Invited paper presented at 5th SPIN workshop, Trento, Italy, 5th July 1999.
Appears in *Theoretical and Practical Aspects of SPIN Model Checking: 5th and 6th International SPIN Workshops*, D. Dams, R. Gerth, S. Leue, and M. Massink eds., Springer Verlag Lecture Notes in Computer Science volume 1680, pp. 1–11.

Integrated Formal Verification: Using Model Checking With Automated Abstraction, Invariant Generation, and Theorem Proving*

John Rushby

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025, USA
rushby@csl.sri.com

Abstract. Mechanized formal methods that use both model checking and theorem proving seem to hold most promise for the future. Effective use of both technologies requires they be recast as methods for calculating properties of specifications, rather than merely verifying them. The most valuable properties are those that contribute to the development of invariants and property-preserving abstractions. We outline an architecture for verification tools based on iterated use of such capabilities.

1 Introduction

The advent of model checking not only introduced useful tools, such a SPIN, it also brought a different focus to the field of formal methods. This new focus stressed tools and automated analyses over mathematical derivations performed by hand, and was targeted towards refutation (i.e., finding bugs) more than verification. Many of those who worked on traditional formal verification tools (those based on theorem proving) welcomed the new interest in pragmatics and the sanction that it provided for use of more effective methods of deduction, such as decision procedures. (Formerly, decision procedures had been spurned in some quarters because mechanized verification had—mistakenly, in my view—been identified with the activity of proof in mathematics, rather than with calculation in engineering.)

Some verification tools were extended to incorporate model checking by making a model checker available as a back-end proof procedure—in PVS, for example, a model checker (due to Geert Janssen) is used as a decision procedure for Park's μ-calculus restricted to finite types, and CTL model checking is implemented as a derived proof procedure on top of that [14]. Conversely, some

* This research was supported by DARPA through USAF Rome Laboratory Contract F30602-96-C-0204, and by the National Science Foundation contract CCR-9509931.

developers of model checkers provided lightweight theorem proving as a front end to overcome limitations in plain model checking [13].

Most users and developers of theorem proving and model checking tools now accept that both technologies are needed, but each tends to see the "other" technology as a supporting player, relegated to the role of a front- or back-end adjunct to their primary tool. In contrast, I believe that significant further advances will be made only by "symmetrically" integrated verification environments in which theorem proving and model checking are seen as equal partners and are used in an iterative, interleaved fashion to provide capabilities such as abstraction and invariant generation, as much as to perform direct verification.

A couple of projects have started to explore the development and use of these symmetric approaches; they include SAL (Symbolic Analysis Laboratory) at SRI and Stanford (with substantial input from Berkeley), and VeriTech at The Technion. In the remainder of this short paper, I will sketch the architectures employed in these systems and some of the techniques used. The systems themselves are still inchoate, and their utility has not yet been demonstrated, so the approach they embody should be regarded as somewhat speculative at present.

2 Common Intermediate Languages

Mechanized formal analysis starts from a description of the problem of interest expressed in the notation of the tool to be employed. Construction of this description often entails considerable work: first to recast the system specification from its native expression in C, Esterel, Java, SCR, UML, Verilog, or whatever, into the notation of the tool concerned, then to extract the part that is relevant to the analysis at hand (the order of these two steps can be reversed), and finally to reduce it to a form that the tool can handle (e.g., finite state). If a second tool is to be employed for a different analysis, then a second description of the problem must be prepared, with considerable duplication of effort. Notice that only the final analysis is automated: the translation, extraction, and reduction steps are all performed by hand (an exception is the *Java PathFinder* [9] which provides model checking for Java via an automated translation into Promela, the language of SPIN). The reasons for this lack of automation seem economic rather than technical: with m source languages and n tools, we would need $m \times n$ translators, either m or n "extractors," and n "reducers." This situation naturally suggests use of a common intermediate language, where the numbers of tools required could be reduced to $m + n$ translators and perhaps as few as one extractor and one reducer.

