Abstract and Model Check while you Prove*

To be presented at the eleventh International Conference on Computer-Aided Verification (CAV99), Trento, Italy, Jul 7-10, 1999

Hassen Saïdi and Natarajan Shankar

Computer Science Laboratory
SRI International
Menlo Park, CA 94025, USA
{saidi,shankar}@csl.sri.com

Abstract. The construction of abstractions is essential for reducing large or infinite state systems to small or finite state systems. Boolean abstractions, where boolean variables replace concrete predicates, are an important class that subsume several abstraction schemes. We show how boolean abstractions can be constructed simply, efficiently, and precisely for infinite state systems while preserving properties in the full μ-calculus. We also propose an automatic refinement algorithm which refines the abstraction until the property is verified or a counterexample is found. Our algorithm is implemented as a proof rule in the PVS verification system. With the abstraction proof rule, proof strategies combining deductive proof construction, model checking, and abstraction can be defined entirely within the PVS framework.

1 Introduction

When verifying temporal properties of reactive systems, algorithmic methods are used when the problem is decidable, and deductive methods are employed, otherwise. Algorithmic methods such as model checking are limited by the state space explosion problem. State space reduction techniques such as symbolic representations, symmetry, and partial order reductions have yielded good results but the state spaces that can be handled in this manner are still quite modest. Deductive methods using theorem proving continue to require a considerable amount of manual guidance. While it is clear that any way out of this impasse must rely on a combination of theorem proving and model checking, specific methodologies are needed to make such a combination work with a reasonable degree of automation. It is known that abstraction is a key methodology in combining deductive and algorithmic techniques. Abstraction can be used to reduce problems to model-checkable form, where deductive tools are used to construct

* This research was supported by the National Science Foundation under Grant Nos. CCR-9509931 and CCR-9712383, and by the Air Force Office of Scientific Research Contract No. F49620-95-C0044. We thank our colleagues John Rushby and Sam Owre for their helpful comments on earlier versions of this paper.

valid abstract descriptions or to justify that a given abstraction is valid. In this paper, we propose a practical verification methodology that is, based on a simple, efficient, and precise form of boolean abstraction generation that preserves properties in the μ-calculus. We extend the boolean abstraction scheme defined in [GS97] that uses predicates over concrete variables as abstract variables, to abstract assertions in the rich assertional language of PVS [OSRSC98]. The PVS language admits the definition of a fixed point operator that is used to define the μ-calculus in PVS [RSS95]. With this definition of the μ-calculus in PVS, model checking implemented as a PVS proof rule can be used as a decision procedure.

Our conservative abstraction scheme is implemented as a proof rule that abstracts any PVS formula over concrete state variables and produces a PVS formula over abstract state variables. Any assertion expressing a general or temporal property of a concrete PVS specification is abstracted into a *stronger* assertion expressing a property over the corresponding abstract specification. The resulting abstract assertion is in a decidable logic, and decision procedures such as model checking can be used to discharge it.

Unlike previous work for the automatic abstraction of infinite state systems using decision procedures [GS97,CU98,BLO98], our algorithm does not always over-approximate the transition relation as is done to preserve only universally quantified path temporal formulas in logics such as $\forall$CTL. Extensions of the preservation results [DGG94,CGL94] to the more expressive logic CTL* are defined using the notion of mixed abstraction which involves multiple next-state relations. Our algorithm abstracts a μ-calculus formula which is not tied to a single transition system. Thus, no distinction is made between universal and existential fragments. The integration of our abstraction algorithm as a PVS proof rule allows us to design powerful proof strategies combining abstract interpretation, model checking and proof checking. We also propose an automatic abstraction refinement algorithm that is applied when model checking fails. This is done by automatically enriching the abstract state with new relevant predicates until the property is proved or a counterexample is found.

The paper is organized as follows. In Section 2 we show how boolean abstractions can be defined in PVS. In Section 3, we present an efficient abstraction algorithm for the computation of the "most precise" abstraction of a given boolean abstraction of a predicate over concrete state variables. In Section 4, we generalize this algorithm to abstract any PVS assertion, including μ-calculus formulas over concrete state variables into assertions over abstract state variables. In Section 5, we present the refinement algorithm.

