Automated Deductive Verification of Parallel Systems

Hassen Saïdi

VERIMAG
Centre Equation, 2, Avenue de la Vignate, 38610 Gières
France
Email: saidi@imag.fr
URL: http://www.imag.fr/VERIMAG/PEOPLE/Hassen.Saidi/
Phone: (+33) 4.76.63.48.44, Fax: (+33) 4.76.63.48.50

Abstract

This paper describes the use of deductive methods for the verification of invariance properties of parallel systems. We show how a combination of proof rules, invariant generation techniques and abstraction techniques integrated to a theorem prover can be used effectively to prove invariants of systems given as a parallel composition of sequential processes with infinite data types. We present an implementation of these various techniques in our tool the Invariant-Checker. The Invariant-Checker is build as a front-end for the PVS theorem prover. The tool is an extension of the PVS specification language to handle the notion of transition systems describing the parallel composition of sequential processes. The PVS prover is also extended with a new proof scheme corresponding to the combination of our proof techniques.

keywords: *Computer-Aided verification, Combination of formal methods, theorem proving,* PVS*, invariants proofs, automatic generation of invariants, abstraction techniques.*

1 Introduction

Formal verification methods form a set of more or less automatic verification techniques for hardware and software computing. Since the use of formal methods is in constant growing, tools for the computer-aided verification supporting formal methods are needed and encouraged. A large variety of formal methods techniques is available, such as Model-checking, theorem proving, programs and circuits synthesis, refinement, abstraction techniques, simulation, testing, etc. Some of these techniques like model checking are used with a lot of success as they are fully automatic but are restricted to finite state systems, whereas others like theorem proving are generally considered labor intensive, which may sow doubts about their use in real life situations. The aim is then to combine several methods and techniques in order to complements them and to take advantage of the benefits of each of them.

Theorem proving has been applied to program verification with relatives success[1], specially systems like [5], [16], [9] and [6]. The use of theorem proving as a computer-aided verification technique consists in the validation of a set of logical formulas expressing the facts that the specification satisfies some properties. Higher order logic which is the specification language of most of the known theorem provers is a rich specification language allowing the definition of usual mathematical objects such as types, sets, functions and propositions. Mathematical objects can be constructed as set of previously declared objects or

[1] see [7] for many examples of the use of PVS.

as functions mapping previously declared objects to previously declared objects.

Other verification methods like model checking are used with success due to complete automatization. Model-checking consist in the exhaustive enumeration of all the states of a system and to check whether all theses states satisfy the desired property. While model-checking is restricted to the verification of finite state systems, theorem proving can deal with both finite and infinite state systems, but with loss of complete automatization and with permanent user guidance. Combination of Model-Checking and Theorem proving is one of the most promising combination of verification techniques since it took benefits from the automatization of model-checking and the powerful of theorem proving.

Many works have been achieved to integrate theorem proving to other verification techniques. In [12] Model checking is used to prove abstract models of specifications, while theorem proving is used to prove that the abstraction relation used is a "good" abstraction which preserve some desired properties. In [13] Theorem proving is used in order to prove the correctness of systems given as a parallel composition of processes by means of compositional rules embedded as inference rules in a theorem prover, while the correctness of each components is established using model checking. In [7] the PVS specification language is extended to support duration calculus syntax, and a set of proof rules dedicated to duration calculus logic are implemented. In [1], the PVS theorem prover is integrated to an environment for the verification of hardware specifications. It is used discharge verification conditions in order to test whether two specifications are equivalents modulo some equivalence relation.

In this work we are interested in the verification of invariance properties of parallel systems given as a parallel composition of sequential process with infinite data types. Such systems can not be verified by model checking. The verification techniques we consider are deductive model checking and abstraction techniques. Deductive model checking [10] consist in computing *inductive* invariants of a system by fix point calculation using a deductive proof rule. This proof rule reduces the problem of finding an appropriate invariant to the validity of first order formulas. Techniques for the automatic generation of invariants [2] are used to accelerate convergence. In the case where the proof rule fails to find an appropriate inductive invariant, abstraction techniques [11] are used to construct automatically an abstract state graph of the original system. The construction of the abstract state graph requires also to discharge first order formulas. An abstract state graph can be seen either as a finite state system on which model checking techniques can be applied, or as a more precise control structure of the original system.

The goal of this work is to show how to design in a fashion and an effective way tools for the computer-aided verification by combining techniques such as theorem proving, model-checking and abstraction in order to verify invariance properties of parallel system. We also show how theorem prover as a general framework can be a bootstrap for a combination of formal verification techniques. We choose for this purpose the PVS theorem prover developed at SRI to discharge the generated formulas.

