AN INTRODUCTION TO THE MECHANICS OF FORMAL METHODS: A FOCUSED EXPOSITION OF JOHN RUSHBY'S PAPER, FORMAL METHODS AND THE CERTIFICATION OF CRITICAL SYSTEMS

Prepared for
Dr. Todd Wilson
Computer Science 198
Spring 2001

Prepared by
Jason J. Simas
Computer Science and Philosophy Undergraduate
California State University, Fresno
September 15, 2001

CONTENTS

INTRODUCTION

Few introductory articles or texts in Formal Methods focus on introducing the mechanics of Formal Methods. Introductions frequently commingle descriptions of the mechanics of Formal Methods, with arguments for the use of Formal Methods, and with descriptions of the successes or failures of Formal Methods applied in industrial practices. Though the commingling of these descriptions together can serve a purpose, each of these can server their own distinct purpose. There are individuals wanting merely to understand *what* Formal Methods are or *how* Formal Methods work and not *why* Formal Methods should be applied nor *where* Formal Methods. These are individuals wanting to understand the *mechanics* of Formal Methods. This paper helps serve this purpose providing an *introduction* to the mechanics of Formal Methods.

Formal Methods is a loose term and can be applied to quite a few disparate engineering practices. These practices can have varying levels of automated or mechanized support. The most interesting and perhaps promising practices are those that can be automated or mechanized in a straightforward way. For these reasons, this paper focuses on describing those practices of Formal Methods for which there exists significant mechanical support or those practices of Formal Methods that are necessary in order to take advantage of mechanical support.

In 1993, John Rushby of the Stanford Research Institute wrote a paper on Formal Methods that served the specific purposes of the paper's sponsors, the Federal Aviation Administration (FAA) and the National Aeronautics and Space Administration (NASA). The paper's content also allowed it to serve as an introduction to Formal Methods for many individuals. However, the paper includes and commingles much more information than is necessary for an introduction to the mechanics of Formal Methods. In writing this focused exposition of Rushby's paper, the relevant information was extracted and re-organized. Within this paper, the reader will find parenthetical indexes (e.g. (57)); the number inside the index cites the page number within Rushby's paper where this information was found. Some quotes from Rushby are also provided.

With any informational paper or text some assumptions must be made about the knowledge of the intended readers. There exist many good introductions to Software Engineering and also many good introductions to Logic. This paper assumes that a reader has an introduction to Software Engineering (e.g. the waterfall model of the software engineering lifecyle, etc.) and an introduction to Logic (e.g. the axiomatic method, first order logic, etc.) Also note that though this paper seeks to introduce the mechanics of Formal Methods for hardware engineering as well as software engineering, it serves better for software engineering.

1: FORMAL METHODS

John Rushby writes, "“Formal Methods” are the use of mathematical techniques in the design and analysis of computer hardware and software." (7) Although this is an arguably correct description of Formal Methods, for the context of this paper and assuming the readers of it have a background in computer systems engineering and in particular the engineering lifecycle of computer systems, another description may be more useful. Formal Methods is a collection of procedures that employ mathematical logic in the specification, verification, and validation of computer systems. In specification, the language in which the specifications are written is (or can be translated into) a mathematical logic. In verification, more concrete specifications are shown to satisfy more abstract specifications. In validation, specifications are analyzed for consistency or for the ability to prove desirable theorems from their axioms.

Mathematical logic is particularly effective with engineering computer systems because computer systems are essentially information processing systems. Information comes in to a computer, it is processed, and then it is sent out. Even control systems can be seen this way (i.e. read information from sensor, process it, send information to actuator). Use of mathematical logic is naturally suited for describing *what* follows from information without having to specify *how* to process the information. (13) Separating the "what" from the "how" assists engineers in dealing with complexity.

The "what" can be described at various levels of abstraction. (42) For example: Level N: "if information then information". Level N/2: "if moving(car) then (closed(doors) & locked(doors))", Level 1: the computer system itself. The specifications written at more concrete levels of abstraction, since it is written in (or can be translated into) a mathematical logic, can be shown to *satisfy* the specifications written at the more abstract levels. The relevant properties of a gate layout or source code of a computer system itself can be translated into mathematical logic and can be analyzed to satisfy the more abstract specifications.

Mathematical logic is also particularly effective at modeling the discrete behavior of computer systems. (8) The outputs of computers can vary widely given narrow variation in the inputs. For example, a sensor reading that changes by one digit, may mean the difference between some warning light routine (or some hazard avoidance routines) from being invoked or not. Once the discrete behavior is specified in a logic, then the logic can use its inference mechanisms to determine the satisfaction of higher level specifications (i.e. verification) and to help determine whether specified system has all the properties desired of it (i.e. validation).

There is one last property of mathematical logic that makes it particularly effective. In mathematical logic, proofs are reduced to symbol manipulation which means that computers are fit for proof checking (i.e. determining whether proofs are valid) or can assist in theorem proving (i.e. finding proofs on its own or providing assistance for finding proofs). (7,8) Automated analysis can help provide assurance of the utility of a specification and of a computer system without having to actually build and test the system. The most straightforward way of providing this level of automation in the engineering of computer systems is to specify systems in a specific syntax and have that syntax correspond to syntax of a mathematical logic. (18) Let's first study the various foundations or mathematical logics upon which Formal Methods can be and are built.

