A Tool for Proving Invariance Properties of Concurrent Systems Automatically

Hassen Saïdi*

1 Introduction

We describe a tool for the verification of invariance properties of concurrent systems. Our tool combines model-checking and theorem-proving in the following sense: in order to prove that a predicate P is an invariant of a system S ($S \models Init \Rightarrow \Box P$) it is necessary and sufficient to find a predicate P' weaker than $Init$ and stronger than P, such that P' is preserved by all transitions τ of S, i.e $P' \Rightarrow \widetilde{Pre}[\tau](P')$[1] is valid for each transition τ.

Model-checking consists in computing iteratively the greatest solution of the equation $P \Rightarrow \widetilde{Pre}[\tau](P)$ starting with $P_0 = P$ and taking $P_{i+1} = P_i \wedge \widetilde{Pre}[\tau](P_i)$. This method can be completely automatized under the condition that the above predicates are decidable. However it suffers from the drawback that in the case of infinite state systems convergence is not guaranteed or too slow. Convergence can be accelerated by using *local invariants* extracted from the program obtained by constant propagation, variable domain information, etc ([2], [1]). Convergence can also be forced by using finite abstraction techniques consisting in replacing $\widetilde{Pre}[\tau](P_i)$ by a lower approximation, but in this case, false negative results are possible.

In our tool theorem proving is used for establishing $P_{i+1} \equiv P_i$ that is for testing if the fixed point has been reached. For this purpose we use the PVS theorem prover [7] which implements several decision procedures for decidable sub-classes of formulas. Systems are described in a syntax using PVS declaration and formula syntax. Verification conditions (VC) are extracted automatically by means a "verification condition generator". This generator avoids to generate "trivially true" VCs. For example, if an action τ does not affect the variables on which predicate P depends, the VC $P \Rightarrow \widetilde{Pre}[\tau](P)$ is not generated. Three kinds of verification conditions are generated automatically. The first ones are the PVS type correctness con-

---

*Verimag, Miniparc-Zirst, Rue Lavoisier, 38330 Montbonnot St-Martin, France. Phone: (+33) 76-90-96-43, Fax: (+33) 76-41-36-20. e-mail: saidi@imag.fr.

[1] The predicate $\widetilde{Pre}[\tau](P)$ defines the smallest set of states that have via transition τ only successors satisfying P.

ditions (TCCs) which are generated once. The others, which are updated at each iteration step are the lemmas expressing the fact that P is the invariant we are looking for, that is $P \Rightarrow \widetilde{Pre}[\tau](P)$ and $Init \Rightarrow P$. Tools based on a similar approach are actually designed [4] [3]. However, they are restricted to simple data types as natural numbers, booleans and arrays. Our aim is to use the power of the specification language used in PVS and the proof techniques available to discharge verification conditions. Thus, we allow complex data-types such as buffers and we use powerful proof mechanisms. Our efforts were concentrated on a maximal automatization of the proof process by defining powerful proof strategies. However, it is clear that complete automatization is only possible in the decidable cases, otherwise, the techniques we use are "just heuristics" improving "sometimes" an intrinsically incomplete proof technique. In this case, interactivity with the PVS prover is used when these proof strategies fails.

PVS

PVS is an environment for writing specifications and developing mechanized proofs. It consists of a specification language integrated with a powerful and highly interactive theorem prover. PVS uses higher order logic as a specification language, the type system of PVS includes uninterpreted type, sub-typing and recursively defined data-types. Four sorts characterize this language: `Theory`, `Type`, `Expression` (*term*), `Formula` (*proposition*). Any PVS specification is structured into parameterized theories. A `theory` is a set of declarations consisting of `type`, variable, constant, function and `formula` declarations. The PVS theorem prover implements a set of powerful primitive inference rules with a mechanism for composing them into proof strategies. PVS has emacs as user interface.

2 General Architecture

The organization of the tool (see Figure 1) is inspired by the architecture of PVS. The main entry is a description of a system and a property we want to be invariant, respecting the syntax[2] defined in figure 2. The user controls the verification process by a set of commands.

Writing specifications

A specification is a parallel composition of programs. For the description of a program we use a simple programming language close to Dijkstra's guarded command language (see Figure 2). A program is a tuple $\langle\, \mathcal{V},\ \mathcal{T},\ \mathcal{I}\,\rangle$

[2] The grammar is presented using the conventions of [7].