This paper is organized as follow: in section 2, we present the principles of the verification techniques used such as deductive proof rules and abstraction techniques. In section 3 we present our approach to implement these methods and how they can be automated in an efficient way. In section 4, we describe our tool the Invariant-Checker implementing these methods. Finally in section 5, we sketch about our use of the Invariant-Checker and some examples we have studied.

2 Deductive Methods

We are interested in using deductive methods to verify invariance properties of parallel systems. This methods consist in applying a set of proof rules to establish that a property P expressed as a first order formulae is an invariant of a system S, that is, P remains true in every state of the system S, starting from its initial state.

2.1 Preliminary definitions

We consider systems which are parallel compositions of processes, where we consider parallel composition by interleaving and synchronization by shared variables as in Unity [3].

As a model of parallel systems we use transition systems. This notion is one of the simplest one used to describe states of a system and elementary steps (transitions) modifying its states. Also, every sequential process including classical algorithmic construction such as loops, tests and assignments can be translated into a transition system. Transitions are guarded commands where each guard is a boolean expression indicating the condition under which the transition is enabled.

Definition 1 (transition system)
A transition system S is a tuple $S = < \mathcal{V}, \mathcal{T} = \{\tau_1, \cdots, \tau_n\}, \mathcal{L}, Init >$, where $\mathcal{V}$ is a set of system variables, $\mathcal{T}$ a set of transitions or actions, $\mathcal{L}$ a set of control locations and $Init$ an initial state.

Each transition τ_i is a guarded command

$$l_i : \ guard_i \ \longrightarrow \ v_1 ::= e_1, \cdots, v_k ::= e_k \ \text{ goto } l_j$$

where $\{v_1, \cdots, v_k\} \subseteq \mathcal{V}$ and $\{l_i, l_j\} \subseteq \mathcal{L}$.
Each guard $guard_i$ is the guard of the transition τ_i. Each variable v_i is assigned with an expression e_i of a compatible type. Locations l_i and l_j are respectively the source and target location of the transition τ_i.

We also give the following informal definition of the weakest precondition and the strongest postcondition predicate transformers.

Definition 2 (predicate transformer)

1. *$\widetilde{pre}\{\tau_i\}(P)$ is the weakest precondition of the predicate P with respect the transition τ_i and is defined as the set of states from which it is not possible to reach via a transition τ_i a state satisfying $\neg P$.*
   $\widetilde{pre}\{\tau_i\}(P)$ is defined as the predicate

   $$guard_i \Rightarrow P[e_1/v_1, \cdots, e_k/v_k]$$

2. *$post\{\tau_i\}(P)$ is the strongest postcondition of the predicate P with respect the transition τ_i and is defined as the set of states which can not be reached from a state satisfying $\neg P$ via a transition τ_i.*

2.2 Invariance proof rule

We use the simple invariance proof rule (Figure 1) which states that a predicate P is an invariant of a system S, if it exists a predicate P' stronger than P, such that P' holds at the initial state and is preserved by all the transitions of the system.

$$\frac{\begin{array}{ll} \models & Init \Rightarrow P' \\ \models & P' \Rightarrow P \\ \models & \forall \tau_i.\ P'\{\tau_i\}P' \end{array}}{Prog \ \models \ \Box P}$$

Figure 1: Invariance proof rule

the predicate $\forall \tau_i.\ P\{\tau_i\}P$ can be expressed using the predicate transformer $\widetilde{pre}$ and $post$ as follow:

- $\bigwedge_{i=1}^{n}(P \Rightarrow \widetilde{pre}\{\tau_i\}(P))$
- $\bigwedge_{i=1}^{n}(post\{\tau_i\}(P) \Rightarrow P)$

We propose to use the predicate transformer $\widetilde{pre}$, since its application does not introduce existential quantifications which are not easy to eliminate.

2.3 Strengthening techniques

This rule does not tell how to find an appropriate P', but many techniques are used to find a P' by refining the predicate P. The most used one is the strengthening method which consists in finding the greatest inductive invariant (implied by all the other inductive invariants of S) implying the predicate P.

That is to find in a model checking like manner the greatest solution of the equation:

$$\forall \tau_i.\ P'\{\tau_i\}P'$$

The strengthening method consists in instantiating P' successively by a sequence $P_0, \cdots, P_n$ of predicates such that:

- $P_0 = P$
- $P_{k+1} = P_k \bigwedge_{i=1}^{n} \widetilde{pre}\{\tau_i\}(P_k)$

At each k^{th} step the second premise of the proof rule is trivially true, and only the two others premises are checked, that is the predicates:

$$k^{th} \text{ step} \begin{cases} Init \Rightarrow P_k \\ P_k \Rightarrow \bigwedge_{i=1}^{n} \widetilde{pre}\{\tau_i\}(P_k) \end{cases}$$

In fact it is not necessary to consider at each new instantiation the predicate $P_k \bigwedge_{i=1}^{n} \widetilde{pre}\{\tau_i\}(P_k)$, but only the predicate $P_k \bigwedge_{j=1}^{m} \widetilde{pre}\{\tau_j\}(P_k)$ where the transitions τ_j are those for which the verification condition $P_k\{\tau_j\}P_k$ is not proved valid.

