Model Checking and Other Ways of Automating Formal Methods*

Position paper for panel on Model Checking for Concurrent Programs
Software Quality Week, San Francisco, May/June 1995

John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

1 Automated Formal Methods

Formal methods can benefit system design in two different ways.

- Concepts and notations from mathematics can provide methodological assistance: they can help us think, and can help us communicate our ideas.
- Mathematical modeling can allow us to *calculate* some of the properties of our designs.

My interest is chiefly in the second of these and, more particularly, in automation of the calculations concerned.[1] Automation provides repeatability and accuracy and—potentially, at least—speed and economy. The motivation and benefits here are similar to those provided by mathematical modeling and automated calculation in other engineering fields: computational fluid dynamics, for example, allows the aerodynamic properties of airplane designs to be thoroughly explored and analyzed prior to construction.

There are many kinds of computer systems, and many different properties of those systems that are of interest. There are correspondingly many ways to model computer systems and to calculate their properties. At one end of the spectrum of different methods, we can build a simulation or rapid prototype of the system and

*This work was partially sponsored by NASA Langley under Contract NAS1-20334, and by Arpa through NASA Ames under contract NASA-NAG-2-891.

[1] I generally use PVS for this purpose. PVS is a verification (i.e., theorem proving) system mainly developed by my colleagues Sam Owre and Shankar [13].

test its behavior on selected inputs; at the other end of the spectrum, we can conduct a mechanically-checked proof that a formally-specified design satisfies its formally-specified requirements. In between these extremes come various (mostly finite-state) methods, including model checking. Although I just indicated that these methods form a spectrum, they actually differ from each other in several dimensions, though these do tend to be correlated with each other. I discuss three of the more important dimensions in the following sections.

1.1 Number of behaviors examined

The main reason computer programming and system design is hard is the sheer complexity of behavior that can be achieved by these means. Individual components can do complicated things, but the real explosion in complexity occurs when components interact with each other or with their environment. In order to understand our designs, and to predict their properties, we need some way to comprehend all the different behaviors that they can exhibit.

For well-structured sequential programs of modest size, this is not too hard to achieve. By selecting suitable test data, it is possible to directly "calculate" through execution a sample of the possible behaviors of the program. If this sample is fairly representative (e.g., satisfying some combination of functional and structural test criteria), we may feel some confidence in extrapolating to the complete set of behaviors.

This extrapolation from a finite number of tests to the (possible infinite) set of all possible behaviors cannot be guaranteed sound in general, but we are fairly comfortable with it for sequential programs because we understand the basis by which the tested behaviors are considered representative of the whole set (e.g., both directions have been examined at each branch in the control flow and each loop has been executed zero, once, and many times). With concurrent or reactive systems, however, it is all but impossible to achieve any degree of comfort in extrapolation from a finite number of tests to the complete set of behaviors. This is because it is hard to define, and even harder to conduct, tests that are representative of all the possible interactions of the components concerned.

Classical program verification methods based on theorem proving do allow us to consider all possible behaviors of such systems in a finite but complete manner, however. For sequential programs, we attach assertions at various points in the control graph (in particular, we identify an "invariant" assertion for each loop) and for each section of program connecting two assertions, we prove that if the first assertion is true when control is at that point, then the program will ensure the that second is true when control reaches that point. One way of generalizing this approach to concurrent programs is to establish "rely" and "guarantee" assertions for each component and to prove that each component will hold its guarantee assertion invariant if the other components do the same for the assertions that it relies on.

The principal difficulties with these program verification methods are inventing suitable invariants, and conducting the proofs without error. The latter can be ensured, to some extent, by using a mechanized proof checker or theorem prover. There are also mechanized techniques that can help in the development of invariants, but even with such assistance it can be a very tedious task to develop these, and the resulting formulas can be very complicated.

So far, we have considered testing and simulating a sample of the behaviors that a system may exhibit, and using verification methods to prove properties about all its behaviors. The first of these is fairly straightforward and automatic, but the extrapolation to all behaviors is questionable; the second is sound and complete, but challenging to undertake. We would like to have a method that falls between these two.

The number of essentially different behaviors that a program can exhibit is related to its state space, that is, to the number of different values that can be assumed by its state variables (including the program counters), considered as a totality. If the state space were finite, and relatively small, it might be possible to systematically enumerate all the possible different behaviors of the program. If two such finite-state programs interact, the size of the state space of the combined system will be the product of the sizes of the individual state spaces: the number of different behaviors of the combined system will be much larger than those of its components, but it may still be feasible to enumerate all of them.

