Invited paper presented at *FM'99: The World Congress on Formal Methods*, Toulouse, France, September 1999. Jeannette Wing and Jim Woodcock eds., Springer Verlag Lecture Notes in Computer Science volume 1708, pp. 48–51.

Mechanized Formal Methods: Where Next? *

John Rushby

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025, USA
rushby@csl.sri.com

1 Where Are We Now?

An author who elects to use the phrase "Where Next?" in a title clearly has predictive or prescriptive intent. In my case it is the latter: I cannot say how formal methods will develop in the next few years, still less whether or how they will be adopted in practice, but I do have views on how they *should* develop and what ought to encourage their adoption in practice, and this World Congress seems an excellent opportunity to impose these views on the public.

To offer a prescription for the future requires the presumption to assess the present and my bias here is indicated in the first part of my title: "mechanized formal methods." While mathematically-based methods can provide several benefits and have elucidated the basic concepts of our field—what it means for a design to be correct with respect to a specification, or for one design to refine another, or for a proof to establish one or another of these relationships—the value, to my mind, of specifically *formal* methods is that they allow some properties of a software or hardware system to be deduced from its design or specification by a process of *mechanized logical calculation*, in much the same way that mechanized numerical calculation is used to predict the properties of traditional engineering artifacts such as bridges and turbine blades. I find engineering calculations a more illuminating analogy for the role of formal methods in computer science than formalized mathematics or logic as practiced by mathematicians and logicians. In particular, the engineering perspective highlights the necessity of efficient, automated, calculation: the stresses in a bridge are calculated by computer, not by pencil and paper, and the calculations are used during design exploration, not just for verification.

So, I consider the most useful technical developments in formal methods to have been those that provide efficient mechanized calculation in logical domains: static analysis, model checking, decision procedures, and so on. And I consider the most useful formal methodologies to be those that help cast analysis of significant properties of a design (or of certain types of design) into forms that allow effective use of these mechanized logical calculations: abstract interpretation,

* Based on research supported by DARPA through USAF Rome Laboratory Contract F30602-96-C-0204, and by the National Science Foundation contract CCR-9509931.

extended static checking, well-formedness checks for tabular specifications, and a host of domain-specific methods for pipelines and other hardware structures, for railroad switching, for translation validation, and for protocols and certain other classes of concurrent systems.[1]

These tools and methods have demonstrated a certain effectiveness, and have been voluntarily adopted by industry to a modest degree (most notably for hardware and communications protocols).

2 Where Should We Go Next?

My prescription for further development of mechanized formal methods is a continuation of the successful trends identified above. I break it into several related headings.

2.1 Mechanize Everything

Some current formal methods, for example those that use model checkers or SAT solvers, are described as "automatic," meaning that they perform their logical calculations without needing interactive guidance (unlike, for example, a tactic-based theorem prover). However, this is a very narrow interpretation of automation, since considerable human guidance and ingenuity is generally required to transform the original problem to a form where the final "push button" automation can be applied. The transformation often requires: translating the description of the problem from a programming or design notation such as C, Java, Verilog, or UML into the input format of the tool concerned; extracting the part of the overall description that is relevant to the problem or property concerned; and reducing the extracted description to a form (e.g., finite-state) consistent with the capabilities of the tool concerned.

Much current research aims to mechanize the manual steps in this process. For example, methods based on backward slicing can yield just the part of a large system specification that is relevant to a given property, and combinations of abstract interpretation and partial evaluation can create a conservative finite-state (or other special-form) approximation from a given program or specification. Techniques employing decision procedures can calculate more precise property-preserving abstractions. These methods work better if provided with invariants of the program or specification concerned; such invariants can be calculated by static analysis or by reachability analysis (a capability of most symbolic model checkers) applied to another abstraction (the reachable states characterize the strongest invariant of that abstraction; their concretization is an invariant—and plausibly a strong one—of the original). This last suggests iterated application of these techniques (one abstraction is used to calculate an invariant that helps calculate another abstraction, and so on), which in turn suggests a blackboard

[1] I omit references due to lack of space; they are provided in a version available at `http://www.csl.sri.com/~rushby/fm99.html`.

architecture in which a common intermediate representation is used to accumulate useful properties and abstractions of a design in a form that many different tools can both use and contribute to. A common intermediate representation also provides a way to support many source notations without inordinate cost.