Figure 2 illustrates this fix point calculation known as the backward analysis using the predicate transformer $\widetilde{pre}$. The dual analysis method is the Forward analysis which consist in the computation of the reachable states from the initial state of the system S iteratively as a fix point using the predicate transformer $post$, and than to check whether the reachable predicate $Reach$ implies the predicate P.

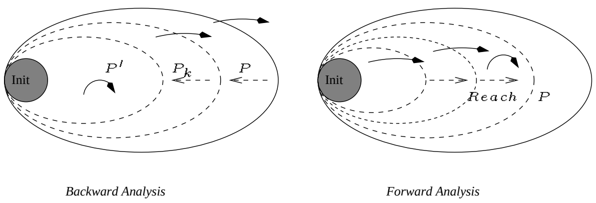

Figure 2: Fixpoint calculation

These two dual techniques may not terminate when the state space of the analyzed system is infinite. Moreover, when one may terminate the other may not. Also, in the forward analysis the generated VCs have no more quantifiers than the predicate P, however in the forward analysis, the generated VCs are existentially quantified formulas. This makes the use of forward analysis technique more difficult than the backward technique due to the quantifier elimination problem.

2.4 Refined strengthening techniques

The drawback of the proposed strengthening techniques is the growing size of the predicates P_k computed at each iteration. To avoid this problem, other strengthening rules have been proposed. In [12] it is proposed to instantiating P' successively by a sequence $P_0, \cdots, P_n$ of predicates such that:

- $P_1 = P$
- $P_{k+1} = \bigwedge_{j=1}^{m}(P_k \Rightarrow \widetilde{pre}\{\tau_j\}(P_k))$

The idea behind the use of this rule is the following: If the proof of the validity of the verification condition $P_k \Rightarrow \widetilde{pre}\{\tau_i\}(P_k)$ fails, it is tried to prove the validity of the simplified form of this VC in the program and not in every model. However, this rule does not accelerate convergence any more. This is due to the fact that at each iteration not all the state from which it is possible to reach a state satisfying $\neg P$ are eliminated.

In [2] a refined strengthening technique is proposed. It consist in using the following rules:
Instantiate P' successively by

- $P_1 = P$
- $P_{k+1} = P_k \wedge Q$

where Q is the "simplified" form of $P_k \Rightarrow \widetilde{pre}\{\tau_i\}(P_k)$. This rule is equivalent to the first one since the predicate P_{k+1} are equivalents in both techniques:
$P_k \wedge (P_k \Rightarrow \widetilde{pre}\{\tau_i\}(P_k)) \;\Leftrightarrow\; P_k \wedge \widetilde{pre}\{\tau_i\}(P_k)$

This method is used to avoid the size increasing of expressions due to the multiple applications of the strengthening techniques. However in practice it does not give interesting results. This is essentially due to the proof strategy used to have a "simplified" form of the considered VC. In fact this strategy should not create additional subgoals. Furthermore since the result of the application of this strategy will be considered in the next step, this strategy should not change significantly the current goal to prove by increasing its size. Thus, proof strategies such as rewriting, induction may not be used. Also, if a non appropriate proof strategy is used such as rewriting a bad instance of a function symbol or an induction on a non relevant variable may create non relevant subgoals which are propagated along the iteration process.

2.5 Automatic generation of invariants

The application of the proof rule in the case of infinite state systems may lead to an infinite computation. Also, when the computation terminates, convergence is often slow. Convergence can be accelerated using some already proved invariants of the system. Let $\mathcal{I}$ be a conjunction of such invariants. The verification conditions generated by the proof rule can be weakened by $\mathcal{I}$, that is by generating at each k^{th} step the following VCs:

$$k^{th} \text{ step} \left\{ \begin{array}{l} \mathcal{I} \wedge Init \Rightarrow P_k \\ \mathcal{I} \wedge P_k \Rightarrow \bigwedge_{i=1}^{n} \widetilde{pre}\{\tau_i\}(P_k) \end{array} \right.$$

In [2] we present techniques for the automatic generation of inductive invariants by static analysis. These techniques allows us to generate invariants of a sequential process as well as for a system given as a parallel composition of sequential processes. The invariant $\mathcal{I}$ represents an upper approximation of the set $Reach$ of reachable states. The proof rule allows us to combine backward and forward analysis, since the backward analysis considers only states satisfying $\mathcal{I}$.

