Invited paper presented at the second IEEE International Conference on Formal Engineering Methods (ICFEM '98), pp. 176–178
Brisbane, Australia, December, 1998.

Ubiquitous Abstraction: A New Approach for Mechanized Formal Verification*
Extended Abstract

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

Formal methods can provide many benefits but, to my mind, the chief benefit of specifically *formal* methods is that they allow some properties of a computational system to be deduced from its design by a process of *logical calculation*, in much the same way that computational fluid dynamics allow properties of aerofoils to be examined by numerical calculation.

Originally, "computational system" meant *computer program*, and the main property of interest was correctness of the program with respect to its specification. More recently, however, these notions have widened to include almost any level of system description (e.g., hardware, algorithms, software architecture, requirements), properties short of full "correctness" (e.g., various notions of internal and external consistency), and refutation (i.e., bug finding) as much as verification. Mechanized formal verification uses the techniques of automated deduction—that is theorem proving and model checking—to perform the "logical calculations" that enable such properties to be checked for such system descriptions. The most successful verification systems combine an interactive theorem prover with powerful automation such as decision procedures for equality and arithmetic, and rewriting: the user directs the overall process, while the automation takes care of the details.

With the aid of a modern verification system, routine formal analyses are, well, routine. By this I mean that if the property of interest follows fairly directly from the system description by reasoning in some previously formalized mathematical domains, then mechanized formal verification is unlikely to be more difficult or to take longer—and may be considerably easier, as well as less error-prone—than a comparably detailed informal examination. Much worthwhile analysis can be accomplished economically and reliably in this way (see, for example, [4], which describes analysis of tables and other requirements specifications for some recent Space Shuttle software), but there is much else that can be accomplished only with great difficulty and effort.

These more challenging problems often involve concurrency, as in protocols and distributed algorithms, and the difficulties are not so much in theorem proving as in ancillary tasks, such as the invention of suitably strong invariants, and diagnosing whether an intractable subgoal indicates an error in the design, an inadequate invariant, or a mistaken proof step. To establish that a concurrent system (typically specified as a transition relation) maintains a desired invariant (expressing some safety property, for example), the basic deductive method is to show that the invariant is implied by the initial system state(s), and that it is preserved by all transitions. Usually, the desired property is *not* preserved in this simple manner, and it is necessary to strengthen it with additional conjuncts to characterize the reachable states (since preservation is required only for states that can be reached from the initial states). These conjuncts can often be found—one at a time—by inspecting a failed proof, extracting a plausible conjunct, and repeating the process until the proof succeeds. In one well-known example, 57 iterations of this kind were required to verify a relatively simple communications protocol known as the "bounded retransmission protocol" [5].

Model checking is an attractive alternative to theorem proving in circumstances such as these. Model checking is largely automatic, but it is applicable only to finite state systems (and to some infinite state systems having special forms); consequently, most system descriptions must be "downscaled" (i.e., aggressively simplified) before they can be subjected to model checking. Unless there is a suitable abstraction (i.e., simulation) relationship between the original system description and the downscaled one, model checking may be unsound or incomplete with respect to the original system: that is, it may fail to detect an error (because it is not present in the downscaled system), or may falsely report errors (that are present in the downscaled sys-

*This work was supported by Darpa through USAF Rome Laboratory Contract No. F30602-96-C-0204, and by the National Science Foundation under contract CCR-9509931.

tem but not in the original). The latter is not much of a problem when refuation is the goal: model checkers generally produce a counterexample in the form of an execution trace that manifests the error in the abstracted system, and it is usually straightforward to check whether a corresponding trace leads to an error in the original system. This may be adequate for refutation, but for verification we need to know that the model checker's inability to find errors in the downscaled system implies satisfaction of the desired property by the original system. For this, it is necessary to establish a suitable abstraction relationship between the original and the downscaled system descriptions—and doing so by traditional means can be almost as hard as proving the property directly. For the example of the bounded retransmission protocol, justification of an abstraction for model checking required 45 of the 57 conjuncts used in the direct proof.

Recently, several researchers, including my colleagues at SRI, have been exploring ways to combine theorem proving, abstraction, model checking, and other techniques more aggressively than before in order to avoid some of the difficulties and costs described above.

One idea is to *calculate* an abstracted system description, so that it is correct by construction, rather than to justify a downscaled description constructed by hand. Given an abstraction function relating the original and abstracted state spaces, a verification condition can be generated for each pair of abstract states that specifies the conditions under which no transition is required between those states in the abstracted system description (the condition is that there is no transition between any pair of original states that map to those abstract states). If the verification condition can be proved (using automatic proof procedures), then the transition can be omitted from the abstracted system description; if not, then it is conservative to include the transition. For the bounded retransmission protocol, this approach is able to compute automatically an abstracted system description suitable for model checking [1]. More sophisticated treatments allow the desired invariant to be used in construction of the abstracted system, and can use information from a failed model check to refine the abstraction.

