Model Checking Guided Abstraction and Analysis[⋆]

In proceedings of the The Seventh International Static Analysis Symposium (SAS2000), Santa Barbara, California.

Hassen Saïdi

System Design Laboratory
SRI International
Menlo Park, CA 94025, USA
Tel: (+1) (650) 859-3810
Fax: (+1) (650) 859-2844
saidi@sdl.sri.com

Abstract. The combination of abstraction and state exploration techniques is the most promising recipe for a successful verification of properties of large or infinite state systems. In this work, we present a general, yet effective, algorithm for computing automatically boolean abstractions of infinite state systems, using decision procedures. The advantage of our approach is that it is not limited to particular concrete domains, but can handle different kinds of infinite state systems. Furthermore, our approach provides, through the use of model checking as a tool for the exploration of the state-space of the abstract system, an automatic way of refining the abstraction until the property of interest is verified or a counterexample is exhibited. We illustrate our approach on some examples and discuss its implementation.

1 Introduction

The combination of abstraction and state exploration techniques is probably the most promising recipe for a successful verification of properties of large or infinite state systems. It is now widely accepted that abstraction techniques are not only useful, but even necessary for a successful verification [19, 6, 21, 13, 12, 9, 14] in order to avoid the limiting factor of using model checking by reducing all the behaviors of a program to a simplified description on which the property of interest can be verified using model checking. While the theoretical frameworks for defining property preserving abstractions such as abstract interpretation [8] have been widely studied in the literature, the *automatic* construction of useful and accurate abstractions preserving useful properties is in an early stage of investigation. Abstract models are usually provided manually, and theorem proving is used to check that the provided abstract mapping preserves the properties.

[⋆] This research was supported by DARPA contract F30602-97-C-0040.

Once the preservation property is established, the abstract model is analyzed by model checking. Recently [14, 7, 1, 25, 11], novel techniques based on abstract interpretation have been proposed in the context of the verification of temporal properties where theorem proving is used to compute automatically finite abstractions. These techniques are quite effective, but require heavy use of theorem proving and decision procedures.

The most general and yet simple and effective abstraction scheme consists of constructing boolean abstractions following the scheme introduced in [14]. Boolean abstractions consist in using predicates over concrete variables as boolean abstract variables. In this abstraction, certain predicates at the concrete level (that might be used in guards, expressions, or properties) can be replaced by boolean variables at the abstract level. An abstract version of the infinite-state transition system is a transition system where the set $\{B_1, \cdots, B_k\}$ of abstract variables is a set of boolean variables corresponding to predicates $\{\varphi_1, \cdots, \varphi_k\}$ over the concrete variables. An abstract state in this transition system is therefore a truth assignment to these boolean variables. Boolean abstractions have very nice properties. In fact, any abstraction mapping that maps an arbitrary system to a finite state system can be expressed as a boolean abstraction. Furthermore, the abstract system can be represented symbolically using Binary Decision Diagrams (BDDs) and therefore can be analyzed using symbolic model checking, allowing an efficient exploration of abstract systems with a large state space. The techniques we developed for the automatic construction of boolean abstractions do not require a preservation check, and ensure that the constructed abstraction indeed preserves various temporal logics properties, including safety properties. Furthermore, boolean abstraction is an efficient and more powerful alternative to static analysis techniques dedicated to the automatic generation of various properties such as invariants like the ones presented in [3, 2, 24].

The drawback of using abstraction followed by model checking as a verification and analysis technology consists in the fact that abstractions are approximations of the original systems that induce false negative results. For instance, a model checker may exhibit an error trace that corresponds to an execution of the abstract program that violates the desired properties. However, this error trace may not correspond to an execution trace in the concrete program. This situation indicates that the abstraction is too coarse, and that the results of model checking the abstract system are not conclusive. That is, too many details were abstracted and the abstraction needs to be refined. The contribution of our work can be summarized as follows:

- We propose an efficient algorithm for the automatic construction of boolean abstractions that requires fewer calls to decision procedures and subsumes the previous and recent work [14, 7, 1, 11] in this topic.

- In all the recent work on the automatic construction of finite abstractions, parallel programs are considered. However, each component are abstracted separately. In our work, the abstraction of a component takes into account its interaction with the environment, allowing the construction of more precise abstractions.

- We propose to use the error trace generated by model checking to automatically refine the abstraction, even more. This methodology consists in successively refining a first abstraction until the property is proved or a concrete error trace violating the property is exhibited. The refinement algorithm generates new predicates that will be used to enrich the abstract state-space.

- The refinement procedure may not always terminate. However, at any refinement step, the reachable states of the constructed abstract system represent an invariant and a new more precise control structure of the concrete system that can be exploited for further analysis. In [20], we use the newly generated predicates to construct a more precise control structure of parameterized systems. Similar ideas are used in [18] for the generation of control structure in the particular case of synchronous linear systems.

Our verification methodology based on abstraction followed by successive model checking guided refinement steps is implemented in a verification environment that combines deduction and state-exploration techniques. We successfully used our methodology to prove safety properties of several systems, including a data-link protocol used by Philips Corporation in one of its commercial products. The original proofs [15, 17, 16] of the protocol required two to six months of work and were entirely done using theorem provers. A boolean abstraction of the protocol can be automatically generated using the predicates appearing in the description of the protocol in about a hundred seconds with the help of the PVS theorem prover [23] and its new efficient implementation of decision procedures. The abstract protocol is then analyzed in a few seconds to check that all the safety properties hold.

This paper is organized as follows: in Section 2, we present the model in which systems are described, and give some basic definitions. In Section 3, we define boolean abstractions in the general framework of abstract interpretation using Galois connections. In Section 4, we show how boolean abstractions can be constructed in an efficient way using decision procedures and compositional reasoning. In Section 5, we show how model checking is used to prove properties on abstract systems and how it can be used as a guide to the automatic refinement of already constructed abstractions. In Section 6, we describe our refinement algorithms. Finally, in Section 7, we describe a tool implementing our methodology.

