Invited paper from *Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE XIII/PSTV XX)*, pp. 369–374, October 2000, Pisa, Italy.

FROM REFUTATION TO VERIFICATION

John Rushby*
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
Rushby@csl.sri.com

Abstract Model checking has won some industrial acceptance in debugging designs. Theorem proving and formal verification are less popular. An approach built around automated abstractions could integrate theorem proving with model checking in an acceptable way and provide a bridge between refutation and verification.

Keywords: Model checking, theorem proving, formal models, assurance, debugging.

1. INTRODUCTION

Formal methods have achieved a modest degree of acceptance in some industries (e.g., communications protocols and processor design), but mainly for "refutation"—that is, for purposes associated with debugging and testing. Formal verification—that is, establishing some some notion of correctness—is much less widely practiced, and is largely restricted to regulated industries.

In this short note, I examine some reasons for the relative success of formal methods in refutation, and their relative failure in verification, and propose that formal verification methods should adopt some of the characteristics of those used for refutation. I further propose that refutation and verification should be seen as points on a continuum and that by adopting techniques that allow moving smoothly along this continuum we may be able to increase both the utility and acceptance of formal methods.

2. FORMAL MODELING AND ANALYSIS

The rôle of formal methods in computer science is analogous to that of mathematical modeling and calculation in traditional engineering disciplines. In

*This work was partially supported by DARPA through USAF Rome Laboratory contract F30602-96-C-0204 and by NASA Langley Research Center contract NAS1-20334.

those disciplines, engineers build mathematical models of some aspects of their design (e.g., the shape of an airfoil) and of its environment (e.g., the properties of fluids in motion) and use calculation to predict the behavior of the design (e.g., its lift and drag). These calculations are used in the design loop, and also to verify the behavior and properties of the final design. The utility of such verification is contingent on how faithfully the finished artifact implements its design, on the faithfulness of the environment models employed, and on the accuracy of the calculations. Validating the faithfulness of the models and the accuracy of the calculations are separate problems from verification, though usually some "end-to-end" checks are performed by testing the final artifact.

Formal methods in computer science follow a similar pattern, except that formal logic provides the mathematical underpinnings for computer science, as opposed to the calculus and differential equations that underpin most traditional engineering disciplines. Hence, the calculations performed in formal methods are based on automated deduction: theorem proving, model checking, and related techniques. These techniques, and the problems they address, are computationally far more demanding than those for differential equations: many problems in automated deduction are undecidable, and those that are decidable have awesome computational complexity. This computational intractability presents practitioners of formal methods with difficult choices: analysis of highly detailed and accurate formal models generally requires the power of human-guided theorem proving, while fully automated procedures such as model checking generally require the formal model to be simplified and approximated to bring it within their capabilities (they may also restrict the analysis to relatively simple properties such as those that can be expressed in a temporal logic).

The difficulty with approximate models is that they may not be faithful the the real system; hence verification performed using such models is of dubious validity. Refutation is also vulnerable to approximation in modeling—a reported bug may be a consequence of over-aggressive approximation and may not be present in the real design—but it is usually straightforward to distinguish whether a bug is real or not: we simply transform the scenario that manifests the bug in the model into a scenario for the real system (or for a simulator, or for a more accurate mathematical model of the system) and test it. This is the basis for the industrial acceptance of model-checking: examining *all* the behaviors of an approximate model often reveals more bugs, or higher-value bugs (those that would otherwise be caught only much later in the lifecycle), than examining *some* of the behaviors of the real system (as in traditional testing or simulation). The model-checking process itself is largely automated, and putative bugs are furnished with a counterexample trace that can be used (often automatically) to construct a scenario for the real system that will determine whether the bug is present there also, or is merely a consequence of the approximate modeling employed.

The difficulty with faithful models is that their analysis is usually outside the scope of fully automated procedures, and must instead rely on heuristic or human-guided theorem proving. Heuristic theorem proving is fine when it succeeds; in the more usual case that it fails, however, its user must try to distinguish whether this is due to a flawed design (i.e., a bug) or to an inadequate heuristic. Theorem proving methods seldom provide a counterexample trace that might help in making this distinction and instead depend on the user having a strong understanding of the operation of the heuristic. If the failure is due to an inadequate heuristic, then strong understanding of its operation is again required to modify either the heuristic or the model in an effective manner.

Human guidance is an alternative to heuristic theorem proving. Human-guided theorem proving is usually directly interactive: the human directs the strategy of the proof, while automated proof procedures take care of the tactical details. Again, it requires strong understanding of the operation of the prover to guide it successfully, and to diagnose and recover from tactical failures.

I believe that the chief reason for the poor industrial acceptance of methods based on theorem proving is that considerable skill is required to operate a theorem prover, and that skill concerns the operation of the prover more than the properties of the model being analyzed. Hence, designers are often uninterested in acquiring the necessary skill: their interests focus on their designs, not on the vagaries of adjuncts such as theorem provers. In contrast, the interactions required to use a model checker chiefly concern construction of an approximate model that is adequately faithful to the original design, yet within the scope of the model checker. This process concerns the design, and is similar to a design activity, and is acceptable to practitioners.

The choices described above present a rather unattractive dilemma: use automated techniques with approximate models that are adequate for refutation but do not extend to verification; or use accurate models that can be used to verify strong properties, but whose analysis depends on theorem proving methods that are not well-accepted in industry. Fortunately, there is an escape through the horns of this dilemma, and this is described in the next section.