2: MATHEMATICAL LOGIC

Though there are many different mathematical logics. For the foundation of Formal Methods, Rushby *expresses* confidence in a higher order, typed logic, *expresses* interest in constructive logics, and *expresses* dissatisfaction with set theory (220). The problems of specifying, verifying, and validating computer systems place burdens on traditional logics, and Rushby advocates research in developing logics and decision procedures tailored to relieve *these* burdens. (220)

A logic suitable for computer systems may be undecidable, incomplete, but sound, and still provide a foundation for formal methods. Though in general decision procedures may be undecidable for a logic, the proof of some theorems may have decision procedures that are decidable. Thus, there may be multiple logics for formal methods, each one (with its decision procedures) a subset of the most general logic that is undecideable. That is, if some steps in a proof are simple enough to have decision procedures, then these should be known and applied; if not then the more powerful, more universal, logic should continue its proof procedures. Godel's first incompleteness theorem establishes that every "reasonably powerful" system is incomplete, and the logic of Formal Methods (since it needs this power in describing powerful systems) need not be complete (i.e. every theorem need not be provable). Rushby writes, "The true but unprovable statements concern sweeping properties of logic itself, not the specific kinds of properties we are concerned about in formal methods." (256-257) Of course the logics must be sound. For otherwise, they would be useless.

A logic suitable for computer systems must support (i)equality relations, (ii)arithmetic, (iii)simple computer systems datatypes, and (iv)set theory. (251) Regarding (i), if whatever is provable of one entity in the logic (e.g. datatype, function, etc.) is provable for another, then *their* equality should be establishable in the logic. Related to this is an inference mechanism called term rewriting. Term rewriting is a procedure where a suitable match for the theorem to be proved is looked for amongst the axioms. Unification is a related inference mechanism, and logics supporting both term rewriting and unification appear necessary for Formal Methods. Regarding (ii), linear arithmetics (i.e. where multiplication may only occur amongst one variable (e.g. 3*x, 7*c, etc.)) that are extended to support some functions while maintaining decidability are useful in formal methods. But some systems will require more powerful arithmetic. Regarding (iii), computer systems datatypes can be supported either by being an abbreviation for some underlying set of definitions and axioms in the logic, or by being embedded in the logic itself. (260) A set of constants, functions, and predicates define a datatype (270); generally there will need to be constructor functions (that build up the datatype), accessor functions (that access the members of and "unbuild" the datatype), an equality relation (to determine whether two instances of the datatype are equal), and an induction scheme (to prove properties about instances of the datatype). Examples of such datatypes are stacks, queues, and trees. Regarding (iv), set theory is a powerful instrument and should be supported with some qualification.

A logic suitable for computer systems should provide higher orders of logic while avoiding paradoxes such as Russell's. Russell's paradox is a particular type of theorem that can show a defining axiom of a logical system to be false. That is, types of Russell's theorem can show a logical system to be unsound. Logical systems must have axioms that define the notions of predicate, set, etc. If one of these axioms can be shown to be false, within that system, then the system is unsound. An example of such an axiom definition written in a higher order logic is: $(\exists P : (\forall Q : P(Q) \text{ iff } \phi\,(Q)))$. The problem occurs on the instance where $P=Q$ and where ϕ = "the predicate that characterizes those predicates that do not exemplify themselves." (279). Special adjustments must be made to make sure $P \neq Q$. One such adjustment is to disallow "impredicative definitions". "An impredicative definition is one that has a particular kind of circularity; it is not the circularity of recursion, but one where a thing is defined by means of a quantification whose range includes the thing being defined." (280) Some adjusted theories are axiomatic set theories, type theories, and constructive theories.

2.1: Axiomatic Set Theories

A well known axiomatic set theory is Zermoelo-Fraenkel (or ZF) set theory. A sometimes preferred axiomatic set theory is Neumann, Bernays, and Godel (or NBG) set theory. Axiomatic set theories disallow impredicative definitions by requiring that new sets (or predicates which are defined by sets in axiomatic set theory) only be made from existing sets. $P \neq Q$ because the P would not yet exist; it would first need to be defined. Rushby notes at least two negative properties of using set theory as a foundation for Formal Methods. (i) In axiomatic set theory, functions are inherently partial; this poses a problem for theorem proving. (91, 287) (ii) Types are very useful for theorem proving, and set theory doesn't naturally support them.

2.2: Constructivist Theories

Constructivist theories require theorems to be proved by showing how they can be derived or constructed. A salient example of this is their abandonment of the “law of non-contradiction” (i.e. assume that P, if a contradiction results, then conclude ~P). “Perhaps P didn’t bring about the contradiction; perhaps one of the other premises are false,” a constructivist might say. For a constructivist, P must be constructed or derived from what is known, not by showing how it would otherwise undermine what was known. Constructivist theories don’t admit of Russell’s paradox (i.e. P≠Q) because the use of P could not occur until the construction of P.

The constructivist theories best suited for Formal Methods are those heavily influenced by the theories of Martin-Lof (because they support arithmetic and sets or types). Detractors argue that constructivist theories burden one with the requirement of existence proofs. The “Curry-Howard isomorphism” shows that constructivist existence proofs can be converted into programs. (296) And having a program, would establish the specification is consistent – something useful for validation (see Section 5.1). Rushby writes, “… similar proof obligations arise in classical logics with rich type systems … Hence, it may be that the actual practice of formal methods in constructive and classical settings is more similar than might at first appear.” (297)

2.3: Type Theories

Type theory is built on a foundation of functions (with predicates being a special case) and allows quantification over them; thus, type theory is a higher order logic. (283) Quantification extends over only a particular sort or type of entities (278), and partial functions are made to be total functions over precisely specified domains. (291) Russell’s “ramified type theory” is a notable example of a higher order type theory. (280)

Type theory disallows impredicative definitions, Rushby writes, “by “stratifying” sets according to their “type” or “level.” Individual elements can be considered to belong to level 0, sets of those elements to level 1, sets of sets of elements to level 2, and so on.” (281). Hence, P≠Q because P does not apply to things P; the quantification wouldn’t be over P.

