Invited paper presented at the 16th Conference on the Foundations of Software Technology and Theoretical Computer Science, Hyderabad, India, December 1996. Springer-Verlag Lecture Notes in Computer Science, volume 1180, pages 43–51.

Mechanized Formal Methods: Progress and Prospects*

John Rushby

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

Abstract. In the decade of the 1990s, formal methods have progressed from an academic curiosity at best, and a target of ridicule at worst, to a point where the leading manufacturer of microprocessors has indicated that its next design will be formally verified. In this short paper, I sketch a plausible history of the developments that led to this transformation, present a snapshot of the current state of the practice, and indicate some promising directions for the future. Mindful of the title of this conference, I suggest how formal methods might have an impact on software similar to that which they have had on hardware.

1 The Past

In their early days (the 1970s—though continuing to the present in some places), formal methods were associated with proofs of program correctness. This is not only a very costly and difficult exercise—it requires formalizing the semantics of real programming languages, and dealing with the scale and characteristics of real imperative programs—but it also adds very little value: traditional methods of code review and testing are highly effective and very few coding bugs of any significance escape detection. For example, of 197 critical faults detected during integration and system testing of the Voyager and Galileo spacecraft, just 3 were coding errors [18]. The large majority of faults arise in requirements, interfaces, and intrinsically difficult design problems (e.g., fault tolerance, and the coordination of concurrent activities). In the spacecraft data just cited, approximately 50% of faults were traced to requirements (mainly omissions), and 25% to each of interfaces and design.

During the 1980s, attention shifted from program correctness to the use of formalism in specifications, exemplified by approaches such as Z [32] and VDM [17]. Although these methods initially stressed the role of proof in development, they came to be used mainly as specification languages, and their advocates commended the utility of mathematical concepts such as sets, functions, and relations in constructing precise yet abstract descriptions of computational systems. The problem with this approach is that it is not necessary to be specifically *formal* to

* This work was supported by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under contract F49620-95-C0044 and by the National Science Foundation under contract CCR-9509931.

make use of such mathematical modeling techniques; conversely, in the absence of formal proof, there are few tangible benefits to a strictly formal approach. By failing to exploit the singular characteristic of truly formal methods—namely, their ability to support deduction—specification-oriented formalisms missed the opportunity to combine mathematical modeling with calculation in the manner that has been so productive in other engineering disciplines.

The value of formal deduction is that it enables many questions about properties of formally specified requirements and designs to be settled by a systematic process that has the character of calculation. The reasons for favoring calculation over informal reasoning or trial and error experimentation are the same in computer science as in other engineering disciplines: calculation allows the properties of designs to be predicted and evaluated prior to construction, it allows analyses to be checked by others, enables large problems to be tackled in a systematic manner, and opens the door to mechanization. And in most engineering disciplines, it is mechanization that releases the full potential of mathematical modeling and calculation: the highly efficient wings of a modern airplane could not be designed without massive mechanization of computational fluid dynamics, finite element analysis, and several other branches of applied mathematics.

It was the arrival of efficient techniques for model checking in the early 1990s [19] (and related methods such as language inclusion) that first made large-scale mechanized calculations a practical reality for formal methods and demonstrated their utility to a wide audience. No less important than the techniques that made model checking practical was the change in approach and outlook that its use engendered. The limited expressiveness of the temporal logics employed in model checking means that it is seldom possible to use them to fully characterize the functionality required of a system; instead, attention is focussed on important properties that it should posses. Similarly, because model checking methods can only explore a limited, finite state space, the full system description must generally be considerably abstracted and simplified before subjecting it to model checking. Partly because of these limitations (and partly because it is able to provide excellent diagnostic information in the form of counterexamples), model checking has generally focused on *in*correctness—on finding bugs—rather than on trying to establish correctness. And find bugs it did: because model checking is well-suited to concurrent systems, it was immediately applied to some of the hardest problems in system design, such as multiprocessor cache-coherence protocols, where “high-value bugs” were quickly detected [8].