2.6 Abstraction techniques

In the case where the combination of the proof rule and the invariant generation techniques fails to prove the property, it is useful to construct a corresponding finite state system called an abstraction where the property we want to verify can be easily checked by model checking techniques. Most abstraction techniques used in program verification are based on abstract interpretation of programs [4]. The main concern of abstract interpretation is the construction of descriptions of concrete systems by "mimicking" the effect of concrete operations with suitable operations defined over descriptions. The construction of an abstract system requires the definition of an abstract state space, where each abstract state represent a possibly infinite set of concrete states.

In [11] we define a particular abstraction scheme, where an abstract state is defined as a valuation of a set of predicates $\varphi_1, ..., \varphi_\ell$ on program variables and the valuation of the concrete control variables. The construction process starts by defining the initial abstract state of the system. This can be done by checking the possible valuations of each predicate φ_i in the concrete initial state. The successors of an abstract state are computed as follows:

1. Select all the possible transitions τ_i from the abstract state.
2. For each such transition compute the valuation of each predicate φ_i after the execution of τ_i , that is whether φ_i or $\neg\varphi_i$ is a postcondition. The successor s' of a valuation s is then equal to:

$$\begin{array}{lll} false & \text{if} & s \Rightarrow \neg guard_i \\ \forall i.\varphi_i = \left\{ \begin{array}{ll} \mathtt{t} & \text{if } s \Rightarrow \widetilde{pre}\{\tau_i\}(\varphi_i) \\ \mathtt{f} & \text{if } s \Rightarrow \widetilde{pre}\{\tau_i\}(\neg\varphi_i) \\ \mathtt{T} & \text{otherwise} \end{array} \right\} & \text{otherwise} & \end{array}$$

In the case where $\mathtt{T}$ is a postcondition of a predicate φ_i, two successors are created corresponding to the case where φ_i or $\neg\varphi_i$ is a postcondition. $\mathtt{t}$ and $\mathtt{f}$ are the boolean values true and false. The successor *false* is interpreted as the empty set, that is no successor exists for the considered abstract state via the transition τ_i.

Using these rule an abstract state graph can be constructed. This graph is an upper approximation of the global reachable state of a system and can be used to prove any property involving the predicate

$\varphi_1, ..., \varphi_\ell$ using model-checking techniques. Also this state graph gives a more precise idea about the behavior of a system given as a parallel composition of processes by eliminating non reachable global control configuration. Thus from a practical point of view it is useful to include in the set $\varphi_1, ..., \varphi_\ell$ of predicates the literals appearing in the guards of each transition of the system and the literals appearing in the property we want to verify. The presented proof rules of the previous sections can then be applied on the system defined by this abstract state graph which is a system with only one component. Notice that our abstraction technique is also based on the proof of a set of first order implications, which can also be weakened by the invariant $\mathcal{I}$. This allows us to combine proof rules, invariant generation and abstraction techniques in a same framework based on discharging verification conditions.

3 Automated deductive methods

We propose to implement the proof rule of figure 1, the techniques for the automatic generation of invariants and the abstraction techniques we have defined. A tool implementing such techniques must have two essential features: a powerful specification language allowing the describtion of transition systems with infinite data types, a verification condition generator for each proposed techniques and finally a proof engine to discharge all the generated VCs. A theorem prover seems to be an appropriate tool for our verification techniques. We choose the PVS theorem prover for this purpose.

3.1 PVS

PVS is an environment for writing specifications and developing 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 types, 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 `Type`, variable, constant, function and `Formula` declarations. The theorem prover or proof checker implements a set of powerful tactics with a mechanism for composing them into proof tacticals. Some of these tactics such as *assert* and and *bddsimp* invoke efficient decision procedures for arithmetic and boolean expressions. PVS has emacs as user interface.

3.2 A first approach

We investigated two approaches to implement the described techniques using the PVS theorem prover. A first approach consists in encoding the proof rule in the specification language of the prover. This approach is the "classical" one adopted when a theorem prover is used in formal verification.

Using the PVS specification language, a system `S` can be described in a PVS theory. Given a predicate `P`, we express the invariance property `S`$\models \Box$`P` as a theorem:

```
th : THEOREM
    is_invariant?(S,P)
```

where `is_invariant?` is a recursive function defined on the structure of the system `S`.

The strengthening techniques can also be defined using a recursive functions parameterized by the maximal number of iteration allowed.

We found this way of proving invariants using a theorem prover not satisfactory for many reasons:

1. The syntax is not adapted to describe systems easily.
2. The proofs are too long, which is due to many unnecessary rewriting steps.
3. A lot of user interaction is required and few automatization is provided.

