if:
GOP (a): the grantor is the owner of the object ; or
GOP(b): the grantor has been granted the object privilege with the GRANT OPTION.

A role represents a group of privileges associated with a business process. The keyword PUBLIC represents all users.Recall that the formulation of an SCR model requires the identification of variables. The various variables identified for modeling the GOP security function are:

(a) Monitored Variables (input variable) - grantor, grantee, selectedObj, selectedObjPriv, granteeType, grantedObj, grantedObjPriv
(b) Controlled Variable (output variable) – grant_obj_priv_OK – A Boolean variable that will have the value TRUE when the conditions for 'granting objective privilege' by one user to another are satisfied.

In addition to the above variables, we need two term variables to complete the GOP function specification in SCR. They are: (a) grantor_owns_object (to incorporate the conditions that affirm the fact that the grantor is the owner of the selected object – the requirement GOP(a)) and (b) has_grantable_obj_privs (to incorporate the conditions that affirms that the grantor hold the privilege in question for the selected object with the ability to propagate (GRANT OPTION) – the requirement GOP(b)). Expressing the conditions that affirm the truth-values for the above discussed term variables in SCR notation we get:

grantor_owns_object – TRUE when grantor = selectedObjOwner (4.2.1)

has_grantable_obj_privs – TRUE when
selectedObj = grantedObj AND selectedObjPriv = grantedObjPriv AND GRANT_OPTION (4.2.2)

Based on our previous discussion, it should be clear that our security functional testing involves not only testing the security function specifications, but also the underlying model semantics (in this case the Discretionary Access Control (DAC) model)). Clearly the DAC model semantics in our case is that the object owner and the holder of the privilege (with GRANT option) are two different entities. This DAC model semantic constraint should be added to the term condition 4.2.2 above to yield:

has_grantable_obj_privs – TRUE when
selectedObj = grantedObj AND
selectedObjPriv = grantedObjPriv AND
GRANT_OPTION AND
selectedObjOwner != grantor AND
selectedObjOwner != grantee (4.12.2')'

Now that the expressions 4.12.1 and 4.12.2' represents our requirements GOP(a) and GOP(b) (along with DAC model semantics), our SCR condition for the entire GOP function becomes:
grant_obj_priv_OK – TRUE when grantor_owns_object OR has_grantable_obj_privs (4.2.3)

Now our SCR specification of the GOP security function fully represents the claimed functionality in the Oracle ST document along with DAC model semantics. However we have still not incorporated constraints that relate directly to the Oracle DBMS domain.

These relate to the fact that the grantee can only be of type user, role or PUBLIC and that the object privilege can only be one of UPDATE, DELETE, SELCT, INSERT or ALL (as they are the valid privilege modes for objects managed by the DBMS). Hence these domain constraints should also be incorporated to complete the GOP function specification. The SCR condition tables dealing with the conditions for the term variables (4.12.1 and 4.12.2') as well as for the controlled variable (grant_obj_priv_OK – 4.2.3) (including the domain constraints) are given in table 4.1.

4.2.4 Verification of SCR Security Specification Model

The SCR model development tool TTM provides a consistency checker that checks the specification for defects such as type errors, missing cases, circular definitions and other application-independent errors. The TTM tool also provides a dependency graph browser that provides a graphical display of the dependencies among the variables in the specification. Running the SCR security specification model through these two processes helps in developing an internally consistent model and in verifying that all the relevant variables have been taken into account.

Table 4.1 – SCR Condition Function Tables for the GOP Security Function

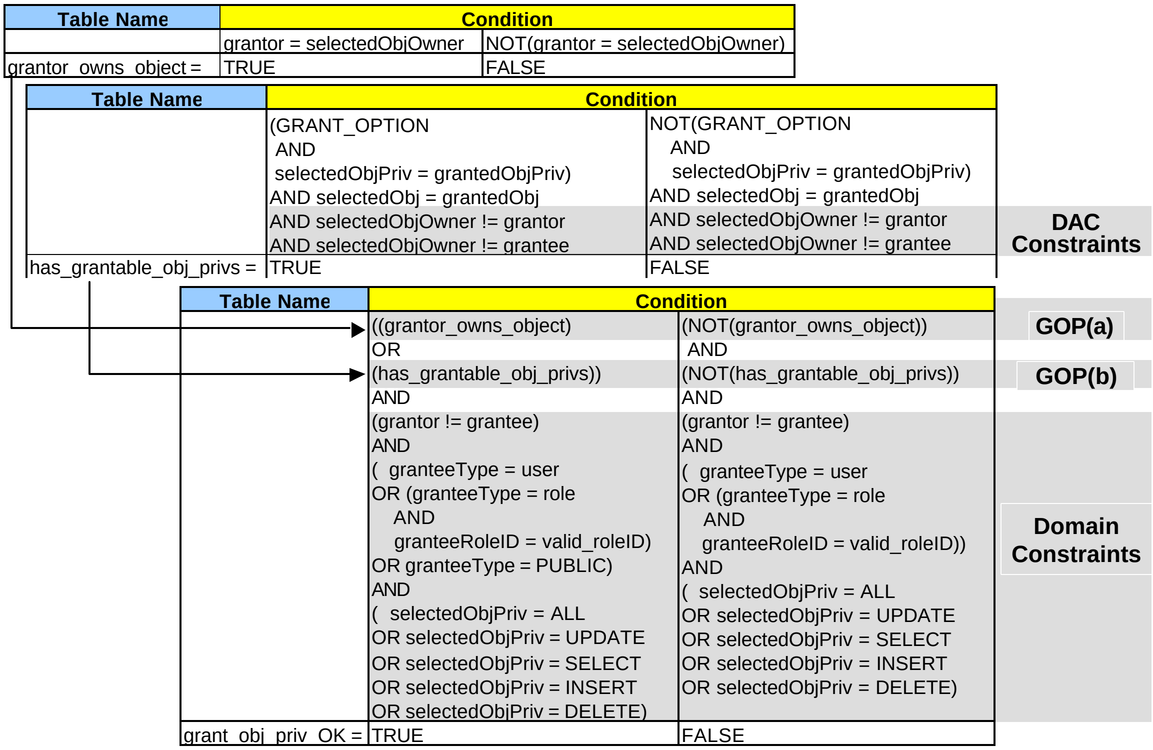


Table Name	Condition	
	grantor = selectedObjOwner	NOT(grantor = selectedObjOwner)
grantor_owns_object =	TRUE	FALSE