2 Boolean Abstractions in PVS

Propositional μ-calculus is an extension of propositional calculus that includes predicates defined by means of least and greatest fixed point operators, μ and ν, respectively. It is strictly more expressive than CTL* which includes both linear and branching time temporal logics such as LTL and CTL. In [RSS95] a detailed description of the encoding of the propositional μ-calculus in PVS is presented. The least fixed point operator is defined as $\mu(F) = \bigcap\{x \mid F(x) \subseteq x\}$, the predicate that is the greatest lower bound of the pre-fixed points of a monotone predicate transformer F. The temporal operators of CTL, such as **AG**, **AF**, **EG**, and **EF**, can be easily defined using their fixed-point characterizations. When the state space is finite, the predicates can be coded in boolean form and model checking of μ-calculus formulas can be done using binary decision diagrams (BDDs).

As a simple example, we consider a simple protocol where two processes are competing to enter a critical section in mutual exclusion using a semaphore. The PVS theory describing the protocol is given as follows.

```
semaphore : THEORY
  BEGIN
IMPORTING  MU@ctlops
 location : TYPE = {idle, wait, critical}
 state : TYPE = [# pc1,pc2:location , sem: int #]
  s,s1,s2 : VAR state

 init(s) : bool= pc1(s)=idle and pc2(s)=idle and sem(s)=1

 N(s1,s2) : bool = ...

  safe: THEOREM
   init(s) IMPLIES
    AG(N, LAMBDA s: NOT (critical?(pc1(s)) AND
                         critical?(pc2(s))))(s)
END semaphore
```

The state is given as a record consisting of two program counters and a semaphore `sem`. The expression `N(s1,s2)` is transition relation of the protocol. We are interested in proving that both processes have mutually exclusive access to the critical section. The property `safe` is expressed as a CTL property using the usual operator **AG**, which is translated into a μ-calculus property. When the state type is finite, the property can be verified using model checking[RSS95]. In this simple example, `sem` is of type integer and cannot be encoded with a finite number of boolean variables and hence the property cannot be directly model checked. We propose to extend the capabilities of PVS with a boolean abstraction mechanism that can conservatively reduce a μ-calculus property of an infinite state system to model checkable form. In this abstraction, certain predicates at the concrete level (that might be used in guards, expressions, or properties) can be replaced by abstract boolean variables. This gives us a general method for constructing abstractions by evaluating any predicate over the variables of the program. Since the set of boolean variables is finite, so is the set of abstract states. Boolean abstraction is defined using a set of predicates of the form $\lambda(s : \texttt{state}) : \varphi(s)$ over the `concrete` state type `state`. An abstraction of the mutual exclusion protocol can be defined using two predicates

$\lambda(s) : sem(s) \leq 0$ and $\lambda(s) : sem(s) > 0$. These predicates define an abstract state type

$$\text{abs\_state} : \text{TYPE} \quad = [\# \text{ pc1}, \text{pc2} : \text{ location}, \text{B1}, \text{B2} : \text{ boolean}\#]$$

where the state components `pc1` and `pc2` are of finite type and therefore are not abstracted, and the state component `sem` referenced by the two predicates defining the abstraction is encoded with two boolean components `B1` and `B2` corresponding to the two predicates. In this particular example, these two predicates happen to be exclusive, but boolean abstractions can be defined more generally with an arbitrary set of predicates over the concrete state type.

3 Efficient Computation of Boolean Abstractions

Abstract interpretation [CC77] is the general framework for defining abstractions using Galois connections[1]. The domain of the abstraction function α consists of sets of concrete states, represented by predicates, and ordered by implication. The range of the abstraction consists of boolean formulas constructed using the boolean variables $B_1, \cdots, B_k$, ordered by implication. If X ranges over sets of concrete states and Y ranges over boolean formulas in $B_1, \cdots, B_k$, then the abstraction and concretization function α and γ have the following properties:

- $\alpha(X) = \bigwedge\{Y \mid X \Rightarrow \gamma(Y)\}$,
- $\gamma(Y) = \bigvee\{X \mid \alpha(X) \Rightarrow Y\}$.

