Mechanizing Formal Methods: Opportunities and Challenges[*]

Reprint of an invited paper presented at the 9th International Conference of Z Users (ZUM '95, Springer Verlag LNCS 967, pp. 105–113), Limerick, Ireland, September 1995

John Rushby

Computer Science Laboratory, SRI International,
Menlo Park, CA 94025, USA

Abstract

Mechanization makes it feasible to calculate properties of formally specified systems. This ability creates new opportunities for using formal methods as an exploratory tool in system design. Achieving enough efficiency to make this practical raises challenging problems in automated deduction. These challenges can be met only by approaches that integrate consideration of its mechanization into the design of a specification language.

1 Introduction

All formal methods rest on the conviction that requirements and designs for computer systems and software can be modeled mathematically, and that many questions concerning properties of those requirements and designs can then be settled by calculation. Advocates of formal methods differ, however, in the extent to which they stress the conceptual and methodological aspects of these methods, as opposed to their calculational aspects.

While appreciating the methodological benefits of mathematical concepts and notations, I believe that the distinctive merit of specifically *formal* methods is that they support calculation. Using automated deduction (i.e., theorem proving) and related techniques (e.g., model checking) it is possible to calculate whether a formally-described design satisfies its specification or possesses certain properties. To be useful, it must be possible to perform these calculations for specifications and properties of practical interest with reasonable ease and efficiency.

It takes a lot of theorem-proving power to do this, and mechanized specification languages must be designed to mesh well with the most effective theorem-proving techniques. For example, reasoning about equality in the presence of uninterpreted

[*]This work was partially supported by the Air Force Office of Scientific research under contract F49620-95-C0044.

function symbols is crucial to most applications of mechanized formal methods, and efficient techniques for achieving this (such as congruence closure [14]) require that functions are total. However, it is draconian and rather unnatural to force all functions (including, for example, division) to be total, so an effectively mechanized formal method requires careful and integrated design choices to be made for both the specification language and its supporting mechanization. In this particular case, it is necessary to find some reasonably attractive treatment for partial functions that does not compromise the efficiency of equality reasoning.

While developing such a treatment poses a challenge, the hypothesized availability of a powerful theorem prover creates new opportunities: for example, the typechecker of our specification language could use the theorem prover to check that partial functions are never applied outside their domains. This balancing of challenges and opportunities, and the corresponding need for integrated design decisions, arises again and again when mechanizing formal methods. In the following sections, I will briefly describe the main opportunities created by mechanized formal methods, and the technical challenges in achieving effective mechanization. My perspective on these topics is influenced by experiences in the development and use of PVS [9].

2 Opportunities for Making Effective Use of Mechanized Formal Methods

It is often assumed that the main goal of mechanized formal methods is "proving correctness" of programs or detailed hardware designs, and this assumption may be reinforced by the term "verification system" that is commonly used to describe mechanized tools for formal methods. In fact, however, this assumption is wrong on both counts: most advocates of mechanized formal methods consider such early-lifecycle products as requirements, architectural designs, and algorithms to be more attractive targets for their tools than finished programs or gate layouts, and their focus is at least as much on finding faults and on design exploration as it is on verifying correctness.

Preference for early-lifecycle applications of mechanized formal methods is partly a consequence of the strength of traditional methods for late-lifecycle activities. Traditional methods for the development and quality assurance of program code and detailed hardware designs are sufficiently effective that very few significant faults are introduced and remain undetected at these late stages of the lifecycle (for example, of 197 critical faults found during integration testing of two JPL spacecraft, only three were programming mistakes [8]), and only a small fraction of overall development costs (typically, less than 10%) are incurred here. In contrast, mechanized formal methods are quite expensive to apply at these stages. Primarily, this is because the products of the late lifecycle are usually large—typically, hundreds of thousands, or even millions, of gates or lines of code—and their sheer size makes formal verification costly.

But if traditional methods are effective in the later stages of the lifecycle, the same cannot be said of the earlier stages. Natural-language, diagrams, pseudocode, and other conventional ways for describing requirements and preliminary designs do not support calculation, so the main way to deduce their properties and consequences is through the fallible processes of inspection and review. The fallibility of these processes is illustrated by the JPL data cited earlier: since only three of the 197 critical faults were due to programming errors, it follows that the other 194 were introduced at earlier stages. Lutz reports that 50% of these faults were due to flawed requirements (mainly omissions) for individual components, 25% were due to flawed designs for these components, and the remaining 25% were due to flawed interfaces between components and incorrect interactions among them [8].