Table Name	Condition	
	(GRANT_OPTION AND selectedObjPriv = grantedObjPriv) AND selectedObj = grantedObj AND selectedObjOwner != grantor AND selectedObjOwner != grantee	NOT(GRANT_OPTION AND selectedObjPriv = grantedObjPriv) AND selectedObj = grantedObj AND selectedObjOwner != grantor AND selectedObjOwner != grantee
has_grantable_obj_privs =	TRUE	FALSE

DAC Constraints

Table Name	Condition	
	((grantor_owns_object) OR (has_grantable_obj_privs)) AND (grantor != grantee) AND (granteeType = user OR (granteeType = role AND granteeRoleID = valid_roleID) OR granteeType = PUBLIC) AND (selectedObjPriv = ALL OR selectedObjPriv = UPDATE OR selectedObjPriv = SELECT OR selectedObjPriv = INSERT OR selectedObjPriv = DELETE)	(NOT(grantor_owns_object)) AND (NOT(has_grantable_obj_privs)) AND (grantor != grantee) AND (granteeType = user OR (granteeType = role AND granteeRoleID = valid_roleID)) AND (selectedObjPriv = ALL OR selectedObjPriv = UPDATE OR selectedObjPriv = SELECT OR selectedObjPriv = INSERT OR selectedObjPriv = DELETE)
grant_obj_priv_OK =	TRUE	FALSE

GOP(a)

GOP(b)

Domain Constraints

4.2.5 Generating Test Vectors From SCR Specification

Our example security function (GOP) specification (in table 4.1) for the Oracle DBMS shows that the SCR security specification model is composed of tables of conditions and events. Now our next task is to examine as to how this model can be used for generating test sequences or test vectors. A test sequence is a sequence of system inputs and their associated outputs. In order to obtain such a sequence of inputs, it would be better if the security specification model is in the form of input-output relations and a set of conditions (or constraints) governing the inputs. The valid sequence of inputs can then be obtained based on the constraints on the input variables and the corresponding output variables obtained using the input-output relation. In other words we are faced with the necessity to convert the SCR security specification model into a security test specification model that consists of a set of input-output relations and associated constraints on inputs.

We used the T-VEC model translator tool [Blackburn, Busser, Fontaine, 1997] to obtain a security test specification model from the SCR security specification model. More specifically, the T-VEC model translator converted our SCR security specification model into what is known as T-VEC linear form. The T-VEC linear form consists of a set of input-output relations (called the functional relationships) and a set of constraints (called relevance predicates) on the inputs associated with a given functional relationship. Processing our SCR model for ‘ Granting Object Privilege Capability’ (GOP security function) through the T-VEC model translator tool we obtained the following functional relationship (input-output relation).

((grantor_owns_object) OR (has_grantable_obj_privs)) AND
<domain_constraints> → grant_obj_priv_OK (4.2.4)

The next item we obtained from the T-VEC translator tool is the relevance predicates. Recall that the relevance predicate groups together all the constraints associated with input values and are expressed in the form of disjunctions of conjunctions and that each component in the expression (i.e. a conjunction) is called the Domain Convergence Path (DCP). In the GOP security function specification context, each DCP should therefore contain either the component GOP(a) or GOP(b) in table 4.1 along with each of the possible value associations given in the domain constraints. A few examples of DCPs are:

(grantor_owns_object) AND (grantee=’user’) AND
(selectedObjPriv = ‘UPDATE’) (4.2.5)
(has_grantable_obj_privs) AND (grantee=’PUBLIC’)
AND (selectedObjPriv = ‘SELECT’) (4.2.6)

Having obtained a transformed security specification model that consists of input-output relations and relevance predicates, the next step is to obtain test vectors (test sequences) using these specifications. We used the T-VEC test generator tool [Blackburn, Busser, Fontaine, 1997] to accomplish this task. Recall that the relevance predicates are constraints on the input variables space and hence a valid input sequence is nothing but a set of input variables values that satisfies the relevance predicates. However since the relevance predicates are in the form of disjunctions of conjunctions (each set of conjunctions is called the DCP), a valid input sequence is one that satisfies at least one DCP. In other words each DCP forms an equivalence class when we categorize all possible set of input sequences. It is this property that is used by our test vector generator to generate test sequences. Since a test sequence (or test vector) is nothing but an input sequence with its associated outputs, our test vector generator generates at least one test sequence for each DCP (since a DCP forms an equivalent class) and thus obtains the coverage of the input space.

The test vector generation process described above also helps to verify the satisfaction of a given specification. Since the entire specification is nothing but a disjunction of DCPs, the specification as a whole is satisfiable if at least one test vector exists for each DCP. Therere may existbes input variables in the input-output relation that are not constrained by relevance predicates. The test vector generator also generates additional test points by incorporating boundary value combinations from these unconstrained inputs (e.g. low bound and high bound for numeric objects, sets for enumerated variable). The incorporation of these additional test points helps to prove that unconstrained inputs do not the affect the expected value of the input-output relation.

However the presence of a test vector for each DCP is no guarantee that collectively the set of test vectors are sufficient to verify all the path conditions for a functional relationship. This scenario may result if contradictions exist among DCPs. Hence in order to ensure that this situation is not present in our specification, we used the T-VEC coverage analyzer tool to detect these contradictions and ensure that the test vectors provide the intended coverage.

In fact we can compute the total number of test vectors for testing our GOP security function, by calculating the number of DCPs in the relevance predicate and the fact that the test vector generator will generate at least one test vector for a DCP. Since a DCP is one disjunction, each of the OR s in our SCR condition function table 4.1 should participate in a DCP. Since the conditions GOP(a) and GOP(b) are connected with OR, each should give rise to a different DCP. On examining the domain constraints we find that there are three OR s for the expressions involving input variable ‘grantee’ (three possible values for grantee) and five OR s for the expressions involving the input variable ‘selectedObjPriv’ (five possible values for selectedObjPriv). Hence the total number of disjunctions or DCPs we will obtain will equal 2*3*5 = 30. There should therefore be a minimum of thirty test vectors for testing the GOP security function (all yielding the value TRUE for the controlled variable grant_obj_priv_OK). Including the test cases for grant_obj_priv_OK being FALSE (by negating at least one predicate in each DCP) and additional test points derived from boundary value combinations of unconstrained input variables like grantor, grantee, grantee_ roleID, the test vector generator generated about 80 test vectors for testing the GOP security function. The test vectors are shown in table 4.2.

