InVeSt : A Tool for the Verification of Invariants*

S. Bensalem[1], Y. Lakhnech[2] and S. Owre[3]

[1] VERIMAG, Centre Equation – 2, avenue de Vignate,
F-38610 Giéres, France. Email: Bensalem@imag.fr
[2] Institut für Informatik und Praktische Mathematik,
Christian-Albrechts-Universität zu Kiel,
Preußerstr. 1-9, D-24105 Kiel, Germany.
Email: yl@informatik.uni-kiel.de
[3] Computer Science Laboratory, SRI International,
Menlo Park, CA 94025, USA. Email : owre@csl.sri.com

1 Introduction

A very important class of properties of reactive systems consists of *invariance* properties which state that all reachable states of the considered system satisfy some given property. Indeed, every safety property can be reduced to an invariance property and to prove progress properties one needs to establish invariance properties [15]. Proving invariance properties is especially crucial for infinite and large finite state systems which escape algorithmic methods. In this paper we present the tool InVeSt which supports the verification of invariance properties of infinite state systems. InVeSt integrates deductive and algorithmic verification principles for the verification of invariance properties as well as abstraction techniques.

2 Methodology

There are basically two approaches to the verification of reactive systems, the *algorithmic* approach on one hand and the *deductive* approach on the other hand. The algorithmic approach is based on the computation of fix-points, on effective representations of sets of states, and on decision procedures to solve the inclusion problem of sets of states. For example the *backward procedure* is an instance of this approach. To prove that a set of states P is an invariant of a system S, the backward procedure computes the largest set Q of states satisfying $Q \subseteq P$ and $Q \subseteq \text{WP}(\tau, Q)$, for every transition $\tau \in \mathcal{T}$ of S. Here $\text{WP}(\tau, Q)$ is the weakest pre-condition of τ with respect to Q. Then, P is an invariant of S if and only if every initial state of S satisfies Q. In general, the algorithmic approach is based on an effective representation $\mathcal{R}$ for sets of states, effective boolean operations, a procedure for deciding inclusion in $\mathcal{R}$, effective predicate

* This work has been partly performed while the first two authors were visiting the Computer Science Laboratory, SRI International. Their visits were funded by NSF Grants No. CCR-9712383 and CCR-9509931.

transformers to guarantee recursiveness of the method, and convergence of fix-points to guarantee completeness.

In general, in case of infinite state systems, first-order logic with Peano arithmetic is considered as representation $\mathcal{R}$. In fact, it can be proved that any weaker logic is not expressive enough (e.g. [8]), when the considered system contains variables that range over infinite domains. Thus, one has effective boolean operations and can define predicate transformers, but inclusion is undecidable. Moreover, convergence of fix-points is not guaranteed. Consequently, the algorithmic approach cannot be applied in general to infinite state systems. On the other hand, the deductive approach is very powerful and gives a complete method even for infinite state systems. It relies upon finding *auxiliary invariants* and proving validity of first-order formulas, called *verification conditions*. The deductive approach is, however, in contrast to the algorithmic approach, difficult to apply. Indeed, it is in general a hard task to find suitable auxiliary invariants and time consuming to discharge all generated verification conditions. Therefore, there is a strong need for tools that support both tasks. InVeSt is such a tool as it supports the verification of invariance properties of infinite state systems.

The salient feature of InVeSt is that it combines the algorithmic with the deductive approaches to program verification in two different ways:

1) It integrates the principles underlying the algorithmic (e.g. [4, 20]) and the deductive methods (e.g. [16]) in the sense that it uses fix-point calculation as in the algorithmic approach but also the reduction of the invariance problem to a set of first-order formulas as in the deductive approach.
2) It integrates the theorem prover PVS [19] with the model-checker SMV [17] through the automatic computation of finite abstractions. That is, it provides the ability to automatically compute finite abstractions of infinite state systems which are then analyzed by SMV or, alternatively, by the model-checker of PVS.

InVeSt supports the proof of invariance properties using the method based on induction and auxiliary invariants (e.g. [16]) as well as the method based on abstraction techniques [5, 13, 7, 11, 12, 6].

InVeSt's approach to finding auxiliary invariants. We use calculation of pre-fix-points by applying the body of the backward procedure a finite number of times and use techniques for the automatic generation of invariants (cf. [16, 14, 1]) to support the search for auxiliary invariants. The tool provides strategies which allow to derive *local invariants*, that is, predicates attached to control locations and which are satisfied whenever the computation reaches the corresponding control point. InVeSt includes strategies for deriving local invariants for sequential systems as well as a composition principle that allows to combine invariants generated for sequential systems to obtain invariants of a composed system.