The changes in approach introduced by model checking opened up new opportunities for all formal methods: whereas previously the goal had been to specify the full functionality required, there was now seen to be a useful spectrum of desired properties; whereas previously the goal had been to describe the system in all its details, there was now seen to be value in isolating key problems and aggressively abstracting away as many details as possible; whereas previously the goal had been to establish unequivocal correctness, there was now seen to be a variety of other useful purposes that could be served by formal analysis; and whereas previously the applications had generally been to routine designs (see, for example, the survey [9]), there was now an enthusiasm for applying formal methods to the hardest and most difficult problems of design.

Mechanized formal methods based on theorem proving, which had become modestly effective by the mid 1980s and were continually improving, benefited from the change in attitude—and the spur of competition—that came with model checking. Decision procedures for basic theories such as linear arithmetic and equality received renewed attention and acceptance, and integrated combinations of decision procedures, rewriting, and customized tactics achieved significant automation and efficiency on interesting classes of problems [22]. Most importantly, the practitioners of these approaches to formal methods followed the lead of the model checkers in applying them to complex, real-world systems [3, 35].

2 The Present

An idea of the current capabilities and accomplishments of mechanized formal methods can be obtained by considering two examples from hardware design.

The Pentium FDIV bug, which attracted a great deal of public interest, also caught the attention of the formal verification community—not least because it caused Intel to take a $475 million charge against revenues. The bug was in the lookup table of an SRT divider [25]. Binary Decision Diagrams (BDDs) have been used successfully to verify many kinds of digital circuits—but not multipliers and dividers, where they grow exponentially large [4]. Nonetheless, Bryant was able to verify a single iteration of an SRT circuit using BDDs [5]. Explosive growth of the BDD representation has generally also precluded application of symbolic model checking to dividers; however, by using a different "word level" representation, Clarke, Khaira, and Zhao were able to apply model checking to this problem [7]. Clarke, German, and Zhao were also able to verify an SRT divider using a special-purpose theorem prover based on the Mathematica symbolic algebra system [6]. Using the PVS general-purpose verification system [21], Rueß, Shankar, and Srivas gave a formally verified treatment of the general theory of SRT division, and then verified a particular circuit and lookup table [27]. While being more general, the theorem proving treatments achieved a level of automation and efficiency comparable to the BDD and model checking approaches, and were equally adept at catching errors in the tables. However, all of these treatments dealt only with the fixed-point core of the divider, and not with the issues of IEEE-compliant floating point representation. Miner and Leathrum extended the PVS treatment to include IEEE-compliance, generalized the whole development to encompass the broader class of subtractive division algorithms that includes SRT, and presented a methodology that enabled specific algorithms to be debugged and verified quite easily—which they demonstrated on various SRT tables [20].

Cache coherence protocols for distributed shared memory multiprocessors are notoriously difficult to design. Some of the early successes with symbolic model checking were in its application to this type of problem. As interest shifted from the "snoopy" to the more scalable—and much more complicated—"directorybased" protocols, the state-explosion problem became quite severe. One response was to "downscale" (aggressively simplify) the problem, so that, for example, only two or three processors, one address, and a 1-bit data word are considered. Another was to use the various symmetries that exist in the problem to allow different, but equivalent, states to be merged. Using these and other techniques, model checkers based on both explicit state-enumeration and symbolic representations are able to tackle cache-coherence problems sufficiently well to be used in the design process for these systems [2, 11]. But although they are effective for detecting bugs, the severely downscaled models used in model checking cannot serve to verify the general case. Theorem proving techniques should be able to do this, but the difficulty of creating appropriate abstractions and sufficiently strong invariants, combined with the labor involved in guiding the theorem prover, had discouraged their application to realistic cache-coherence protocols. Recently, however, by using a method called "aggregation" to guide construction of the abstraction function, Park and Dill have been able, using PVS in a quite straightforward manner, to verify the behavior of the protocol used in the Stanford FLASH processor [24]. Furthermore, using the Murϕ explicit state-enumeration system they were able to construct an executable model for the non-sequentially-consistent memory behavior of the processor. In similar work for the Sparc V9 memory model, they were able to verify the behavior of synchronization code using Murϕ, and were able to verify the executable Murϕ model against its axiomatic specification using PVS [23].