4.2.6 Develop Additional artifacts for Test Driver (executable test code) generation

We now have the translated SCR model containing the security function specification of security functions and the associated test vectors. These two documents by themselves do not provide sufficient information to the test driver to generate executable test code in a procedural language. We do need to provide the test driver generator the knowledge of the product's interface API (that pertains to the test code language) and any other relevant APIs needed for extraction of information pertaining to the product's state. This is exactly the information that is provided by the 'Object Mapping' file. More specifically, the 'Object mapping file' provides the mapping between the behavioral model variables and the interface elements needed to set, retrieve or evaluate the values of those model variables. The combination of the behavioral model and the object mapping information is called the 'verification model' since it represents the complete specification required for carrying out the product's functional verification process.

Table 4.2 – Test Vectors generated for testing the GOP Security Function

#	TSP	grant_obj _priv_OK	grantor	grantee	grantee Type	grantee RoleID	valid_roleID	selected ObjPriv	objOwner	GRANT_ OPTION	granted ObjPriv	selected Obj	granted Obj
1	1	TRUE	1	2	user	2	2	ALL	1	TRUE	ALL	4	4
2	1	TRUE	4	3	user	1	1	ALL	4	FALSE	SELECT	1	1
3	2	TRUE	1	2	user	2	2	UPDATE	1	TRUE	ALL	4	4
4	2	TRUE	4	3	user	1	1	UPDATE	4	FALSE	SELECT	1	1
5	3	TRUE	1	2	user	2	2	SELECT	1	TRUE	ALL	4	4
6	3	TRUE	4	3	user	1	1	SELECT	4	FALSE	SELECT	1	1
7	4	TRUE	1	2	user	2	2	INSERT	1	TRUE	ALL	4	4
8	4	TRUE	4	3	user	1	1	INSERT	4	FALSE	SELECT	1	1
9	5	TRUE	1	2	user	2	2	DELETE	1	TRUE	ALL	4	4
10	5	TRUE	4	3	user	1	1	DELETE	4	FALSE	SELECT	1	1
. . .													
77	39	FALSE	1	2	role	1	1	INSERT	3	FALSE	ALL	1	1
78	39	FALSE	4	3	role	2	2	INSERT	2	FALSE	SELECT	4	4
79	40	FALSE	1	2	role	1	1	DELETE	3	FALSE	ALL	1	1
80	40	FALSE	4	3	role	2	2	DELETE	2	FALSE	SELECT	4	4

The last but not the least important piece of information that the test driver generator needs is the generic sequence of steps needed for executing any test. It is this piece of information that is provided in the 'Test driver Schema' file. The test driver schema file describes the simple algorithmic pattern that is used to load, execute and receive test data and other environmental information pertaining to the target test environment.

As far as interface information (for object mapping file) for our DBMS product is concerned, we need the following:

(a) knowledge of the Java APIs to interface with Oracle DBMS
(b) knowledge of the structure of the data stores that contain the security state information (Data Dictionary tables)
(c) knowledge of Content extraction API needed to extract and verify information from those data stores.

Fortunately since Oracle is a relational DBMS, it supports the standardized Java Database Connectivity (JDBC) [JDBC, 2000] interface API, and Structured Query Language (SQL) [ISO/IEC 9075, 1999] as the content extraction API.

Now our test driver for our Oracle DBMS, in order to perform its intended function, has to contain Java code that verifies the conditions in our SCR security function specification model (using the data from our test vectors) by extracting the security state information stored in data dictionary views through the JDBC API library calls and SQL commands. In order to generate such a test driver, we need to combine the security function specification and test vectors with the interface API, content-extraction API and the data dictionary views in Oracle DBMS. In other words we need information that maps the model variables in the behavioral specification to the commands in JDBC API and SQL API and the data dictionary views against which these commands must be executed. It is this mapping information that is specified in our 'Object Mapping File'.

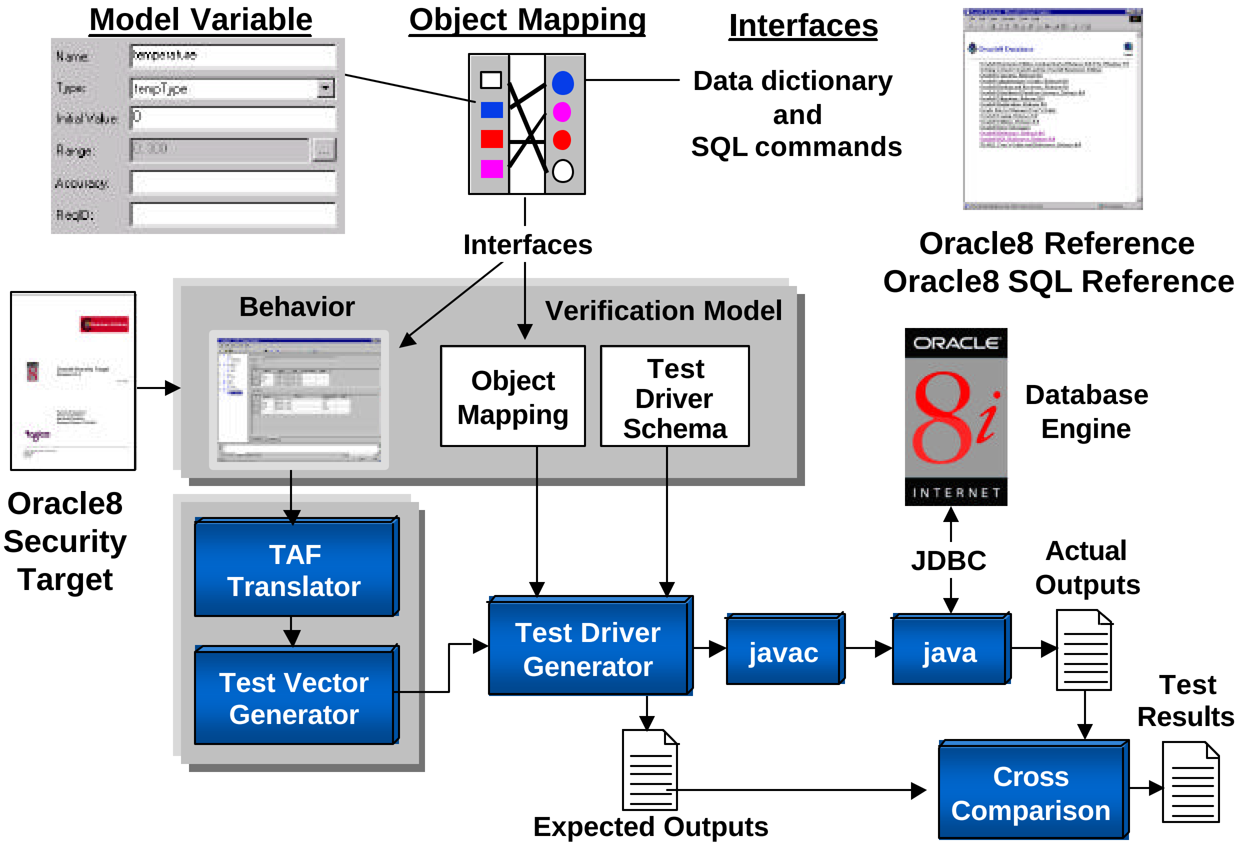
With the development of the 'Object Mapping' file, the SCR behavioral specification and the test vectors we have the constituent ingredients of the verification model. The only other artifact that we need for the test driver to generate security function tests for the Oracle DBMS environment is the 'Test driver schema'.