However, we use a simpler and precise concretization function γ which consists simply in substituting each abstract variable B_i by its corresponding predicate φ_i, and each abstract state variable `abs_s` by the corresponding concrete state variable `s`. That is

$$\gamma(Y) = Y[\varphi_i(s)/B_i(abs\_s)].$$

We propose to apply boolean abstractions to any predicate (assertion or transition relation) written in a rich assertional language.

Abstraction of assertions. For any predicate P over the concrete variables, the abstraction $\alpha(P)$ of P can be computed as the conjunction of all boolean expressions b satisfying the condition:

$$P \Rightarrow \gamma(b) \tag{1}$$

[1] A Galois connection is a pair (α, γ) defining a mapping between a concrete domain lattice $\wp(\mathcal{Q})$ and an abstract domain lattice $\wp(\mathcal{Q}^A)$, where α and γ are two monotonic functions such that $\forall (P_1, P_2) \in \wp(\mathcal{Q}) \times \wp(\mathcal{Q}^A).\ \ \alpha(P_1) \subseteq P_2 \ \Leftrightarrow\ P_1 \subseteq \gamma(P_2)$.

Note that there are 2^{2^k} distinct boolean truth functions in k variables, and testing all of these could become very expensive. This set is designated as the set of *test points*. An abstraction is *precise* with respect to the considered abstract lattice, if the set of test points is the entire set of the boolean expressions forming the abstract lattice. Any over-approximation of the $\alpha(P)$ can be computed with a smaller set of test points for which the implication (1) must be valid. For example, in [GS97], the abstract lattice considered is the lattice of monomials[2] over the set of boolean variables. In this case, it is not necessary to prove (1) for all the monomials over the set $\{B_1, \cdots, B_k\}$, but only for the atoms $B_1, \cdots, B_k$ and their negations. We can efficiently compute $\alpha(P)$ for any predicate P by choosing the abstract space as the whole boolean algebra over B or by choosing a sub-lattice of B and the corresponding test points, using the following fact:

Theorem 1. *Let $B = \{B_1, \cdots, B_k\}$ be a set of boolean variables, and let $\mathcal{B}_A$ be the boolean algebra defined by the structure $< B, \wedge, \vee, \neg, true, false >$. Let $\mathcal{D}_B$ be the subset of $\mathcal{B}_A$ containing only literals[3] and disjunctions of literals. To compute the most precise image by α of any set of concrete states P (given as a predicate), it is sufficient to consider as a set of test points, the set $\mathcal{D}_B$ instead of the whole set $\mathcal{B}_A$ of boolean expressions. That is, testing*

$$P \Rightarrow \gamma(b)$$

for all boolean expressions in $\mathcal{B}_A$ is equivalent to test this implication only for b in $\mathcal{D}_B$. That is, 2^{2^k} tests can be reduced to at most only $3^k - 1$ tests.

Proof. We consider the fact that each boolean expression b can be written in a conjunctive normal form $d_1 \wedge \cdots \wedge d_j$, where each d_i is a disjunction of literals. Thus, the proof of the implication (1) for each element b can be first decomposed to simpler proofs $P \Rightarrow \gamma(d_i)$. This implication can be proved for each d_i by first testing one disjunct, that is a literal, or more than one disjunct if necessary. That is, only for disjunctions in $\mathcal{D}_B$. ∎

This theorem gives us an efficient way of computing precise abstractions by reducing the set of proof obligations from 2^{2^K}, the number of elements of $\mathcal{B}_A$, to only $3^k - 1$, the number of elements of the smaller set $\mathcal{D}_B$, and also gives us an order in which the proof obligations should be generated and proved. In fact, when the set of predicates $\{\varphi_1, \cdots, \varphi_k\}$ is properly chosen, the actual number of tests is far fewer than $3^k - 1$. When a proof for any element b_i of the set $\mathcal{D}_B$ succeeds or fails, then the number of tests will decrease due to the fact that for many elements b_j of $\mathcal{D}_B$, the test is redundant due to subsumption. Figure 1(a) shows how the image by α of a set $P(s)$ of concrete states is computed. The variable α is initialized to *true*. The variable *fail* consists of the set of elements of $\mathcal{D}_B$ that have not been proved to be in the abstraction of P. The set *fail* is

---

[2] Monomial are the expressions $\bigwedge_{i \in \{1 \cdots k\}} b_i$ where each b_i is either B_i or $\neg B_i$.

