Verifying Invariants Using Theorem Proving

Susanne Graf and Hassen Saïdi

Verimag *
{graf,saidi}@imag.fr

Abstract. Our goal is to use a theorem prover in order to verify invariance properties of distributed systems in a "model checking like" manner. A system S is described by a set of sequential components, each one given by a transition relation and a predicate $Init$ defining the set of initial states. In order to verify that P is an invariant of S, we try to compute, in a model checking like manner, the weakest predicate P' stronger than P and weaker than $Init$ which is an *inductive* invariant, that is, whenever P' is true in some state, then P' remains true after the execution of any possible transition. The fact that P is an invariant can be expressed by a set of predicates (having no more quantifiers than P) on the set of program variables, one for every possible transition of the system. In order to prove these predicates, we use either automatic or assisted theorem proving depending on their nature.
We show in this paper how this can be done in an efficient way using the Prototype Verification System PVS. A tool implementing this verification method is presented.

1 Introduction

Using a theorem prover to do model checking is not a new idea[2]. Theorem proving has been used successfully for the verification of temporal logic formulas on programs, specially systems like [BM88], [OSR93a][3], [GM93] and [CCF+95].

In most of these approaches, it is mainly emphasized how to define the syntax of a specification formalism and its semantics (in terms of sets of computations) as well as the satisfaction of temporal logic formulas on computations. Then, a system S satisfies a property f if every computation of S satisfies f. In general, not much is told about *how to verify* the obtained formulas.

[RSS95] explains how model checking (for *finite* state systems) is implemented in PVS as a tactic (which consists in transforming the model checking problem into a decidable μ-calculus formula and to run a decision procedure on this formula). In [RSS95], [DF95] and [HS96] model checking is used to prove abstract descriptions of systems, while "ordinary" theorem proving is used to show the

* Verimag is a joint laboratory of CNRS, Institut National Polytechnique de Grenoble, Université J. Fourier and Verilog SA associated with IMAG.

[2] See [RSS95], where a set of combination attempts are mentioned.

[3] see [CLN+95] for many examples of the use of PVS.

correctness of this abstract description with respect to a more concrete (in general infinite state) description. In [Hun93] it is proposed to verify the correctness of each component using model checking, and then to deduce the correctness of the composed system by means of compositional rules embedded as inference rules in a theorem prover. [BGMW94] describes an integration of the PVS theorem prover in an environment for the verification of hardware specification. It is used for discharging verification conditions expressing the fact that a specification simulates another.

1.1 Our approach

Our intention is not to verify arbitrary temporal logic formulas, but particular formula schematas corresponding to useful property classes. In order to prove that a system S satisfies a property expressed by a temporal formula f, we do not use its semantics, but a proof rule generating a set of first order logic formulas (without temporal modalities and without new quantifiers) such that their validity is sufficient to prove that S satisfies f. Here, we mention only safety properties expressible by formulas of the form "$\Box P$" (invariants) or "$\Box(P \Rightarrow P_1 \mathcal{W} P_2)$", where P, P_1, P_2 are predicates[4].

For example, in order to prove that P is an invariant of S ($S \models \Box P$) — where S is defined by a set T of transitions and a predicate $Init$ defining the set of initial states — it is necessary and sufficient to find a predicate P' weaker than $Init$ and stronger than P which is an inductive invariant, that is P' is preserved by any computational step of S, i.e $P' \Rightarrow \widetilde{pre}[\tau](P')$[5] is valid for each transition τ of T. Model checking consists in computing iteratively the weakest predicate satisfying the implication $Q \Rightarrow \widetilde{pre}[T](Q)$ starting with $Q_0 = P$ and taking $Q_{i+1} = Q_i \wedge \widetilde{pre}[T](Q_i)$ that is by strengthening the proposed solution at each step. This method can be completely automatized under the condition that the above predicates are decidable. However, in the case of infinite state systems convergence is not guaranteed, and in real life systems with this very simple tactic, convergence is too slow, anyway. Convergence can be accelerated by replacing the predicate transformers $\widetilde{pre}[\tau]$ by some (lower) approximation or by using structural invariants (see Section 4.3) extracted from the program obtained by constant propagation, variable domain information, etc. Theorem proving (or an appropriate decision procedure) is used for establishing $Q_i \Rightarrow Q_{i+1}$ that is for verifying that a fixed point has been reached.