Successful use of an intermediate language seems more promising in this domain than in programming (where the failure of the UNCOL project in the 1960s is still remembered) because we are interested primarily in concurrency issues, where transition relations already provide a widely-accepted semantic foundation.

A useful intermediate language cannot provide just bare transition relations, however, for we will want to support various transformations and analyses on the intermediate representation and these will be inefficient or ineffective if too much of the structure and "intent" of the original specification is lost by flattening it out into a plain transition relation. For these reasons, the SAL intermediate language provides both synchronous and asynchronous forms of composition; distinguishes input, local, and output variables; provides state update via both single-variable assignments (a style often used for hardware specification) and guarded commands (where several state variables may be updated simultaneously—a style often used in protocol specification), and a module structure. Because SAL is an environment where theorem proving as well as model checking is available, absence of causal loops in synchronous systems is ensured by generating proof obligations, rather than by more restrictive syntactic methods. For example, the following pair of assignments is acceptable in SAL because we can prove that `X` is causally dependent on `Y` only when `A` is *true*, and vice-versa only when it is *false*—hence there is no causal loop.

```
X = IF A THEN NOT Y ELSE C ENDIF
Y = IF A THEN B ELSE X ENDIF
```

While there are as yet no automated translators into the SAL language (we hope others will write those if the rest of the system proves effective), we do have translators from SAL to SMV (for model checking), to PVS (for theorem proving), and to Java (for animation).

3 Syntactic Transformations

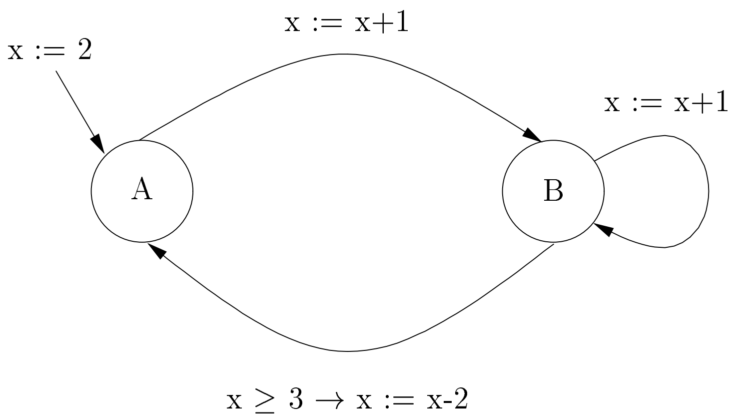
A number of generally-useful transformations can be defined on systems specified in an intermediate representation. A canonical "extractor" could perform backward slicing on variables to yield just that part of a large system specification that is relevant to a given property [7]. We suspect that many of the complexities of program slicing (e.g., interprocedural data flow and global variables) are much reduced when this technology is applied to specifications rather than imperative programs. (Conversely, we believe that effective automated theorem proving could improve the precision of classical program slicing.)

Other transformations could perform syntactic "reductions" on the state space of a specification. These may, but do not always need to be, property-preserving. For example, a reduction commonly employed in model checking arbitrarily reduces the size of large or infinite data types to small concrete values: the maximum number of messages in a queue may be reduced from n to 3, for instance. Such *downscaling* may preserve neither soundness (model checking the downscaled instance may fail to reveal bugs in the original) nor completeness (bugs may be found in the downscaled instance that are not present in the original), but can nonetheless be effective in exposing real bugs quite cheaply.

Yet other reductions may be property-preserving: examples may include some defined by abstract interpretation and those associated with partial order reductions. One of the attractions of an intermediate representation is that it may make techniques such as partial order reduction, presently provided only by SPIN, more widely available.

4 Semantic Transformations

The real power of integrated verification environments only becomes apparent when we consider transformations whose own construction or justification exploits the model checking or deductive capabilities available in the environment. Property-preserving abstractions are the main transformations of this kind. The utility of such abstractions in model checking is well known [5, 12], and the topic of interest is how to develop useful abstractions by largely automatic means. To illustrate some of the issues, I use the following trivial transition system.