Type theory is well suited for formal methods because it supports the automation of verification and validation activities. The computer systems community is well aware of the advantages of having a strongly typed system. These advantages carry over into Formal Methods. Specification languages based on type theory, can require an engineer to specify types for the variables, functions, and predicates.

Automated typechecking, then, can catch many errors that would be more difficult to detect. Typing provides a safe-harbor from Russell's paradox, yet provides the power of quantification. The support for partial functions in the form of total functions on strongly typed domains, provides flexibility without damaging effectiveness of automated analysis.

Furthermore, the application of typed theory in Formal Methods overcomes objections that couldn't be overcome if applied to other endeavors (i.e. mathematics for example). Computer systems engineers are already used to dealing with the complexity of typing; learning how to write typed statements will not be overly difficult. (286) The incompleteness of type theory is moot because the formal methods must be founded on a powerful system that includes arithmetic anyway. (287) The undecidability associated with typechecking a system based on typed total functions, is made acceptable when coupled with a powerful theorem prover (see Section 4.1). (60, 292)

3: FORMAL SPECIFICATION

In the specification stage of the engineering lifecycle, the system is designed and described in a formal notation, a formal specification language. In the context of this paper, specification *will include* a stage in the lifecycle that is *usually not included*, System Design, and *will mostly exclude* implementation (though implementation can be seen as a very concrete specification). Formal Specification (and Formal Methods) in general can be applied at any stage (or level of abstraction) in specification and for any sub-section of the system. For example, perhaps Formal Specification (and Formal Methods in general) will be applied to only one very critical algorithm (to develop some assurance that algorithm is suitable before implementing it). In this context Formal Specification will be spoken of as existing throughout the specification stage.

The specification will need to describe the expected (relevant) environmental parameters within which the system will be expected to perform correctly (e.g. a system for an ant-arctic rover will need to withstand extreme cold). This is necessary not only for making sure the system will operate correctly but for making sure the system doesn't have unnecessary functionality. The functional expectations of the system itself will need to be specified (e.g. what to do in the environment). A suggestive formal depiction is: (environment -> (system -> requirements)). (9) The functional specifications of a system must carefully incorporate both how a system should behave (positive properties) with how it shouldn't (negative properties). For example, moving(car) -> ~pop(hood). And example that falls under the system class, but appears to be environmental is, extreme(wind) -> stop(car), where wind really is just a sensor feeding information to the system (i.e. wind is just as much a part of the system as car or hood which are sensor based as well).

Formal specification languages are founded on a mathematical logic. Type theory and set theory are two popular foundations. Formal specifications of a system are a collection of definitions and axioms (written in the specification language). (58) Definitions create or define new members of the specification language (i.e. constants, functions, or predicates) out of existing members. Axioms assert a property over existing entities. (58) Maintaining consistency in a specification is paramount. An inconsistent specification is worthless since anything can be proven from an inconsistent specification. Inconsistency can be introduced, for example, when defined entities are defined twice, representing two different things (i.e. imagine that one can be true while the other is false, because they are different things yet have the same name in the specification). This is part of the rationale for requiring specifications to include explicit definitions (as opposed to having definitions merely be axioms). A specification can then be checked for this problem. More complex ways of introducing inconsistencies are handled by forcing definitions to conform to a definitional principle which is like a template. Axioms can introduce inconsistencies as well, and there are various ways of handling this including model checking (see Section 12).

A specification language that is based on types assists a specifier by allowing the specifier to embed much of the specification in the typing. (55) For example, the moving (car) predicate is meaningless with moving (locks). Axioms involving quantification can be written more simply (i.e. excluding extra predicates so moving() only ranges over speed sensors) if specification languages are typed. Types are both explicitly introduced in definitions, explicitly and implicitly used in definitions, and implicitly used in axioms. One example, a definition explicitly using a type might be: type predicate: moving (type speed: car). Specification languages built on powerful foundations like type theory can "encode other formalisms such as finite state machines or temporal logics relatively conveniently." (80) This is important because, "Concurrent and distributed systems are generally based on temporal logic." (29)

There may be multiple specifications of a system, each at their own level of abstraction. A top-level specification captures all the necessary behavior of a system and the relevant assumptions about the environment in which it will operate. Making more concrete (or lower level) specifications that satisfy the top-level specification is generally considered part of the design stage of the lifecycle; yet this design process can be carried out using specification languages by adding more and more detail to a top-level specification. The top-level specification should specify *what* the system should do. The lower levels of specifications will increasingly specify *how* the system will do it by defining more detailed parts of the system and the interrelationships amongst the parts. (39, 64)

Specifications have "loose semantics" which means that they may be satisfied by multiple more concrete specifications. (64) The loose semantics of a specification language may pose problems when it needs to be interpreted (i.e. understood by a person). For these reasons formal specifications generally should use descriptive language in the naming of their functions, predicates, and constants (and are usually accompanied by a corresponding statement in a natural language such as English). If there still is a problem, the semantics of a formal specification can sometimes be fixed by challenging an interpretation (i.e. "if this function means this, then this should follow") (see Section 5.2). (74) Regarding the levels of abstraction in which specifications can be written the initial levels are generally more "property oriented", while the lower levels are increasingly more "model-oriented". (28) Property-oriented specifications describe a system by stating the properties of the system; model-oriented specifications describe a system by providing a model that has the properties of the system.

