The Invariant Checker: Automated Deductive Verification of Reactive Systems

Hassen Saïdi
saidi@imag.fr
VERIMAG*

Abstract

The Invariant Checker is a tool for the computer aided verification, dedicated to the proof of invariance properties of reactive systems. The aim of the tool is to provide a framework combining theorem-proving techniques (Pvs theorem prover) and deductive verification methods. Systems are described as a parallel composition of sequential programs described in a language close to the Dijkstra guarded command language. Program variables can be of any type definable in Pvs. The tool provides automatic generation of invariants, automatic strengthening of invariants and automatic abstraction. The tool is interfaced with ALDÉBARAN, a tool for the analysis of state graphs.

keywords: *theorem proving, invariants proofs, abstraction, automatic generation of invariants*

1 Design Philosophy

The Invariant Checker ($\mathcal{IC}$) [GS96, Saï96] is a tool for the computer-aided verification dedicated to 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 [CLN$^+$95] theorem prover. The $\mathcal{IC}$ 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 appears at two different levels: the Pvs specification language is extended with the notion of a `system`, or a parallel composition of transition systems instead of a `theory`. The Pvs prover is also extended with a deductive 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]). This proof rule also provides a strengthening method for P if for some of the generated VCs the proof fails. This method can be completely automatized under the condition that the generated VCs are decidable predicates.

This type of invariant verification makes a different use of theorem proving from the "classical" one where program's semantics are encoded in the prover's specification language (usually higher order logic or set theory). In this "classical" approach the proof process is complicated due to the heavy encoding of semantics and the unnecessary rewriting of semantics definitions, while usually 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 and less user intervention using a set of features. The architecture of the tool is presented in Figure 1.

Syntax: Systems can be described in a Simple Programming Language (SPL), close to the one used in [MP95], but with the rich data types and expressions definition mechanism available in Pvs. Systems described in SPL are translated automatically to guarded commands with explicit control. Program variables can be of any type definable in Pvs, and can be assigned by any definable Pvs expression of compatible type. Also, it is possible to import any defined Pvs theory.

*Centre Equation, 2, Avenue de la Vignate, 38610 Grenoble-Gières, Tel: (+33) 4.76.63.48.44, Fax: (+33) 4.76.63.48.50 http://www.imag.fr/VERIMAG/PEOPLE/Hassen.Saidi/