This system is truly trivial, in that it does not involve concurrency; it features two control locations, `A` and `B`, and a single integer-valued state variable `x`, which can be incremented by one when control is at either location, and decremented by two when control is at `B` and `x` is at least three. Initially, control is at `A` and `x` has the value two; the goal is to show that `x` is at least two whenever control is at `B`.

This system cannot be model checked directly because it is not finite-state (its state variable `x` can grow arbitrarily large). We could downscale the system by restricting `x` to some arbitrary minimum and maximum values—say zero and five, respectively—but then we will have to modify the description to do something other than increment when `x` is at its maximum value, and to do something other than decrement if this would take it below its minimum value, and then we may wonder about the relationship between the original and the downscaled system descriptions, and the interpretation of any results obtained by model checking.

If we wish to draw conclusions about the original system on the basis of model checking a reduced system, then we must ensure that the reduced system is a property-preserving abstraction of the original. Before doing this, however, I wish to set the stage for combined approaches by illustrating how properties of infinite-state systems can be established by theorem proving.

Theorem proving can be used to establish an invariant property (implicitly, those are the only kind of properties considered here) by showing that the property is true in the initial system state and that it is preserved by all transitions of the system. If we feed our example into PVS [15], it quickly discharges three of the four proof obligations and terminates with the following unproven subgoal.

```
[-1]    pc(pre!1) = A
[-2]    pc(post!1) = B
[-3]    x(pre!1) = 0
  |-------
[1]     x(pre!1) + 1 ≥ 2
```

PVS displays proof goals in this "sequent" form; the interpretation is that the conjunction of the formulas above the line should imply the disjunction of formulas (here there is only one) below the line. An expression such as `pc(pre!1) = A` is interpreted as asserting that the control location (or "program counter," hence the abreviation `pc`) in state `pre!1` is `A` (the `!1` suffix is added to ensure that Skolem constants have unique names). This particular sequent is inviting us to consider the case of a transition from `A` to `B`, where the value of `x` is initially zero. Plainly, the invariant is not satisfied in this case (hence the unprovable formula below the line, which reduces to $1 \geq 2$). From our understanding of the system, however, we can see that this case cannot arise, because `x` must be at least one when control is at `A`. In order to convince the prover of this, however, we must add it to the invariant as an additional fact to be proven: the invariant is said to be *strengthened* by adding the extra conjunct. In general, the process of establishing an invariant by theorem proving involves inspecting failed subgoals to deduce plausible additional conjuncts, strengthening the invariant by adding these conjuncts, and repeating the process until the proof succeeds. In one example of this kind, 57 such strengthenings were required to verify a bounded retransmission communications protocol [10]; each conjunct was discovered by inspecting a failed proof, and the process consumed several weeks.

A more integrated approach to this problem is one that uses theorem proving to justify a property-preserving finite state abstraction of the system, and then model checks the abstraction. Given a concrete state space with transition relation $\mathtt{tr_c}$ and initial states and desired invariant characterized by predicates $\mathtt{init_c}$ and $\mathtt{p_c}$, respectively, we need to find a (finite) abstract state space with an abstraction function `abs` from concrete states to abstract ones, together with an abstract transition relation $\mathtt{tr_a}$ and predicates $\mathtt{init_a}$ and $\mathtt{p_a}$ on the abstract states, such that the following verification conditions hold.