[3] A literal is either a boolean variable B_i or its negation $\neg B_i$

```
α+(P(s), C)
Initialization
   α := TRUE;
   fail := ∅;
   i := 1;
Iteration
   while  i < k do
     D := disjuncts(i, α);
     while  D ≠ ∅ do
       let  b = choose_in D   in
       remove b from D
       If  ¬(α ∧ b ∈ fail)
         Then
             If  ⊢ P(s) ∧ C ⇒ γ(b)
               Then  α := α ∧ b
               Else  fail := fail ∪ b
         Else  skip
     od
     i := i + 1
   od
 return α
```

(a)

```
α+(P(s1, s2), C)
Initialization
   α := TRUE;
   fail := {FALSE};
   i := 1;  j := 1;
Iteration
   while  j < k do
    C := conjuncts(j, α);
    while  i < k do
     D := disjuncts(i, α);
     while  D ≠ ∅   ∧   C ≠ ∅ do
       let  (b1, b2) = choose_in C ×  D   in
       If  ¬(α ∧ (b1 ⇒ b2) ∈ fail)
         Then
             If  ⊢ P(s1, s2) ∧ C ∧ γ(b1) ⇒ γ(b2)
               Then  α := α ∧ (b1 ⇒ b2)
               Else  fail := fail ∪ (b1 ⇒ b2)
           Else  skip
   ...
   return α
```

(b)

Fig. 1. Efficient computation of $\alpha(P)$

initially just the singleton $\{FALSE\}$. It is assumed that there has already been a prior check to ensure that $P(s) \wedge C$ is not equivalent to $FALSE$. The construction starts by using disjunctions of length 1, i.e., the literals B_i and $\neg B_i$ for b. The literals b for which the proof obligation $P(s) \Rightarrow \gamma(b)$ succeeds, are added to α. At each iteration, when such a proof succeeds, it is possible to eliminate from the current set of test points the elements for which the test is no longer necessary. This is done by the test $\alpha \wedge b \in \mathit{fail}$. For instance, in the first iteration when we consider only literals, if the proof succeeds for B_i, it is not necessary to test $\neg B_i$. The test for $\neg B_i$ can only fail, otherwise, both $\neg B_i$ and B_i would be added to α, and $\alpha(P)$ would be equivalent to $FALSE$. In the next iteration, the test points that are disjunctions of two literals and not already subsumed by the disjunctions in α, are considered. Once again, the successful test points are added to α, i is incremented and the iteration is repeated for disjunctions of length i. The image α of a set of concrete states is computed incrementally and can be interrupted at any moment, providing an over-approximation of the precise image. Furthermore, we use additional heuristics to avoid unnecessary tests. For instance, if the intersection of the set of free variables of P and those of $\gamma(B_i)$ is empty, it is not necessary to consider the boolean expressions constructed using B_i.

Abstraction of a Transition Relation. Transitions are expressed as general assertions over a pair of concrete states (s_1, s_2). The abstraction of a predicate $P(s_1, s_2)$ describing such a transition relation is defined as a predicate $B(abs\_s_1, abs\_s_2)$ over the abstract pair (abs_s_1, abs_s_2). Figure 1(b) shows how a concrete predicate $P(s_1, s_2)$ representing a transition relation is abstracted. The algorithm constructs a transition relation over the variables $\{B_1, \cdots, B_{2k}\}$ by constraining the current and the next abstract states. This is done by considering as set of test points the set of implications $b_1 \Rightarrow b_2$, where b_1 and b_2 represent formulas in the current and the next abstract state variables, respectively. Again, the abstraction of P is computed incrementally by first constraining the next state, that is by enumerating the disjunctions b_2. When all the proofs fail for a given choice of b_1, the current state is constrained by considering a longer conjunction for b_1. Consider for instance the expression