Rapid prototyping and simulation provide more repeatable and systematic examination of these issues, but often force premature consideration of implementation questions and thereby divert attention from the most important topics. Mechanized formal methods, on the other hand, can support direct analysis and exploration of the products of the early lifecycle: as soon as we have written down a few logical formulas that describe some aspect of the system, so we can begin to check their consequences—such as whether they entail some expected property, or are mutually contradictory.

The purposes for which mechanized formal methods may be used in the early lifecycle are as much those of validation and exploration as verification: the opportunities are to validate requirements specifications, to explore different system architectures and the interactions of their components, to debug critical algorithms and to understand their properties and assumptions, and to cope with changes.

Mechanized formal methods can assist requirements validation by checking whether a formal statement of requirements entails other expected properties: intuitions such as "if I've got that right, then this ought to follow," can be examined in a formal manner using theorem proving or model checking. While confirmation of an expected property is gratifying, a more common outcome—at least in the early stages—is the discovery that the requirements must be revised, or that some new assumption must be adopted. The rigor of mechanized analysis renders the discovery of such oversights far more systematic than is the case for informal reviews. Furthermore, mechanization allows us to check, rapidly and reliably, that previously examined properties remain true following each revision to the requirements.

The assurance derived by checking that expected properties are entailed by a requirements specification may be specious if the requirements are inconsistent (i.e., mutually contradictory): every property is entailed by an inconsistent specification. Consistency of formal specifications can be demonstrated by exhibiting a model; an equivalent demonstration can be checked mechanically using theory interpretations: the basic idea is to establish a translation from the types and constants of the "source" specification (the one to be shown consistent) to those of a "target" specification and to show that the axioms of the source specification, when translated into the terms of the target specification, become provable theorems of that target specification. If this can be done, then we have demonstrated *relative* consistency:

the source specification is consistent if the target specification is. Generally, the target specification is one that is specified definitionally, or one for which we have some other good reason to believe in its consistency.

Following requirements validation, mechanized formal methods can be used to explore candidate system architectures. Architectures consist of interacting components; the concerns at this stage are generally to verify that the properties assumed of the components and of their interaction are sufficient to ensure satisfaction of the overall requirements. The chief benefits of applying mechanically checked formal verification to this task are the ability to explore alternative designs and assumptions, and to prune unnecessary assumptions. For example, formal examination of an architecture for fault masking and transient recovery in flight control systems reveals the need for interactive consistency on sensor inputs [11]. This can be achieved by a Byzantine Agreement algorithm [5]. Inputs to the majority vote function must also satisfy interactive consistency and it may therefore appear as if these, too, need to be run through a Byzantine Agreement algorithm. In fact, this is not required: it is possible to prove that interactive consistency of voter inputs is an inherent property of the architecture. Mechanized formal analysis allows this attribute of the architecture to be determined with certainty, and it also allows determination of the exact circumstances under which a modified architecture provides recovery from transient faults [11].

As with requirements validation, exploration of architectural design choices is an iterative process that is greatly facilitated by the rigor and repeatability of mechanized formal methods. Simulation and direct execution share the repeatability of formal methods but can examine only a few cases, whereas deductive formal methods allow consideration of *all* cases. But, unfortunately, the price of this generality can be less automated and more difficult analysis: we may need to invent invariants and to undertake proofs by induction in order to cover an infinite state space. Often, however, an effective alternative is available: we may be able to abstract or "downscale" a specification to a finite state space that can be examined, exhaustively and automatically, by model checking or by explicit state exploration. The great automation of formal finite-state methods creates new opportunities for applying formal methods in the design loop. Experience indicates that examining *all* the cases of such an abstracted system description is generally more effective at finding faults than testing or simulating *some* of the cases of the full description [3].

Once a preferred system architecture has been selected we may, recursively, explore architectures for its components or—once a sufficiently detailed level has been reached—investigate algorithms for those components. As with the earlier stages, the great benefits of using mechanized formal methods to examine algorithms are the abilities to explore alternatives, to prune assumptions, and to adapt to design changes. For example, the journal presentation of the interactive convergence clock synchronization algorithm [4] has an assumption that all initial clock adjustments are zero. Friedrich von Henke and I retained this assumption when we formally verified the algorithm [13]. Subsequently, when contemplating design of a circuit to implement part of the algorithm, it became clear that this assumption is exceedingly inconvenient. I explored the conjecture that it is unnecessary by simply striking it out of our formal specification and rerunning the proofs of all the lemmas and theorems that constitute the verification. (There are about 200 proofs in the full verification and it takes about 10 minutes for the theorem prover to check them all.) It turned out that the proofs of a few lemmas failed without the assumption, but examination showed that those lemmas could easily be restated, or given different proofs. A few hours of work were sufficient to make these adjustments to the formal specification and mechanically checked verification.