3.1: Property-Oriented Specifications

Property-oriented specifications describe the properties of a system by defining functions and exhibiting relationships between these functions, to which a system will have to conform. A system will have to perform certain operations. These operations can be described as functions. The relationships between these operations, stated as axioms in the specification, provide the semantics of them, and a system must have operations that conform to these semantics. The following example is adapted from Rushby. (26) It is a partial definition of a computer systems datatype, a stack.

```
push: [elem, stack -> stack]  //specifies the type for push
pop:  [stack       -> stack]  //specifies the type for pop
top:  [stack       -> elem ]  //specifies the type for top

pop (push (e, s)) = s         //pop returns the last pushed stack
top (push (e, s)) = e         //top returns the last pushed element
```

Frequently theorems can be stated more easily within a property-oriented specification. Many theorems can be stated using quantification that would otherwise require specifying an explicit search procedure in model-oriented specifications. The ease of stating theorems is important for validation (See Section 5.2).

3.2: Model-Oriented Specifications

Model-oriented specifications describe a system by giving a model that has the properties required of the system. This method is based on the notion of an explicit state. This specification will be built out of more general components that are part of the specification language. Specifying a model includes specifying the components of the state (i.e. variables), how the components are related so that they have a fixed semantics, the initial state of the components, and the operations that access and modify the components of the state (i.e. the state transitions). (33-34) If a system satisfies the model, then it will have every property of the model. The following example is adapted from Rushby. (27) It is a partial definition of a computer systems datatype, a stack.

```
stack:
  array:  [nat -> elem],  //maps natural numbers to elements
  pointer: nat            //an index into the array

push (e, s) = s with                              //changing stack
  [pointer: = s.pointer + 1,                      //pointer to next
   array:   = s.array with [s.pointer + 1 := e]]  //puts element in
pop  (s   ) = s with                              //changing stack
  [pointer := pred (s.pointer)]                   //pointer to previous
top  (s   ) = s.array (s.pointer)                 //get this element
```

Model-oriented specifications specify *what* is to be done by specifying *how* it is to be done using the constructs of a given specification language. This method guarantees consistency of the specification but it does make verification and validation more difficult (see Sections 5.1.3). (27-28)

4: FORMAL VALIDATION AND VERIFICATION

Formal Validation and Formal Verification (V&V) have in common at least three properties (i)they are both processes that analyze a system for correctness, (ii)the automation of their analyses are made possible by system specifications being written in formal specification languages (19), and (iii)some of the automation of their analyses is carried out (or can be carried out) using the same underlying tools or "machinery" (75). Briefly stated, validation seeks to determine whether a system satisfies a customer's requirements; verification seeks to determine whether a higher level specification is satisfied by a lower level specification, the lowest level specification being the implementation (see Sections 5 and 6).

The analyses of V&V are advanced by proof. Proof gives V&V the ability to establish properties of correctness for a specification in a way that is absolute and unequaled by conventional testing. Formal analyses based on proof, implicitly "test" all possible behaviors of a system specification, not just for particular inputs or scenarios. (178). However, the analysis is lengthy and technical; automation is most required. Thankfully, proof can be automated or assisted using computers. Two such classes of computer systems (or tools) are proof checkers and theorem provers. (19) Proof checkers merely check the validity of proofs that are usually written by an engineer. Theorem provers can produce proofs themselves when given a particular theorem to prove that are sometimes provided by an engineer. Proofs are essential to the analyses of V&V, powerful theorem provers are essential for tools that support the automation of these analyses, and automation of these analyses are essential for the success of V&V. Otherwise, V&V would be too labor intensive and error prone. (87) Rushby characterizes an effective automation system as one that: (89)

(i) "Supports a specification language as attractive and rich as those that have found favor in unmechanized applications,"
(ii) "Provides support tools to assist in the validation and documentation of specifications, including parsing and typechecking of sufficient strictness that most simple faults are quickly flushed out, and "
(iii) "Is supplied with a theorem prover or proof checker that allows conjectures of all kinds (large, small, easy, difficult, valid and invalid) to be proved or found in error with an efficiency at least equal to that of the best stand-alone theorem provers."

There is one significant caveat that is very relevant to Formal Methods and specifically to the V&V analyses of correctness. As has been noted, that Formal Validation and Verification are made possible by the specifying the system in a formal specification language, that some of the specified properties are properties of the environment, and that the properties of the environment must be true in order for the system to be guaranteed to meet the requirements. This question should pertain: "What then if the assumptions about the behavior of the environment are incorrect?" This question introduces a problem more generally occurs whenever information is translated into or out of a formal system. If the translations do not correctly capture the relevant properties of the real world or if the translations incorrectly apply the specifications in the real world, the analyses of V&V are undercut and do not provide assurance of correctness.

