The Invariant Checker: Automated Deductive Verification of Reactive Systems

Hassen Saïdi
VERIMAG[1]
saidi@imag.fr
http://www.imag.fr/VERIMAG/PEOPLE/Hassen.Saidi/Invariant-Checker.html

1 Design Philosophy

The Invariant Checker [GS96,Saï96] is a tool for the verification of invariance properties of reactive systems using theorem-proving techniques and tools. The system is designed as a front-end for the PVS [OSR93a] theorem prover. The Invariant Checker can be seen as an extension of the PVS verification system to handle the notion of transition systems and invariants as well as the usual mathematical objects. These extensions appear at two different levels: the PVS specification language is extended by the notion of a **system**, that is a program given as a transition system or a parallel composition of transition systems. The PVS prover is also extended with a proof rule (cf. [MP95]) dedicated to invariance properties. To check whether a predicate P is an *inductive* invariant of a system S, it is sufficient to check the validity of a set of first order formulas called verification conditions (VCs) (cf. [GS96]), expressing the fact that each transition of the program preserves P. This proof rule also provides a strengthening method for P: if some of the generated VCs are not provable, P is replaced by $P \wedge \widetilde{pre}(P)$ in a model checking like manner. This method can be completely automatized, but convergence is not guaranteed.

This kind of invariant verification makes a different use of theorem proving than the "classical" one where the program semantics is encoded in the prover's specification language. In this "classical" approach the proof process is complicated by the encoding of semantics and the rewriting of semantics definitions, while the most important and difficult part of the verification process is the reasoning about the program variables and their values. Also, it requires too much user intervention. The objective of our tool is to provide more automatization using a set of features. The architecture of the tool is presented in Figure 1.

2 Features

Syntax: Programs can be described in a Simple Programming Language (SPL), close to the one used in [MP95], where program variables can be of any type definable in PVS, and can be assigned by any PVS expression of compatible type. Also, it is possible to import any defined PVS theory. Programs described in SPL are translated automatically to guarded commands with explicit control.

[1] Centre Equation, 2, Avenue de la Vignate, 38610 Grenoble-Gières, Tel: (+33) 4.76.63.48.44, Fax: (+33) 4.76.63.48.50