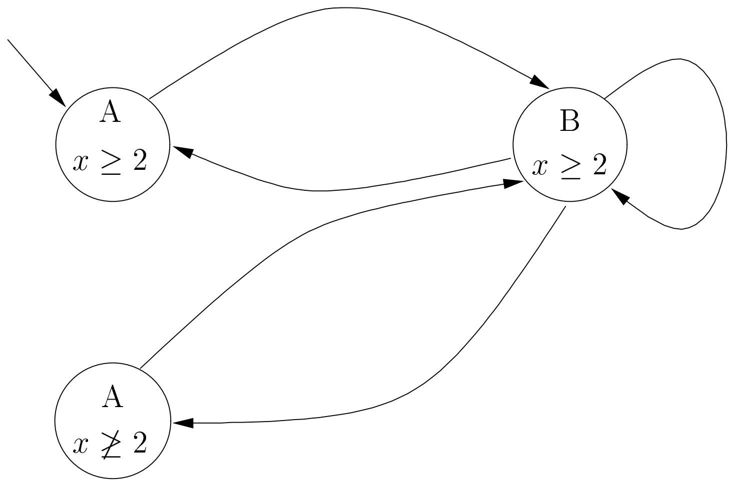
1. $\mathtt{init_c(cs)} \supset \mathtt{init_a(abs(cs))}$,
2. $\mathtt{tr_c(pre_c, post_c)} \supset \mathtt{tr_a(abs(pre_c), abs(post_c))}$, and
3. $\mathtt{p_a(abs(cs))} \supset \mathtt{p_c(cs)}$

where `cs`, $\mathtt{pre_c}$, and $\mathtt{post_c}$ are concrete states, formulas are implicitly universally quantified over their free variables, and $\supset$ indicates logical (material) implication. Then we will have

$$\mathtt{invariant(p_a)(init_a, tr_a)} \supset \mathtt{invariant(p_c)(init_c, tr_c)},$$

and the antecedent can be established by model checking.

A particularly useful form of abstraction uses a collection of boolean variables to define the abstract state space, where each variable abstracts some predicate over the concrete state space [17]. This is called *predicate abstraction* (or sometimes *Boolean abstraction*) and we can apply it to our example by using just a single abstract variable that abstracts the predicate $\mathtt{x} \geq 2$. Using insight, we can postulate the following as a suitable transition relation under this abstraction.



We can see by inspection that the desired invariant is satisfied in this system, so it remains to show that the postulated transition relation is a true abstraction of the original. The first and third of the verification conditions listed previously are discharged trivially by PVS, but the second yields the following sequent.

```
[-1]     pc(post_c!1) = B
[-2]     x(pre_c!1) = 0
  |-------
[1]      x(pre_c!1) + 1 ≥ 2
```

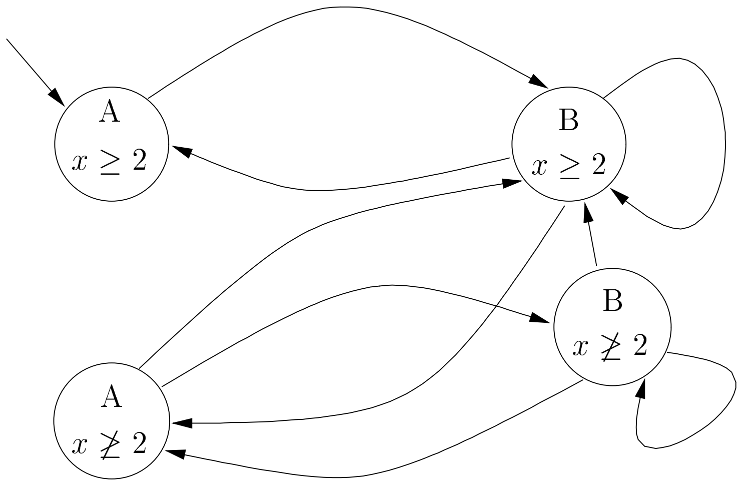
Similar to the subgoal that arose in the invariance proof, this is inviting us to consider the case of a concrete transition into `B` from a state in which `x` is zero. Also similar to the original invariance proof, we see that this troublesome case can be eliminated by using the fact that $\mathtt{x} \geq 1$ is a global invariant of the concrete system—but then we will again need to establish that this is indeed an invariant. As this example illustrates, justification of abstractions generally requires invariants. For the bounded retransmission protocol mentioned earlier, 45 invariants were needed to justify a proposed finite-state abstraction.

Now static analysis can easily establish that $\mathtt{x} \geq 1$ is an invariant of our example system, so one way to enrich an integrated verification environment is by adding a static analysis component (e.g., [4]) that can deduce useful invariants. Justification of a given abstraction can then make use of such known invariants.