Unfortunately, the state space of a typical program is vastly too great for such brute-force enumerations to be feasible, and the state space of a formal specification can be infinite (since it can have true, mathematical, integers as state variables). However, it is often possible to "downscale" the state space of a program or specification to a fairly small finite size in such a way that the reduced program exhibits essentially all the behaviors of the original. For example, a sorting program may be designed to handle an arbitrary number of arbitrary sized records, but we may lose little if we downscale the number of records to, say, five, each only two bits long. The number of different inputs to the downscaled sorting program is 4^5 and it is feasible to test them all. Obviously, great care must be taken in the downscaling if we are not to lose essential properties of the original program: for example, if the sorting program is a quicksort that reverts to linear insertion when the number of records is less than five, then we must be sure to consider more than five records. In some cases, it is possible to perform the downscaling in such a way that the reduced program is a true abstraction of the original (relative to some property); this means that if the downscaled program can be shown to satisfy the property concerned, then the original will satisfy it also. More often, it will be unsound to make this extrapolation, but any bugs found in the downscaled system are likely to indicate a bug in the original. Experience suggests that enumerating *all* the behaviors of a downscaled system is a much more potent debugging method than exploring merely *some* of the behaviors of the original system.

There are several ways to organize the enumeration of behaviors for finite-state systems and to check their properties. The example of a sorting program suggested above is atypical: a sequential program whose behavior is determined by its input, and which can therefore be checked by presenting it with all possible inputs. More usually, these methods are applied to concurrent or reactive systems; the behaviors of such systems are explored by interleaving execution of each of their components. The interleaving order is chosen nondeterministically in each run (if a component can select from any of a number of different steps—for example, it may perform its ordinary next step, or fail in one of a number of different ways—then these are chosen nondeterministically also). By backtracking over many runs, all the different nondeterministic choices—and hence all behaviors—can be explored.

An explicit state-exploration system of this kind (examples are Murϕ [12]—pronounce "Murphy"—and Spin [8]) can typically explore some millions of states in a few minutes.[2] The properties to be checked are generally specified as explicit assertions or error-checks programmed into the component specifications; this approach is generally known as "reachability analysis" since it aims to explore all the reachable states and to check that they satisfy their assertions. An alternative is to provide a separate assertion language that can characterize required properties of the set of behaviors generated by the system. Typically, the assertion language is a temporal logic for which the finite-state system description is putatively a Kripke model—this approach is therefore known as "model checking" [3]. Temporal logic specifications can seldom express the full correctness requirement of a program, but they can often express important safety and liveness properties. Related methods are based on language inclusion: implementation and specification are described by automata and we show that the language (of behaviors) accepted by the implementation is a subset of that accepted by the specification [5].

The requirement to severely downscale a system description to make it suitable for finite-state exploration, coupled with limitations on the properties that can be checked in this way, mean that finite-state methods are generally most suitable for examining questions related to control, rather than data, complexity. A valuable characteristic of these methods is that they can often construct a counterexample when a design is found not to satisfy a specified property.

[2]Using this type of system, the sorting program would be explored by composing it with an input generator program that nondeterministically generates an input for it to sort. By exploring all the execution sequences of the generator program, the state-exploration system will cause it to generate all possible inputs within the finite space concerned.

1.2 Concreteness of description

Testing, in the usual sense, can generally be applied only to specifications that are directly executable. Such specifications contain a lot of concrete detail and are close to being programs. Yet much of the benefit from formal methods comes from examining early lifecycle products such as requirements, architectures, abstract specifications of designs, and algorithms. One way of testing these products is to develop "rapid prototypes" that are directly executable—but this can be costly and it is not easy to distinguish those properties that truly belong to the requirement or specification under considerations from those that are accidental to the prototype.

Using theorem proving techniques, however, it is often possible to "test" early lifecycle products symbolically by challenging them with putative theorems. For example, conventional tests can only examine a specific sorting program (or at best an algorithm), whereas we may be interested in whether our requirements specification for the sorting function is correct. If these requirements are expressed formally, we can challenge them by attempting to prove theorems such as `sort(sort(x)) = sort(x)`. This challenge would reveal an inadequacy of many early formulations of sorting (they required the output to be ordered, but omitted to state that it should be a permutation of the input), and could also lead us to consider "stability" of sorting (i.e., whether the sort can reorder records that are equal with respect to the sort criterion, but distinguishable in other ways) and to ask whether this is important for our application.