2 Preliminaries

We consider systems that are parallel compositions of sequential processes, where each process is modeled as a transitions system.

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 including a program counter pc.*

– *$\mathcal{T}$ is a set of transitions.*
– *$\mathcal{L}$ is a set of control locations, that is, the possible values of pc.*
– *Init is a predicate characterizing the set of initial states.*

Each transition τ is a guarded command

$$l_i : \quad guard \;\longrightarrow\; v_1 ::= e_1, \cdots, v_n ::= e_n \;\; \text{goto } l_j$$

where $\{v_1, \cdots, v_k\} \subseteq \mathcal{V}$ and $\{l_i, l_j\} \subseteq \mathcal{L}$. The boolean expression *guard* is the guard of the transition τ. 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 locations of transition τ_i. A state of a system S is a valuation of the system variables of $\mathcal{V}$. A system can be a parallel composition of components described as transition systems. The system can be described as a single transition system where the set of variables is the union of the set of variables of each component, the set of transitions is the union of all the transitions of all components, the program counter is a tuple formed by the program counters of all components, and the initial state is the conjunction of the initial states of each component. Figure 1 shows the description of the Bakery protocol in our specification lan-

```
bakery : SYSTEM
  BEGIN
    process_1 : PROGRAM
        y1 : VAR nat
      BEGIN
      p1_Try  1: TRUE                → y1 := y2+1  GOTO  2
      p1_In   2: y2 = 0  ∨  y1 ≤ y2  →   SKIP      GOTO  3
      p1_Out  3: TRUE                → y1 := 0     GOTO  1
      END process_1
  ||
    process_2 : PROGRAM
        y2 : VAR nat
      BEGIN
      p2_Try  1: TRUE                → y2 := y1+1  GOTO  2
      p2_In   2: y1 = 0  ∨  y2 < y1  →   SKIP      GOTO  3
      p2_Out  3: TRUE                → y2 := 0     GOTO  1
      END process_2
  INITIALLY  :  y1 = 0  ∧  y2 = 0  ∧  pc2 = 1  ∧  pc1 = 1
END bakery
```

Fig. 1. Bakery transition system (version A)

guage. The algorithm is called the Bakery algorithm, since it is based on the idea that customers, as they enter a bakery, pick numbers that form an ascending sequence. Then a customer with a lower number has higher priority in accessing its critical section, which in this case is control location 3. Each process process_i modifies its local variable yi, and can read the other's variable.

We also recall the definitions of predicate transformers over transition systems. The predicate transformers *post* and *pre* expressing, respectively, the strongest postcondition and precondition by a transition τ of a predicate P over the state variables of $\mathcal{V}$ are defined as follows:

$$\begin{aligned} post[\tau](P) &= \exists \mathcal{V}'.action_\tau(\mathcal{V}',\mathcal{V}) \land P(\mathcal{V}') \\ pre[\tau](P) &= \exists \mathcal{V}'.action_\tau(\mathcal{V},\mathcal{V}') \land P(\mathcal{V}') \end{aligned}$$

where $action_\tau(\mathcal{V},\mathcal{V}')$ is defined as the relation between the current state and next state, that is, the expression

$$pc = l_i \land guard \land \bigwedge_{i=1}^{k} v'_i = e_i,\ pc' = l_j$$

Defining the transition relation of a system as a relational predicate for each transition is a more general alternative to the use of guarded commands. The semantics of a transition system S is given by its computational model $K_S = (Q, \mathcal{T}, R)$, where Q is the set of valuations of the program variables $\mathcal{V}$, and $R \subseteq Q \times \mathcal{T} \times Q$ a transition relation. A set of states of a program can be represented by its corresponding predicate over the state variables of $\mathcal{V}$.

3 Boolean Abstractions

Boolean abstraction is a simple abstraction scheme defined in [14] that consists of using predicates over concrete variables as boolean abstract variables. In an abstract version of the infinite-state transition system, the set $\{B_1, \cdots, B_k\}$ of abstract variables is a set of boolean variables corresponding to predicates $\{\varphi_1, \cdots, \varphi_k\}$ over the concrete variables. An abstract state in this transition system is therefore a truth assignment to these boolean variables. Since the set of boolean variables is finite, so is the set of abstract states. Boolean abstractions can easily be defined in the framework of abstract interpretation using Galois connections.

Definition 2 (Galois connection). *A pair of monotonic functions (α, γ) defining a mapping between a concrete domain lattice $\wp(\mathcal{Q}, \subseteq)$ and an abstract domain lattice $\wp(\mathcal{Q}^a, \sqsubseteq)$, is a Galois connection if and only if*

$$\forall (P, P^a) \in \wp(\mathcal{Q}) \times \wp(\mathcal{Q}^a).\ \ \alpha(P) \sqsubseteq P^a \ \Leftrightarrow\ P \subseteq \gamma(P^a)$$

Sets of states in $\wp(\mathcal{Q})$ and $\wp(\mathcal{Q}^a)$ are represented by their corresponding predicates. Thus, $\wp(\mathcal{Q})$ and $\wp(\mathcal{Q}^a)$ correspond to lattices of concrete and abstract predicates ordered by the logical implication. A boolean abstraction can be expressed as a Galois connection as follows:

- $\alpha(P) = \bigwedge\{B^a \mid P \Rightarrow \gamma(B^a)\} = P^a$, where B^a is any boolean expression over

the set $\{B_1, \cdots, B_k\}$.
- γ is defined as a substitution function. That is, $\gamma(P^a) = P^a[\varphi_1/B_1, \cdots, \varphi_k/B_k]$, where each boolean variable B_i is substituted by its corresponding concrete predicate φ_i.