The alternative to justifying a given abstraction *a posteriori* is to construct one that is sound *a priori*: that is, the construction technique ensures that if the abstracted property is true of the abstraction so constructed, then the original property is true of the original system (the converse may not be so, however).

Several methods have been proposed for constructing abstracted transition relations for a given abstraction function. One way starts with the universal relation on the abstracted states; then, for each pair of abstract states, generates the verification condition that asserts that the arc between those two states can be removed (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 it. [2]. An alternative approach (specialized for predicate abstraction) develops the abstract transition relation by a forward reachability analysis; at each point it generates the verification conditions that lead to successor states with each given predicate true and with it false [18]. Both approaches can generate an abstracted specification for the bounded retransmission protocol in minutes; the first approach (which is implemented in an extension to PVS called InVeSt [3]) has the advantage that it preserves the structure of the original system description, the second (which is implemented in PVS itself) does not preserve structure but is generally faster and develops abstracted systems with fewer states.

Both these methods require the user to propose the abstraction function (and both work better when supplied with a collection of known invariants). For the case of predicate abstraction, it is natural to start by abstracting on subexpressions of the desired invariant that involve nonfinite variables. With our example system, abstracting on $\mathtt{x} \geq 2$ yields the following abstracted system.



Unfortunately, we see that the abstracted invariant is not satisfied in this system (i.e., the state at the lower right is reachable). If, however, we take a counterexample trace generated by model checking the abstracted system—for example

$$\{\mathtt{A},\ x \geq 2\} \rightarrow \{\mathtt{B},\ x \geq 2\} \rightarrow \{\mathtt{A},\ x \ngeq 2\} \rightarrow \{\mathtt{B},\ x \ngeq 2\}$$

and then *concretize* it [1], we will obtain a trace such as

$$\{\texttt{A, 2}\} \rightarrow \{\texttt{B, 3}\} \rightarrow \{\texttt{A, 1}\} \rightarrow \texttt{?}$$

in which it is impossible to find a final state that both concretizes the final abstract state and satisfies the concrete transition relation. This suggests that our initial abstraction is too coarse and that we should abstract on some additional predicates. Often, those appearing in guards are plausible candidates, but that is not the case here; however, inspection of the counterexample trace suggests $\texttt{x} \geq 1$ as a candidate. Abstracting on both $\texttt{x} \geq 2$ and $\texttt{x} \geq 1$ is a correct solution: it generates the same abstracted system that we previously generated by hand and sought to justify.

Using the abstractor built in to PVS version 2.3, this example is dispatched by a single proof strategy that combines abstraction and model checking when instructed to abstract on both $\texttt{x} \geq 2$ and $\texttt{x} \geq 1$.

```
  |-------
{1}   FORALL (s: state): initial(s) IMPLIES AG(transition, prop)(s)

Rule? (ABSTRACT-AND-MC ("LAMBDA s: x(s)>=2" "LAMBDA s:  x(s)>=1"))

Starting computation of abstraction...abstraction computed in 4 seconds

calls to the decision procedure: possible: 38   performed: 34

MU simplification took 0.08 real, 0.05 cpu seconds
Q.E.D.
```

A difficulty in all these approaches involving abstraction is that the user must propose an abstraction function (or, equivalently, a set of predicates to be abstracted) that is strong enough to solve the entire problem: this is a substantial challenge that requires considerable insight or experimentation. A variant 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 [16]. 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. Another benefit from developing the abstractions in the context established by a theorem prover is that the prover keeps track of all outstanding proof obligations and subgoals, so that none can be forgotten or overlooked.