InVeSt's approach to computing abstractions. InVeSt provides also a module that allows to compute an abstract system from a given concrete system and an abstraction function. The method underlying this module is presented in [2]. The main features of this method is that it is automatic and compositional. Moreover, it generates an abstract system which has the same structure as the concrete one. This gives the ability to apply further abstractions and techniques to reduce the state explosion problem and facilitates the debugging of the concrete system. The computed abstract system is optionally represented in the specification language of PVS or in that of SMV. A graphical interface allows to interact with InVeSt and SMV in a uniform way.

Finally, it is important to understand that our use of the theorem-prover PVS is limited to discharging the verification conditions. This shows a difference of our approach to the approach followed in most of the work using theorem-proving for verifying invariance properties (e.g. [9, 10]). That is we do not encode the invariance problem in the specification language of the considered theorem-prover and then use the theorem-prover to solve it, but we use the deductive approach to reduce the problem to a set of first-order formulas whose validity is proved using the theorem-prover. Moreover, the construction of the verification conditions as well as the generation of an auxiliary invariant are performed outside the theorem-prover.

3 Design Principles

The structure of InVeSt is motivated by a number of design decisions. These decisions are :

- **Minimization of user's intervention.** This decision is motivated by our belief that the success of the algorithmic approach is partly due to the fact that it does not require user's intervention. To support this choice we have
  - developed techniques for the generation of auxiliary invariants,
  - implemented the strengthening method and its refined version,
  - investigated strategies for proving first-order predicates, and
  - developed a method for computing abstractions of infinite state systems.
- **Use of an existing theorem-prover.** We build on an existing theorem-prover for the following reasons. The first is that we want to rely on a widely used tool; this increases our trust and confidence in the prover. Our particular choice is to use PVS, since PVS gives us the possibility to combine decision procedures and interactive proofs (see [18]). The examples we have considered show that this feature is a prerequisite to reach a high level of automation.
- **Theorem-prover as a "Decision Procedure".** Most of the theorem-provers, including PVS, are general purpose provers. This means that they have general specification languages and if they include pre-defined strategies, then these are in general not tuned to a particular application. This is often a source of inefficiency and prevents a higher level of automation. There is always a trade off between generality on one side and efficiency and automation on the other side. We use the theorem-prover in a particular way and for a particular task, namely to discharge the verification conditions. This should be seen in contrast to the alternative approach where one

encodes the verification problem within the specification language of PVS and tries to solve it completely within this theorem-prover, usually by expanding the definitions of the semantics of the programs and the definition of invariance and using an induction argument. Our approach is different, since we use deductive rules to reduce the invariance problem to a set of first-order formulas whose validity is proved using PVS. This design decision allows us to implement the components of our tool outside PVS.

- **Modularity with respect to the theorem-prover.** Our tool builds on PVS in two different ways. The first obvious point is that it uses PVS to discharge verification conditions as explained above. There is another, less transparent dependency, which lies in the fact that we use the internal representations of PVS of all objects constructed by the PVS type-checker including programs, actions, formulas, expressions, etc.. In order to be modular with respect to PVS, the interface between the components of the tool and PVS itself has to be defined precisely. As interface, we use a module which contains functions that allow to access internal PVS variables. This ensures that even when the data structure used in PVS is modified, the functioning of our tool is still guaranteed as long as the accessor functions maintain their semantics.

4 Tool structure

The main components of InVeSt are: a front-end that translates guarded command-like programs into a PVS-theory, a module of functions for generating invariants, a module of proof strategies, and a module for computing abstractions. In the sequel, we discuss each of these components.