Thus, the abstraction of a concrete set of states represented by a predicate P over concrete variables is defined as the smallest boolean formula P^a over the abstract variables B_i. That is, an overapproximation of P. In [25], we presented an efficient algorithm for computing the most precise boolean abstraction with respect to a set of predicates, for systems where the transition relation is given as a relational predicate. The algorithm consists of an efficient enumeration of all boolean combinations B^a to test the assertion $P \Rightarrow \gamma(B^a)$. The algorithm abstracts systems where the transition relation is given as a predicate. Each implication $P \Rightarrow \gamma(B^a)$ is submitted to the decision procedure to test its validity. In [25], we proved that in order to compute P^a it is not necessary to consider all the possible B^a, that is 2^{2^k} expressions, but at most $3^k - 1$. However, this is still a high price to pay for the construction of an abstract system. Notice that any approximation of P^a is a valid abstraction of P.

```
bakery : SYSTEM
B3 : VAR bool
 BEGIN
  process_1 : PROGRAM
     B1 : VAR bool
    BEGIN
    p1_Try  1 : TRUE       → B1 := F,  B3 := F  GOTO 2
    p1_In   2 : B2 ∨ B3 →   SKIP               GOTO 3
    p1_Out  3 : TRUE       → B1 := T,  B3 := T  GOTO 1
    END process_1
 ||
  process_2 : PROGRAM
     B2 : VAR nat
    BEGIN
    p2_Try  1 : TRUE        → B2 := F,  B3 := T                  GOTO 2
    p2_In   2 : B1 ∨ ¬B3 →   SKIP                               GOTO 3
    p2_Out  3 : TRUE        → B2 := T,
                               B3 := if  B1
                                         then  T
                                         else  if  (¬B1 ∨ ¬B3)
                                                 then F
                                                 else  ?
                                                                  GOTO 1
    END process_2
 INITIALLY  :  B1 ∧ B2 ∧ B3 ∧ pc2 = 1 ∧ pc1 = 1
END bakery
```

Fig. 2. Abstract version of Bakery transition system (version A)

Thus, in order to compute for a concrete system S, an abstract system S^a, it is sufficient to abstract the initial state $Init$ by computing $\alpha(Init)$, and to abstract each transition τ as follows:

$$\tau^a = \alpha(\tau) = \alpha(action_\tau(\mathcal{V}, \mathcal{V}')) = \bigwedge\{(B^a, B^{a'}) | \vdash post[\tau](\gamma(B^a)) \Rightarrow \gamma(B^{a'})\}$$

that is, the pair $(B^a, B^{a'})$ characterizing the abstraction of the set of possible predecessors by τ and the abstraction of the set of possible successors by τ. In this case, the complexity of the computation of τ^a is $(3^k - 1) * (3^k - 1)$ calls to the decision procedure, $(3^k - 1)$ calls to test the successors, and $(3^k - 1)$ calls to test the potential predecessors.

The preservation of properties expressed in temporal logic is widely studied in [21, 10, 5]. Preservation results are established via equivalences and preorders between the concrete and abstract models. The following theorem establishes the preservation of safety properties expressed in the logic CTL^* via simulation.

Theorem 1 (weak preservation). *Let S be a concrete system, and let S^a be a boolean abstraction of S using any set of predicates. We have*

$$S^a \models \alpha(\varphi) \quad \Rightarrow \quad S \models \varphi$$

for each formula $\varphi \in \forall CTL^*$, that is, temporal formulas with universal quantification over paths, including safety properties such as invariants.

Proof. This result can be established by proving that S^a simulates S. This can be done by proving that the following holds for each transition τ of S:

$$\forall P.\ post[\tau](P) \ \Rightarrow \ \gamma(post[\alpha(\tau)](\alpha(P)))$$

that is, each set of successor states by an abstract transition is an overapproximation of the corresponding set of states of the concrete system.

Intuitively, $\forall CTL^*$ properties hold in all execution paths. Since S^a simulates S, that is, all the executions of S are executions of S^a, then if a property holds along all execution paths of S^a, it holds in all execution paths of S. Theorem 1 indicates that when a property is established in the abstract system, its corresponding concrete property holds in the concrete system. However, nothing can be concluded when the property does not hold in the abstract system. Strong preservation results can be applied in this case under some conditions.

Theorem 2 (strong preservation). *Let S be a concrete system, and let S^a be a boolean abstraction of S using any set of predicates that includes all the literals appearing in the guards of S and in the property φ. If S^a is deterministic, we have*

$$S^a \models \alpha(\varphi) \quad \Leftrightarrow \quad S \models \varphi$$

That is, S^a and S are equivalent.

Proof. By construction S^a simulates S. Thus, it is sufficient to prove now that S simulates S^a. To show this, it is sufficient to prove that for each pair of abstract states s_1^a and s_2^a, if s_2^a is a successor of s_1^a by τ^a in the abstract system, then, for every pair s_1 and s_2 of states in the concretization of s_1^a and s_2^a, s_2 is the successor of s_1 by τ in the original system. Every concrete state s_1 in the concretization of s_1^a satisfies the guard of τ, and every successor s_2 of s_1 is in the concretization of s_2^a. Thus, S simulates S^a.

The strong preservation result allows us to avoid false negative results by mapping abstract error traces to concrete executions violating the property. However, the condition for strong preservation requires that S^a be deterministic. This is usually not the case. However, we will see later how we exploit Theorem 2 to generate boolean abstractions to verify properties, and also to generate counterexamples when a formula is not a property of the concrete system. As we mentioned earlier in the introduction, boolean abstraction subsumes abstractions where the abstract domain is finite.

Theorem 3 (generality). *Let S be a system and let α be an abstraction function where the abstract domain is finite. Then, α can be expressed as a boolean abstraction.*

Proof. The proof is based on the fact that a finite domain can be encoded by a set of boolean variables. Each abstract state is then a conjunction of a subset of the set of boolean variables. The concretization of an abstract state is a set of concrete states that can be represented as a predicate.

Figure 2 shows the abstraction of the Bakery protocol using predicates $y1 = 0$, $y2 = 0$, and $y1 \leq y2$ appearing in the guards. Notice that all the assignments are deterministic except the assignment for the variable $B3$ in the transition **p2_Out**.