3.3 Our approach

To deal with the weakness reported previously, we choose to built a verification system using the PVS theorem prover as a back-end where the verification conditions are generated automatically and submitted to the PVS prover in a fully automatic way. The objectives of this approach are the following:

- Full automatization whenever it is possible.
- Use all the power of the PVS specification language and the PVS prover.
- Extend the PVS verification system with new features, including our particular proof scheme.

Following these objectives, we have designed our tool the Invariant Checker [10, 19]. The system is designed as a front-end for the PVS [7] 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 appears at two different levels: the PVS specification language is extended with the notion of a `system`, or a parallel composition of transition systems. The PVS prover is also extended with the deductive proof rule we use.

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 interaction. The objective of our tool is to provide more automatization and less user intervention using a set of features. The principals of our tool are illustrated in Figure 3. To extend the prover with our particular proof schemes and techniques, a front end is required, where VCs are only submitted to the prover without user interaction. Notice that this extension scheme is general and can be applied to any verification scheme where discharging first order assertions is required. Of course the verification process can be completely automatized under the condition that the generated VCs are decidable predicates.

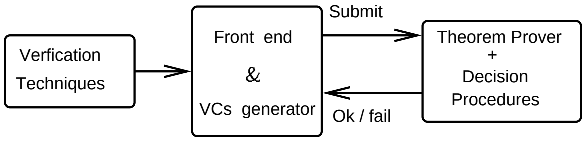

Figure 3: Design principal

4 A verification tool: The Invariant-Checker

4.1 Design Philosophy

As reported previously, the objective of the design of such tool is the complete automatization of the verification process whenever it is possible, and also to take advantage of the PVS specification language and its rich type system and the PVS prover and its powerful proof strategies.

In order to be efficient it is only necessary to interact automatically with PVS, and then to generate VCs in a format acceptable by the prover. Systems can then be described in any other specification language, but in order to have an efficient integration with PVS we decided to use the PVS specification language to describe systems. However it was necessary to extend this language to support the definition of transition systems in a syntax close to the one defined in section 2.2. Thus, the manipulated variables can be defined as PVS variables and the manipulated expressions as PVS expressions. The typechecker is also extended in order to parse and to typecheck specifications written in this syntax. This allow us to use all the power of the PVS specification language with a suitable syntax. It is also necessary to consider a new notion of proof status of a system, that is which are the assumed invariants (axioms) of the considered system and which are the already proved invariants and the non yet proved ones. The following correspondence scheme gives the design philosophy of the tool where Pvs is extended with the notion of `system`, `invariant` and `invariant proof` :

PVS		*Invariant − Checker*
`Theory`	$\longleftrightarrow$	`System`
`Theorem`	$\longleftrightarrow$	`Invariant`
`Proof`	$\longleftrightarrow$	`Invariant proof`

4.2 Features

Figure 4 describes the general scheme of the Invariant-Checker. In this section we explain the role of each component of the tool and how the proof process is organized.

Syntax: Systems can be described in a Simple Programming Language (SPL), close to the one used in [17], but with the rich data types and expressions definition mechanism available in Pvs. Our SPL language includes common algorithmic constructions such as single and multiple assignments statements, conditionals If-Then-Else and loop statements. We also allow parallel composition by interleaving and synchronization by shared variables as in Unity [3]. Systems described in SPL are translated automatically into 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 a compatible type. Also, it is possible to import any defined Pvs theory. Figure 5 shows a simple example of a system given as a parallel composition of two processes Process_1 and Process_2. this system is knows as the bakery system. It guarantees the access to the critical section 3 in a mutual exclusion between the two processes. The example is given in the concrete syntax allowed for transition systems[2]. **Req_i**, **In_i**, **Out_i** are labels of the transitions of Process_i.

[2] The example is presented in the generated LaTeX format for guarded commands. Variables $pc1$ and $pc2$ are respectively the control variable of Process_1 and Process_2.