Most finite-state enumeration and model-checking methods are comparable to direct execution in that they require concrete representations of the algorithms and data structures concerned. Because of the downscaling required to make these methods feasible, the data representations and manipulations used are vestigial and it is generally only the control aspects of algorithms that are fully developed—and this is reasonable, since it is what these methods are designed to examine. However, the need to provide even vestigial implementations of unimportant elements of the design is unattractive and a potential source of errors. For example, a standard demonstration of model checking concerns processor pipeline control [2]. The processor design is abstracted to a register file, an ALU, and a pipeline with, say, three stages. An external source generates the addresses of pairs of source data values to be extracted from the register file, an ALU operation is applied to the data values, and the result is written back to an externally generated destination address. Since it is possible for the address of a source value for one operation to be the destination address for an earlier operation that is still in the pipeline, bypass circuitry is provided to extract the appropriate data value from the pipeline rather than from the register file (to which it will not yet have been written back). The main interest in this exercise is the bypass circuitry: the exact operation performed by the ALU, the width of the data buses, and the size of the register file, are all irrelevant to this concern and would best be left uninterpreted (i.e., given no specific implementation). To apply model checking or other finite-state methods, however, we must fix values for the size of the data buses and register file, and must give an implementation of the ALU. If we are too aggressive in giving a vestigial implementation for the ALU (e.g., if it always returns zero, or the value of one of its input arguments) we will lose the ability to detect certain errors, so we are forced to provide an implementation that combines its input values in some fairly complicated way.

1.3 Degree of automation

Direct testing and model-checking methods are generally presented as fully automatic, whereas those based on theorem proving are generally considered labor-intensive. This characterization needs to be qualified, in my view. If, for example, we are interested in exploring a formal requirements specification, then we might be able to go straight to work with a theorem proving method, whereas direct execution and model checking will require us to develop the more concrete representations that those methods require.

The size of the state spaces that can be explored by model checking has been increased dramatically recently through the introduction of symbolic representations based on BDDs (SMV [11] is an example of a system that uses a BDD representation). These representations are not always superior to explicit state enumeration, however. For some large state spaces, an explicit enumeration method may be able to explore a large number of states before it runs out of memory, whereas a symbolic method may exhaust its memory before it can even construct the BDD. If some of the states visited by the explicit enumeration reveal errors (as is commonly the case early in the design cycle), then this will have provided useful information that the symbolic method did not. The symbolic method is likely to prove more effective in the long run, however, and it will be worth the effort needed to get it working. This may involve experimenting with the variable ordering used by the BDD, using advanced methods for symmetry reduction or reduction of interleavings using partial orderings, further downscaling of the specification, or partitioning the problem into pieces that can be attacked separately. These manipulations require skill and experience and cannot be considered automatic.

Theorem proving is not automatic either (even provers called "automatic" require indirect guidance through the invention and ordering of lemmas, the orientation of equations, and selection of strategies). However, it is often possible to provide strategies that are very effective for certain classes of problems. For example, a single PVS strategy is very effective against a wide variety of problems in hardware verification [4]. This strategy can deal with the pipeline example mentioned above in a few tens of seconds—furthermore, it does not require an implementation for the ALU operation (it can be left as an uninterpreted symbolic operation), nor specific sizes for the data buses and register file. In contrast, symbolic model checking requires concrete representations for all of these, and CPU time running into tens of minutes when the register file has more than about 16 registers.

2 Integrated Automation of Formal Methods

Automated calculation of properties of formal descriptions of computer systems can support validation of requirements (by checking that they entail some expected properties), early debugging of specifications and designs, identification of assumptions, exploration of design alternatives, verification that designs satisfy their specifications, and adaptation to changes.

Each of the methods mentioned in the previous section—executable (or simulatable) specifications, finite-state methods including model checking, and those based on theorem proving—as well as other related methods including fault-tree analysis, offers valuable benefits and practitioners of formal methods should generally be prepared to use all of them. In many applications where full verification through theorem proving is the ultimate goal, it will be sensible to first debug the design through direct execution and then through state exploration or model checking. In this way we can avoid the expense and frustration of applying theorem proving to designs that contain errors that can be found more easily by other methods. At SRI, we have explored a number of different combinations of methods that are outlined in the sections below.