As already stated, the test driver schemas are templates containing generic execution steps for each of the tests. In a database environment the security state is determined by a combination of security data that consists of user attributes, roles (entities that represent collection of privileges), database objects (tables, views etc) and privilege assignments to users and roles for various database objects. This security data is stored in database dictionary tables (also called system tables or database catalogues). The data in these tables cannot be created or deleted using the traditional data manipulation SQL commands but only through some privileged SQL commands. Hence definition of generic execution steps for each of the 'security function tests' against the database involves a set of these privileged SQL commands to systematically populate the database dictionary tables with security state-defining data as well as other relevant data. In other words appropriate database conditions must be established prior to the execution of each of the 'security function tests' by accessing the database as administrative-level system user.

4.2.7 Generate test drivers, execute tests and generate test results report

We used the test driver generator tool [Blackburn, Busser, 1996] (from T-VEC) that operates on the transformed SCR security specification model, test vectors, object mapping information and test execution template definitions (in the test driver schema file) to generate the executable test code. In our TAF-SFT toolkit, we configured the test driver generator to generate executable test code in Java, though conceptually any language generator module can be incorporated within the test driver generator. The test driver generator also generates the 'Expected Outputs' file whose format is again specified in the test driver schema.

The generated test driver code is then executed against the product by incorporation of the appropriate run-time libraries (e.g. Java Virtual Machine and Java run-time libraries). This process generates the 'Actual Outputs' File. The last process in our TAF-SFT approach for automated security functional testing is the ‘Cross Comparison’ step that compares the expected outputs with the actual outputs to generate the test results report.



Name: temperature
Type: tempType
Initial Value: 0
Range: 0..300
Accuracy:
ReqID:

Figure 4.1 – TAF-SFT Process Steps for security functional testing of a DBMS product

4.2.8 Lessons Learned

Our TAF-SFT approach (and an associated toolkit implementation) for automated security functional testing addressed the economics aspect of testing (though automated test vector, executable test code and test report generation) as well as the quality aspect (through a validated formal model of specifications, coverage criterion based test vector generation etc). The fact that the approach was scalable was established by using the TAF-SFT toolkit for developing and conducting security functional tests for a large, complex commercial DBMS product.

The major disadvantages of our approach are the detailed knowledge of the security function semantics required on the part of the modeler to develop good behavioral models and the complexity of object mapping information that may result in case of products with complex interfaces. These disadvantages can be partially overcome in situations where the following are possible:

(a) Partial reuse of SCR security behavioral model
(b) Partial reuse of Object Mapping information

Since the SCR behavioral model is based on the security function specifications, reuse of parts of this model is possible if security function specifications in the different products under security testing are based on an interoperable security API like CDSA [CDSA,1998]. Partial reuse of Object Mapping information is possible if the different products under security testing support a common interface API (like the different relational DBMS products supporting the JDBC API and the SQL API).

4.3 Cost and Practicality of Specification Based Test Generation

Test development is typically an enormous expense, often half of total development cost. But because of the wide variation in the extent of testing by different developers, and variance in testing requirements for different types of software, testing costs can range from 30% to 90% of total labor resources [Beizer,1990]. Thus any increases in the efficiency of test development can have a significant impact on product cost.

Although not widely used in system assurance, formal verification is an added cost beyond system testing. The cost of formal verification has two components: development of a formal system specification, and analysis of the specification against requirements. The analysis may take the form of a computer-assisted proof, or an almost fully automated verification through model checking. Formal verification of this type can add 10% - 20% to the system cost

But formal specifications can be used for more than analysis and proof. Using methods we and others have developed, formal specifications can be used to generate complete test cases, both input data and expected results. This can result in a significant reduction in the cost of testing. Table 4.3 shows system development cost under various assumptions:

- Traditional development, testing but no formal verification
- Development with formal verification
- Development with formal verification and automated test generation, assuming automated generation reduces the cost of test coding by 50%.
- Development with automated test generation but no formal verification, assuming automated generation reduces the cost of test coding by 50%.
- Development with automated test generation but no formal verification, assuming automated generation reduces the cost of test coding by 66%.

To date, most research on automated software testing has focused on structural testing, i.e., testing based on execution paths within the code that implements a specified function. However, structural testing is not possible with many systems, as there is no access to source code. An alternative is to use specification-based testing, in which tests are derived from the specification alone. Faults are inserted into a specification and a model checker is used to generate counterexamples that can be post-processed into complete test cases. Experiments show test coverage to be as good or better than hand crafted tests.