The interesting feature of these examples is the diversity of approaches employed—and the diversity would be even greater if I had considered other examples such as pipelines, microcode, communications and switching protocols, or hybrid systems. There simply is no single best method: we are dealing with problems that are at the limit of what is computationally feasible, and different applications yield to different approaches. Thus, symbolic model checking using BDDs works well for some problems, but explicit state enumeration is better for others; some state spaces can be reduced significantly by symmetry reductions, others require partial-order reductions; some problems are best dealt with by model checking, others are better suited to theorem proving.

Just as different approaches work better for different problems, so different approaches work better for the *same* problem at different stages of its "verification lifecycle." When first encountered, a design (or its formalization) will often be full of bugs. These should be identified as quickly and as cheaply as possible. Methods that require relatively little preparation, such as typechecking, animation, or explicit state enumeration are effective here. Once the simple bugs have been eliminated, it becomes necessary to explore more and more of the state space to discover those that remain, and explicit state exploration methods that use hashing, and symbolic model checking methods, start to become more effective. Once the complete state space of downscaled instances of the problem can be explored without finding a bug, then the aggressiveness of the simplifications can be reduced, and the size of the problem instances can be increased. The "state explosion" problem is likely to hit at this point, and reduction methods based on symmetry, partial orders, or abstraction may need to be invoked. When the largest problem instances that can be examined by finite state methods no longer reveal bugs, then it is time to consider theorem proving. For concurrent systems, it is generally necessary to develop abstractions and to strengthen the desired invariant to obtain one that is inductive. Special-purpose tools can help with these activities, and finite-state methods can be invoked during the proof process to check that proposed invariants really are so, and that subgoals are true (on finite instances) [12].

Different methods come into play on a single problem as easy bugs are eliminated and those that remain become harder to find; in a related progression, different methods come into play in the treatment of *classes* of problems as our understanding and techniques improve. For example, as enumerated above, treatments of SRT division evolved from BDD-based analysis of individual iterations, to treatment of the core of a specific algorithm by special-purpose theorem proving, to general treatment of the entire class of algorithms with a general-purpose theorem prover.

3 The Future

I offer some suggestions on likely, or promising, directions for future developments in mechanized formal methods under two headings: applications to software, and tools.

3.1 Applications to Software

Compared to hardware, software is more of a challenge for successful application of mechanized formal methods. Hardware has a relatively small number of stereotypical problems (pipeline control, floating point ALUs, microcode, cache coherence), so that the cost of developing really effective solutions can be recouped over many applications, whereas software has a vastly larger supply of problems and a correspondingly smaller community of interest for any one of them. Nonetheless, we can adopt some of the strategies that seem to have been successful for hardware.

– *Go where the bugs are.* Formal methods have been effective for hardware because their use has been targeted at areas where they can offer a real payoff: areas that experience has shown to be error-prone and where other methods are ineffective. The targeted areas concern some of the *hardest* challenges in design (e.g., the stereotypical problems mentioned above). For software, correspondingly difficult and worthwhile challenges include those where local design decisions have complex global consequences, such as the fault-tolerance and real-time properties of concurrent distributed systems, and those where independently designed systems interact, such as the problems of feature interactions, protocol stacks, and component interfaces. It is generally most productive to examine these issues at the level of the algorithms concerned, rather than at the detailed design or coding level. It also helps to target applications where the costs of bugs are unacceptably high. These include applications that share with hardware the characteristic that design errors cannot be repaired in the field (e.g., embedded systems in consumer products), and those where failure is intolerable (e.g., safety and other kinds of critical systems).

- *Target the early lifecycle.* The requirements for hardware (especially processors) are quite simple (i.e., "implement a given instruction set architecture"), whereas those for software are generally complex (e.g., "control air traffic") and subject to change. The most damaging and costly errors discovered late in the software development lifecycle can usually be traced back to faulty requirements. Consequently, requirements validation consumes considerable resources (in avionics, for example, more than half the development costs can go into requirements; programming, in contrast, consumes less than 10%). Formal methods are singularly well-adapted to the specification and analysis of requirements, because they allow precision without premature detail (unlike pseudocode and prototyping), and they allow useful analyses to be performed on very abstract or incomplete descriptions [29].
- *Use powerful tools, and a spectrum of methods.* Without tools, formal methods are just documentation; it is tools that make formal methods useful, and powerful tools that make them productive. Many of the tools that have been effective in applications of formal methods to hardware can also be used for software (see, for example, [33], where the SMV model checker is applied to a software requirements specification); alternatively, *ideas* from those tools can be incorporated into new tools that are specifically tailored to the characteristics of software [16]. Even less than for hardware, no single tool or method provides universally effective support for all the diverse applications of formal methods to software, so a spectrum of tools and methods should be employed.