Rushby writes, "[An absolute guarantee] is impossible for any enterprise that uses modeling to predict the behavior of real-world artifacts; we can have no absolute guarantees that the upper specification captures all the requirements perfectly, nor that the lower specification (which may be a program or gate layout) exactly describes the behavior of the actual system: both of these are issues that require validation, not verification." (75) The particular problems Rushby mentions do fall in the domain of validation; interestingly some formal analysis can assist in validating the questionable formal analysis. But other problems do appear to fall more in the domain of verification. Recall that more concrete specifications can be shown to satisfy more abstract ones, that the more concrete specifications are more model-oriented, and that more model-oriented specifications have a different specification language than more property-oriented specifications. Though most of these specification languages may have a common underlying mathematical logic (i.e. type theory) at least one such specification language will not (i.e. the language of implementation). The *translation* from the programming language or gate layout into a more abstract specification language may go awry. The *modeling* of the language of implementation may be erroneous. This problem of correctness of translations and fidelity of modeling does appear to a common problem of both Formal Validation and Verification and hence, of V&V. There are some ways to detect faults of poor modeling (see Section 5).

4.1: Theorem Proving

As has been suggested, Formal Validation and Formal Verification offer powerful support to the processes of validation and verification, but their power will only be as strong as the theorem prover that supports them. (85) Theorem proving in this context refers to an activity in which theorems are proven with some *degree* of mechanical support by tools called theorem provers. Theorem provers basically take as inputs a formal specification of a system and an arbitrary theorem, and outputs information including whether the theorem is provable from the specification (recall that specifications are basically definitions and axioms).

Rushby cites several properties that theorem provers should have in order to be effective in the application of formal methods. First, theorem provers must have as a foundation a very rich logic such as higher order type theory; otherwise, theorems that express the more complex properties of computer systems will not be possible. (85) Second, theorem provers should be more than proof-checkers; they must be able to discharge many of the theorems themselves; this leads to great increases in productivity. (91). Third, theorem provers should be just as powerful in the discovery that a theorem cannot be proved or is false as they are in the discovery that a theorem is true. This property of a theorem prover is helpful when trying to develop more concrete specifications for the satisfaction of more abstract ones. (92) Fourth, theorem provers should provide four ways to control how they go about trying to prove a theorem (i.e. useful if the theorem is too complex to be performed automatically). These ways include control by "direct instruction", control by specifying the order in which certain theorem proving techniques are tried, control by specifying or programming explicit "proof strategies", and control by specifying search depths. (94) Rushby writes, "Large and complex theorems will require human control of the proof process, and we would like this control to be as straightforward and direct as possible." (93) Fifth, theorem provers must provide effective automation of arithmetic; for many theorems involve arithmetic. (93) Rushby notes that a theorem prover's effectiveness or productively (in general) is a result of "tight integration" of decision procedures such as "term-rewriting and arithmetic" decision procedures. (93) Rushby mentions this tight integration in a way that appears to imply that significant research-effort should be applied in finding integrations between powerful techniques that maintain some decidability. Seventh, theorem provers should be able to produce a humanly readable proof so that, for instance, proofs about the expected behavior of the system can be evaluated for quality control purposes by humans. (94) Lastly, Rushby seems to discourage integration of heuristic procedures; this is perplexing. (92) There will not exist decision procedures for every theorem; so heuristic procedures appear to be the next best thing.

Theorem provers are general tools that are applied in many ways in Formal Validation and Verification. They are applied in "theory interpretation" (see Section 4.2), "typechecking" see Section (5.1.1), in "challenging" a specification (see Section 5.2), and elsewhere.

4.2: Theory Interpretation

Theory interpretation is an activity whereby one abstract formal specification is translated into another more concrete formal specification. This activity is used in Formal Verification (e.g. showing the satisfaction of formal specifications) and also in Formal Validation (e.g. showing that a formal specification is consistent). Theory interpretation proceeds by finding (or building) constants, functions, and predicates in the more concrete specification with the same semantics as those found in the more abstract specification. Then the axioms of the more abstract specification are added to the axioms of the more concrete specification. If the more concrete specification can prove all the axioms that were translated into it, then the more concrete specification is said to be an interpretation of the more abstract formal specification (or theory). (63, 229) A theorem prover can be used in the proof process of theory interpretation. But note that the finding (or building) of corresponding constants, functions, and predicates will generally require a human with a keen understanding of the semantics of both specifications.

Theory interpretation can generally only be undertaken when going from the more abstract to the more concrete. Trying to go from the more concrete to the more abstract will generally fail, because the more abstract doesn't have the granularity to allow for translation without losing detail. If detail is lost, then the more concrete specification could not be satisfied by the more abstract specification. This explains what will be said later about Formal Verification, that a more concrete specification cannot be proven to exclusively satisfy a more abstract specification. (see Section 6). Theory interpretation is used in "model checking" (see Section 5.1.3 and 5.2), and "satisfaction checking" (see Section 6.1).

4.3: State Exploration

State Exploration is an activity where theorems can be proved about a more model-oriented specification. (12) Model-oriented specifications have state; the data contained in the variables of a model defines its state. If there are a finite number of states in a model, a theorem about the relations between the data in the model can be proved by exploring all possible states of the model. If the theorem holds under this exploration, then it holds for the model. Models with very large (but still finite) numbers of states can be explored. (8, 289) Generally, more abstract specifications will either have too many states or will be infinite (leaving the detail of choosing just how finite to later in the specification process). Thus state exploration cannot replace theorem proving. However, state exploration can be applied effectively in some special cases. For example, state exploration can assist in Formal Validation by proving the consistency of a formal specification (see Section 5.1.3).