2.1 Novel Uses of State Exploration

These examples in this section do not combine different methods, but illustrate novel uses of finite-state exploration for problems that had previously been examined by hand or by theorem proving.

2.1.1 Exploration of Fault-Diagnosis Algorithms

I described earlier how a sequential sorting algorithm could be exhaustively tested over some finite input space by combining it with a nondeterministic input generator and using a state-exploration system to enumerate all possible behaviors of the input generator. Of course, sorting algorithms have a very regular structure and behavior, so this kind of exploration is unlikely to be enlightening in their case. However, there are other kinds of sequential programs that have very complex behavior and where this kind of exploration can be very valuable. One such class of programs is concerned with fault diagnosis: the symptoms of failure of some device are provided to the program and it is expected to diagnose the faults that could have caused those symptoms. These programs are notoriously hard to test and tend to be "brittle" (they are often developed as "expert systems" and fail catastrophically when confronted with an unanticipated circumstance). Using a state-exploration system, it is possible to program a model of the device to be diagnosed and to non-deterministically inject faults; these will generate symptoms that can be supplied to the diagnostic program and its diagnosis can then be compared with the injected fault. The state-exploration system will systematically enumerate all faults within the class considered, thereby providing complete coverage of (this aspect of) the diagnosis program.

The language of Murϕ is rule-based and rather similar to those used for expert systems. We are therefore considering applying Murϕ to a diagnostic program for the NASA Manned Maneuvering Unit that we have previously studied by hand [15].[3] I believe that an approach similar to that described for diagnosis programs can also be used to explore decision support systems and the procedural rules provided to the operators of complex plants.

2.1.2 Exploration of Fault-Tolerant Algorithms

Whereas a fault-diagnosis algorithm is required to diagnose a fault in some external mechanism, a fault-tolerant algorithm is required to continue operation despite faults in the mechanisms of its own execution. Typically, the algorithm is replicated on different processors and some form of voting is used. *Interactive Consistency* (also known as *Byzantine Agreement* and *Distributed Consensus*) is the problem of distributing sensor values consistently to such replicated components despite the presence of faults. We have formally verified several interactive consistency algorithms under increasingly complex fault models using PVS [9,10,13]. As the algorithms and fault models get more complex, worst-case analyses of the fault tolerance achieved become rather uninformative—it seems more useful to compare algorithms according to the total number of different fault scenarios they can tolerate (there are some tens of thousands of scenarios for the cases we are interested in).

We again used Murϕ to perform the necessary exhaustive examination of scenarios. We considered the five processor case and modeled each algorithm as a five-process distributed system; a sixth process performs the role of the environment and nondeterministically injects faults. In running the exhaustive enumeration of all possible fault scenarios, we rediscovered a bug in one algorithm that we had previously found, with much greater effort, while attempting to formally verify it. We also found that one of the algorithms under consideration is able tolerate significantly more faults than the others in regions beyond the simple worst-case bounds.

[3] In fact, I suspect that the efficient state enumeration provided by Murϕ could be used to *do* model-based diagnosis in a way that would be competitive with other methods.

2.2 Combining Methods at the Language Level

When applying formal verification with PVS to distributed algorithms, and to protocols in particular, we have found it advantageous first to check a downscaled instance of the protocol using Murϕ; we also find it useful to check proposed invariants in this way. Constructing and maintaining both a PVS and a Murϕ specification of a protocol is tedious and error-prone, so we have developed a translator from Murϕ to PVS. The translator was built by Seungjoon Park of Stanford University who has used it for two applications.

- A distributed list protocol that is representative of the core of several directory-based cache coherence protocols was debugged using Murϕ for the three-processor case, and then translated into PVS and formally verified for the general n-processor.
- A Murϕ model of Sparc International's multiprocessor memory consistency model that is used to verify synchronization routines was proved consistent with the axiomatic definition of the memory model using PVS.

Shankar has also developed a translation from the internal representation used by ObjecTime (a Statechart-like CASE notation and system) to Murϕ. The translation (done by hand at present) supplements the simulation capability of ObjecTime with explicit state exploration.