1.2 Related work

Tools like STeP [MAB+94], TPVS [BLUP94] and CAVEAT [GR95] use this technique. In CAVEAT systematic strengthening of invariants is not foreseen. STeP

[4] in [MP95] many such schemata and corresponding verification rules are presented for which we will implement strategies in the future

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

provides a lot of automatization and implements most of the rules presented in [MP95].

In [HS96], a new strengthening method has been proposed, in order to avoid the fast growth of the formulas due to the systematic strengthening: suppose that Q_i is not inductive for some transition τ, that is, the proof of the goal $Q_i \Rightarrow \widetilde{pre}[\tau](Q_i)$ does not reduce to *true* but to some formula R. Then, instead of checking in the next step the formula $Q_{i+1} = Q_i \wedge \widetilde{pre}[\tau](Q_i)$, it is proposed to check R (which is often simpler) for invariance. However, this method does not accelerate convergence.
This paper is organized as follows: in Section 2, we recall some general ideas concerning theorem proving and give a small overview on PVS. In Section 3, it is explained how to define our method completely within PVS and also, why we have abandoned this approach. Finally, in Section 4, we give a short presentation of our tool which acts like an interface with PVS. In Section 5, we demonstrate our method and tool on two examples: a finite state program implementing a mutual exclusion algorithm, and an infinite state program implementing a simple buffer using lists as data type.

2 The theorem proving paradigm

Theorem proving is the paradigm of developing and verifying mechanically mathematical proofs. The specification languages used (higher order logic) allow to define usual mathematical objects such as sets, functions, propositions and even proofs[6], and can be generally understood as a mixture of predicate calculus, recursive definitions à la ML and inductively defined types. These languages are strong enough to model systems and express properties on them. Theorem provers provide an interactive environment for developing mathematical proofs using a set of tactics (elementary proof steps) and tacticals (combination of tactics). Possible tactics are implementations of either a deduction rule, rewriting rule, induction scheme or a decision procedure.

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 PVS theorem prover implements a set of powerful tactics with a mechanism for composing them into proof tacticals. The tactics available are combinations of deduction rules and decision procedures. Some of

[6] See [CCF+95] for this purpose.

these tactics such as *assert* and *bddsimp* invoke efficient decision procedures for arithmetic and boolean expressions. PVS has emacs as user interface.

3 Specification and verification within PVS

One of the drawbacks when using theorem provers is the tedious encoding of semantics and writing of specifications. In Coq [CCF+95], grammar extension is allowed which makes specifications easier to write and to read.

In PVS, this technique can be generalized to allow user-defined specification syntax (e.g. [Saï95]). The defined specification syntax can be a combination of the PVS specification syntax and user specification syntax since it can be constructed using non-terminals of the PVS grammar.

To prove that a predicate is an invariant of a system is usually done by embedding the semantics of transition systems and the notion of invariance of a property in the specification language of a theorem prover. In PVS, this can be done by means of the following definitions:

```
Program [State : TYPE] : THEORY
BEGIN
  Action : TYPE = [guard:bool, assignments:State]
  System : TYPE = [vars:State, acts:list[Action], init:bool]

  is-inductive? (S:System, P:Pred[State]) : bool =
    (init(S)  =>  P(vars(S)))          AND
    (P(vars(S)) => WPC-System(acts(S),P))

  WPC-System(L:list[Action], P:Pred[State]) : RECURSIVE =
    CASES L of
      null           : TRUE
      cons(act,rest): WPC-Action(act,S) AND WPC_System(rest,P)
    END CASES

  WPC-Action(act:Action, P:Pred[State]) : bool =
    guard(act) => P(assignments(act))
END Program
```

The PVS theory named `Program` is parameterized by the type `State` defining the tuple type of the state vector, that means, its i^{th} component defines the type of the i^{th} state variable. `System` is given as list of actions, where `Action` is defined as a record type with two fields, a guard and an assignment. `guard` is the condition under which the given action is activated. `assignments` is a tuple of type `State` representing the new value of the state vector after the execution of the given action. The predicate `is-inductive?` taking as arguments a system S and a predicate P, yields the result `true` if P is an inductive invariant of S.

In order to show that P is an invariant of S, we have to prove the following obligation:

```
prove-invariant : OBLIGATION
 EXISTS (P':PRED[State]):
  (FORALL (t: State) : (P'(t) => P(t)) AND is-inductive?(S,P'))
```

This proof obligation does not tell us how to find a satisfactory predicate `P'`. This is the reason why we use the iterative computation described in Section 1.1 which replaces the above (second order) obligation by an (infinite) suite of first order obligations such that the proof of any obligation of this suites validates the initial obligation.