In other revisions to this algorithm and its verification, I have tightened the bound on the achieved clock skew, and extended the fault model so that the algorithm tolerates larger numbers of simple faults, without compromising its ability to resist arbitrary (i.e., Byzantine) faults [12]. In each case, the effort required to investigate the proposed revision and to rework the formal specification and verification was on the order of a day or two. In another example, Lincoln and I developed the formal specification and verification of a Byzantine Agreement algorithm for an asymmetric architecture in less than a day by modifying an existing treatment for a symmetric architecture [7].

The ability to make these enhancements to complex algorithms, rapidly and reliably, is an opportunity created by mechanized formal methods. Informal methods of proof are unreliable in these domains (see [6, 9, 13] for examples) and it requires superhuman discipline to bring the same level of care and skepticism to the scrutiny of a modified algorithm as to the original. A formal specification and verification, on the other hand, is a reusable intellectual resource: its properties can be calculated, and those calculations can be mechanized.

In summary, the opportunities created by mechanized formal methods are similar to those offered by mechanized calculation in other fields, such as computational fluid dynamics: exploration of design alternatives, early detection of design errors, identification of assumptions, the ability to analyze the consequences of changes and responses to them, and the acquisition of an enhanced understanding that can lead to further improvements. In the next section, I will briefly examine some of the challenges in developing mechanizations of formal methods that can make these opportunities reality.

3 Technical Challenges in Mechanizing Formal Methods

The primary challenge to achieving the benefits described above is the difficulty of mechanizing formal deduction in a way that is both efficient and enlightening. Since automated deduction—i.e., theorem proving—is a fairly well-developed field, applying this technology to formal methods might seem to be simply a matter of engineering. In fact, most of the challenges are those of engineering, but they are not simple. Theorem proving in support of formal methods raises issues that are quite different from those that have traditionally been of interest in the theorem-proving community. Most notably, candidate theorems generated by formal methods in their exploratory and debugging roles are very likely to be false. In these cases, mechanization is expected to quickly reveal the falsehood, and to help identify its causes. The traditional concern of theorem proving, however, has been to prove true theorems, and most off-the-shelf theorem provers are therefore ill-suited to the needs of formal methods.

But although an existing theorem prover is unlikely to be useful in support of formal methods, we must not ignore the component techniques developed for automated theorem proving. One of the principal reasons that many attempts at mechanizing formal methods have failed is that their developers did not appreciate the raw power and speed that is needed from their theorem-proving components, and did not make use of the relevant techniques. The most important of these techniques are decision procedures for specialized, but ubiquitous, theories such as arithmetic, equality, function updates (i.e., overriding), and propositional calculus. Decision procedures are helpful in discovering false theorems (especially if they can be extended to provide counterexamples) as well as in proving true ones, and their automation dramatically improves the efficiency of proof.

There are decision procedures for many useful theories, but few problems fall precisely in the domain of any single one of them, so one of the big engineering challenges in mechanizing formal methods is to develop effective combinations of decision procedures. This requires very careful selection of the individual procedures. For example, the decision procedure for Presburger arithmetic (i.e., the first order theory of linear arithmetic with relation symbols such as $<$) does not consider uninterpreted function symbols. Since function symbols are pervasive in formal specifications, a better choice than true Presburger arithmetic is the theory of ground (i.e., unquantified) linear arithmetic, which can be combined with the theory of equality over uninterpreted function symbols [15].

Small extensions to decidable theories can have considerable value. For example, it is not possible to add full nonlinear multiplication to the decision procedures for linear arithmetic, but it is possible to add the ability to reason about the signs of products (e.g., "a minus times a minus is a plus") and this proves to be a significant benefit in practice.