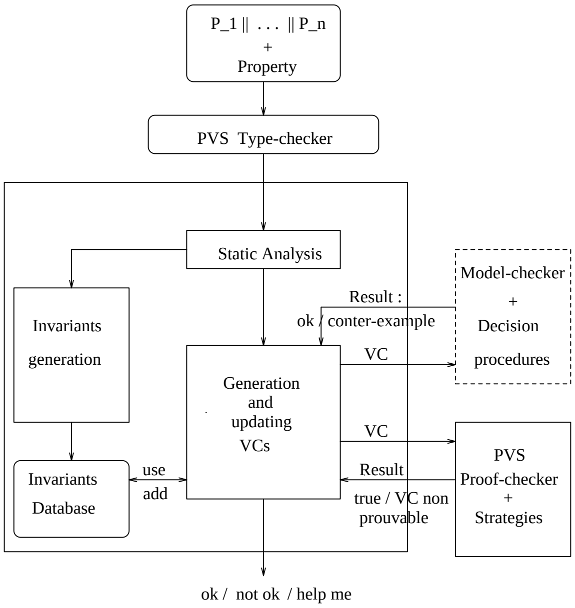

FIGURE 1. Tool architecture

where $\mathcal{V}$ is a set of local program variables, $\mathcal{T}$ is a set of transitions corresponding to the different actions of the program, and $\mathcal{I}$ an initial condition. $global-declaration$ and $local-declaration$ are sequences of PVS declarations. This shows clearly the power of the language, since we allow program variables of any type definable in PVS (that is natural, rational and real numbers, booleans, enumerated or abstract data-types and functions[3]). Programs may use as basic functions any function definable in PVS by importing predefined or user-defined theories.

Static analysis

Static analysis provides "structural" invariants. They provide informations about values of variables at particular control points. In [2] and [5] several techniques are used to generate automatically this kind of invariants. In our tool we implemented the more powerful ones presented in [1]. Given such a structural invariant I, it is sufficient to prove for each transition τ and predicate P, $I \wedge P \Rightarrow \widetilde{Pre}[\tau](P)$ instead of proving $P \Rightarrow \widetilde{Pre}[\tau](P)$. Since the predicate I is generally a huge formula of the form $\bigwedge_{i=1}^{n}\ pc = i => Q_i$ where pc is a control variable, it is necessary to eliminate as far as possible the non-relevant conjuncts. For example, when we prove $P \Rightarrow \widetilde{Pre}[\tau](P)$, where action τ moves from control point i to j, we only use the conjuncts of the form $pc = i \Rightarrow Q_i$.

[3] This is the way we define arrays and other multi-dimensional data structures.

```
specification  ⇒ id−system [ PARAMETER id ] : SYSTEM
                 BEGIN
                 ⟨ global−declarations ⟩
                      BEGIN
                      ⟨ components ⟩
                      END
                 SAFETY : ⟨ pvs−boolean−formula ⟩
                 INITIALLY : ⟨ pvs−boolean−formula ⟩
                 END id−system
components     ⇒ ⟨ program ⟩ | ⟨ program ⟩ || ⟨ components ⟩
program        ⇒ id−program : PROGRAM
                 BEGIN
                      ⟨ local−declarations ⟩
                    ACTIONS :
                      BEGIN
                      ⟨ action ⟩+
                      END
                 INITIALLY : ⟨ pvs−boolean−formula ⟩
                 END id−program
action         ⇒ ⟨ pvs−boolean−formula ⟩ --> ⟨ assignement ⟩+
assignement    ⇒ id := ⟨ pvs−expression ⟩
```

FIGURE 2. Specification syntax

Proving verification conditions

A proof session starts by typing the command `M-x prove-invariant` (`M-x pi`). A PVS theory containing the lemmas to prove is automatically generated, and the following proof strategy is applied to each of them. First an efficient but incomplete proof strategy for first order predicates is used. It combines rewriting, boolean simplification using BDDs[4], and an arithmetic decision procedure: after rewriting all definitions, the BDD procedure breaks formulas into elementary ones, where other decision procedures such as arithmetic ones can be applied. If the proof fails, another strategy combining automatic induction and decision procedures is applied. If some of them cannot be proved, the user can either try to prove the unproved lemmas using the PVS interactive prover, or start a new proof session with new verification conditions corresponding to the iteration step defined above. In the case that the user knows that the generated VC are decidable predicates – that is a VC that cannot be proved is not true –, he can use the

[4] A BDD simplifier is available in PVS as a proof strategy.

command `M-x prove-invariant-and-loop` (`M-x pl`) which tries to compute the greatest solution of $P \Rightarrow \widetilde{Pre}[\tau](P)$ implying the property to be proven an invariant, with the risk of non convergence but with the benefits of complete automatization.

3 Future work

We verified with our tool some mutual exclusion examples with finite and infinite state variables, studied in [5]. We also verified some examples involving abstract data types such as buffers. Many features are planned to deal with effective and large systems. We plan to enforce modularity of the descriptions, allowing to verify properties of subsystems of a given complex system. Using compositional rules, we can deduce global properties of systems using some previously proved properties of their components.

Acknowledgments: The author wants to thank Susanne Graf for her helpful comments.

4 References

[1] S. Bensalem, Y. Lakhnech, H. Saïdi. *Powerful Techniques for the Automatic Generation of Invariants.* Submitted to CAV'96.

[2] N. Bjørner, A. Browne and Z. Manna. *Automatic Generation of Invariants and Intermediate Assertions.* In U. Montanari, editor, First International Conference on Principles and Practice of Constraint Programming, LNCS, Cassis, France, September 1995.

[3] A. Blinchevsky, B. Liberman, I. Usvyatsky, A. Pnueli. *TPVS: Documentation and Progress Report.* Weizmann Institute Of Science, 1994.

[4] Z. Manna and al. *STeP: The Stanford Temporal Prover.* Department of Computer Science, Stanford University, 1994.

[5] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[6] S. Owre, N. Shankar, and J. M. Rushby. *A Tutorial on Specification and Verification Using PVS.* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[7] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language.* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.