The first of my prescriptions is to continue and extend this line of development so that the human verification engineer will direct and select a sequence of mechanized calculations, rather than manually create input for a single tool. A radical element of this proposal is that it shifts the focus of formal analysis from verification (or refutation) to the calculation of *properties* that can assist further analyses: for example, a static analyzer will be used to calculate (rather than verify) the range of values that a variable can assume, and a model checker will be used to calculate an invariant rather than verify a property.

2.2 Mechanize More Powerfully

Traditional techniques for mechanized formal methods were based on interactive theorem proving. As the role of automated deduction in this activity came to be recognized as more closely resembling calculation in engineering than proof in mathematics, so more powerful automation such as decision procedures came to be accepted. But the overall pattern of activity was (and is) still that of an interactively-guided proof, where the main activity is one of generating and reducing subgoals to the point where they can be discharged by decision procedures or other automation. In contrast, techniques that combine automated abstraction, invariant generation, and model checking in the manner outlined above, even though they may be undertaken in the context provided by a theorem prover, have a rather different character: the steps are bigger, and fewer, and are aimed at quickly reducing the problem to a single decidable subgoal. The effectiveness of this approach depends on having techniques for "reduction" (e.g., abstraction and invariant generation) and a decidable class of formulas (e.g., μ-calculus formulas over finite transition systems) that are well-suited to the class of problems at hand.

Different classes of problems need different underlying capabilities in order to use this "big step" approach. For example, translation validation and certain kinds of pipeline correctness arguments need massively efficient decision procedures for the combination of propositional calculus with equality over uninterpreted function symbols (whereas a traditional theorem prover will generally provide a less efficient decision procedure for a richer combination of theories).

The second of my prescriptions is to develop and exploit new and highly efficient decision procedures or other algorithmic techniques for pragmatically useful combinations of theories. These need to be tens of thousands of times faster, and to scale better, than the techniques employed in current verification systems: the efficiency is to be gained by more precise targeting of the theories to be decided, thereby allowing better data structures and better algorithms. As suitable theories are identified and a customer base develops, an industry may emerge to supply the necessary components, much as BDD packages and SAT solvers are becoming commodities today.

2.3 Make Mechanized Formal Analysis Ubiquitous

Tools with the capabilities described in the last two sections can serve wider purposes than classical verification or refutation. For example, they can be used to support testing: either by generating test cases (e.g., using counterexamples generated by a model checker for the negation of the property of interest), pruning test cases (e.g., do not use tests that are provably satisfied by a conservative approximation of the design), or by serving as a test oracle. Another example is static analysis, where decision procedures can be used to increase the strength of typechecking, or to improve the precision of slicing. And another example is use of these techniques to examine security properties of mobile code (e.g., "proof carrying code").

Thus, my third prescription is to use mechanized logical calculation in all aspects of software and hardware development: I suspect that almost every tool used in development could be improved if it made some use of mechanized formal analysis. We should aim to make our techniques ubiquitous throughout software and hardware engineering; this particularly means that we should attempt to serve all stages of the development lifecycle—from requirements analysis through maintenance—and all classes of products (not just those deemed "critical").

2.4 Adapt to Engineering Practice

Successful transition from research to practice is unpredictable, and especially so where methodology, rather than technology, is concerned. While it is possible that formally-based methods will become widespread (and the possibility will surely be greater if they are supported by tools that offer compelling advantages over those of other methods), a more plausible scenario uses tools made possible by mechanized formal analysis to *add value* to existing (or modestly enhanced) practices: this may mean finding more bugs, or higher-value bugs, more quickly than otherwise, increasing the quality of test-cases, or verifying the satisfaction of certain properties. These accomplishments are rather far from the proofs of correctness that were among the original goals of formal methods, and from the view that programming and hardware design should become mathematical activities, but they are consistent with the way mathematics and calculation are used in other engineering disciplines.

I suggest we envisage and work towards a scenario where designers (or specialist verification or test engineers) equipped with an eclectic knowledge of formal methods and familiarity with several tools will turn to these methods and tools, much as they currently turn to traditional tools such as test coverage analyzers, when the need or opportunity arises. Only by adapting to existing "design flows" and by coexisting with traditional tools and methods will our methods have a chance to be assimilated into the normal state of practice. By developing the more powerful, flexible, and integrated mechanization suggested earlier, I believe that formal methods of analysis could increase their value added and reduce the cost incurred so significantly that they would then have a chance to become a dominant part of that normal practice.