$$s_2 = s_1 \text{WITH [sem := sem}(s_1) + 1].$$

This assertion over a pair of concrete state variables (s_2, s_2) of type `state` is abstracted with respect to the predicates $\lambda(s) : sem(s) \leq 0$ and $\lambda(s) : sem(s) > 0$ to the following assertion over a pair of abstract state variables (abs_s_1, abs_s_2) of type `abs_state`:

$$\begin{aligned}
&(B_1(abs_s_1) \Rightarrow (B_1(abs_s_2) \vee B_2(abs_s_2))) \\
\wedge\ &(B_2(abs_s_1) \Rightarrow B_2(abs_s_2)) \\
\wedge\ &(\neg B_2(abs_s_1) \Rightarrow (\neg B_2(abs_s_2) \vee \neg B_1(abs_s_2)) \wedge (B_1(abs_s_2) \vee B_2(abs_s_2))) \\
\wedge\ &(\neg B_1(abs_s_1) \Rightarrow B_2(abs_s_2) \wedge \neg B_1(abs_s_2)).
\end{aligned}$$

4 Abstract Interpretation as a Proof Rule

Our abstraction algorithm computes the most precise over-approximation of an assertion over concrete states, using a validity checker for the generated assertions. We implemented this algorithm in the PVS verification system as a primitive proof rule. Our goal is to approximate a PVS formula over concrete state variables, that is a PVS boolean expression, by a formula over abstract state variables. This generated theorem is stronger than the original one. However, it is expressed in a decidable theory that can be handled by model-checking, BDD simplification, or the ground decision procedures available in PVS. To do so, we generalize the abstraction algorithm defined in [PH97] for the μ-calculus to the PVS assertion language and we use our abstraction algorithm to approximate assertions. This algorithm abstracts propositional μ-calculus formulas using over-approximation of predicates and under-approximation of negated predicates. Under-approximation of an assertion is defined as follows:

$$\alpha_-(P(s)) \; = \; \bigvee \{b \mid \gamma(b) \Rightarrow P(s)\}$$

We use only the over-approximation algorithm relying on the following lemma.

Lemma 1. *Let φ a predicate defining a set of states. For all predicate φ*

$$\alpha_+(\neg\varphi(s)) \;\Leftrightarrow \neg\alpha_-(\varphi(s)).$$

We now formally define the abstraction function $[\![\ ]\!]^\sigma$ which approximates a PVS boolean expression f such that, $[\![\ f\ ]\!]^+$ denotes an *over* approximation of f, and $[\![\ f\ ]\!]^-$ an *under* approximation of f. We also use a context c consisting of a PVS formula that is valid at the PVS subformula that is being approximated. The intuition behind using such a context expression is that when an expression $e_1 \wedge e_2$ is being abstracted, one can assume that e_1 is valid when abstracting e_2 and vice-versa. The context when omitted is just the boolean constant *TRUE*. $[\![\ f\ ]\!]_c^\sigma$ denotes the approximation of f under the context c.

Approximation of PVS assertions. The abstraction function $[\![\ ]\!]$ is defined recursively on the structure of the PVS assertion language as follows.

$$\begin{array}{lll}
\textit{propositions}: & [\![e_1 \wedge e_2]\!]_c^\sigma & \longrightarrow [\![e_1]\!]_{c\wedge e_2}^\sigma \wedge [\![e_2]\!]_{c\wedge e_1}^\sigma \\
 & [\![\neg e]\!]_c^\sigma & \longrightarrow \neg[\![e]\!]_c^{-\sigma} \\
 & & \\
\textit{quantifiers}: & [\![\exists(s) : e]\!]_c^\sigma & \longrightarrow \exists(abs_s) : [\![e]\!]_c^\sigma \\
 & [\![\forall(s) : e]\!]_c^\sigma & \longrightarrow \forall(abs_s) : [\![e]\!]_c^\sigma \\
 & [\![\lambda(s) : e]\!]_c^\sigma & \longrightarrow \lambda(abs_s) : [\![e]\!]_c^\sigma \\
 & & \\
\textit{fixpoints}: & [\![\mu/\nu(\lambda(Q) : \mathtt{F}(Q))]\!]_c^\sigma & \longrightarrow \mu/\nu(\lambda(abs_Q) : [\![\mathtt{F}(Q)]\!]_c^\sigma) \\
 & & \\
\textit{atoms}: & [\![e(s)]\!]_c^+ & \longrightarrow \alpha_+(e(s), c) \\
 & [\![e(s_1, s_2)]\!]_c^+ & \longrightarrow \alpha_+(e(s_1, s_2), c) \\
 & [\![e(s)]\!]_c^- & \longrightarrow \alpha_-(e(s), c) \\
 & [\![e(s_1, s_2)]\!]_c^- & \longrightarrow \alpha_-(e(s_1, s_2), c) \\
 & [\![\varphi_i(s)]\!]_c^\sigma & \longrightarrow B_i(abs_s) \\
 & & \\
\textit{constants}: & [\![\ e\ ]\!]_c^\sigma & \longrightarrow \ e \qquad \text{if } \textit{free variables}(e) = \emptyset
\end{array}$$