But we found that such an embedding of the semantics of transition systems directly in PVS is still not satisfactory for the verification of large systems. Writing programs is tedious, proofs are very slow since much time is lost in expanding the definitions of `is-inductive?`, `WPC-System` and `WPC-Action`. We also found that we cannot perform static analysis on programs written in this way.

Therefore, we prefered to describe programs in a more natural way and not to translate them into a PVS theory, but just to generate automatically proof obligations equivalent to `is-inductive?(S,P')` and to submit them to the PVS proof checker.

4 A verification tool

Figure 1 shows the architecture of our tool for computer-aided verification. We first present how systems are described in this tool and how the verification process works. We also show how both specification and verification are connected with the PVS system.

4.1 A specification formalism

In our tool, systems are discribed in a formalism close to Dijkstra's language of guarded commands. In fact, a system is defined as a set of components where each component is given by a set of transitions defining conditional data transformations, where program variables are of any data type definable in PVS and allowed value expressions are any expressions definable in PVS. The grammar defining this specification formalism is the following[7]:

system ⇒	*id_system* [`PARAMETER` *id*] : `SYSTEM`
	`BEGIN`
	⟨ *pvs_declarations* ⟩
	`BEGIN`
	⟨ *sys_components* ⟩
	`END`
	`INITIALLY` : ⟨ *pvs_boolean_formula* ⟩
	`END` *id_system*

[7] This grammar is presented using the conventions of [OSR93b]

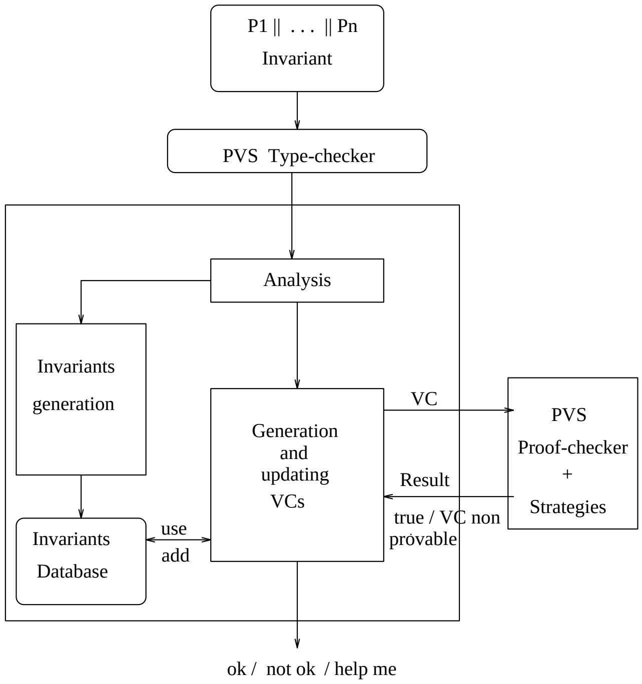

Fig. 1. Tool architecture

sys_components	⇒	⟨ *program* ⟩ | ⟨ *program* ⟩ `||` ⟨ *sys_components* ⟩
program	⇒	⟨ *action* ⟩+ | ⟨ *named_program* ⟩
named_program	⇒	*id_program* : `PROGRAM`
		`BEGIN`
		⟨ *pvs_declarations* ⟩
		`BEGIN`
		⟨ *action* ⟩+
		`END`
		`END` *id_program*
action	⇒	⟨ *pvs_boolean_formula* ⟩ `--->` ⟨ *assignment* ⟩+
assignment	⇒	*id* `:=` ⟨ *pvs_expression* ⟩

where all declarations are global, but the variables declared within a component of the form *named_program* are only used locally.

This grammar uses some non-terminals of the grammar of the PVS specification language[8]. This allows to type check easily all PVS declarations and expressions by invoking the PVS parser and type checker. There are additional type correctness conditions for actions which have the form of invariants. For example, an action of the form

```
guard ---> x:= x-1,
```

[8] The non-terminals of the form ⟨ *pvs_*$\cdots$ ⟩

where `x` is declared as natural number, is type correct if `guard` $\Rightarrow$ `x` > 0 is a valid formula; but it is sufficient that `guard` $\Rightarrow$ `x` > 0 is an invariant of the whole system under consideration.