4 Automatic Construction of Boolean Abstractions

Decision procedures can be used for the automatic construction of a boolean abstraction of a concrete, infinite state system described as a transition system. The abstraction of a concrete system $S = \ <\mathcal{V}, \mathcal{T} = \{\tau_1, \cdots, \tau_n\}, \mathcal{L}, Init>$ is an abstract system $S^a = \ <\mathcal{V}^a, \mathcal{T}^a = \{\tau_1^a, \cdots, \tau_n^a\}, \mathcal{L}, Init^a>$ such that

- $\mathcal{V}^a$ is the set $\{B_1, \cdots, B_k, pc\}$.
- $\mathcal{T}^a$ is a set of abstract transitions.
- $Init^a$ is the abstract initial state computed as $\alpha(Init)$.

The abstraction algorithm consists in computing $Init^a$ and for each concrete transition τ

$$l_i : \ \ guard \ \longrightarrow \ v_1 ::= e_1, \cdots, v_n ::= e_n \ \ \text{goto } l_j$$

a corresponding abstract transition τ^a

$$l_i : \quad guard^a \;\longrightarrow\; B_1 ::= b_1, \cdots, B_k ::= b_k \;\; \text{goto } l_j$$

such that:
- The abstract guard $guard^a$ is computed as $\alpha(guard)$. When using the literals of the guards as abstract boolean variables, $\alpha(guard)$ is an *exact* abstraction, where each literal of *guard* is substituted by its corresponding abstract boolean variable. - Each assignment $B_i := b_i$ is defined as follows:

$$B_i := \begin{cases} \mathbf{T} \text{ if} & post[\tau](true) \Rightarrow \gamma(B_i) \quad (1) \\ \mathbf{F} \text{ if} & post[\tau](true) \Rightarrow \neg\gamma(B_i) \; (2) \\ ? \;\text{ otherwise} & \end{cases}$$

that is, for each abstract variable B_i, the strongest postcondition by τ of any arbitrary state is in $\gamma(B_i)$ or $\neg\gamma(B_i)$, that is, in φ_i or $\neg\varphi_i$. When neither of the above implications is valid, the variable is nondeterministically assigned the value ?.
- The variable *pc* is not abstracted since it is of a finite type.

When a variable is assigned the value ?, it is possible to compute a more refined assignment by taking into account the dependencies between the abstract variables. Thus, the assignment B_i :=? can be redefined as follows:

```
$B_i := if\ b_i^{\mathbf{T}}$
        $then\ \ \mathbf{T}$
        $else\ if\ \ b_i^{\mathbf{F}}$
            $then\ \ \mathbf{F}$
            $else\ \ ?$
```

where $b_i^{\mathbf{T}}$ and $b_i^{\mathbf{F}}$ are defined as follows:

$$b_i^{\mathbf{T}} \equiv \bigvee \{B^a \mid post[\tau](\gamma(B^a)) \Rightarrow \gamma(B_i)\}$$

$$b_i^{\mathbf{F}} \equiv \bigvee \{B^a \mid post[\tau](\gamma(B^a)) \Rightarrow \neg\gamma(B_i)\}$$

That is, $b^{\mathbf{T}}$ and $b^{\mathbf{F}}$ are, respectively, the smallest boolean combination over the abstract variables $\{B_1, \cdots, B_k\}$ that defines the abstract state from which, if the transition τ is executed, the variable B_i gets either the value **T** or **F**. In the worst case, both $b_i^{\mathbf{T}}$ and $b_i^{\mathbf{F}}$ are equivalent to true. Thus, the variable B_i is assigned with the value ?.

In [25], the complexity of the abstraction algorithm for a transition is $3^k - 1 * 3^k - 1$. In our case, this is reduced to at most $3^k - 1 * 2 * k$.

Theorem 4 (complexity). *The complexity of the abstraction of a transition τ using k predicates $\{\varphi_1, \cdots, \varphi_k\}$ requires checking the validity of $3^k - 1 * 2 * k$ implications.*

Proof. For each abstract variable B_i assigned in the abstraction of τ, The boolean expressions $b_i^{\mathbf{T}}$ and $b_i^{\mathbf{F}}$ are computed. Thus $2 * k$ implications have to be proved. For each of the expressions $b_i^{\mathbf{T}}$ and $b_i^{\mathbf{F}}$, all possible boolean expressions B^a over $\{B_1, \cdots, B_k\}$ have to be considered. There are $3^k - 1$ possible expressions as illustrated in the following Figure with $k = 2$. The elements of the set of possible

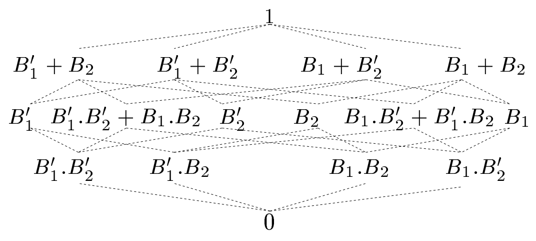

Fig. 3. Boolean algebra for 2 boolean variables B_1 and B_2

boolean expressions B^a are the elements of the boolean algebra defined by the k boolean variables, that is, 2^{2^k} expressions. However, the expression B^a appears on the left hand side of an implication. Thus, it is necessary to consider only expressions that are conjunctions of boolean variables. That is is only $3^k - 1$ possible expressions that can be tested incrementally by first testing each boolean variable B_i and its negation, and then testing conjunctions of the set of variables for which both tests fail.

The results in [25], shows that the enumeration of $3^k - 1$ expressions subsumes the enumeration of the possible 2^{2^k} expressions. However, the enumeration of the possible B^a satisfying the above implications can be done only for the expressions B^a such that

$$FV(post[\tau](\gamma(B^a))) \cap FV(\gamma(B_i)) \neq \emptyset$$

where $FV(P)$ is the set of free variables of the predicate P.