Rewriting is another technique that is essential to efficient mechanization of formal methods. Unrestricted rewriting provides a decision procedure for theories axiomatized by terminating and confluent sets of rewrite rules, but few such theories arise in practice. Consequently, rewriting cannot be unrestricted, in general, but must be performed under some control strategy. For example, one of the control strategies used in PVS will rewrite a definition whose body involves a top-level *if-then-else* only if the condition to the *if* can be reduced to *true* or *false*. This reduction may involve use of decision procedures and further rewriting, and it is possible to expend considerable resources on a search that is ultimately unsuccessful (because it does not succeed in reducing the condition). These resources will not have been wasted, however, if they allow the theorem prover to avoid making an unprofitable rewrite: in most contexts, the heuristic effectiveness of a good control strategy is likely to be more beneficial than the raw speed of a blind rewriter. As this description suggests, rewriting and decision procedures cannot stand apart: truly effective theorem provers must integrate them very tightly. A classic account of the issues in such integration is given by Boyer and Moore [1].

Integration is a pervasive theme in the effective mechanization of formal methods: many individual techniques work well on selected examples, but fail in more realistic contexts because problems seldom fall exactly within the scope of one method. Sometimes the integration must be tight, as in cooperating decision procedures, and the integration of decision procedures with rewriting. In other cases, the integration can be less tight; a good example is model checking within a theorem-proving context. Whereas theorem proving attempts to show that a formula follows from given premises, model checking attempts to show that a given system description is a model for the formula. As noted in the previous section, an advantage of model checking is that, for certain finite state systems and temporal logic formulas, it is much more automatic and efficient than theorem proving. The additional benefits of a system that provides both theorem proving and model checking are that model checking can be used to discharge some cases of a larger proof or, dually, that theorem proving can be used to justify the reduction to finite state that is required for automated model checking [10].

Model checking can provide a further benefit: before undertaking a potentially difficult and costly proof, we may be able to use model checking to examine some restricted or special cases. Any errors that can be discovered and eliminated in this way will save time and effort in theorem proving. This is a particular case of a more general desideratum: mechanized formal methods should provide a graduated collection of tools and techniques that apply increasingly strict scrutiny at correspondingly increasing cost. Representative techniques include typechecking, animation or direct execution, model checking, and theorem proving. These techniques become more effective if they are integrated so that each can use capabilities of the others. For example, typechecking can be made more strict if it is allowed to use theorem proving, rather than being restricted to trivially decidable properties; certain specifications can be executed on test cases by using the theorem prover to perform rewriting (in this case, fast "blind" rewriting may be desirable); and theorem proving can be more efficient if it makes use of type information provided by the typechecker.

Achieving the necessary integrations involves more than careful engineering of theorem proving and support tools: it extends to the design of the specification language itself. As noted in the introduction, efficient equality reasoning, for example, requires that functions are total. If we allow the concerns of mechanization to dominate language design, we may then decide that our specification language should provide only total functions. Similar considerations may lead us to restrict quantification to first-order (or to eliminate explicit quantification altogether), to restrict recursive definitions to the syntactic form of primitive recursion, or to require all formulas to be equations. In the limit, we may provide a raw logic devoid of the features expected of a specification language. Conversely, concerns for an expressive notation may lead us to provide a specification language that cannot be mechanized effectively. This is not to say that expressiveness cannot be combined with mechanization, but that expressiveness must not be considered in isolation from mechanization. For this reason, I consider it dangerous to look to the classical foundations of mathematics for guidance when designing a specification language. These formal systems (notably, first-order logic with axiomatic set theory) were created in order to be studied, not in order to be used—the "...interest in formalized languages being less often in their actual and practical use as languages than in the general theory of such use and its possibilities in principle" [2, page 47]. Unsurprisingly, therefore, set theory has characteristics that pose difficulty for mechanization—for example, as already noted, functions are inherently partial in set theory (they are sets of pairs). Also, it is difficult to provide really strict typechecking (and hence, early error detection) for set theory without sacrificing some of its flexibility: for example, a function is a set, so it can (sometimes) make sense to form its union with another set, and it is therefore not clearcut whether type restrictions should prohibit or allow this sort of construction.