4.2 A proof methodology

We implemented some of the verification rules presented in [MP95] such as the `Inv` rule and the `Waiting` rule corresponding respectively to the proof of properties of the form $\Box P$ and $\Box(P \Rightarrow P_1 \mathcal{W} P_2)$, where P, P_1 and P_2 are predicates. Verification conditions are extracted automatically from the considered specification S and the property we want to verify by a proof obligation or verification condition (VC) generator. The VCs generated for the `Inv` rule are respectively $Init \Rightarrow Q_i$ and $\{Q_i \Rightarrow \widetilde{pre}[\tau](Q_i) \mid \tau \in T\}$ where $Init$ is the predicate defining the set of initial states, T the set of transitions of S and Q_i defined as in Section 1.1. We start with $i = 0$ and increase it until a provable set of verification conditions is obtained or $Init \Rightarrow Q_i$ is not provable anymore (a counter example for this obligation proves that P is *not* an invariant of S).

The VC generator generates only VCs which are not "trivially true". For example, if an action τ does not affect the variables occurring in Q_i, then the VC "$Q_i \Rightarrow \widetilde{pre}[\tau](Q_i)$" is not generated. If Q_i is of the form "$(pc = i) \Rightarrow Q$", where pc a control variable and i a possible value, it is only necessary to prove that Q_i is preserved by every action leading to control point i. In fact, it is often the case that predicates of the form $\widetilde{pre}[\tau](Q_i)$ are of the form $(pc = i) \Rightarrow Q$. Also, the auxiliary invariants (see Section 4.3) are of this form.

The generated obligations are submitted to the PVS proof checker, which tries to prove their validity by means of a set of tacticals we have defined. First an efficient but incomplete tactical for first order predicates is used. It combines rewriting with boolean simplification using Bdds[9] 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 tactical combining automatic induction and decision procedures is applied. If the proof fails again, a set of non-reducible goals is returned and one iteration step is performed. The user can always suspend this process and try to prove the unproved obligation in an interactive manner using the PVS proof checker.

4.3 Use of auxiliary invariants

It is in general essential to use already proved invariants or systematically generated structural invariants obtained by static analysis ([MAB+94], [BBM95], [MP95] and [BLS96]). Let $\mathcal{I}$ stand for the conjunction of all these invariants. In order to prove that P is inductive, it is sufficient to prove

$$\mathcal{I} \wedge P \Rightarrow \widetilde{pre}[\tau](P) \qquad (*)$$

[9] A Bdd simplifier is available in PVS as a tactic.

instead of $P \Rightarrow \widetilde{pre}[\tau](P)$. As $\mathcal{I}$ is usually a huge formula, we have to use it in an efficient way, that is only its "relevant conjuncts". Invariants of the particular form $(pc = i) \Rightarrow Q$, providing information about values of variable at some control point i, are only relevant for (*) when τ starts at control point i. In [Gri96], a more refined strategy is defined which selects in a formula of the form $h_1 \wedge h_1 \cdots \wedge h_n \Rightarrow c$, formulas h_i which are relevant for establishing the validity of c.

4.4 An efficient implementation

The implementation language of PVS is Lisp. Theories, expressions and formulas are defined as Lisp classes. In our tool, programs are also defined as Lisp classes. Type checking a program creates a class containing the corresponding declarations and actions. A current list of type checked programs is maintained. Static analysis described in Section 4.3 is performed using the internal representation of programs. The fact that our internal structures are very close to the internal PVS representation, allows to use many PVS features.

5 Examples

We present two examples. The first one, which is finite state, is a mutual exclusion algorithm studied in [Sif79].

```
mutex : SYSTEM
BEGIN
  ina, inb, PAB : VAR bool
  p1, p2        : VAR nat
  BEGIN
    p1=1                  ---> p1 := 2  ; ina := true        (t11)
    p1=2 AND inb          ---> p1 := 3  ;                    (t12)
    p1=3 AND NOT(PAB)     ---> p1 := 4  ; ina := false       (t13)
    p1=4 AND PAB          ---> p1 := 2  ; ina := true        (t14)
    p1=3 AND PAB          ---> p1 := 2  ;                    (t15)
    p1=2 AND NOT(inb)     ---> p1 := 5  ;                    (t16)
    p1=5                  ---> p1 := 6  ; ina := false       (t17)
    p1=6                  ---> p1 := 1  ; PAB := false       (t18)
     ||
    p2=1                  ---> p2 := 2  ; inb := true        (t21)
    p2=2 AND ina          ---> p2 := 3  ;                    (t22)
    p2=3 AND PAB          ---> p2 := 4  ; inb := false       (t23)
    p2=4 AND NOT(PAB)     ---> p2 := 2  ; inb := true        (t24)
    p2=3 AND NOT(PAB)     ---> p2 := 2  ;                    (t25)
    p2=2 AND NOT(ina)     ---> p2 := 5  ;                    (t26)
    p2=5                  ---> p2 := 6  ; inb := false       (t27)
    p2=6                  ---> p2 := 1  ; PAB := true        (t28)
  END
  INITIALLY : p1=1 AND p2=1
END mutex
```