	Traditional	Formal spec & verification w/out test generation	Formal spec & verification w/ test generation (a)	Formal spec w/ test generation (b)	Formal spec with test generation (c)
Design, code, other costs	50%	50%	50%	50%	50%
Test coding	30%	30%	15%	15%	10%
Test execution	20%	20%	20%	20%	20%
Formal specification	---	10%	10%	10%	10%
Formal verification	---	10%	10%	---	---
Cost compared to traditional		120%	105%	95%	90%

Table 4.3. Estimated Costs of Using Automated Test Generation Under Conservative Assumptions

Methods for generating tests from specifications can make formal methods cost effective for a much larger class of systems. Use of formal methods is largely confined to secure systems or safety-critical systems, i.e., those systems whose failure can have catastrophic cost. But if the high cost of formal methods can offset the possibly higher cost of test development, formal techniques become much more attractive. The past two decades have seen great advances in methods and tools for formal verification. More effective tools can do a great deal to increase the use of formal techniques in industry. In addition, better methods and tools for specification based testing could reduce the cost and increase the effectiveness of system testing.

5 Formal Methods and Certification Standards

Is it possible to develop a market for high assurance components for M&S systems? If so, what certification and accreditation schemes are effective, and how can they be used to encourage the transfer of high assurance technology into the commercial market? This section provides some lessons learned from the computer security field that can help answer these questions. In particular, we focus on the relationship between government assurance standards – which will clearly be required for M&S components used by DoD systems – and the commercial market. Our aim is to identify approaches that work in an environment where COTS systems and components are to be incorporated into larger systems. We describe early, unsuccessful, efforts to encourage the industrial use of formal methods through assurance standards, and discuss lessons learned. We also describe a current assurance standard that has resulted in the adoption of formal methods by industry; that is, developers are using formal techniques strictly for the marketing advantage of obtaining a higher rating under the assurance standard.

5.1 Standards and Technology Transfer

Formal methods have been included in government standards for nearly 20 years. One of the first was the DoD Trusted Computer Security Evaluation Criteria (TCSEC), commonly known as the "Orange Book". The TCSEC defined requirements for operating system security, in a series of levels with progressively stronger security objectives. The lower levels provided a minimal level of security, with a correspondingly basic degree of assurance. At the highest levels, B3 and A1, formal methods were required. B3 and A1 systems required the same security features; the key difference between them was the degree of formal assurance required.

In the terminology introduced earlier in this paper, B3 systems required a formal statement of security requirements, with an informal system specification and an informal proof of correspondence between the two. A1 systems required that both the requirements and system specifications be formally defined, and a formal proof of correspondence between the two. To encourage the development of these high assurance systems, the National Security Agency sponsored the creation of several high-quality verification tools, some of which were forerunners of today's best-known tools.

The TCSEC was a tremendous success in encouraging research on security and assurance methods. Its multiple evaluation levels gave customers the ability to compare the features and assurance of secure systems. In terms of technology transfer, the TCSEC was successful in getting basic security features into the commercial market. But its technology transfer success did not extend to the assurance side of security. Despite the availability of good tools and a recognized need for secure systems, only a handful of A1 systems were ever produced. All of these were for government customers, and nearly all developed under government contract. Only one A1-rated product is commercially available today.

Why was the TCSEC not more successful in promoting the use of formal methods for high assurance systems? An often-cited reason for the paucity of high assurance TCSEC-rated products is the lengthy evaluation period required An evaluation often required two years or more, making it difficult for developers to get products into the market. A second problem was likely the significantly increased work required to produce an A1 level product, as compared with the B3 level. As one developer explained,

> *Given the functionality of an A1 system and the functionality of a B3 system are *identical*, we decided the added documentation/mathematical proofs required to attain the A1 evaluation were not worth the extra effort and cost, particularly as most users seem willing to take on the added risk of using a B3 to meet what would be, according to the Yellow Book matrix, an A1 requirement.* [Goertzel, 95]

Later standards, informed by the TCSEC experience, have taken somewhat different approaches to assurance requirements. The Information Technology Security Evaluation Criteria (ITSEC) is a single uniform standard adopted by the UK, France, Germany, the Netherlands and the European Commission. From an assurance standpoint, the significant difference between the ITSEC and the TCSEC is that functionality and assurance have been decoupled. That is, a system with minimal security functions could be evaluated to a high assurance level, or a system with extensive features could be evaluated at a low assurance level. A second significant difference from the TCSEC is that ITSEC evaluations are done by third-party accredited laboratories. This framework helps to eliminate the evaluation bottleneck that occurs when evaluation resources are not sufficient for the volume of products submitted for evaluation.

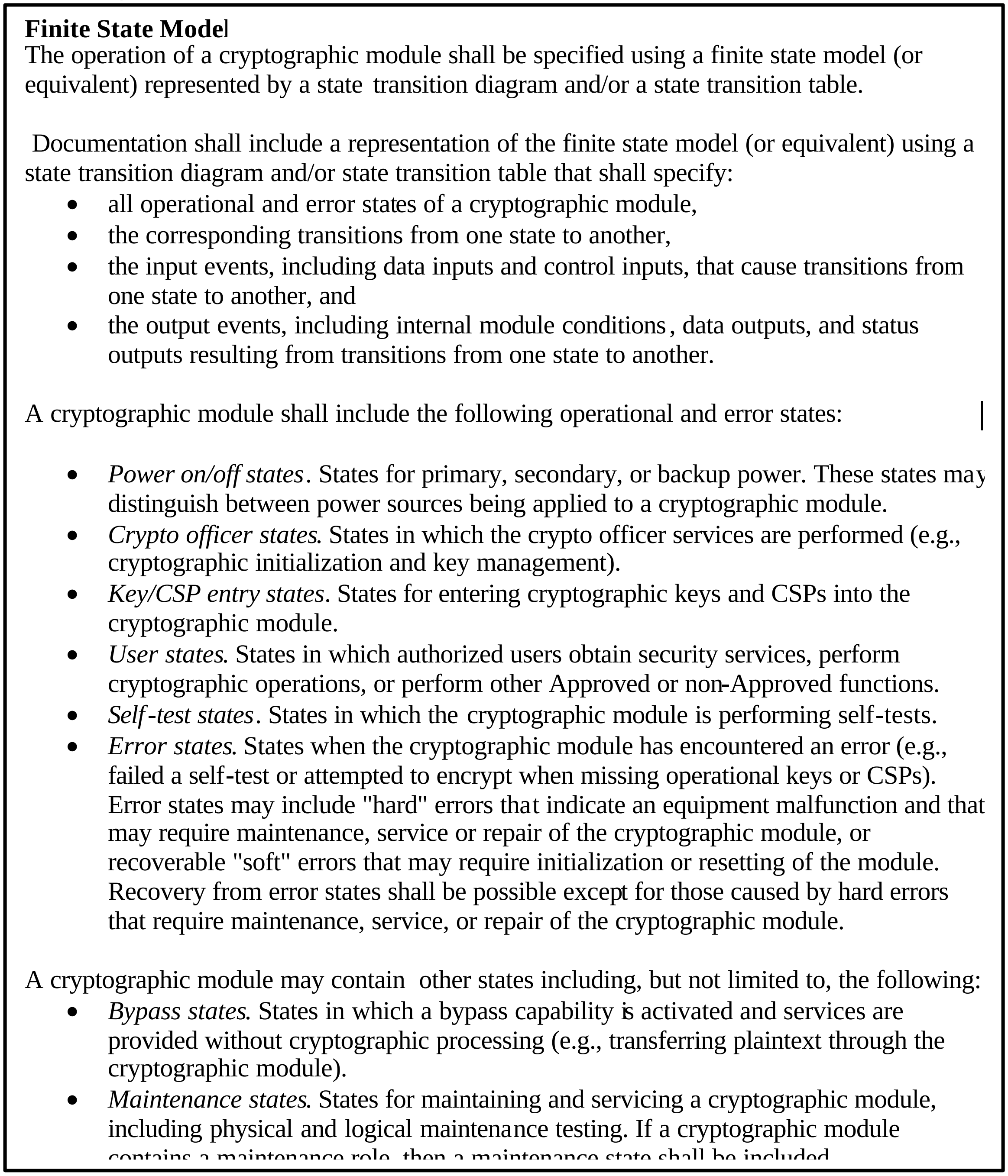
5.2 Cost and Practicality of Mandating Use of Formal Methods