2.3 Combining Theorem Proving With Model Checking

Using an off-the-shelf BDD-based decision procedure for the μ-calculus (i.e., quantified Boolean logic with least and greatest fixed-point operators), we have recently implemented CTL model checking in PVS [14]. That is to say, given a state-transition relation specified in PVS on a hereditarily finite type (that is a type that is Boolean, or an enumeration type, or a finite subrange of integers, or a tuple, array, or record of hereditarily finite type), a start state, and a CTL specification, the PVS rewriter and decision procedures expand the specification to yield a formula that can be translated into the form required by the μ-calculus decision procedure.[4] Availability of model checking within a theorem proving system creates new opportunities.

- We have worked with NRL to provide support within PVS for checking tabular specifications of the kind used in the SCR method for requirements specification [7]. For example, PVS provides a table construct, and can generate the proof obligations for mutually disjoint and exhaustive row and column conditions that are required for well-formedness of SCR mode transition tables [6].

[4]The μ-calculus is strictly more expressive than CTL, and can also be used to define fair versions of the CTL operators. Language inclusion methods can also be described in the μ-calculus and may be added to PVS at a later date.

Using the model checker we are now also able to check certain application properties of these tables automatically, with regular theorem proving available for the harder properties.[5]

- Shankar has verified an n-place mutual exclusion protocol by induction using the PVS theorem prover, but with the inductive step discharged automatically by the model checker. This is the first example we are aware of that uses combined methods in a truly integrated way.

3 Conclusion

Model checking and finite-state methods are essential components of any toolkit for automated formal methods. Such a toolkit should also provide a variety of other capabilities including simulation and theorem proving and, for maximum benefit and utility, the various different methods should be integrated with each other. Loose integration provides a common language (or at least the ability to translate between the languages used by the different components); tighter integration allows a combination of methods to be applied to a single problem and is probably most naturally constructed in a theorem proving environment where model checking can be treated as a decision procedure.[6]

I believe that the future lies with even tighter integrations of these different methods: rather than combining a theorem prover with a model checker, we need to integrate their component techniques. For example, theorem proving could apply some of the efficient symbolic representations developed in model checking, and model checking should be extended so that state-equivalence testing can use theorem proving and so that some aspects of a design can be left uninterpreted.

These are significant research challenges, but exciting progress is already being made. In the future, I expect the line between model checking and theorem proving to become blurred (e.g., it will be possible to apply model checking to certain infinite-state systems) and that combined methods will be the rule. I also expect these methods to become sufficiently effective that their industrial use will become routine.

References

[1] Joanne M. Atlee and John Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.

[5] Atlee and Gannon were the first to note that model checking could be applied to SCR mode transition tables [1].

[6] Theorem proving systems also already have the bookkeeping necessary to keep track of the outstanding obligations when a complex analysis is broken into many pieces.

[2] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design*, 13(4):401–424, April 1994.

[3] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[4] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In Ramayya Kumar and Thomas Kropf, editors, *Theorem Provers in Circuit Design (TPCD '94)*, volume 910 of *Lecture Notes in Computer Science*, pages 203–222, Bad Herrenalb, Germany, September 1994. Springer-Verlag.

[5] Zvi Har'El and Robert P. Kurshan. Software for analytical development of communications protocols. *AT&T Technical Journal*, 69(1):45–59, January/February 1990.

[6] Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. Scr*: A toolset for specifying and analyzing requirements. In *COMPASS '95 (Proceedings of the Ninth Annual Conference on Computer Assurance)*, Gaithersburg, MD, June 1995. IEEE Washington Section. To appear.

[7] K. L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering*, SE-6(1):2–13, January 1980.

[8] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[9] Patrick Lincoln and John Rushby. Formal verification of an algorithm for interactive consistency under a hybrid fault model. In Costas Courcoubetis, editor, *Computer-Aided Verification, CAV '93*, volume 697 of *Lecture Notes in Computer Science*, pages 292–304, Elounda, Greece, June/July 1993. Springer-Verlag.

[10] 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.

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

[12] Ralph Melton and David L. Dill. *Murϕ Annotated Reference Manual.* Computer Science Department, Stanford University, Stanford, CA, March 1993.

[13] 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.

[14] 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*, Lecture Notes in Computer Science, Liege, Belgium, June 1995. Springer-Verlag. To appear.

[15] John Rushby and Judith Crow. Evaluation of an expert system for fault detection, isolation, and recovery in the manned maneuvering unit. Contractor Report 187466, NASA Langley Research Center, Hampton, VA, December 1990. (Work performed by SRI International).