5: FORMAL VALIDATION

Validation is a process that provides assurance that a finished system does (or will) satisfy the customer's requirements. A key insight here is that validation (if began early) can provide early assurance that a system is "on its way" to satisfy the customer's requirements, or more specifically, early validation can help detect if a system will not be satisfying the customer's requirements. For example, a top-level specification can be scrutinized to determine whether it really does capture all the functional requirements of the system; this is early-validation. Formal Validation can assist this process by assisting in the detection of formal specifications that if implemented would *not* satisfy the customer's requirements. That is, formal validation is like a debugging technique for formal specifications. If a formal specification contains a fault, then Formal Validation can help find that fault before any time is spent implementing the specification. If Formal Validation is applied on the actual system, the system will need to be translated into a formal specification using a process known as theory interpretation; then this specification can be examined for faults. Formal Validation cannot do anymore than what has been described here, yet Formal Validation is a powerful process. It can help stop faults at an abstract level of specification (including the top-level specification itself) from propagating down into more concrete specifications (including the system itself).

Rushby says that Formal Validation does two things (i)it helps provide assurance that a specification is internally consistent and (ii)it helps provide assurance that a specification says what was intended. (53) Regarding (i), an inconsistent specification could not be implemented and hence, would not produce a system that would satisfy a customer's requirements. (63) Regarding (ii), formal specifications can be very large or complex. One might wonder whether some desirable property (that wasn't explicitly stated as an axiom) holds (e.g. ~speed(100)). That is, Formal Validation can challenge a specification to determine whether it really says what was intended. Techniques for (i) have to do with type-checking, definitional principles, and model checking. Techniques for (ii) are what Rushby calls challenging a specification. These are discussed in the order mentioned, in the following sections.

5.1: Consistency

A formal specification is inconsistent if it is possible to derive a contradiction from (or within) it. Inconsistency exhibits itself as a problem in two ways: (i)any theorem can be proved to be true or proved to be false depending on the steps taken in the proof (ii)an inconsistent specification cannot be implemented. Regarding (i), this is clearly a problem because the purpose of proving a theorem is to determine whether the theorem holds; if whether a theorem holds depends on the steps taken in a proof, then the purpose of proving a theorem is undercut. Regarding (ii), this is clearly a problem because formal specifications are implemented; if there is no such implementation, then effort applied to implement the specification is wasted.

There are two main ways of providing assurance that a formal specification is consistent: (i)a (consistent) model can be shown to satisfy the formal specification and (ii) a specification language may restrict the form or syntax in which formal specifications are written in ways that provide early detection *of* problems that can *lead* to inconsistency. (54) Regarding (i), this way can provide full assurance that a specification is consistent, if the model is consistent. Regarding (ii), these ways include the use of types, definitional principles, and sometimes axiom principles to catch many faults that may infect the specification to the point of inconsistency. (20, 62) These are discussed in the following sections.

5.1.1: Typechecking

Recall that specifications that require variables, functions, and predicates to be defined before their use can also require that their *use* be defined through the notion of typing (e.g. pop [stack -> stack]) as well. Formal specification languages with such rules support the detection of many faults that can lead to inconsistency. (39) This class of faults can be detected through typechecking. Typechecking is an activity where the *use* of variables, functions, and predicates is checked to match their defined use. Typechecking is performed by a typechecker. In order to give two examples of typechecking, consider if the property-oriented stack definition given earlier was extended to deal with empty stacks by redefining pop [stack -> stack] to be pop [non_empty_stack -> stack] and the types non_empty_stack and empty_stack were added to the specification. An axiom containing pop (empty_stack) would easily be detected because pop only operates on entities of type non_empty_stack. A more complex example, one that suggests the need for theorem proving in typechecking, is if the following axiom was part of a specification: pop (pop (push (push (e, s)))) = s. This axiom is no doubt true, but it requires that the second pop be shown to return a non_empty_stack and by definition, pop is only guaranteed to return a stack. Theorem proving could prove this theorem type correct, by proving that the second pop really does return a non_empty_stack. (57) Incorrect *use* of functions and predicates can accidentally introduce inconsistencies, and Rushby writes, “Typechecking is a very strong check on internal consistency.” (39)

5.1.2: Definitional Principles

Definitional principles are rules by which definitions of functions, predicates, or datatypes must conform and in their enforcement, a specification is guaranteed against becoming inconsistent through the *definition* of some new function, predicate or datatype. (59) Note that the technique of assuring consistency through enforcement of definitional principles is complementary to the technique of typechecking; typechecking attacks introduction of inconsistency by *use* (i.e. in the axioms); the enforcement of definitional principles attacks introduction of inconsistency by definition (i.e. in the definitions). Basically definitional principles should ensure that nothing new, besides notation, is added to the specification (i.e. the set of theorems that are provable before a definition is introduced should be the same as the set that are provable after the definition). (267-268)

Definitional principles for functions and predicates are quite simple. An example of a definitional principle for a predicate is: $P(x_1, \ldots, x_n) \equiv \phi$ where P is the predicate being defined, ϕ is a syntactically correct predicate written in the specification language, and no variables except $x_1, \ldots, x_n$ are free (i.e. not quantified over) in ϕ. (269) Definitional principles for datatypes are more complex, and some datatypes cannot be defined using a general definitional principle. Datatype principles force a datatype to be defined in a way that the definition can be proved to be consistent using known techniques. That is, instead of providing principles that ensure the consistency of a datatype, datatype definitional principles ensure that the consistency of the defined datatype can be determined for instance, by a formal validation tool. Basically, a datatype definitional principle requires that a datatype have constructors, accessors, and "recognizers" and that specification of them conform to a certain form. A tool can then "expand" these definitions and prove their consistency. (61-62)