5 Model Checking Guided Analysis

Once an abstract system is constructed, model checking is used to explore its state-space. We use both symbolic and explicit-state model checking techniques. Figure 4 shows the reachable abstract states of the Bakery protocol. It is easy to show that the protocol does guarantee mutual exclusion for both processes since there is no reachable state where both control variables *pc*1 and *pc*2 have the value 3.

The advantage of model checking over other verification techniques is its ability to generate counterexamples when a property is violated. The error trace is a sequence of states and transitions starting from the initial state of the system

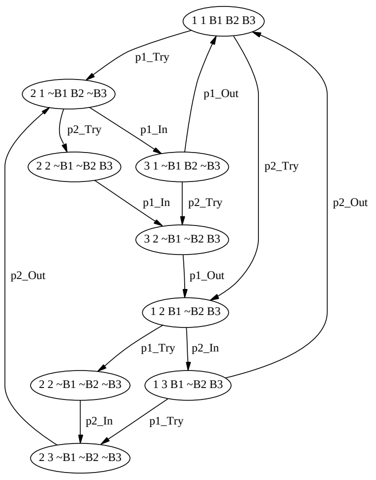

Fig. 4. Abstract state graph for the Bakery protocol

leading to a state violating the property. Error traces of an abstract system can be mapped to executions of a concrete system since each abstract transition corresponds to a single concrete one with the same label.

Figure 5 shows a more complex version of the Bakery protocol (known as *Bakery_C*) where the critical section corresponds to control location 7. This version was proposed to avoid the long wait of one process at location 2 in the previous version (known as *Bakery_A*) before the process enters its critical section. The abstraction of the protocol with respect the guards $y1 = 0$, $y2 = 0$, $y1 \leq y2$, $x1 = 0$, and $x2 = 0$ is given in Figure 6. Figure 7 shows an error trace from the initial abstract state 0 to abstract state 30 violating the mutual exclusion property, where for both processes the program counter has value 7. The simulation of the error trace on the concrete system indicates that it does not correspond to an execution of the concrete system. However, this does not rule out the possibility that the property is violated. In the next section, we show how model checking can guide the automatic refinement of an abstract system until the property is verified or a counterexample corresponding to a concrete execution violating the property is generated.

6 Automatic Refinement of Abstractions

Unlike current model checking tools, the error trace we generate is a tree indicating the states where abstract variables are nondeterministically assigned. In Figure 7, states 9 and 12 indicate loss of information on, respectively, the abstract variables B_1, B_3, and B_2. The concrete system is deterministic. Thus,

```
bakery : SYSTEM
BEGIN
 process_1 : PROGRAM
 $y1,\ x1,\ t1$ : VAR nat
 BEGIN
 p1_init_x1  1: TRUE                       $\rightarrow x1 := 1$      : 2
 p1_init_t   2: TRUE                       $\rightarrow t1 := y2{+}1$ : 3
 p1_init_y   3: TRUE                       $\rightarrow y1 := t1$     : 4
 p1_init_x0  4: TRUE                       $\rightarrow x1 := 0$      : 5
 p1_Wait     5: $x2 = 0$                   $\rightarrow$ $SKIP$       : 6
 p1_In       6: $y2{=}0 \ \vee \ y1 \leq y2$ $\rightarrow$ $SKIP$     : 7
 p1_Out      7: TRUE                       $\rightarrow y1 := 0$      : 1
 END process_1
    ||
 process_2 : PROGRAM
 $y2,\ x2,\ t2$ : VAR nat
 BEGIN
 p2_init_x1  1: TRUE                       $\rightarrow x2 := 1$      : 2
 p2_init_t   2: TRUE                       $\rightarrow t2 := y1{+}1$ : 3
 p2_init_y   3: TRUE                       $\rightarrow y2 := t2$     : 4
 p2_init_x0  4: TRUE                       $\rightarrow x2 := 0$      : 5
 p2_Wait     5: $x1 = 0$                   $\rightarrow$ $SKIP$       : 6
 p2_In       6: $y1{=}0 \ \vee \ y2 < y1$  $\rightarrow$ $SKIP$       : 7
 p2_Out      7: TRUE                       $\rightarrow y2 := 0$      : 1
 END process_2
INITIALLY : $y1{=}0 \ \wedge \ y2{=}0 \ \wedge \ x1{=}0 \ \wedge x2{=}0 \ \wedge t1{=}0 \ \wedge t2{=}0 \ \wedge pc1{=}1 \ \wedge \ pc2{=}1$
END bakery
```

Fig. 5. Bakery transition system (version C)

in an execution of the concrete system, each abstract state s, such as the abstraction of s is state 9, has only one successor by the transition **p1_init_y**. Also, each state s such as the abstraction of s is state 12, has only one successor by the transition **p2_init_y**. However, if the error trace is a sequence and not a tree, that is all assignments in the sequence are deterministic, the following theorem allows us to conclude that the error trace corresponds to a sequence of concrete transitions violating the property. The theorem is a corollary of Theorem 2.