The following theorem establishes the fact that the abstraction provides, respectively, an over and under approximation of any PVS boolean expression.

Theorem 2. *Let f be a PVS assertion, $[\![\ ]\!]$ an abstraction function. We have:*

$$\vdash f \Rightarrow \gamma([\![\ f\ ]\!]^+) \qquad \textit{and} \qquad \vdash \gamma([\![\ f\ ]\!]^-) \Rightarrow f$$

Proof. The proof is established by induction on the structure of the assertion f. It is easy to show that by the definitions of α_+ and α_-, both implications hold when f is an atom. The other cases can be deduced by monotonicity of the logical connectives, and the fixed point operators. ∎

The soundness of the abstraction function is established by the following theorem.

Theorem 3 (preservation). *Let $[\![\ ]\!]$ be the abstraction function defined above as a boolean abstraction, and let f be any PVS boolean formula. Then*

$$\vdash\ [\![\ f\ ]\!]^{-}\quad \textit{implies}\quad \vdash\ f$$

Theorem 2 ensures that for an assertion f, the abstraction algorithm produces a stronger assertion $\gamma([\![\ f\ ]\!]^{-})$. Note that $\vdash [\![\ f\ ]\!]^{-}$ trivially implies $\vdash \gamma([\![\ f\ ]\!]^{-})$, which then justifies the preservation result of Theorem 3.

The abstraction algorithm where a formula f is under-approximated is implemented as a PVS proof rule `abstract`. This atomic proof rule takes a goal given by a PVS formula (a μ-calculus formula) and a set of state predicates, and translates this to a propositional formula (a propositional μ-calculus formula) which is returned as a new goal. This goal can be discharged using any other PVS proof command including BDD simplification and model checking.

We have defined a PVS proof strategy that carries out a sequence of inference steps that simplify goal formulas by rewriting all definitions, including constant definitions such as the temporal operators of the logic CTL in terms of the μ and ν operators, and applies the abstraction function on the resulting goal.

$$\begin{array}{l}
\forall\ (s:\ \text{state}):\\
\quad \text{init}(s) \supset\\
\quad\quad \neg\mu.\lambda\ (Q:\ \text{pred[state]}):\\
\quad\quad\quad (\lambda\ (u:\ \text{state}):\\
\quad\quad\quad\quad (\neg\lambda\ s:\\
\quad\quad\quad\quad\quad\quad \neg(\text{critical?}(\text{pc1}(s)) \wedge\\
\quad\quad\quad\quad\quad\quad\quad \text{critical?}(\text{pc2}(s))))\\
\quad\quad\quad\quad\quad (u) \vee\\
\quad\quad\quad\quad \exists\ (v:\ \text{state}):\\
\quad\quad\quad\quad\quad (Q(v) \wedge N(u,v)))(s)
\end{array}
\qquad
\begin{array}{l}
\forall\ (abs_s:\ \text{abs_state}):\\
\quad \neg[\![\text{init}(s)]\!]^{+} \vee\\
\quad\quad \neg\mu.\lambda\ (abs_Q:\ \text{pred[abs_state]}):\\
\quad\quad\quad (\lambda\ (abs_u:\ \text{abs_state}):\\
\quad\quad\quad\quad (\neg\lambda\ abs_s:\\
\quad\quad\quad\quad\quad\quad \neg(\text{critical?}(\text{pc1}(abs_s)) \wedge\\
\quad\quad\quad\quad\quad\quad\quad \text{critical?}(\text{pc2}(abs_s))))\\
\quad\quad\quad\quad\quad (abs_u) \vee\\
\quad\quad\quad\quad \exists\ (abs_v:\ \text{abs_state}):\\
\quad\quad\quad\quad\quad (abs_Q(abs_v) \wedge [\![N(u,v)]\!]^{+}))(abs_s)
\end{array}$$