The techniques sketched in this section make fairly substantial use of both theorem proving and model checking, but they are still not truly integrated in

[1] "Concretization" is the inverse of abstraction; the inverse of the abstraction function is not a function, in general, so some approximation is required.

that model checking is used only to verify the final abstraction (though I did suggest how a counterexample trace could be used to aid diagnosis of an inadequate abstraction). The reason for this is that a traditional model checker returns very little information that can be used in subsequent analyses (basically, true/false plus possibly a counterexample in the latter case). However, it typically will have calculated a great deal of interesting information during its internal computation, and then thrown it away. For example, a symbolic model checker operating in the "forwards" mode calculates[2] the reachable states of its model. Now, the reachable states characterize the strongest invariant of a system, so a concretization of the reachable states of an abstracted system is certainly an invariant, and possibly a strong one, for the original system. 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 that can help construct further abstractions. 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 CMU version of the SMV model checker, where a `print` function, implemented by Sergey Berezin, provides external access to the reachable states.

An integrated combination of abstraction and reachability analysis similar to that sketched above is described in a recent paper by Abdulla et al [1]. Another related approach is described by Das, Dill, and Park [6].

5 Conclusion

A significant advance in the effectiveness and automation of verification for concurrent systems seems possible by combining techniques from model checking and theorem proving in a truly integrated environment. A key idea is to change the perception (and implementation) of model checkers and theorem provers from tools that perform verifications to ones that calculate *properties* (such as abstractions and invariants). By iterating through several such tools (together with tools for static analysis) operating on a common intermediate representation, it becomes possible to incrementally accumulate properties (e.g., invariants) that eventually enable computation of a substantial new property (e.g., an abstraction)—which in turn enables accumulation of further properties. By exploiting the combined power of several forms of mechanized formal analysis, proofs and refutations that currently require significant human effort (either in guiding a theorem prover or in constructing clever abstractions) will, I believe, soon become routine calculations.

[2] Or can calculate—it may terminate early if it discovers a reachable state that falsifies the desired property.

Acknowledgments

None of the work described here is mine; I have drawn freely on the research and insights of my colleagues Saddek Bensalem, David Dill, César Muñoz, Sam Owre, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Yassine Lakhnech. The SAL project is led at SRI by N. Shankar, and at Stanford by David Dill, and has benefitted from collaboration with Tom Henzinger at UC Berkeley.

I am grateful also to Dennis Dams and Mieke Massink for inviting me to the Fifth Spin Workshop, and for their careful reading of the first version of this paper.

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, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In Hu and Vardi [11], pages 319–331.
3. Saddek Bensalem, Yassine Lakhnech, and Sam Owre. InVeSt: A tool for the verification of invariants. In Hu and Vardi [11], pages 505–510.
4. Saddek Bensalem, Yassine Lakhnech, and Hassen Saïdi. Powerful techniques for the automatic generation of invariants. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 323–335, New Brunswick, NJ, July/August 1996. Springer-Verlag.
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. Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In Halbwachs and Peled [8], pages 160–171.
7. Matthew B. Dwyer and John Hatcliff. Slicing software for model construction. In *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, January 1999.
8. Nicolas Halbwachs and Doron Peled, editors. *Computer-Aided Verification, CAV '99*, number 1633 in Lecture Notes in Computer Science, Trento, Italy, July 1999. Springer-Verlag.
9. Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *Software Tools for Technology Transfer*, 1999. To appear.
10. 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.
11. Alan J. Hu and Moshe Y. Vardi, editors. *Computer-Aided Verification, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, Vancouver, Canada, June 1998. Springer-Verlag.

12. 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.
13. Ken McMillan. Minimalist proof assistants: Interactions of technology and methodology in formal system level verification. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD '98)*, volume 1522 of *Lecture Notes in Computer Science*, Palo Alto, CA, November 1998. Springer-Verlag. Invited presentation—no paper in proceedings, but slides available at `http://www-cad.eecs.berkeley.edu/~kenmcmil/`.
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*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995. Springer-Verlag.
15. John Rushby, Sam Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, September 1998.
16. Vlad Rusu and Eli Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 178–192, Amsterdam, The Netherlands, March 1999. Springer-Verlag.
17. Hassen Saïdi and Susanne Graf. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997. Springer-Verlag.
18. Hassen Saïdi and N. Shankar. Abstract and model check while you prove. In Halbwachs and Peled [8], pages 443–454.