3. AUTOMATED ABSTRACTION

The approximations that render a model suitable for automated analysis such as model checking do not necessarily have to sacrifice the ability to establish correctness. The approximations that are typically performed are Draconian: large or unbounded data structures are chopped down to small finite sizes—a queue may be restricted to length three, or a datapath to two bits. But more subtle approximations may be equally effective in rendering the model tractable to automated analysis, while preserving certain correctness properties: for example, a queue could be modeled as being in one of three states—empty, full, or in-between. Use of such *abstractions* in model checking has been understood for some time [5, 10]. Typically, a number of verification conditions (putative theorems) must be proved to establish that the proposed abstraction is a conservative approximation to the original model and preserves the property of interest, and then model checking or some other automated procedure can be applied to the abstracted model. As usual, reported bugs must be checked to see if they are real or the consequence of too aggressive an approximation, but if analysis of the abstracted model reveals no bugs, then we have succeeded in verifying the property for the original model. The problem with this approach is that is is often almost as hard to prove the verification conditions as it would be to establish the property directly for the original model.

An alternative to post-hoc justification of an approximate model constructed by hand is to use automated methods to *construct* the approximate model. One method for doing this performs abstract interpretation over the data structures appearing in the model (e.g., [6]); another, generally more powerful, method replaces selected predicates in the model by Boolean variables (e.g., [3, 12]). Calculation of the abstracted model in the latter case requires checking a large number of verification conditions; compared to those for post-hoc justification of an approximate model constructed by hand, these verification conditions tend to be more numerous, but simpler. Heuristic theorem proving techniques combined with powerful decision procedures are often able to discharge most of these conditions automatically. Furthermore, it is not a catastrophe if theorem proving fails to discharge a true verification condition: the approximate model will be more conservative than necessary, but it may still be good enough for model checking to succeed. If it does not, then the counterexample can help refine the construction, or can suggest additional predicates on which to abstract [11].

4. ITERATED, INTEGRATED, ANALYSIS

Construction of accurate abstractions often requires knowledge of some invariants of the original model—and the combination of automated abstraction and model checking can sometimes be used in a recursive fashion to help develop these. The idea is to construct an even simpler abstraction than the one desired, and then use a model checker to calculate the reachable states of that abstraction (most model checkers can do this). The reachable states characterize the strongest invariant of that abstraction; their concretization (the inverse of abstraction) is an invariant—and plausibly a strong one—of the original model. An invariant discovered in this way can often be strengthened further by a few iterations of a strongest postcondition computation (performed by theorem proving).

This approach 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 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. Verification systems that explore this approach include SAL [2], which is under development at SRI, and InVeSt [4], which is a collaboration between Verimag and SRI (InVeSt is built on PVS and is being integrated with SAL). Other experiments using related methods have been reported recently [1, 7].

The great benefit of this approach is that human guidance to the process of iterated abstraction and model checking is conducted in terms of properties related to the design: the user suggests predicates to be abstracted, examines counterexample traces or derived invariants, and suggests new predicates. Theorem proving is a central element in the mechanization, but is fully automated and conducted behind the scenes in circumstances that can tolerate the occasional failure to prove a true theorem.

Furthermore, by providing additional methods of abstraction that are *not* property-preserving (such as those that simply chop data structures down to a small size), this architecture and approach can be used for refutation as well as for verification. By increasing the sophistication of the abstractions employed, and by accumulating invariants, we can proceed smoothly from fairly crude approximate models that may be adequate for finding bugs early in the design loop, to more faithful models that reveal deeper bugs, and eventually to property-preserving abstractions that are able to verify aspects of the final design.

Even in environments where refutation is all that is required, this approach can increase the power and convenience of the refutation process by automating the construction of a graduated series of approximate models. In regulated environments, or others where true verification is required, this approach may make it possible to involve the design engineers more fully in the process (rather than having a separate formal verification group that operates largely outside the design loop) and may also bring to these environments the cost-benefits of employing formal methods for refutation as well as for verification.

References

[1] Parosh Aziz Abdulla, Aurore Annichini, Saddek Bensalem, Ahmed Bouajjani, Peter Habermehl, and Yassine Lakhnech. Verification of

infinite-state systems by combining abstraction and reachability analysis. In Halbwachs and Peled [8], pages 146–159.

[2] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, NASA Langley Research Center, Hampton, VA, June 2000. Available at `http://shemesh.larc.nasa.gov/fm/Lfm2000/Proc/`.

[3] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In Hu and Vardi [9], pages 319–331.

[4] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. InVeSt: A tool for the verification of invariants. In Hu and Vardi [9], pages 505–510.

[5] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[6] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering*, pages 439–448, IEEE Computer Society, Limerick, Ireland, June 2000.

[7] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In Halbwachs and Peled [8], pages 160–171.

[8] Nicolas Halbwachs and Doron Peled, editors. *Computer-Aided Verification, CAV '99*, Volume 1633 of Springer-Verlag *Lecture Notes in Computer Science*, Trento, Italy, July 1999.

[9] Alan J. Hu and Moshe Y. Vardi, editors. *Computer-Aided Verification, CAV '98*, Volume 1427 of Springer-Verlag *Lecture Notes in Computer Science*, Vancouver, Canada, June 1998.

[10] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:11–44, 1995.

[11] Hassen Saïdi. Model checking guided abstraction and analysi. In *Seventh International Static Analysis Symposium (SAS'00)*, Santa Barbara CA, June 2000. To appear.

[12] Hassen Saïdi and N. Shankar. Abstract and model check while you prove. In Halbwachs and Peled [8], pages 443–454.