Fig. 2. An example of abstraction for a PVS assertion

Figure 2 shows how the μ-calculus formula corresponding to the theorem `safe` presented in the PVS theory `semaphore` in Section 2 is approximated. The property of mutual exclusion $\lambda(s)\ :\ \neg(\text{critical?}(\text{pc1}(s)) \wedge \text{critical?}(\text{pc2}(s)))$ is expressed as an invariance property. As expected for such properties, the initial state and the transition relation are over-approximated. For instance, we have

$$[\![\text{init}(s)]\!]^{+}\ \longrightarrow\ \text{idle?}(\text{pc1}(abs\_s)) \wedge \text{idle?}(\text{pc2}(abs\_s)) \wedge \neg B_1(abs\_s) \wedge B_2(abs\_s)$$

We have tried other examples including a simple snoopy cache-coherence protocol with an arbitrary number of processes [Rus97] and a variant of the alternating-bit communication protocol called the bounded retransmission protocol [HS96]. The main invariant of the cache coherence protocol is proved by an abstraction defined in terms of five predicates. The preservation of the invariant is then proved by abstraction and BDD-based propositional simplification.

The bounded retransmission protocol is verified using an abstraction also defined in terms of five predicates. The construction of the abstract description takes about 100 seconds in PVS. The resulting abstract assertion is discharged using model checking. In contrast, Havelund and Shankar's verification [HS96] of this example required 57 invariants to justify the validity of a manually derived abstraction.

5 Refining an Abstraction

The abstraction proof rule is used in PVS to generate new goals that depend only on finite state variables. Such goals can be discharged using a PVS proof rule such as the BDD simplifier or the μ-calculus simplifier. However model checking on the new goal can fail because the abstraction is too coarse. It is then necessary to refine the abstraction using a richer abstract domain. Since our abstraction algorithm presented in Section 4 allows us to compute the most precise abstraction with respect the predicates $\varphi_1, \cdots, \varphi_k$, refining the abstraction requires additional predicates. The refinement algorithm takes as arguments the original PVS assertion f, a new list of predicates $\varphi_{k+1}, \cdots, \varphi_l$, and a context Γ_α computed previously. The context Γ_α is a hash-table which associates to each atom the BDD representing its abstraction, that is the BDD α, and the set *fail* of BDDs. The refinement algorithm descends through the structure of f and refines each sub-formula with the new predicates. The refinement algorithm is similar to the algorithm computing $\alpha_+(P)$ of Figure 1. However the variables α and $fail$ are initialized with their already computed values. This allows us to take advantage of the success or failure of already executed proofs. The new set of test points is defined as the disjunctions formed using the literals $B_{k+1}, \cdots, B_l$ and their negation. This set is augmented with the boolean expressions over the old variables $B_1, \cdots, B_k$ for which the proof previously failed. The algorithm returns a more precise approximation of P.

We implemented our abstraction and refinement algorithms as a proof strategy defining a semi-decision procedure that abstracts an original PVS formula and then applies model checking. If model checking fails, the abstraction is refined until model checking succeeds. This strategy is expressed as follows in the PVS strategies language

```
(TRY (THEN  (abstract (phi_1...phi_k)) (model-check))
            (skip)
            (REPEAT
             (LET ((Φ (new-list-of-predicates)))
             (THEN (refine Φ) (model-check)))))
```