Theorem 5. *Let Let S be a concrete system, and let S^a be a boolean abstraction of S using any set of predicates that includes all the literals appearing in the guards of S and in the property φ. every sequence of transitions in S^a where all assignments are deterministic is a sequence of transitions of S. We call such a sequence a deterministic trace.*

Our refinement methodology consists in computing a new abstract system with more abstract variables. This is done by enriching the current abstract state by adding additional predicates, and therefore additional abstract boolean variables.

```
bakery : SYSTEM
B3 : VAR bool
BEGIN
 process_1 : PROGRAM
 B1, B4 : VAR bool
 BEGIN
 p1_init_x1 1: TRUE      → B4 := F                : 2
 p1_init_t  2: TRUE      →  SKIP                  : 3
 p1_init_y  3: TRUE      → B1 :=?, B3 :=?         : 4
 p1_init_x0 4: TRUE      → B4 := T                : 5
 p1_Wait    5: B5        →  SKIP                  : 6
 p1_In      6: B2 ∨ B3   →  SKIP                  : 7
 p1_Out     7: TRUE      → B1 := T, B3 := T       : 1
 END process_1
    ||
 process_2 : PROGRAM
 B2, B5 : VAR bool
 BEGIN
 p2_init_x1 1: TRUE      → B5 := F                                    : 2
 p2_init_t  2: TRUE      →  SKIP                                      : 3
 p2_init_y  3: TRUE      → B2 :=?,
                           B3 := if B1 then T else ?                  : 4
 p2_init_x0 4: TRUE      → B5 := T                                    : 5
 p2_Wait    5: B4        →  SKIP                                      : 6
 p2_In      6: B1 ∨ ¬B3  →  SKIP                                      : 7
 p2_Out     7: TRUE      → B2 := T,
                           B3 := if B1
                                   then T
                                   else if ¬B1 ∨ ¬B3 then F else ? : 1
 END process_2
INITIALLY : B1 ∧ B2 ∧ B3 ∧ B4 ∧ B5 ∧ pc1=1 ∧ pc2=1
END bakery
```

Fig. 6. Abstract version of Bakery transition system (version C)

We use Theorem 5 in order to construct a new abstract system that may produce more error traces that are deterministic. That is, by eliminating the nondeterminism in the current error traces. This is done by computing the constraints under which the system may execute one of the nondeterministic transitions. These constraints are captured as preconditions and computed using the predicate transformer *pre*. We use the following lemma, allowing an efficient computation of preconditions for assignments.

Lemma 1. *Let τ be a transition. If $guard(\tau)$ is equivalent to true, then*

$$\forall P.\ pre[\tau](P) \ \equiv\ \neg pre[\tau](\neg P)$$

This lemma indicates that when computing a precondition for assignments, it is not necessary to compute it for both the predicate and its negation. Let us consider the case of the Bakery protocol. The error trace indicates that nondeterminism is created for transitions **p1_init_y** and **p2_init_y** at, respectively, states 9 and 12. The refinement technique is applied to each of these states by computing the preconditions for each boolean variable that is assigned the value ? as follows:

– refining state 9:

$$pre[\mathbf{p1\_init\_y}](y1 = 0) \ \equiv\ t1 = 0$$

$$pre[\mathbf{p1\_init\_y}](y1 \leq y2) \equiv t1 \leq y2$$

– refining state 12:

$$pre[\mathbf{p2\_init\_y}](y2 = 0) \ \equiv\ t2 = 0$$

Three new predicates $t1 = 0$, $t2 = 0$ and $t1 \leq y2$ corresponding to the new abstract variables $B6$, $B7$, and $B8$ are generated. Each transition where a variable is not assigned with the value **T** or **F** is refined. The refinement of the transition **p1_init_y**

$$3{:}\ \ \text{TRUE} \rightarrow B1 :=?,\ B3 :=? : 4$$

where $B1$ and $B3$ correspond to $y1 = 0$ and $y1 \leq y2$ is the transition

$$\begin{array}{ll} 3{:}\ \ \text{TRUE} \rightarrow & B1 := if\ B6\ then\ \mathbf{T}\ else\ if\ \neg B6 \vee \neg B8\ then\ \mathbf{F}\ else\ ?, \\ & B3 := if\ B6\ then\ \mathbf{T}\ else\ ? \qquad\qquad\qquad\qquad : 4 \end{array}$$

The refinement algorithm uses a refined way of computing the values $b_i^{\mathbf{T}}$ and $b_i^{\mathbf{F}}$

$$b_i^{\mathbf{T}} \equiv \bigvee\{B^a \mid \gamma(\mathcal{R}_\tau^a) \wedge post[\tau](\gamma(B^a)) \Rightarrow \gamma(B_i)\}$$

$$b_i^{\mathbf{F}} \equiv \bigvee\{B^a \mid \gamma(\mathcal{R}_\tau^a) \wedge post[\tau](\gamma(B^a)) \Rightarrow \neg\gamma(B_i)\}$$

where $\mathcal{R}_\tau^a$ is a boolean expression representing the set of reachable states of the already constructed abstract system at the source location of τ. For instance, $\mathcal{R}_{\mathbf{p1\_Try}}^a$ of *Bakery_A* is equal to $B_1 \vee B_2$. The expression B^a is any expression over the union of the new set of variables and set of the old one that satisfy the invariant $\mathcal{R}_\tau^a$. Thus, each refinement step uses the results of model checking the