Calculation of abstracted system descriptions often requires, and is usually made easier, if known invariants can be supplied to the process. Some useful invariants can be calculated by static analysis [3], but others can be extracted from model checking. As part of its computation, a model checker will almost certainly calculate the set of reachable states of the system description presented to it. Now, the reachable states of a system characterize its strongest invariant, so a concretization of the reachable states of an abstracted system is certainly an invariant, and possibly a strong one, for the original system.[1] This suggests a new way to calculate invariants that may help in the construction of abstracted system descriptions: construct some simpler abstraction (one for which already known invariants are adequate for its construction), and use a concretization of its reachable states as a new invariant. A practical difficulty in this approach is that the reachable state set calculated by a model checker is not usually made available externally and, in any case, it is usually represented by a data structure (a BDD) that is not directly suitable for input to a theorem prover. This difficulty has been overcome in the current version of the SMV model checker, where a `print` function, implemented by Sergey Berezin, provides external access to the reachable states.

The techniques described so far allow calculation of invariants and of abstracted systems, but they require the user to supply suitable abstraction functions. Some guidance in doing this can be obtained by inspecting the predicates that appear in the original, concrete, system description (particularly those in the guards on transitions): if a predicate such as $x = y + 1 \wedge x \neq z$ appears in the concrete system description, then an abstraction can be constructed having a boolean state variable that records the truth or falsity of this predicate [8]. Even with the aid of heuristics such as this, however, it can still require great insight to design a tractable abstraction that preserves the property of interest.

An alternative approach does not seek to construct an abstraction that directly preserves the property of interest: instead, this approach uses theorem proving as its top-level technique, and employs abstraction and model checking to help discharge the subgoals that are generated [7]. The attraction here is that theorem proving will have performed some case analysis in generating the subgoals, so that they will be simpler than the original problem. Therefore the abstraction needed to help discharge a given subgoal can be much simpler than one that discharges the whole problem; furthermore the predicates that appear in the formulas of the subgoal provide useful hints for the construction of a suitable abstraction.

In summary, "ubiquitous abstraction"—that is constructing many different abstracted system descriptions at many different points in an analysis, and for several different purposes—has great promise as a way to ease difficulties and increase productivity and automation in the formal analysis of concurrent systems. The approach also provides a new way to combine different tools, such as theorem provers and model checkers, though full exploitation of this opportunity requires modification to the tools so that they can exchange symbolic values (e.g., the reachable state set,

[1] "Concretization" is the inverse of abstraction; the inverse of the abstraction function is not a function, in general, so some approximation is required to find a set of concrete states who image under the abstraction function includes all the reachable abstract states.

or a counterexample) rather than merely report the success or failure of their own local analysis. Some of the capabilities I have described are already integrated in a system called InVeSt [2] and initial experiments with this and other prototypes developed as part of our "Symbolic Analysis Laboratory" (SAL) are quite promising. Our current plans are to evaluate the approach on more challenging examples.

Acknowledgements

The ideas outlined here, and the systems that implement them, are the work of my colleagues Saddek Bensalem, Sergey Berezin, Yassine Lakhnech, Sam Owre, Hassen Saïdi, and Natarajan Shankar.

References

Papers by SRI authors can generally be found at `http://www.csl.sri.com/fm.html`.

[1] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In Hu and Vardi [6], pages 319–331.

[2] S. Bensalem, Y. Lakhnech, and S. Owre. InVeSt: A tool for the verification of invariants. In Hu and Vardi [6], pages 505–510.

[3] S. Bensalem, Y. Lakhnech, and H. Saïdi. Powerful techniques for the automatic generation of invariants. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 323–335, New Brunswick, NJ, July/Aug. 1996. Springer-Verlag.

[4] J. Crow and B. L. Di Vito. Formalizing Space Shuttle software requirements: Four case studies. *ACM Transactions on Software Engineering and Methodology*, 7(3):296–332, July 1998.

[5] K. 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, Mar. 1996. Springer-Verlag.

[6] A. J. Hu and M. Y. Vardi, editors. *Computer-Aided Verification, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, Vancouver, Canada, June 1998. Springer-Verlag.

[7] V. Rusu and E. Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. Submitted for publication, Sept. 1998. Available at `http://www.csl.sri.com/~singermn/integration.html`.

[8] H. Saïdi and S. Graf. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Computer-Aided Verification, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997. Springer-Verlag.