- **Front-end:** As formalism for describing systems we consider a language similar to Unity. Variables are allowed to be of any type of the specification language of PVS. The role of the front-end consists of translating the extended transition system into a PVS-theory that can be type-checked. The components of the system are translated into PVS-constants of type "Program" consisting of a list of actions. An action can be a function or a relation between states. When the PVS-theory corresponding to the system is type-checked a list of LISP-objects corresponding to the declarations in the PVS-theory is constructed. By accessing to the elements of this list, we have at our disposal PVS-representations corresponding to the relevant syntactic objects of the system. Our tool works with these representations. Accessors functions have been implemented that allow to access these representations and their components. For instance, there is a function GET-GUARD that is used to extract the guard of an action, and a function GET-AFFECTED-VARIABLES to extract the list of variables to which a value is assigned by the action.
- **Generation of Invariants:** A central component in the tool is a module consisting of functions that implement strategies used to automatically generate invariants. Basically, we implemented the strategies presented in [1]

which allow us to derive *local invariants*, that is, predicates attached to control locations such that these predicates are satisfied whenever the computation reaches the corresponding control point. We have strategies for deriving local invariants for sequential systems and a composition principle that allows to combine invariants generated for sequential systems to obtain an invariant of a composed system.

- **Proof strategies:** A proof strategy determines which verification conditions are constructed and how to handle failed proofs. The choice of a strategy is determined by:
  1 whether auxiliary invariants are generated automatically,
  2 whether in case the proof of a verification condition fails, the strengthening method [16] or its refined version [3] is applied.

  In case auxiliary invariants are generated, verification conditions are weakened by taking the generated predicate as assumption in the left-hand side of the implication. Thus, if φ is the generated invariant, then to prove that P is preserved by a transition τ it suffices to prove $(\varphi \wedge P) \Rightarrow \text{WP}(\tau, P)$. We implemented a function that takes as arguments two predicates φ and ψ, a transition τ, and a PVS-proof strategy *str* and which calls the PVS-prover on the formula $(\varphi \wedge P) \Rightarrow \text{WP}(\tau, \psi)$ with the strategy *str*.
- **Computing Abstractions:** The abstraction module implements the method presented in [2] for computing abstractions of infinite state systems. For a given concrete system and a given abstraction function, it computes an abstraction of the concrete system compositionally and automatically. The process of generation of the abstract system does not depend on the assumed semantics of the parallel operator; it works for the synchronous as well as for the asynchronous computation model. The generated abstract system has the same structure as the concrete one and there is a clear correspondence between the transitions of both systems. This does not only allow to apply further abstractions and techniques to mitigate the state explosion problem but also facilitates the debugging of the concrete system.

References

1. S. Bensalem and Y. Lakhnech. Automatic generation of invariants. Accepted in Formal Methods in System Design. To appear.
2. S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems automatically and compositionally. Accepted in CAV'98, 1998.
3. S. Bensalem, Y. Lakhnech, and H. Saidi. Powerful techniques for the automatic generation of invariants. In *CAV'96*, volume 1102 of *LNCS*. Springer-Verlag, 1996.
4. E.M. Clarke, E.A. Emerson, and E. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *10th ACM symp. of Prog. Lang.* ACM Press, 1983.
5. E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5), 1994.
6. D. Dams. *Abstract interpretation and partition refinement for model checking.* PhD thesis, Technical University of Eindhoven, 1996.

7. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems: Abstractions preserving ACTL*, ECTL* and CTL*. In *PROCOMET*. IFIP Transactions, North-Holland/Elsevier, 1994.
8. J.W. de Bakker. *Mathematical Theory of Program Cortrectness*. Prentice-Hall, NJ., 1980.
9. K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *FME'96*, volume 1051 of *LNCS*. Springer-verlag, 1996.
10. J. Hooman. Verifying part of the access.bus protocol using PVS. In *Proc. 15th Conference on the Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *LNCS*. Springer-Verlag,, 1995.
11. R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes, the automata theoretic approach*. Princeton Series in Computer Science. 1994.
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(1), 1995.
13. D. E. Long. *Model Checking, Abstraction, and Compositional Reasoning*. PhD thesis, Carnegie Mellon, 1993.
14. Z. Manna, A. Anuchitanukul, N. Bjøner, A. Browne, E. Chang, M. Colon, L. de Alfaro, H. Devarajan, H. Sipma, and T. Uribe. STeP : The Stanford Temporal Prover. Technical report, Stanford Univ., Stanford, CA, 1994.
15. Z. Manna and A. Pnueli. Completing the temporal picture. *Theoretical Computer Science*, 83(1):97–130, 1991.
16. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.
17. K.L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, Boston, 1993.
18. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. volume 1102 of *LNCS*, pages 411–414. Springer-Verlag, 1996.
19. S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, Feb. 1995.
20. J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int. Sym. on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1982.

This article was typeset using the LaTeX macro package with the LLNCS2E class.