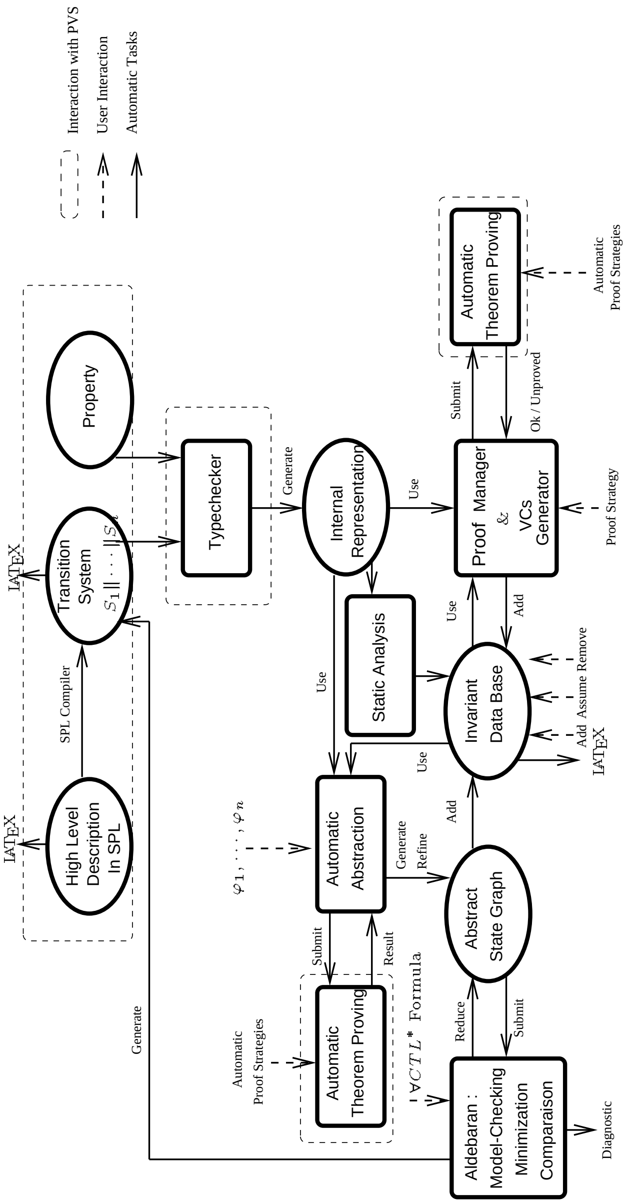

Figure 4: The Invariant Checker Architecture

```
bakery : SYSTEM
 BEGIN

   comp1 : PROGRAM
      y1 : VAR nat

     BEGIN
       Req_1
       1: TRUE                  → y1 := y2+1    GOTO 2
       In_1
       2: y2 = 0 ∨ y1 ≤ y2      →   SKIP        GOTO 3
       Out_1
       3: TRUE                  → y1 := 0       GOTO 1
     END comp1

 ||

   comp2 : PROGRAM
      y2 : VAR nat

     BEGIN
       Req_2
       1: TRUE                  → y2 := y1+1    GOTO 2
       In_2
       2: y1 = 0 ∨ y2 < y1      →   SKIP        GOTO 3
       Out_2
       3: TRUE                  → y2 := 0       GOTO 1
     END comp2

 INITIALLY  :  y1 = 0 ∧ y2 = 0 ∧
               pc2 = 1 ∧ pc1 = 1

END bakery
```

Figure 5: Bakery transition system

Typechecking: Typechecking concerns only systems described as a parallel composition of guarded commands since the translation of an SPL system into a transition system is only a syntactical transformation. 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`. Since the manipulated expressions are PVS expressions, typechecking a system may lead to the generation of type correctness conditions (TCCs). In PVS a generated TCC must be proved before proving any assertions of the corresponding theory. This is due to the fact that a theory is a logical specification and a generated TCC indicates some correctness criterion of the specification.

Our description of systems can be seen as a description at a hight (specification) or at a low (implementation) level. They are not logical specification. Therefore the generated TCCs are predicates that guarantee "absence of run-time errors" (division by zero, application of the tail function to the empty list...). Moreover, these predicates have to be proved as invariants and not as valid formulas as it is the case of TCCs generated for a PVS theory. It is just required that they are valid in the model generated by the system and not in every model. In the case where they cannot be proved, absence of run-time errors is not guaranteed but this does not affect the proof of any invariant.

Consider the following transition where x is declared as a positive integer:

$$i: \ \mathrm{x} > 0 \quad \longrightarrow \quad x := x - 1 \ \ \text{GOTO } j$$

The expression $x-1$ is assigned to x and should be of a compatible type, that is $x - 1 \geq 0$. This constraint is not required to be true at any moment of the execution of the program but only when a transition is activated, that is only when the program counter equals i and the guard $\mathrm{x} > 0$ is true. Thus the following TCC is generated:

Tcc_i : OBLIGATION
$$pc = i \ \wedge \ x > 0 \ \Rightarrow \ x - 1 \geq 0$$

this TCC is trivially valid and can be discharged using the PVS arithmetics decision procedures.

Of course, one can encode the semantics of guarded commands within the PVS specification language, and then use the "classical" PVS typechecker to generate such TCCs. However they must be proved as invariants. The generated TCCs can be used in this case to prove other invariants of the considered system, and can also be proved using some other invariants, which is not the case of TCCs generated for a theory, which must be proved independently of any other assertions.