3.2 Tool Building

As noted several times already, most applications of mechanized formal methods require a range of capabilities and make use of a number of tools. Rather than loose integration of a number of different tools, however, what is really required is tight integration of a number of different capabilities [28, 31]. For example, loose integration of a theorem prover and a model checker might allow one to use a single specification of a problem and to examine specific instances with the model checker and to prove the general case with the theorem prover, whereas tight integration might allow the theorem prover actively to *use* the model checker—so that the theorem prover could set up the induction to prove the general case, with the base case and inductive step then being discharged by model checking [30]. Such an integration of theorem proving and model checking has been achieved [26], but it required extending the implementation of a complex verification system. Future systems should be designed in a much more "open" manner, so that components can be added, modified, interconnected, and accessed in a modular fashion. For example, an attractive application of formal reasoning to software requirements is to check consistency and completeness of the conditions that label the rows and columns of tabular specifications [15]. Depending on the logic and theories used in specifying these conditions, the deductive capabilities needed to perform the checks range from propositional tautology checking, though decision procedures for ground linear arithmetic, to full interactive theorem proving. When tautology checking proved inadequate for an example derived from the TCAS II specification [13], Czerny and Heimdahl turned to the PVS verification system in order to make use of its decision procedures. However, because those decision procedures could not be accessed separately, they had to invoke the entire PVS system, which entailed more baggage and less performance than they desired [14]. What is really needed is an open environment that provides access to components such as decision procedures and the other building blocks of theorem provers and model checkers, and in which customized combinations can be quickly constructed. The hub of such an environment must be a theorem prover, since that is what has the capability to check that problems are decomposed appropriately, that constraints on the application of certain procedures are satisfied, and that all the pieces come together to solve the whole problem in a sound manner [10]. In collaboration with David Dill of Stanford University, we are about to begin construction of such an environment.

4 Conclusion

These are exciting times for mechanized formal methods, with opportunities for rapid and significant progress in the capabilities of tools and the quality and scale of their applications. Theoretical research can assist these developments by, for example, providing better characterizations for the complexities of the various problems and algorithms encountered (almost every problem in model checking and theorem proving is NP-hard or worse), and by identifying useful special cases that admit fast solutions.

References

Papers by SRI authors are generally available from `http://www.csl.sri.com/fm.html`.

[1] Rajeev Alur and Thomas A. Henzinger, editors. *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[2] Ásgeir Th. Eiríksson and Ken L. McMillan. Using formal verification/analysis methods on the critical path in system design: A case study. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 367–380, Liege, Belgium, June 1995. Springer-Verlag.

[3] Bishop Brock, Matt Kaufmann, and J Strother Moore. ACL2 theorems about commercial microprocessors. In Srivas and Camilleri [34], pages 275–293.

[4] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[5] Randal E. Bryant. Bit-level analysis of an SRT divider circuit. In *Proceedings of the 33rd Design Automation Conference*, pages 661–665, Las Vegas, NV, June 1996.

[6] E. M. Clarke, S. M. German, and X. Zhao. Verifying the SRT division algorithm using theorem proving techniques. In Alur and Henzinger [1], pages 111–122.

[7] E. M. Clarke, Manpreet Khaira, and Xudong Zhao. Word level symbolic model checking—avoiding the Pentium FDIV error. In *Proceedings of the 33rd Design Automation Conference*, pages 645–648, Las Veqas, NV, June 1996.

[8] Edmund M. Clarke, Orna Grumberg, Hiromi Haraishi, Somesh Jha, David E. Long, Kenneth L. McMillan, and Linda A. Ness. Verification of the Futurebus+ cache coherence protocol. *Formal Methods in System Design*, 6(2):217–232, March 1995.