We want to verify that the predicate

$$P = (p1 = 2) \Rightarrow ((p2 = 2) \Rightarrow (ina \vee inb))$$

expressing the impossibility that both processes may enter the critical section (pi =5) at the same moment, is an invariant for this program[10]. Since $Q_0 = P$ is not inductive for the transitions `t15` leading to $p1 = 2$ and `t25` leading to $p2 = 2$, the predicate $Q_1 = P \wedge \widetilde{pre}[t15](P) \wedge \widetilde{pre}[t25](P)$ is calculated:

$$\begin{aligned} Q_1 = &(p1 = 2 \Rightarrow (p2 = 2 \Rightarrow (ina \vee inb))) \\ &\wedge (p1 = 3 \wedge PAB \Rightarrow (p2 = 2 \Rightarrow (ina \vee inb))) \\ &\wedge (p2 = 3 \wedge \neg PAB \Rightarrow (p1 = 2 \Rightarrow (ina \vee inb))) \end{aligned}$$

$Q_1 \Rightarrow \widetilde{pre}[\tau](Q_1)$ is a valid formula for all transitions τ leading to $pi = 2$ or $pi = 3$ and the proof of this fact succeeds using our tactical. In this example, iteration is not necessary when using the following structural invariant obtained by an extension of the method described in [BLS96]:

$$I = \ (p1 = 3 \Rightarrow ina) \wedge \ (p2 = 3 \Rightarrow inb)$$

The proof of $Q_0 \wedge I \Rightarrow \widetilde{pre}[\tau](Q_0)$ succeeds also for the transitions $\tau =$ `t15` and $\tau =$ `t25`. This example was treated automatically by our tool.

The second example, which is infinite state, describes a simple buffer with two actions "input" and "output".

```
simple_buffer :  SYSTEM
BEGIN
  elem : TYPE
  outp, e, x, y : var elem
  IMPORTING Buffer[elem]
  B : var Buffer[elem]

  BEGIN
    TRUE            ---> B := cons(e,B)
    NOT(null?(B))   ---> outp := first(B) ; B := tail(B)
  END
  INITIALLY : B = null
END simple_buffer
```

The variable `e` represents the input of the the buffer. The imported PVS theory `Buffer` that contains the definition of buffers and some basic functions operating on them, is defined as follows:

```
Buffer [elem:TYPE]   : THEORY
BEGIN
  IMPORTING list[elem]
```

[10] Using the predicate $\neg(p1 = 5) \wedge (P2 = 5)$ to express the mutual exclusion property, leads to exactly one more iteration step

```
  Buffer : TYPE = list[elem]

  isin(B1:Buffer, e1:elem) : RECURSIVE bool =
    CASES B1 OF
      null : FALSE,
      cons(e2,B2) : IF (e1=e2) THEN TRUE ELSE isin(B2,e1) ENDIF
    ENDCASES

  first(B:(cons?)) : RECURSIVE elem =
    IF null?(cdr(B)) THEN car(B) ELSE first(cdr(B)) ENDIF

  tail(B:(cons?)) : RECURSIVE Buffer =
    IF null?(cdr(B)) THEN null ELSE cons(car(B),tail(cdr(B))) ENDIF

  isbefore(x,y:elem, B1:Buffer) : RECURSIVE bool =
    CASES B1 OF
      null : FALSE,
      cons(e1,B2) :
        IF null?(B2) THEN (e1=x)  ELSE
           IF (e1=x) THEN NOT(isin(B2,y)) OR isbefore(x,y,B2)
                     ELSE isbefore(x,y,B2)
           ENDIF
        ENDIF
    ENDCASES

  Buffer-lemma : OBLIGATION
    FORALL (B: Buffer, x: elem, y: elem):
      NOT(null?(B)) AND NOT(isin(B,y))   =>   NOT(isin(tail(B),y))

END Buffer
```