An evaluation standard that is focused on a particular class of product, and thus more relevant to the needs of high assurance M&S components, is FIPS 140-2, Security Requirements for Cryptographic Modules. FIPS 140-2 was defined by NIST and is required for commercial cryptographic products used in both national security and unclassified government systems. It specifies requirements for both hardware and software, including the use of formal methods at all levels. Like the ITSEC, the FIPS 140-2 framework uses third-party laboratories, accredited by NIST, to evaluate modules according to requirements of the standard. To date, more than 200 products from over 40 vendors have been evaluated by accredited labs. The number of testing labs has grown from three in 1995 to six, as demand has increased. Products include radios, telephones, cryptographic co-processors, PDAs, smart cards, routers, toolkits, accelerators, and postal systems.

Despite the variety of these products, all share some common requirements in their cryptographic functions, making it possible to provide greater specificity in evaluation requirements than is possible for other standards, like the ITSEC, that must be sufficiently broad to cover the range of IT products. For example, specific V&V requirements can be established for features that all cryptographic modules must have, such as operator authentication, random number generation, key generation and storage, and key zeroization. The FIPS 140-2 experience may thus have some lessons for assurance of M&S components. As noted by Balci, Nance, and Arthur [2002], software tools designed specifically for VV&A of M&S system features such as experimental design or random variate generation are likely to be more effective than general-purpose V&V tools. Similarly, certification requirements specific to particular types features have advantages for the inclusion of formal methods in V&V, as demonstrated by experience with FIPS 140-2.

Formal methods have been incorporated into all four levels of FIPS 140-1 assurance. At levels 1 – 3, a basic formal specification of the system, in the form of a finite state machine model, is required (a portion of these requirements can be seen in Figure 5.1). Notice that even this basic level of formal specification is more rigorous than specifications for most software. Properly constructed, the finite state machine transition table, with state transition conditions, expected inputs and outputs, should provide sufficient detail for a test generation tool such as that described in Section 4.

Figure 5.1. FIPS 140-1 Level 1-3 Formal Methods Requirements



Finite State Model

The operation of a cryptographic module shall be specified using a finite state model (or equivalent) represented by a state transition diagram and/or a state transition table.

Documentation shall include a representation of the finite state model (or equivalent) using a state transition diagram and/or state transition table that shall specify:

- all operational and error states of a cryptographic module,
- the corresponding transitions from one state to another,
- the input events, including data inputs and control inputs, that cause transitions from one state to another, and
- the output events, including internal module conditions, data outputs, and status outputs resulting from transitions from one state to another.

A cryptographic module shall include the following operational and error states:

- *Power on/off states*. States for primary, secondary, or backup power. These states may distinguish between power sources being applied to a cryptographic module.
- *Crypto officer states*. States in which the crypto officer services are performed (e.g., cryptographic initialization and key management).
- *Key/CSP entry states*. States for entering cryptographic keys and CSPs into the cryptographic module.
- *User states*. States in which authorized users obtain security services, perform cryptographic operations, or perform other Approved or non-Approved functions.
- *Self-test states*. States in which the cryptographic module is performing self-tests.
- *Error states*. States when the cryptographic module has encountered an error (e.g., failed a self-test or attempted to encrypt when missing operational keys or CSPs). Error states may include "hard" errors that indicate an equipment malfunction and that may require maintenance, service or repair of the cryptographic module, or recoverable "soft" errors that may require initialization or resetting of the module. Recovery from error states shall be possible except for those caused by hard errors that require maintenance, service, or repair of the cryptographic module.

A cryptographic module may contain other states including, but not limited to, the following:

- *Bypass states*. States in which a bypass capability is activated and services are provided without cryptographic processing (e.g., transferring plaintext through the cryptographic module).
- *Maintenance states*. States for maintaining and servicing a cryptographic module, including physical and logical maintenance testing. If a cryptographic module contains a maintenance role, then a maintenance state shall be included

FIPS 140-2 provides four levels of assurance, with increasingly stringent requirements for both hardware and software. As with other security standards, the use of multiple levels of assurance is designed to provide developers and purchasers with a metric that can be used to compare products. Those products that achieve higher ratings on the standard can be expected to provide better security, and therefore be more attractive to customers.

Level 4 adopts a much stronger requirement, which might be described as “modified A1”, to adopt the TCSEC terminology. Table 5.1 summarizes key differences between these requirements and the TCSEC A1 and B3 levels.