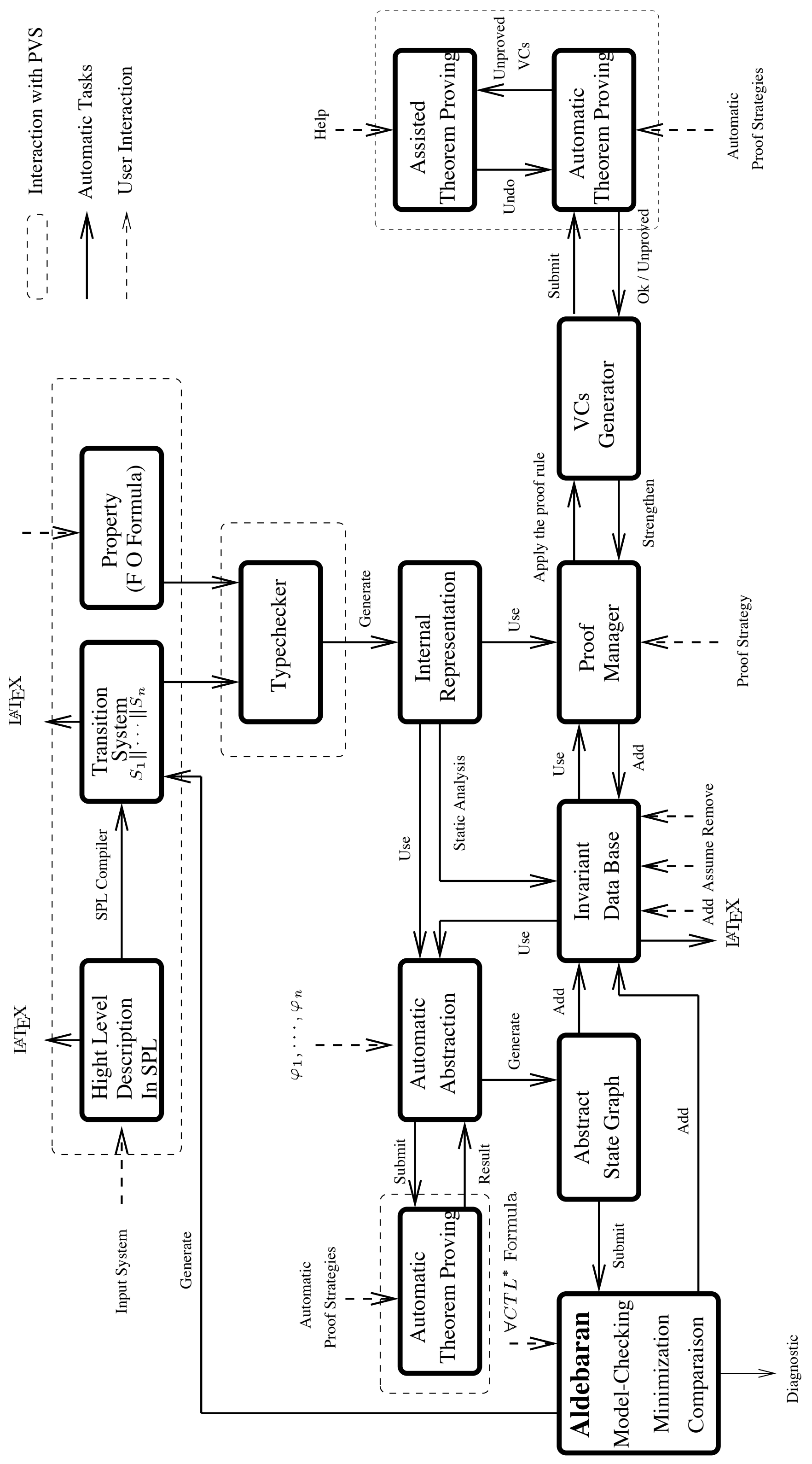

Figure 1: The Invariant Checker Architecture

Typechecking: Typechecking a system consists in checking that every guarded command is well typed according to a typing context. This typing context consists of all variable declarations and may be some imported PVS `theory`. Typechecking a specification is undecidable as it is the case in PVS. The generated type correctness conditions (TCCs) have to be proved as invariants and not as valid formulas. The generated TCCs guarantee "absence of run-time errors" (division by zero, application of the tail function on the empty list...). In the case where they cannot be proved, absence of run-time errors is not guaranteed but this does not affect the proved invariants.

Proof session: A proof session starts with typechecking the system and the property we want to verify. The system is then translated into an internal representation which will be used by all components of the tool. One can then apply static analysis to the system in order to extract auxiliary invariants using the techniques described in [BLS96]. The user can then start the proof by indicating to the proof manager, which strategy he wants to invoke. That is for example the maximal number of strengthening steps, the automatic use or not of some of the invariants in the data base. At each call of the proof rule, a set of VCs is submitted to the prover. If the applied proof strategy fails to prove some of them, the user can either prove them interactively or automatically strengthen the invariant and apply the proof rule again. For every generated VC, a set of relevant generated invariants is automatically selected to achieve the proof.

Automatic theorem proving: 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 can be applied. The user can defined such strategies, by combining pre-defined PVS strategy and user defined ones. Non provable assertions are considered non valid.

Invariants data base The invariant data base contains the invariants generated using the techniques described in [BLS96]. The user can always enrich the invariant data base with some invariants. with each invariant is associated a status which may change during the proof. The status has three possible values:

- "assumed": assumed invariants are user defined and no proof is required for them. They play the role of axioms, and can therefore lead to inconsistent proofs.
- "unproved".
- "proved": with each proved invariant its proof is associated. It consists of the applied proof strategy and the invariants used during the proof. If some invariant is removed from the data base, all the already proved invariants whose associated proof depends on the removed invariant, 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 system, 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 can be used as a global control graph from which stronger invariants can be generated and added to the invariants data base then from the initial system.
- It can be minimized modulo strong equivalence using the ALDÉBARAN tool [FGK+96]. The reduced graph defines a new system with a single component, on which we can prove the property we want to verify using the implemented proof rule.
- It is possible to prove a temporal formula involving $\varphi_1, ..., \varphi_\ell$ using the model checker of ALDÉBARAN.

Features The main features of available in the $\mathcal{IC}$ tool can be resumed as follows:

- Convenient and simple syntax for the description of parallel system.
- Automatic generation of invariants.
- Automatic generation of proof obligations.
- Automatic generation of abstract state graph.
- Automatic proof procedures du to PVS.
- Generation of LaTeX.

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 some emacs command.

2 Experiments

Using the $\mathcal{IC}$ we verified various classical mutual exclusion algorithms [MP95], a read and write buffer using complex data types [GS96]. The use of abstraction techniques allow us to prove in a fully automatic way an alternating bit and a bounded retransmission protocol [GS97].

Additional information can be found on the Invariant Checker home page:
`http://www.imag.fr/VERIMAG/PEOPLE/Hassen.Saidi/Invariant-Checker.html`

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.

[CLN+95] D. Cyrluk, P. Lincoln, P. Narendran, S. Owre, S. Ragan, J. Rushby, N. Shankar, J. U. Skekkebæk, M. Srivas, and F. von Henke. *Seven Papers on Mechanized Formal Verification.* Technical Report SRI-CSL-95-3, Computer Science Laboratory, SRI International, 1995.

[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.

[Gra94] S. Graf. *Characterization of a sequentially consistent memory and verification of a cache memory by abstraction.* Journal of Distributed Computing, 1995.

[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. Submitted to CAV'97 as an A-category paper.

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

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