5.1.3: Model Existence

Model existence checking is a way of further providing assurance for the consistency of a formal specification. If there exists some model for which there is good reason to believe that it is consistent and if the axioms of the formal specification in question are satisfied by it, then the formal specification is consistent. At first this technique may appear to have little value; the value of this technique lies in the relatively few necessary conditions the model must satisfy. Unlike all the necessary conditions the finished system must satisfy (i.e. be expressed in a programming language), the consistent model need only be known to be consistent and be proved to satisfy the formal specifications axioms. Hence, the model for consistency may be quite simple. This technique employs the activity of theorem interpretation to interpret the formal specifications' axioms into the model and the functionality of a theorem prover to prove those axioms are satisfied. (63)

5.2: Captures Intension

Formal specifications may fail to capture the properties of the system or of the environment – properties that should be captured if the finished system is to satisfy the customer's requirements. Complete assurance that a specification does not contain such faults cannot be established by Formal Validation, but Formal Validation can assist in the location of properties either that a specification fails to have or shouldn't have but does. This activity is what Rushby calls "challenging" a specification. (66)

Challenges can be expressed as theorems that a formal specification should or should not satisfy. (38) Understanding how challenging a specification using theorems is useful requires the understanding that all properties of a system need not be explicitly stated as axioms in a formal specification. (9) There will be some properties that are implicitly specified through inference relations between the axioms that were specified. Challenging a specification through the theorems makes sure that these inference relations produce the desired implicitly specified properties and do not produce undesired properties. For example, if I wanted to make sure that a specified automobile controller system would not allow a car to go over 100 miles per hour, I might challenge the system with the following theorem: speed (car) <= 100.

Challenges can also be expressed as models that should or should not satisfy the formal specification in question. (69) Rushby uses an illuminating example where the formal specification is of a clock. The formal specification algorithm of the clock may allow for some small error to be in the time of the clock because a real clock system will have some error. The formal specification of the real clock can be challenged by: (i)seeing if the specification is satisfied by the model or specification of a perfect (i.e. not realizable and much simpler) clock and (ii)seeing if the specification is satisfied by a clock with gross error. Regarding (i), the technique established that the formal specification has certain desirable properties. Regarding (ii), the technique establishes the formal specification does not have certain undesirable properties. This process may use state exploration or theory interpretation.

These techniques are used to detect faults in formal specifications. Though not guaranteed to detect all such faults, they can detect numerous faults and keep them from propagating further down into the system itself. Two special classes of the faults discussed should be mentioned: implementation over-constrainment (i.e. writing a specification to concretely and having it exclude some intended implementations) and implementation under-constrainment (i.e. writing a specification to abstractly or incompletely and having it include some unintended implementations). (73-74) Both of these should be avoided.

6: FORMAL VERIFICATION

Verification is a process that provides assurance that a finished system satisfies its top-level specification fully and exclusively. Formal Verification can assist with providing assurance for full satisfaction of a specification, but generally not for exclusive satisfaction. (50) There are other techniques some more formal than others that provide for assurance of exclusive satisfaction; one such technique will be briefly described later (see Section 6.2). Analogous to the relation between Formal Validation and validation, Formal Verification can assist in early detection of specifications that if implemented won't satisfy the top-level specification and likely won't satisfy the customer's requirements.

Formal Verification can be applied to any two formal specifications. Recall that there may be more than one formal specification; each specification becomingly increasingly more detailed and concrete with the implementation being the most concrete. Hierarchical verification is a name for applying Formal Verification to all these levels, showing that every abstract specification is satisfied by its next less abstract (or more concrete) specification, all the way down to a specification constructed directly from analysis of the implementation itself. (28) Hierarchical verification can catch the faults described, at the earliest possible time. (28) Although Rushby writes, "... it is certainly technically feasible to apply formal methods all the way from requirements capture through to code or circuit verification." (34) He also warns of the daunting complexity that a truly hierarchical verification requires. "Formal verification top to bottom will necessitate formal modeling of a host of low-level mechanisms, such as the communication and synchronization primitives of the programming language concerned, the interface between its run time support system and the process-management features of the operating system, and the characteristics of certain hardware devices." (37) For formal verification to be useful, it need not be hierarchical. In general, Formal Verification can be applied to any two specifications at different levels of abstraction and can prove useful in catching faults that would otherwise have propagated into the system.

<u>6.1: Full Satisfaction</u>

Recall that part of Formal Verification is showing that a more concrete specification satisfies a more abstract specification. This can be accomplished through what has been called theory interpretation (i.e. the more abstract specification is translated into the more concrete specification). If every axiom translated from the more abstract specification is proved within the more concrete specification (generally through the help of a theorem prover), then the more concrete specification satisfies the more abstract specification. For Formal Verification between two property-oriented specifications, one merely applies the process described above (35), but for Formal Verifications between specifications where one of them is model-oriented, the process could use some more description.