[9] Dan Craigen, Susan Gerhart, and Ted Ralston. Formal methods reality check: Industrial usage. *IEEE Transactions on Software Engineering*, 21(2):90–98, February 1995.

[10] David A. Cyrluk and Mandayam K. Srivas. Theorem proving: Not an esoteric diversion, but the unifying framework for industrial verification. In *International Conference on Computer Design: VLSI in Computers and Processors (ICCD '95)*, pages 538–544, Austin, TX, October 1995. IEEE Computer Society.

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

[12] Klaus Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe FME '96*, volume 1051 of *Lecture Notes in Computer Science*, pages 662–681, Oxford, UK, March 1996. Springer-Verlag.

[13] Mats P. E. Heimdahl. Experiences and lessons from the analysis of TCAS II. In Steven J. Zeil, editor, *International Symposium on Software Testing and Analysis (ISSTA)*, pages 79–83, San Diego, CA, January 1996. Association for Computing Machinery.

[14] Mats P. E. Heimdahl and Barbara J. Czerny. Using PVS to analyze hierarchical state-based requirements for completeness and consistency. In *IEEE High-Assurance Systems Engineering Workshop (HASE '96)*, Niagara on the Lake, Canada, October 1996.

[15] Mats P. E. Heimdahl and Nancy G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.

[16] Daniel Jackson and Craig A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7):484–495, July 1996.

[17] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International Series in Computer Science, Hemel Hempstead, UK, 1990.

[18] 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.

[19] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1993.

[20] Paul S. Miner and James F. Leathrum, Jr. Verification of IEEE compliant subtractive division algorithms. In Srivas and Camilleri [34], pages 64–78.

[21] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Alur and Henzinger [1], pages 411–414.

[22] 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.

[23] Seungjoon Park and David L. Dill. An executable specification, analyzer and verifier for RMO (Relaxed Memory Order). In *7th ACM Symposium on Parallel Algorithms and Architectures*, pages 34–51, July 1995.

[24] Seungjoon Park and David L. Dill. Verification of the FLASH cache coherence protocol by aggregation of distributed transactions. In *8th ACM Symposium on Parallel Algorithms and Architectures*, pages 288–296, Padua, Italy, June 1996.

[25] Vaughan Pratt. Anatomy of the Pentium bug. In *TAPSOFT '95: Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 97–107, Aarhus, Denmark, May 1995. Springer-Verlag.

[26] 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.

[27] H. Rueß, N. Shankar, and M. K. Srivas. Modular verification of SRT division. In Alur and Henzinger [1], pages 123–134.

[28] John Rushby. Automated deduction and formal methods. In Alur and Henzinger [1], pages 169–183.

[29] John Rushby. Calculating with requirements. In *3rd IEEE International Symposium on Requirements Engineering*, Annapolis, MD, January 1997. IEEE Computer Society. To appear.

[30] N. Shankar. Computer-aided computing. In Bernhard Möller, editor, *Mathematics of Program Construction '95*, volume 947 of *Lecture Notes in Computer Science*, pages 50–66. Springer-Verlag, 1995.

[31] Natarajan Shankar. Unifying verification paradigms. In Bengt Jonsson and Joachim Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1135 of *Lecture Notes in Computer Science*, pages 22–39, Uppsala, Sweden, September 1996. Springer-Verlag.

[32] J. M. Spivey, editor. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, Hemel Hempstead, UK, 1993.

[33] Tirumale Sreemani and Joanne M. Atlee. Feasibility of model checking software requirements. In *COMPASS '96 (Proceedings of the Eleventh Annual Conference on Computer Assurance)*, pages 77–88, Gaithersburg, MD, June 1996. IEEE Washington Section.

[34] Mandayam Srivas and Albert Camilleri, editors. *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, Palo Alto, CA, November 1996. Springer-Verlag.

[35] Mandayam K. Srivas and Steven P. Miller. Formal verification of the AAMP5 microprocessor. In Michael G. Hinchey and Jonathan P. Bowen, editors, *Applications of Formal Methods*, Prentice Hall International Series in Computer Science, chapter 7, pages 125–180. Prentice Hall, Hemel Hempstead, UK, 1995.

This article was processed using the LaTeX macro package with LLNCS style