If mechanization is a goal, then design of the specification language and selection of its underlying logic should not be undertaken independently of consideration of its mechanization. This does not mean that concern for mechanization must inevitibly restrict or impoverish a notation—rather, I believe that availability of powerful mechanization creates new opportunities for the design of expressive, yet mechanically tractable, notations. An example is provided by the treatment of partial functions such as division in PVS, whose specification language supports the notion of *predicate subtypes*. These allow the nonzero real numbers to be defined as follows.

```
nonzero_real: TYPE = { r: real | r ≠ 0 }
```

We can then give the signature of division as

```
/: [real, nonzero_real → real]
```

and the function is total on this precisely specified domain. We can then state and prove the following result.

```
inverse_sum: LEMMA
   ∀ (a, b: real):
      a ≠ 0 ∧ b ≠ 0 ⊃ (1/a + 1/b) = (a + b)/(a × b)
```

PVS allows a value of a supertype to appear where one of a subtype is required, provided the value can be proved, in its context, to satisfy the defining predicate of the subtype concerned. In this case, PVS will generate the following three proof obligations, called Type Correctness Conditions (TCCs), that must be discharged before inverse_sum is considered fully type-correct. The three TCCs correspond to the three appearances of division in the formula inverse_sum and collectively ensure that the value of the formula does not require division by zero. The first two TCCs can be discharged automatically by a decision procedure for linear arithmetic; the third requires extensions mentioned earlier that reason about the signs of nonlinear products.

tcc1: OBLIGATION $\forall$ (a, b: real): $a \neq 0 \wedge b \neq 0 \supset a \neq 0$

tcc2: OBLIGATION $\forall$ (a, b: real): $a \neq 0 \wedge b \neq 0 \supset b \neq 0$

tcc3: OBLIGATION $\forall$ (a, b: real): $a \neq 0 \wedge b \neq 0 \supset (a \times b) \neq 0$

Predicate subtypes in PVS provide many more capabilities than are suggested by this simple example: in particular, injections and surjections are defined as predicate subtypes of the functions, and state-machine invariants can be enforced by the same mechanism. The source of these conveniences and benefits is allowing typechecking to require theorem proving (i.e., to become algorithmcally undecidable), which is only feasible if a powerful and automated theorem prover is assumed to be available. Conversely, the power of the theorem prover is enhanced by the precision of the type information that is provided by the language and its typechecker.

4 Conclusion

Mechanization creates new opportunities for formal methods: by making it feasible to calculate properties of formally specified designs, mechanization allows exploration of alternative designs, examination of assumptions, adaptation to changed requirements, and verification of desired properties. These opportunities are likely to have maximum benefit when applied early in the development lifecycle, and to the hardest and most important problems of design.

To realize these benefits, mechanizations of formal methods must provide several capabilities ranging from very strict typechecking, to powerfully automated theorem proving. These individual capabilities need to be closely integrated with each other, and with the specification language. Because of the integration required, consideration of its mechanization must be factored into the design of a specification language.

References

[1] R. S. Boyer and J S. Moore. Integrating decision procedures into heuristic theorem provers: A case study with linear arithmetic. In *Machine Intelligence*, volume 11. Oxford University Press, 1986.

[2] Alonzo Church. *Introduction to Mathematical Logic*, volume 1 of *Princeton Mathematical Series*. Princeton University Press, Princeton, NJ, 1956. Volume 2 never appeared.

[3] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, 1992. Cambridge, MA, October 11-14.

[4] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[5] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[6] Patrick Lincoln and John Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Fault Tolerant Computing Symposium 23*, pages 402–411, Toulouse, France, June 1993. IEEE Computer Society.

[7] Patrick Lincoln and John Rushby. Formal verification of an interactive consistency algorithm for the Draper FTP architecture under a hybrid fault model. In *COMPASS '94 (Proceedings of the Ninth Annual Conference on Computer Assurance)*, pages 107–120, Gaithersburg, MD, June 1994. IEEE Washington Section.

[8] Robyn R. Lutz. Analyzing software requirements errors in safety-critical embedded systems. In *IEEE International Symposium on Requirements Engineering*, pages 126–133, San Diego, CA, January 1993.

[9] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[10] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995. Springer-Verlag.

[11] John Rushby. A fault-masking and transient-recovery model for digital flight-control systems. In Jan Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Kluwer International Series in Engineering and Computer Science, chapter 5, pages 109–136. Kluwer, Boston, Dordecht, London, 1993. An earlier version appeared in [16, pp. 237–257].

[12] John Rushby. A formally verified algorithm for clock synchronization under a hybrid fault model. In *Thirteenth ACM Symposium on Principles of Distributed Computing*, pages 304–313, Los Angeles, CA, August 1994. Association for Computing Machinery.

[13] John Rushby and Friedrich von Henke. Formal verification of algorithms for critical systems. *IEEE Transactions on Software Engineering*, 19(1):13–23, January 1993.

[14] Robert E. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(7):583–585, July 1978.

[15] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[16] J. Vytopil, editor. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, Nijmegen, The Netherlands, January 1992. Springer-Verlag.