Our refinement algorithm tries to eliminate as much of the nondeterminism created by the over-approximation of the transition relation as possible. Absence of nondeterminism can be easily detected by checking that when the abstraction of a transition $\alpha_+(P(s_1, s_2), C)$ is computed, the index i will never reach a value greater than 1. For instance, the abstraction of the assertion

$$e(s_1, s_2) \equiv s_2 = s_1 \text{WITH } [\text{sem} := \ \text{sem}(s_1) \ + \ 1]$$

presented in Section 3 is nondeterministic since it contains the conjunct

$$(B_1(abs\_s_1) \Rightarrow (B_1(abs\_s_2) \vee B_2(abs\_s_2)).$$

Refining such an abstraction involves translating the predicate characterizing the next state, that is $(B_1(abs\_s_2) \vee B_2(abs\_s_2))$ into a disjunctive normal form. Then, for each disjunct, the pre-image is computed with respect the concrete assertion $e(s_1, s_2)$. In this particular case, the pre-images for $B_1(abs\_s_2)$ and $B_2(abs\_s_2)$ are, respectively, $\exists(s_2) : e(s, s_2) \wedge \varphi_1(s_2)$ and $\exists(s_2) : e(s, s_2) \wedge \varphi_2(s_2)$. Their simplified forms are respectively $sem(s) < 0$ and $sem(s) = 0$.

6 Conclusion

We have presented a general abstraction/refinement algorithm that preserves the full μ-calculus as the basis for an integration of abstract interpretation, model checking, and proof checking. We have implemented this boolean abstraction algorithm as an extension to the PVS theorem prover. This allows us to define powerful proof strategies combining deductive proof, induction, abstraction, and model checking within a single framework. It also allows our abstraction algorithm to be used in the framework of a richly expressive specification language encompassing finite, infinite-state, and parametric systems. The computation of the abstraction is completely automatic, and uses the PVS decision procedures to test the generated implications.

We are currently investigating cases where it is possible to detect whether a constructed abstraction *strongly* preserves fragments of the μ-calculus so that abstract counterexamples yield concrete ones. This is done by finding sufficient conditions allowing us to use the various preservation results presented in [LGS+95,DGG94].

The new PVS version includes code generation capabilities, and as future work, we plan to define abstraction construction in the PVS specification language, and to automatically extract the code implementing the abstraction operation. Such experiments are similar to the ones presented in [vHPPR98] where, for instance, the code implementing a BDD simplifier is extracted automatically from its formal specification.

References

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Proceedings of the 9th Conference on Computer-Aided Verification, CAV'98*, LNCS. Springer Verlag, June 1998.

[CC77] 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.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CU98] Michael Colon and Thomas Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Proceedings of the 9th Conference on Computer-Aided Verification, CAV'98*, LNCS. Springer Verlag, June 1998.

[DGG94] D. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems: Abstractions preserving ∀CTL*, ∃CTL* and CTL*. In Ernst-Rudiger Olderog, editor, *IFIP Conference PROCOMET'94*, pages 561–581, 1994.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Conference on Computer Aided Verification CAV'97*, LNCS 1254, Springer Verlag, 1997.

[HS96] Klaus Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe FME '96*, number 1051 in Lecture Notes in Computer Science, pages 662–681, Oxford, UK, March 1996. Springer-Verlag.

[LGS+95] 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, Vol 6, Iss 1, January 1995*, 1995.

[OSRSC98] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *The PVS Specification Language.* Computer Science Laboratory, SRI International, Menlo Park, CA, August 1998.

[PH97] A. Pardo and G.D. Hachtel. Automatic abstraction techniques for propositional μ-calculus model checking. In *Conference on Computer Aided Verification CAV'97*, LNCS 1254, Springer Verlag, 1997.

[RSS95] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model checking with automated proof checking. In *Computer-Aided Verification, CAV '95*, number 939 in Lecture Notes in Computer Science, Liège, Belgium, 1995. Springer-Verlag.

[Rus97] John Rushby. Specification, proof checking, and model checking for protocols and distributed systems with PVS. In *FORTE/PSTV '97*, Osaka, Japan, November 1997.

[vHPPR98] Friedrich von Henke, Stephan Pfab, Holger Pfeifer, and Harald Rueß. Case studies in meta-level theorem proving. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs '98*, volume 1479 of *Lecture Notes in Computer Science*, pages 461–478, Canberra, Australia, September 1998. Springer-Verlag.