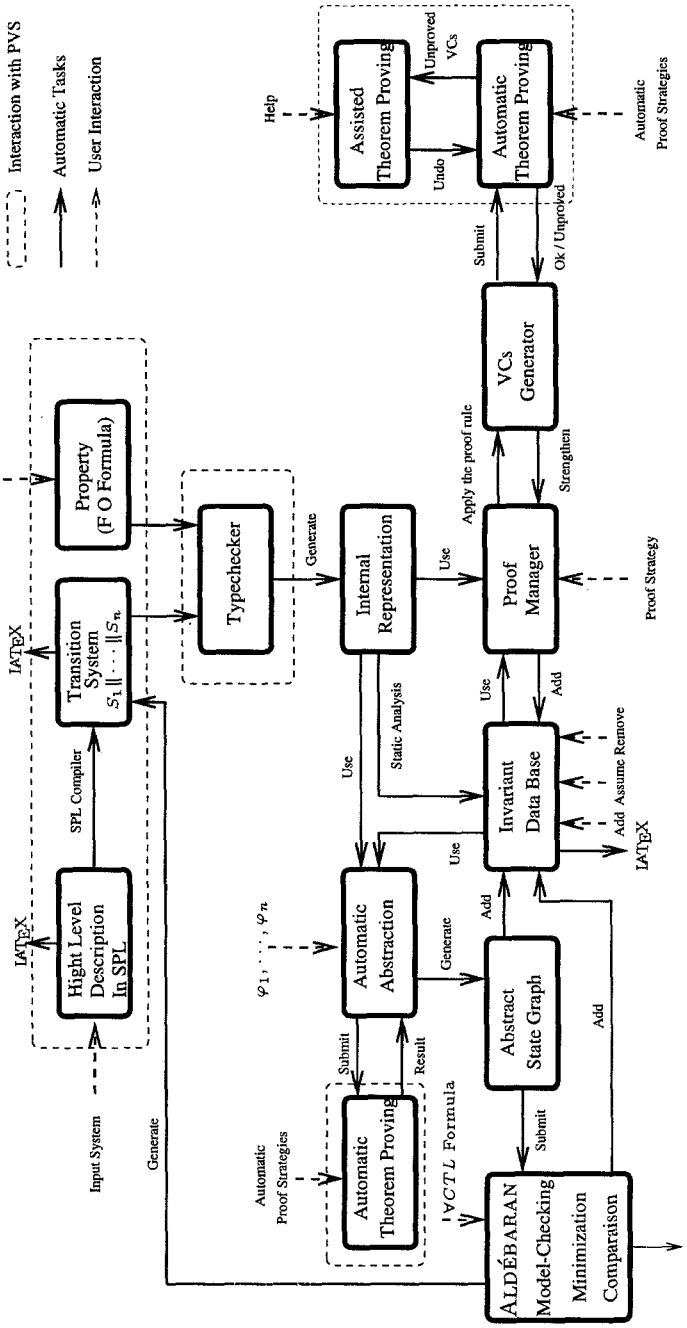

Fig. 1. The Invariant Checker Architecture

Typechecking: Typechecking a program consists in checking that every guarded command is well typed according to a typing context, that is each guard is a boolean expression, and each variable is assigned by an expression of a compatible type. This typing context consists of all variable declarations and the imported Pvs `theory`. Typechecking a specification leads to the generation of type correctness conditions (TCCs) which have to be proved as *invariants* and not as valid formulas. If all generated TCCs are proved, this guarantees "absence of run-time errors", such as division by zero or the application of the tail function to the empty list.

Proof session: A proof session starts with typechecking the program and the property to be verified. The program is translated into an internal representation which is used by all components of the tool. One can apply static analysis methods in order to extract inductive invariants using the techniques described in [BLS96]. The generated invariants are stored in the invariant data base.
In order to prove that the considered property is an inductive invariant of the program, the user can apply the proof rule using two different modes:

- Interactive mode: In this mode the user invokes the proof rule which generates a set of VCs. Each VC is submitted to the prover, where a proof strategy is applied. If some of the generated VCs are not provable, the user can either try to prove them interactively or he can apply the proof rule again. In this case, the invariant is automatically strengthened, and a fresh set of VCs is generated.
- Automatic mode: In this mode the user indicates to the proof manager the maximal number of strengthening step. The proof rule is then applied without user interaction until this maximal number is reached, or until an inductive invariant is computed.

In both modes, the user can decide to use the invariants in the data base in order to weaken the generated VCs. In this case, for every generated VC, a set of relevant generated invariants is automatically selected to achieve the proof.

Automatic discharging of VCs: The generated VCs are submitted to the Pvs prover, where automatic proof strategies combining automatic induction, automatic rewriting, boolean simplification using Bdds and decision procedures are applied. The user can defined such strategies, by combining pre-defined Pvs strategies and user defined ones. Non provable assertions are considered non valid.
Invariant data base: It contains the invariants generated using the techniques described in [BLS96]. Each already proved invariant is automatically added to the data base. Also, the user can always enrich the data base. Therefore, with each invariant is associated a status which changes during a proof session. The status has three possible values:

- *assumed*: these are user defined invariants for which no proof is required. They play the role of axioms, and can therefore lead to inconsistent proofs.
- *unproved*.

- *proved*: a proof is associated with such an invariant. It consists of the applied proof strategy and the invariants used during the proof. In order to maintain a coherent data base, if some invariant is removed, all the already proved invariants depending on it, become *unproved*.

Automatic abstraction: Recently, we added a new feature, which consists of the use of abstraction techniques [GS97]. Given a set of predicates $\varphi_1, ..., \varphi_\ell$ on the variables of a program, an abstract state graph (where states are valuations of $\varphi_1, ..., \varphi_\ell$) is constructed in an automatic way using user defined proof strategy. An abstract state graph can be used in many ways:

- It defines an invariant of the program.
- Any verification technique for finite state systems can be applied. We have interfaced our tool with ALDÉBARAN [FGK+96], which allows:
  - minimization of the abstract state graph modulo bisimulation.
  - evaluation of any temporal logic formula without existential quantification over paths.
- The abstract state graph (or its minimization w.r.t to strong bisimulation) can be used as a global control graph from which stronger invariants can be generated and added to the invariant data base.

User interface: PVS has emacs as user interface. We found convenient to use the same user interface for our prototype. All the functions of the tool can be invoked by emacs commands.

3 Experiments

Using our tool we verified various classical mutual exclusion algorithms , a read and write buffer using complex data types [GS96]. The use of abstraction techniques allow us to prove in a fully automatic way parameterized versions of an alternating bit and a bounded retransmission protocol [GS97].

References

[BLS96] S. Bensalem, Y. Lakhnech, and H. Saïdi. *Powerful Techniques for the Automatic Generation of Invariants.* In *Conference on Computer Aided Verification CAV'96*, LNCS 1102, July 1996.

[FGK+96] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier and M. Sighireanu. CADP (Cæsar/Aldébaran Development Package): A protocol validation and verification toolbox. In *CAV' 1996.* LNCS 1102, 1996.

[GS96] S. Graf and H. Saïdi. Verifying invariants using theorem proving. In *Conference on Computer Aided Verification CAV'96*, LNCS 1102, July 1996.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In this volume.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[OSR93a] S. Owre, N. Shankar, and J. M. Rushby. *A Tutorial on Specification and Verification Using PVS.* Computer Science Laboratory, SRI International, February 1993.

[Saï96] H. Saïdi. *A Tool for Proving Invariance Properties of Concurrent Systems Automatically.* In TACAS'96, LNCS 1056, Springer-Verlag.