	A1	**B3**	**FIPS 140-2**
Requirements Statement	Formal	Formal	Formal
System Specification	Formal	Informal	Formal
Spec to Requirements Proof	Formal	Informal	Informal

Table 5.1. Comparison of TCSEC and FIPS 140-2 Requirements

Rather than require a formal, machine checked, proof of correspondence between requirements and system specification, FIPS 140-2 level 4 requires an informal proof that is sufficiently detailed to convince the evaluator. Figure 5.2 shows major requirements for Level 4.

Figure 5.2 FIPS 140-1 Level 4 Formal Methods Requirements

> In addition to the requirements for Security Levels 1, 2, and 3, the following requirements shall apply to cryptographic modules for Security Level 4.
>
> - Documentation shall specify a formal model that describes the rules and characteristics of the cryptographic module security policy. The formal model shall be specified using a formal specification language that is a rigorous notation based on established mathematics, such as first order logic or set theory.
> - Documentation shall specify a rationale that demonstrates the consistency and completeness of the formal model with respect to the cryptographic module security policy.
> - Documentation shall specify an informal proof of the correspondence between the formal model and the functional specification.
> - For each cryptographic module hardware, software, and firmware component, the source code shall be annotated with comments that specify (1) the preconditions required upon entry into the module component, function, or procedure in order to execute correctly and (2) the postconditions expected to be true when execution of the module component, function, or procedure is complete. The preconditions and postconditions may be specified using any notation that is sufficiently detailed to completely and unambiguously explain the behavior of the cryptographic module component, function, or procedure.
> - Documentation shall specify an informal proof of the correspondence between the design of the cryptographic module (as reflected by the precondition and postcondition annotations) and the functional specification.

This approach was adopted to make the use of formal methods more cost effective. It takes advantage of two considerations:

- The specification of a cryptographic module is likely to be smaller than that of an operating system kernel, reducing the work required for proofs.

- It has been observed that most of the benefit of using formal methods often comes from the process of developing the specification. In formalizing the system description, ambiguities and omissions are detected. Although no study has actually quantified these values, the "80/20 rule" seems to be in effect: perhaps 20% of the effort of a formal verification goes into developing the formal specification, but around 80% of the value of formal techniques can come from this process. (In practice, however, some developers have used fully formal, machine-assisted proofs, rather than develop informal proofs. Theorem proving tools have become sufficiently advanced to make computer assisted theorem proving practical.)

This simplified approach has paid off. In 1998, the IBM 4758 cryptographic coprocessor became the first device to achieve a Level 4 rating. Developers defined and mechanically verified a formal mathematical model of the coprocessor's internal software [Smith and Weingart, 1999; Smith et al., 1999]. It is notable that the IBM 4758 is a mass-produced product for the commercial market, developed without government funding. The developers sought the Level 4 rating strictly for its marketing advantage.

Since 1998, eight Level 4 evaluations have been completed and more are expected. Although these eight products represent only about 4% of all products evaluated, it is significant that all were done for the competitive advantage a Level 4 certificate would provide in the marketplace, rather than to fulfill a government contract. From a technology transfer standpoint, FIPS 140-2 is thus a useful model for encouraging the adoption of formal methods for commercial products.

6 Conclusions

We have reviewed the role that formal methods and associated tools can play in the verification and validation of software specifications and implementations. We then proceeded to give a brief narrative on the real-world applications of formal V & V methodologies. We also discussed in detail a case study involving test code generation for security functional testing of a commercial product based on formal specification. Lastly we dealt with the issue of formal methods in evaluation and certification of software products. Our conclusion was that the use of formal methods brings with it significant technical and cost impacts in situations where product evaluations involve a small and compact set of specialized functions (e.g. cryptographic functions in FIPS 140-2 evaluations).

Three decades of research and practical experience have demonstrated two truths of formal methods – they are not the "silver bullet" to eliminate all software failures, but neither are they beyond the budget constraints of software developers. Formal methods are usually the only practical means of demonstrating the absence of undesired behavior, an essential property of critical systems. Industrial-quality model checkers and advanced theorem provers make it possible to do sophisticated analyses of formal specifications in an automated or semi-automated mode, making these tools attractive for commercial use. The ability to generate complete test cases from formal specifications can result in overall savings, despite the cost of developing the specification. Experience has shown that formal techniques can be applied productively even without full-blown proofs. The process of developing a specification is often the most valuable phase of a formal verification, and "lightweight formal methods" approaches make it possible to formally analyze partial specifications and early requirements definitions. Experience with mandated use of formal techniques in FIPS 140-1 and other standards provides empirical evidence that these methods can be successfully incorporated into the development process for commercial products.

7 References

S. Agerholm and P.G. Larsen, "Modeling and Validating SAFER in VDM-SL", Proceedings, Fourth NASA Langley Formal Methods Workshop, Sept. 1997.

Paul E. Ammann, Paul E. Black, and William Majurski, "Using Model Checking to Generate Tests from Specifications", Proceedings of 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98), Brisbane, Australia (December 1998), edited by John Staples, Michael G. Hinchey, and Shaoying Liu, IEEE Computer Society, pages 46-54.

R. J. Anderson. The formal verification of a payment system. Technical report, Computer Lab, Univ. of Cambridge, UK, 1997.

Balci, O., "A Methodology for Certification of Modeling and Simulation Applications", *ACM Transactions on Modeling and Computer Simulation*, Vol. 11, No. 4 (Oct.).

Balci, O., R.E. Nance, J.D. Arthur, "Expanding Our Horizons in Verification, Validation, and Accreditation Research and Practice", Proceedings 2002 Winter Simulation Conference, San Diego, CA, Dec. 8-11, 2002 (to appear).

B. Beizer, *Software Testing Techniques*, Van Nostrand Rheinhold, New York, second edition, 1990.

G. Bella, F. Massacci, L. C. Paulson, and P. Tramontano. Formal verification of cardholder registration in SET. In F. Cuppens, Y. Deswarte, D. Gollman, and M. Waidner, editors, Computer Security | ESORICS 2000, LNCS 1895, pages 159{174. Springer, 2000.

G. Bella, F. Massacci, L. C. Paulson, Verifying the SET Purchase Protocols, University of Cambridge, 2001.

M.R.Blackburn, R.D. Busser, “T-VEC: A Tool for Developing Critical System”, Proc. 11th International Conference on Computer Assurance, Gaithersburg, Maryland,USA pages 237-249, June, 1996.