Internal representation: PVS is implemented in LISP. every object manipulated in PVS such as a theory, a theorem or a proof is represented as an instance of a predefined object class. We have defined for all the abject manipulated in the Invariant-Checker an internal representation which is also a class such as: `system`, `process`, `transition`, `invariant` and `proof`. An important aspect in these structures is that they are independent from the PVS internal structures and make our implementation independent from the possible changes in the PVS internal representation. However the expression manipulated and the verification condition generated are represented as PVS expressions and PVS obligations. This is necessary for the automatic interaction with the prover we want to achieve.

Static analysis As it is shown, any known auxiliary invariant $\mathcal{I}$ of a system can be used to accelerate convergence. Static analysis consists in a set of techniques for the automatic generation of such invariants. Theses techniques are based on propagation of guards and assignments through program control points. two kinds of invariants are generated: local invariants for each process and global invariants of the parallel composition. The techniques we used are described in [2]. These invariants are added automatically to the invariant database. For example, the following global invariants are automatically generated for the bakery system:

$\mathrm{pc1} = 1 \;\Rightarrow\; y_1 = 0$

$\mathrm{pc1} = 2 \;\Rightarrow\; \exists\,(y_2):\; y_1 = y_2 + 1$

$\mathrm{pc1} = 3 \;\Rightarrow\; \exists\,(y_2):\; y_1 = y_2 + 1$

$\mathrm{pc2} = 1 \;\Rightarrow\; y_2 = 0$

$\mathrm{pc2} = 2 \;\Rightarrow\; \exists\,(y_1):\; y_2 = y_1 + 1$

$\mathrm{pc2} = 3 \;\Rightarrow\; \exists\,(y_1):\; y_2 = y_1 + 1$

$\mathrm{pc1} = 2 \wedge \mathrm{pc2} = 3 \;\Rightarrow\; y_1 = y_2 + 1 \;\vee\; y_1 = 0 \;\vee\; y_2 < y_1$

$\mathrm{pc1} = 1 \wedge \mathrm{pc2} = 3 \;\Rightarrow\; y_1 = 0 \;\vee\; y_2 \;<\; y_1$

$\mathrm{pc1} = 3 \wedge \mathrm{pc2} = 2 \;\Rightarrow\; y_2 = 0 \;\vee\; y_1 \leq y_2 \;\vee\; y_2 = y_1 + 1$

$\mathrm{pc1} = 2 \wedge \mathrm{pc2} = 2 \;\Rightarrow\; y_1 = y_2 + 1 \;\vee\; y_2 = y_1 + 1$

$\mathrm{pc1} = 1 \wedge \mathrm{pc2} = 2 \;\Rightarrow\; y_1 = 0 \;\vee\; y_2 = y_1 + 1$

$\mathrm{pc1} = 3 \wedge \mathrm{pc2} = 1 \;\Rightarrow\; y_2 = 0 \;\vee\; y_1 \leq y_2$

$\mathrm{pc1} = 2 \wedge \mathrm{pc2} = 1 \;\Rightarrow\; y_1 = y_2 + 1 \;\vee\; y_2 = 0$

Proof manager: The proof manager is the module which coordinates the proof process by applying the proof commands invoked by the used. This module controls the number of iterations, controls the validity of each VCs and invokes the proof rule whenever it is necessary.

VC generator: Each application of a proof rule leads to the generation of a set of first order formulas to discharge. For a predicate P and a system S only non trivial VCs are generated. For example no VC is generated when a transition does not affect the free variables of P. Also, for every generated VC, a set of relevant generated invariants is automatically selected to achieve the proof.

Proof session: A proof session starts with type-checking 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 invariants during the static analysis . 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 generated and submitted automatically 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. Non provable assertions are considered as non valid.

Automatic theorem proving: The generated VCs are submitted automatically 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 predefined PVS strategy and user defined ones. We have defined a set of powerful proof strategies which allows us to discharge non trivial goals, especially using heuristics for the automatic induction.

Invariants data base The invariant data base contains the invariants generated during the static analysis. The user can always enrich the invariant data base with some invariants. With each invariant a status is associated 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 The abstraction technique we use in our tool are those described in [11] and explained in section 2.6. 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.
- It can be minimized modulo strong equivalence using the ALDÉBARAN tool [8]. 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.

The abstract state graph of figure 6 is obtained using the predicates appearing in the guards of the bakery transition system, that is the predicates $y_1 = 0$, $y_2 = 0$, $y_1 \leq y_2$ and $y_2 < y_1$. The control configuration $pc1 = 3 \wedge pc2 = 3$ indicating violation of the mutual exclusion property is not reachable.

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.