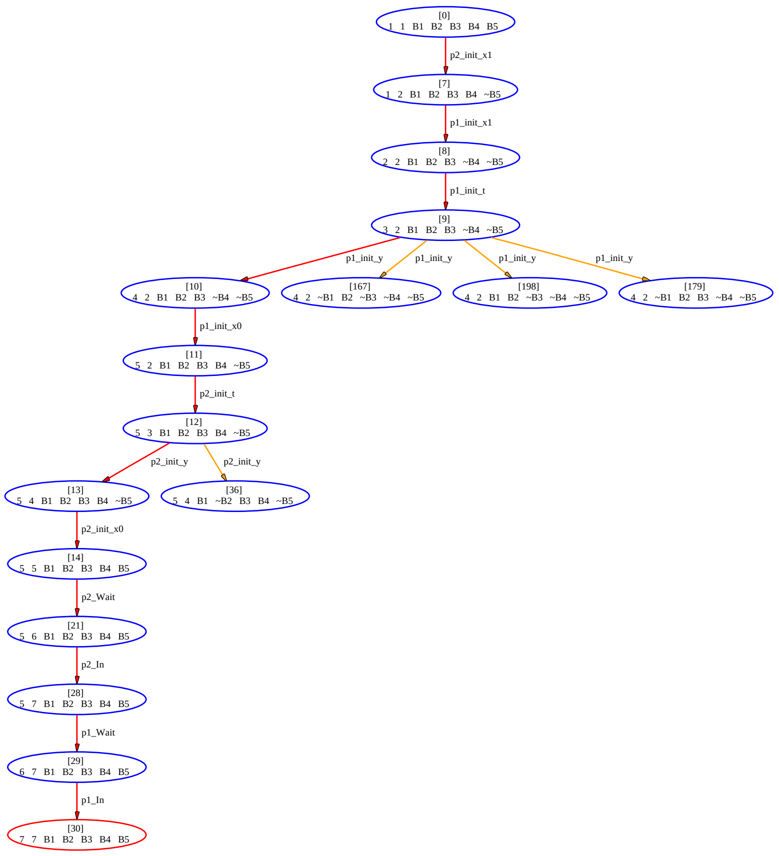

Fig. 7. Error trace for the Bakery Protocol

constructed abstract system to generate new abstract variables and to reduce the cost of the refinement algorithm. Furthermore, the invariant $\mathcal{R}^a_\tau$ refers to variables written by the component where τ belongs and to variables that are modified by other components that form its environment. The new generated predicates are used as new abstract boolean variables to compute a refined abstract system. The new abstract system is then analyzed and a new error trace indicates that mutual exclusion is violated. However, the trace is not deterministic, and a new refinement step is performed where two new predicates $y1 \leq t2$ and $t1 \leq t2$ corresponding to the new boolean variables $B9$ and $B10$ are generated. A new abstract system is then generated and analyzed, and the property is proved to be a property of the abstract system. Thus by Theorem 1, it is a property of the Bakery protocol.

In general, the an abstract system obtained after refinement is a more precise an accurate abstraction of the corresponding original system.

Theorem 6 (refinement simulation). *Let S^a be an abstraction of a system S using a set of predicates $\{\varphi_1, \cdots, \varphi_k\}$. Let S^a_r be a refinement of S^a using the additional predicates $\{\varphi_{k+1}, \cdots, \varphi_j\}$. Then, S^a simulates S^a_r.*

Proof. The proof of the theorem can be established by proving that for each abstract predicate P^a, the set of successors of P^a with respect to an abstract transition τ^a is smaller that the set of successors of P^a with respect to the corresponding refined transition τ^a_r of S^a_r. That is:

$$\forall P^a.\ post[\tau^a_r](P^a) \ \Rightarrow\ post[\tau^a](P^a)$$

Thus, the concretization of the set of reachable states of the abstract system is a more refined invariant of the concrete system. It is a more precise approximation of the reachable state of the concrete systems. Even when a property can not be established after a number of successive refinement steps, one can use this invariant as a starting point for a more elaborate proof and analysis technique using for instance a theorem prover. It is in fact necessary for even very simple systems and property to provide an invariant in order to be able to achieve a correctness proof.

7 Implementation and Analysis Methodology

We have implemented the abstraction/model checking/refinement methodology in a tool dedicated to the verification of infinite state systems. Figure 8 shows the architecture of the tool. Our tool is built on top of the PVS theorem prover. We explain the role of each component of the tool and how the analysis process is organized.

Syntax: Systems can be described in a Simple Programming Language (SPL), close to the one used in [22], but with the rich data types and expression definition mechanism available in PVS. Our SPL language includes common algorithmic constructions such as single and multiple assignment statements, conditionals If-Then-Else, and loop statements. We also allow parallel composition by interleaving and synchronization by shared variables as in Unity [4]. 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. It is possible to import any defined PVS theory. The examples in this paper are presented in the automatically generated LaTeX format for guarded commands.

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 transition systems a representation that is also a class. An important aspect of such a structure is that it is

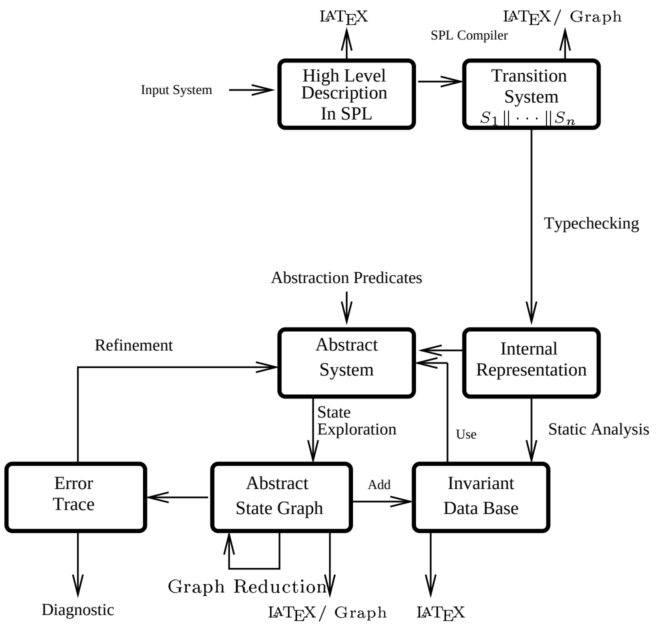

Fig. 8. Analysis methodology

independent of the PVS internal structure, and makes our implementation independent of 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 decision procedures.

Static analysis: We use the techniques developed in [24] to generate useful invariants of the concrete system. Static analysis consists in a set of techniques for the automatic generation of such invariants. These techniques are based on propagation of guards and assignments through program control points. The techniques we use computes invariants for each component and are composed using a novel composition rule presented in [24] to form invariants of the global system. These invariants are used to weaken all the implications that are generated when an abstraction is computed. When used, the allow a more efficient construction of abstractions. That is, one can decide with the help of these invariants that a variable is not assigned the value ? but either **T** or **F**, and thus, allows to generate less implications.

Automatic abstraction: The abstraction module takes a transition system and builds a first abstraction using the predicates appearing in the guards and the property to verify, and then submits the abstract system to our model checker. This module is also used for automatic refinement.