We want to verify that

```
         BOX ( NOT(null?(B)) AND (x=car(B)) AND NOT(isin(B,y))
                  =>
               isbefore(x,y,B) WEAK-UNTIL (outp=x)              )
```

is an invariant. It expresses the fact that elements leave the buffer in the same order they have entered it, that is, the FIFO property. The following VCs are generated by our tool using the `Waiting` rule:

```
 VC-1 : OBLIGATION
          isbefore(x,y,f)
             =>
          isbefore(x,y,cons(e,f)) OR (outp=x)

 VC-2 : OBLIGATION
          isbefore(x,y,f) AND  NOT(null?(f))
             =>
          isbefore(x,y,tail(f)) OR (first(f)=x)
```

```
VC-3 : OBLIGATION
         NOT(null?(f))  AND  (x=car(f))  AND
         NOT(isin(f,y)) AND  NOT(x=y)
            =>
         isbefore(x,y,f) OR (outp=x)
```

The obligations `VC-1` and `VC-3` are proved automatically in one single step proof using our tactical. `VC-2` is proved automatically with the same tactical using `Buffer-lemma`, which expresses a trivial property of buffers. That means the property can be verified without iteration.

6 Conclusions and future work

In this paper, we have presented a method and a tool allowing to do model checking using a theorem prover. Our approach takes advantage of the automatizability of algorithmic model checking and of the power of axiomatic methods which allows to deal with infinite state programs. It is clearly only a partial method as the fixed point may never be reached by the algorithmic method. Sometimes, the user will be able to guess a solution (which often can be checked easily).

In this paper we have hardly mentioned compositionality; however, for example for the verification of the mutual exclusion program (consisting of the parallel composition of two components) no product is built; also the method deriving structural invariants [BLS96] is compositional. In the future, more compositionality will by added by means of well-known rules.

Another interesting direction is the use of abstraction in the manner proposed for example in [Gra94]. The present framework is appropriate for this approach as in the above mentioned paper, the most difficult part was to argue that the considered abstract operations are in fact abstractions of the concrete operations. Here, all the necessary proofs can be done with PVS. Similar proposals have been made in [DF95] or in [HS96].

References

[BGMW94] 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.

[BLS96] S. Bensalem, Y. Lakhnech, H. Saïdi. *Powerful Techniques for the Automatic Generation of Invariants.* In this volume.

[BBM95] 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, September 1995.

[BLUP94] A. Blinchevsky, B. Liberman, I. Usvyatsky, A. Pnueli. *TPVS: Documentation and Progress Report.* Weizmann Institute Of Science, Department of Applied Mathematics and Computer Science, 1994.

[BM88] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook.* Academic Press, 1988.

[CCF+95] 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.

[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.

[DF95] J. Dingel and Th. Filkorn. *Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving.* Computer-Aided Verification, CAV'95, LNCS 939, Liège, June 1995.

[GM93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL.* Cambridge University Press, 1993.

[Gra94] S. Graf. *Characterization of a sequentially consistent memory and verification of a cache memory by abstraction,* CAV'94. To appear in Journal of Distributed Computing.

[GR95] E. P. Gribomont and D. Rossetto. *CAVEAT: technique and tool for Computer Aided VErification And Transformation.* Computer-Aided Verification, CAV'95, LNCS 939, Liège, June 1995.

[Gri96] E. P. Gribomont. *Preprocessing for invariant validation.* AMAST'96.

[HS96] K. Havelund, and N. Shankar. *Experiments In Theorem Proving and Model Checking for Protocol Verification.* Proceedings of Formal Methods Europe. 1996.

[Hun93] H. Hungar. *Combining Model checking and Theorem proving to Verify Parallel Processes.* Computer-Aided Verification, CAV'93, LNCS 697, Elounda, June 1993.

[MAB+94] Z. Manna and al. *STeP: The Stanford Temporal Prover.* Department of Computer Science, Stanford University, 1994.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1991.

[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.

[OSR93b] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language.* Computer Science Laboratory, SRI International, February 1993.

[RSS95] S. Rajan, N. Shankar, and M. K. Srivas. *An integration of model checking with automated proof checking.* Computer-Aided Verification, CAV'95, LNCS 939, Liège, June 1995.

[Rus95] J. M. Rushby. *Model Checking and Other Ways of Automating Formal Methods.* Position paper for panel on Model Checking for Concurrent Programs Software Quality Week, San Francisco, May/June 1995.

[Saï95] H. Saïdi. *Syntax extentions in PVS, some suggestions.* Unpublished notes. September 1995.

[Sif79] J. Sifakis. *Le Contrôle des Systèmes Asynchrones: Concepts, Propriétés, Analyse Statique.* Phd Thesis. INPG, Grenoble, 1979.