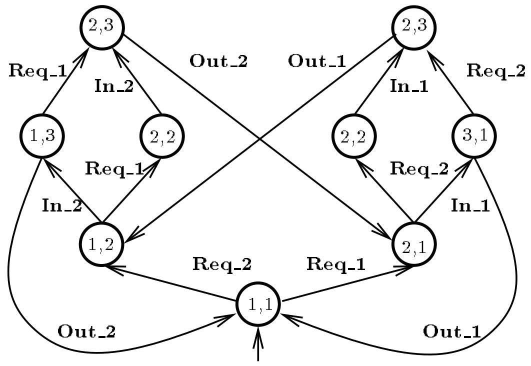

Figure 6: Abstract state graph for the bakery system

5 Experiments with the Invariant-Checker

Using the Invariant Checker we verified various classical mutual exclusion algorithms [17], a read and write buffer using complex data types [10]. The mutual exclusion property of the bakery system has been proved using the proof rule and using abstraction techniques in a fully automatic way. The use of abstraction techniques allow us to prove in a fully automatic way an alternating bit and a bounded retransmission protocol [11] developed by Philips and used in one of its commercial products. This example has been already proved using theorem proving techniques, but a big amount of user interaction has been necessary to provide powerful enough auxiliary invariants. We have also used successfully the Invariant Checker tool for the verification of parameterized networks following the techniques described in [15]. Additional information can be found on our experiences using the tool in the Invariant Checker home page [14].

6 Conclusion

This paper describes a tool supporting the automated formal verification of invariance properties of parallel processes. It shows how the use of theorem proving techniques can be done in an effective and efficient way with a combination of other verification techniques such as model checking and abstraction. It shows that automatization of deductive methods can be effective and leads to effective proof tools. This work can be extended to other verification techniques which can be combined with theorem proving. The first step of the combination process is to allow different specification languages in the same tool. The second one is to allow different verification techniques. The verification process can be than a combination of such techniques and can be decomposed into different steps. Our design philosophy is a consequence of the following remarks: Effective verification systems requires generally an enough powerful specification language and adequate and powerful verification techniques, but only the combination of formal verification techniques and tools can lead to effective use of formal methods. We found that a theorem prover can play a crucial role, by bootstrapping the combination process. The different specification languages can be formally defined using the PVS specification language. The PVS prover can then be used partially or totally in the verification process. It can also be used to prove that the combination process is correct.

References

[1] H. Barringer, G. Gough, B. Monahan, and A. Williams. The ELLA Verification Environment: A Tutorial Introduction. Technical Report UMCS CS-94-12-2. University of Manchester.

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

[3] K. M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, 1988.

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis

of programs by construction or approximation of fixpoints. In *4th POPL*, January 1977.

[5] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[6] C. Cornes, J. Courant, J. C. Filliâtre, G. Huet, P. Manoury, C. Paulin-Mohring, C. Muñoz, C. Murthy, C. Parent, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual. Version 5.10. Technical Report, I.N.R.I.A, February 1995.

[7] 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.

[8] 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'96*. LNCS 1102, 1996.

[9] M. J. C. Gordon and T. F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.

[10] S. Graf and H. Saïdi. Verifying invariants using theorem proving. In *Conference on Computer Aided Verification (CAV'96)*, Rutgers University, New Jersey, July 1996. LNCS 1102.

[11] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. Accepted for publication. In *CAV'97*.

[12] K. Havelund, and N. Shankar. Experiments In Theorem Proving and Model Checking for Protocol Verification. In *the proceedings of Formal Methods Europe 96*. 1996.

[13] H. Hungar. Combining Model checking and Theorem proving to Verify Parallel Processes. In *Computer-Aided Verification, CAV'93*, LNCS 697, Elounda, June 1993.

[14] The Invariant-Checker home page. `http://www.imag.fr/VERIMAG/PEOPLE /Hassen.Saidi/INVARIANT-CHECKER/`

[15] D.Lesens and H. Saïdi. Automatic Verification of Parameterized Networks of Processes by Abstraction. In *Proceedings of the 2nd International Workshop on the Verification of Infinite State Systems (INFINITY'97)*, Bologna, Italy, July 1997.

[16] S. Owre, N. Shankar, and J. M. Rushby. The PVS Specification Language. Computer Science Laboratory, SRI International, February 1993.

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

[18] J. M. Rushby. Model Checking and Other Ways of Automating Formal Methods. *TPosition paper for panel on Model Checking for Concurrent Programs Software Quality Week*, San Francisco, May/June 1995.

[19] H. Saïdi. A Tool for Proving Invariance Properties of Concurrent Systems Automatically. In *Proceedings of 2nd International Workshop, Tools and Algorithms for the Construction and Analysis of Systems*, Passau, Germany, March 1996. LNCS 1056, Springer-Verlag.

[20] H. Saïdi. . The Invariant-Checker: Deductive Verification of Reactive Systems. Accepted for publication. In *CAV'97*.