Model checking: The state-space of the constructed abstract system can be explored in two ways. In the symbolic approach, the system is translated into a boolean function represented by a BDD that represents the successor function. The exploration consists in applying the function recursively starting from the initial abstract state, represented also by a BDD until a fix point is reached. In the explicit approach, it consists in translating the abstract system into an executable form and then running it and by hashing the visited states. Both approaches can be exploited to construct the corresponding abstract state graph. The abstract state graph can then be reduced using simulation and bisimulation minimization algorithms as a way of performing additional abstractions.

Experiments: We have used our analysis methodology to verify several communication protocols such as the alternating bit protocol and a data link protocol. We also applied our methodology on several parametrized systems that are compositions of arbitrary numbers of identical processes. Figure 7 shows our experiments with three versions of the Bakery protocol. The versions *Bakery_A* and *Bakery_C* were described previously and illustrated in Figures 1 and 5. The version *Bakery_B* is obtained by removing the transitions **init_x0** and **Init_x1** from the description of *Bakery_C*. Figure 7 shows the number of predicates used in the compute a first abstraction, the refinement steps used to reach a conclusive result, that is either the property is verified, or to generate a deterministic error trace. It shows, the number of predicates computed each refinement step. It shows the numbers of implications generate and proved for each abstraction/refinement step, and the duration of each step. It also shows a comparison with our previous work in [25] where transitions systems are given as relational predicates, and where the numbers of implications is much higher as shown by Theorem 4. Notice that in general the complexity of each refinement step is less than the complexity of the computation of the first abstraction. The version *Bakery_B* is shown to violate the mutual exclusion property, and a deterministic error trace is generated after two refinements steps.

	#of initial predicates	#of refinements steps	#of new predicates	#of calls to the decision procedure	comparison with [25]	time (s)
Bakery_A	3	0	_	27	33	1.8
Bakery_B	3	2	_	72	100	5.1
			3	32	178	3.3
			4	134	366	15.5
Bakery_C	5	2	_	120	168	12
			3	35	94	3.2
			2	32	136	3.4

Fig. 9. Experiments results for 3 versions of the Bakery protocol

8 Conclusion and Future Work

We presented a general, yet effective, methodology for the verification of large systems, based on abstraction followed by model checking. The novelty of our methodology consists of an efficient algorithm for the automatic construction of boolean abstractions and an efficient algorithm for automatically refining a coarse abstraction when model checking the abstract system fails. This methodology also allows in many cases the generation of counterexamples, that is executions violating the property of interest. Our abstraction algorithm can be used to compute abstraction for any abstract domain which is a boolean algebra. Our verification tool represents the core of a verification and analysis technology for large software. The first step will be to translate source code into transition systems. For large programs, thousands of calls to the decision procedure are necessary. This can be done in few minutes or at most few hours.

References

1. 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.
2. S. Bensalem, Y. Lakhnech, and Hassen Saïdi. Powerful techniques for the automatic generation of invariants. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 323–335, New Brunswick, NJ, July/August 1996. Springer-Verlag.
3. Nikolaj Bjorner, Anca Browne, and Zohar Manna. Automatic Generation of Invariants and Intermediate Assertions. *Theoretical Computer Science*, 1997.
4. K. Mani Chandy and Jayadev Misra. *Parallel Program Design*. Addison-Wesley, Reading, Massachusetts, 1988.
5. Ching-Tsun Chou. Simple proof techniques for property preservation via simulation. *Information Processing Letters*, 60(3):129–134, 1996.
6. 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.
7. 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.
8. 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.
9. D. Dams. *Abstract interpretation and partition refinement for model checking.* PhD thesis, Technical University of Eindhoven, July 1996.
10. D. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems: Abstractions preserving $\forall$CTL*, $\exists$CTL* and CTL*. In Ernst-Rudiger Olderog, editor, *IFIP Conference PROCOMET'94*, pages 561–581, 1994.
11. S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. *Lecture Notes in Computer Science*, 1633:160–??, 1999.

12. J. Dingel and Th. Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In *Proc. of 7th CAV 95, Liège*. LNCS 939, Springer Verlag, 1995.
13. S. Graf. Characterization of a sequentially consistent memory and verification of a cache memory by abstraction. *Distributed Computing*, 1995.
14. 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.
15. J.F. Groote and J. van de Pol. A bounded retransmission protocol for large data packets. Technical report, Department of Philosophy, October 1993.
16. 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.
17. L. Helmink, M.P.A. Sellink, and F.W. Vaandrager. Proof-checking a data link protocol. Technical report, Department of Philosophy, Utrech University, The Netherlands, March 1994.
18. Bertrand Jeannet, Nicolas Halbwachs, and Pascal Raymond. Dynamic partitioning in analyses of numerical properties. In Agostino Cortesi and Gilberto Filé, editors, *Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 39–50. Springer, 1999.
19. R.P. Kurshan. *Computer-aided verification of coordinating processes, the automata theoretic approach.* Princeton Series in Computer Science. Princeton University Press, 1994.
20. David Lesens and Hassen Saïdi. Automatic verification of parameterized networks of processes by abstraction. In Faron Moller, editor, *2nd International Workshop on Verification of Infinite State Systems: Infinity '97*, volume 9 of *Electronic Notes in Theoretical Computer Science*, Bologna, Italy, July 1997. Elsevier.
21. 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.
22. Zohar Manna and Amir Pnueli. *The Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, 1995.
23. S. Owre, N. Shankar, and J. M. Rushby. A tutorial on specification and verification using pvs. Technical report, Computer Science Laboratory, SRI International, February 1993.
24. H. Saïdi. Modular and incremental analysis of concurrent software systems. In *14th IEEE International Conference on Automated Software Engineering*, pages 92–101, Cocoa Beach, FL, October 1999. IEEE Computer Society Press.
25. Hassen Saïdi and Natarajan Shankar. Abstract and model check while you prove. In *Computer-Aided Verification, CAV '99*, Trento, Italy, July 1999.