Recall that in state-exploration, theorems of more abstract specifications can be proved. Hence, using theorem interpretation and state exploration, some models can be formally verified. But in general models have too many states for state exploration to be effective. Some other more powerful technique must be used. The general idea of this more powerful technique is to build a property-oriented specification from the model-oriented one. C. A. Hoare invented a method that assisted in this kind of verification, but Haore's method generally isn't used in automated Formal Verification. (25) In automated Formal

Verification, the programming language constructs of the source code or the constructs of a gate layout are studied for the effects on the state of the model; in particular what is looked for is what can be guaranteed to be true at each particular line, and this is stated in terms of the target specification language. (22) For example, the assignment statement z:=0 would guarantee that z equals 0 at that point in the code and also exited while loops with the loop condition (z=0) would guarantee that z did not equal 0 at the instant the loop was exited. These assertions are combined in a way that allow for a property-oriented specification to be developed.[1] The axioms of the specification to be satisfied are then translated in this newly developed specification (i.e. using theory interpretation), and a theorem prover assists in determining whether these axioms are satisfied.

6.2: Exclusive Satisfaction

One approach to solving the problem of providing assurance that a more concrete specification exclusively satisfies a more abstract specification (i.e. it doesn't introduce any undesirable properties that were not ruled out in the abstract specification) is Software Fault Tree Analysis (SFTA); the name is unfortunate because SFTA appears to apply to hardware as well. SFTA doesn't require the foundation of a mathematical logic or that the properties of a system be specified in a formal specification language; so SFTA is perhaps out of the scope of this paper, but briefly describing it will make the discussion of Formal Verification more complete.

SFTA is applied on the finished system (or perhaps prototypes of it), and it does not provide full assurance that a system exclusively satisfies its specification. SFTA is analogous to the "challenging" of Formal Validation that was described earlier. The system is studied and particular outputs of the system are decided to be worth preventing. The system is studied starting from the outputs to determine what might immediately cause them (i.e. a function call with a certain value); if they could have any immediate causes, then the system is studied to determine whether its immediate causes might have any immediate causes of their own; and so on until the initial state or inputs of the system are reached. In this way, SFTA can determine whether it is possible for these outputs to materialize and if so, under what circumstances. (49) SFTA can be seen as providing assurance for exclusive satisfaction of a specification because these studied outputs (if they could be materialized) would likely not have been specified to occur.

[1] This vague description is suitable because Formal Verification down to the gate level or source code level is rare and arguably provides insufficient benefit to be employed.

FINAL REMARKS

Formal Methods attempt to garner support from mechanical analysis in the engineering of computer systems and the most straightforward way to do this is by using a mathematical logic as a foundation. Founding Formal Methods on a mathematical logic increases the technical complexity of engineering, but provides for many ways to detect a wide range of faults that can be introduced in the design of a system. Perhaps Formal Methods is most widely thought of as a way to prove correctness of a system. A more accurate description is that Formal Methods provide a framework in which faults in the specification and design stages of an engineering lifecycle are more likely to be detected and provide techniques for this detection. In short, Formal Methods are techniques for debugging specifications and designs. Of course this last statement is an over simplification, and to a lesser degree this paper is an over simplification. Yet, this paper has provided a fairly concrete description of some of the significant mechanics of Formal Methods, and in particular, those mechanics which enable or are supported by some degree of automation.

WORKS READ

Bowen, J. P., M. G. Hinchey. 1999. “Formal Methods.” In High Integrity System Specification and Design, Pp. 127-133. London: Springer.

Bowen, J. P., M. G. Hinchey. 1994. “Seven More Myths of Formal Methods.” IEEE Software, 12.4: 34-31. In Bowen & Hinchey 1999, Pp. 135-152.

Bowen, J. P., M. G. Hinchey. 1995. “Ten Commandments of Formal Methods.” IEEE Computer, 28.4: 56-63. In Bowen & Hinchey 1999, Pp. 217-230.

Clarke, E. M., J. M. Wing. 1996. “Formal Methods: State of the Art and Future Directions.” ACM Computing Surveys, 28.4: 626-643.

Fetzer, J. H. 1998. “Philosophy and Computer Science: Reflections on the Program Verification Debate.” In The Digital Phoenix: How Computers are Changing Philosophy, T. W. Bynum and J. H. Moor, ed. Pp. 253-273. Oxford: Blackwell, 1998.

Hall, Anthony. 1990. “Seven Myths of Formal Methods.” IEEE Software, 7.5: 11-19. In Bowen & Hinchey 1999, Pp. 153-167.

Holloway, C. M. 1997. “Why Engineers Should Consider Formal Methods.” Unpublished paper. NASA Langley Research Center, NASA. Pp. 1-7.

Hoare, C. A. 1987. “An Overview of Some Formal Methods for Program Design.” IEEE Computer, 10.9: 85-91. In Bowen & Hinchey 1999, Pp. 210-216.

Pressman, R. S. 2001 “Formal Methods.” Software Engineering, Pp. 673-698. New York: McGraw.

Rushby, John. 1993. “Formal Methods and the Certification of Critical Systems.” Unpublished paper. Computer Science Laboratory, SRI International. Pp. 1-312.

Rushby, John. 1999. “Integrated Formal Verification: Using Model Checking With Automated Abstraction, Invariant Generation, and Theorem Proving.” Theoretical and Practical Aspects of SPIN Model Checking: 5th and 6th International SPIN Workshops, ed. D. Dams, et al. Pp. 1-11. London: Springer, 1999.

Wing, J. M. 1990. “A Specifier’s Introduction to Formal Methods.” IEEE Computer, 23.9: 8-24. In Bowen & Hinchey 1999. Pp. 167-199.