M.R.Blackburn., R.D. Busser, J.S. Fontaine. “Automatic Generation of Test Vectors for SCR-Style Specifications”, Proc. 12th h Annual Conference on Computer Assurance, Gaithersburg, Maryland, pages 54-67, June, 1997.

D. Bolignano. Towards the formal verification of electronic commerce protocols. In *Proc. 10th IEEE Computer Security Foundations Workshop (CSFW*). IEEE Computer Soci-ety Press, 1997.

Bonatti, P., Vimercati, S.D.C., Samarati, P., “A modular Approach to Composing Access Control Policies“, Proceedings of Computer Communications Conference, 2000.

Bose, P. “Automated Translation of UML Models of Architectures for Verification and Simulation Using SPIN”,

J.R. Burch, E.M. Clarke, D.L. Dill, B. Misra, “Automatic Verification of Temporal Circuits Using Temporal Logic”, IEEE Transactions on Computers, C-35, V. 12, pp. 1035 – 1044.

E.M. Clarke and J. Wing, “Formal Methods: State of the Art and Future Directions”, ACM Computing Surveys, 1996.

Common Data Security Architecture (CDSA), http://www.opengroup.org/security/l2-cdsa.htm, 1998

Defense Modeling and Simulation Office, “Conceptual Model Development and Validation”, www.msiac.dmso.mil/vva/Special_Topics/ Conceptual/conceptual-pr.PDF, Nov. 30, 2000.

[DMSO] Defense Modeling and Simulation Office, *VV&A Recommended Practices Guide*, http://www.msiac.dmso.mil/vva/default.htm, 2001.

S. M. Easterbrook and J. R. Callahan, "Formal Methods for Verification and Validation of partial specifications: A Case Study," *Journal of Systems and Software*, vol. 40, (3), 1998.

S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton “Experiences Using Lightweight Formal Methods for Requirements Modeling,” *IEEE Transactions on Software Engineering* , Vol. 24, No. 1, January, 1998.

S.R.Faulk, P.C.Clements. “The NRL SCR requirements specification”, Proc. 4th International Workshop on Software Specification and Design, Monterey, California,USA, 1987.

C.Heitmeyer, J.Kirby,B.Labaw and R.Bharadwaj. “SCR: A toolset for specifying and analyzing software requirements”,Proc. 10th Annual Conference on Computer-Aided Verification, Vancouver, Canada, 1998.

JDBC Data Access API, http://java.sun.com/products/jdbc/download.html, 2000

ISO/IEC 9075, "Information Technology --- Database Languages --- SQL", http://www.iso.org, 1999

ISO/IS 15408 International Standard (IS), http://csrc.nist.gov/cc/ccv20/ccv2list.htm, 2000

D. Jackson, "Lightweight Formal Methods", *International Symposium of Formal Methods Europe,* Berlin, Germany, March 12-16, 2001, Proceedings.

W.Jansen. "Security Testing Characteristics", http://csrc.ncsl.nist.gov/sectest/Security_Testing.html, April 1998

W. Janssen, R. Mateescu, S. Mauw, P. Fennema, P. v.d. Stappen, "Model Checking for Managers", *6th International SPIN Workshop on Practical Aspects of Model Checking*, Toulouse, France, 21 and 24 September 1999.

R.R. Lutz, "Reuse of a Formal Model for Requirements Validation", Proceedings, Fourth NASA Langley Formal Methods Workshop, Sept. 1997.

Mastercard and VISA. SET Secure Electronic Transaction Specification:
Formal Protocol Definition, May 1997.
http://www.setco.org/set specifications.html.

C. Meadows and P. Syverson, "A Formal Specification of Requirements for Payment Transactions in the SET Protocol", *Proceedings, Financial Cryptography*, 1998.

S. Merz, "Model Checking, a Tutorial Overview", F. Cassez et al. (eds): *Modeling and Verification of Parallel Processes.* Springer LNCS 2067, pp. 3-38. (2001)

National Institute of Standards and Technology, FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION 140-2. SECURITY REQUIREMENTS FOR CRYPTOGRAPHIC MODULES (Supersedes FIPS PUB 140-1, 1994 January 11)

P.G. Neumann, R.S. Boyer, R.J. Feiertag, K.N. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs. Technical report, Computer Science Laboratory SRI International, Menlo Park, California, May 1980. 2nd ed., Report CSL-116.

Oracle Corporation, Oracle8 Security Target Release 8.0.5, April 2000.

Pace, D.K., "Conceptual Model Development for C4ISR Simulations", 5th International International Command and Control Research and Technology Symposium, Dept. of Defense, 2001. http://www.dodccrp.org/2000ICCRTS/cd/papers/Track2/059.pdf

Pace, D.K., "Simulation Conceptual Model Role in Determining Compatibility of Candidate Simulations for a HLA Federation", *Proc. Spring 2001 Simulation Interoperability Workshop*. Paper 01S-SIW-024. 25-30 March 2001. Orlando, FL. http://www.dmso.mil/briefs/war/vva/prod/01S-SIW-024.doc.

Rushby, J. Formal Methods and their Role in the Certification of Critical Systems, SRI Technical Report CSL-95-1, March 1995.

Safford, E.L., " Key Applications of Test Automation Framework (TAF)", Proceedings of the 12th Annual Software Technology Conference, April 30-May 5, 2000.

Sargent, R.G. “Validation and Verification of Simulation Models”, *Proceedings, 1999 Winter Simulation Conference.*

S.W. Smith, S.H. Weingart. ``Building a High-Performance, Programmable Secure Coprocessor." *Computer Networks (Special Issue on Computer Network Security).* 31: 831-860. April 1999.

S.W. Smith, R. Perez, S.H. Weingart, V. Austel. ``Validating a High-Performance, Programmable Secure Coprocessor." *22nd National Information Systems Security Conference.* October 1999.

S.W. Smith, V. Austel. ``Trusting Trusted Hardware: Towards a Formal Model for Programmable Secure Coprocessors." *3rd USENIX Workshop on Electronic Commerce.* August 1998.

S. Stepney. “New HoriZons in Formal Methods”, The Computer Bulletin, British Computer Society, January 2001.

Stevenson, D.E. "A Critical Look at Design, Verification, and Validation of Large Scale Simulations ", *IEEE Computational Science and Engineering* (to appear).

Warmer, J.B., A.G. Kleppe, *The Object Constraint Language: Precise Modeling With UML*, Addison-Wesley, 1998.