# Modular and Incremental Analysis of Concurrent Software Systems*

**In proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE'99), Cocoa Beach, Florida.**

Hassen Saïdi
System Design Laboratory, SRI International
Menlo Park, CA 94025, USA
saidi@sdl.sri.com

## Abstract

*Modularization and abstraction are the keys to practical verification and analysis of large and complex systems. We present in an incremental methodology for the automatic analysis and verification of concurrent software systems. Our methodology is based on the theory of abstract interpretation. We first propose a compositional data flow analysis algorithm that computes invariants of concurrent systems by composing invariants generated separately for each component. We present a novel compositional rule allowing us to obtain invariants of the whole system as conjunctions of local invariants of each component. We also show how the generated invariants are used to construct, almost for free, finite state abstractions of the original system that preserve safety properties. This reduces dramatically the cost of computing such abstractions as reported in previous work. We finally give a novel refinement algorithm that refines the constructed abstraction until the property of interest is proved or a counterexample is exhibited. Our methodology is implemented in a framework that combines deductive methods supported by theorem proving techniques and algorithmic methods supported by model checking and abstract interpretation techniques.*

## 1 Introduction

Modularization and abstraction are the keys to practical verification and analysis of large and complex systems. Finite state verification techniques such as model checking have reached a saturation point. Deductive methods supported by theorem proving tools, despite their generality, are still limited in their ability to handle large systems. Only the combination of algorithmic verification techniques and deduction, through the extensive use of modularization and abstraction, can handle the verification of large and complex systems. Invariants – that is, approximations of the reachable states – are necessary for the success of the verification process in both software and hardware analysis. When an invariant of a program is known, it can be used to reduce the search space for model checking, or it can be used to constraint the logical formula representing the program. In fact, to prove any temporal property, it is often the case that the property cannot be established without the use of auxiliary invariants.

In this paper we describe a practical approach to the verification of safety properties of infinite state systems based on abstract interpretation [5]. We first propose a compositional data flow analysis algorithm that computes invariants associated with each control location in each component. We present a novel compositional rule allowing us to obtain invariants of the whole system as conjunctions of local invariants of each component. We also show how the generated invariants allow us to construct, almost for free, finite abstractions of the analyzed system that preserve safety properties. That is, the property holds in the concrete system *if* it holds in the abstract one. This reduces dramatically the cost of computing such abstractions as reported in previous work. We finally give a novel refinement algorithm that refines the constructed abstraction until the property of interest is proved or a counterexample is exhibited. We also give sufficient conditions under which a computed abstraction strongly preserves a large class of temporal properties including safety properties. That is, the property holds in the concrete system *if and only if* it holds in the abstract system. The contributions of this work can be summarized as follows:

- Our invariant generation algorithm is a generalization of static analysis methods investigated in the early 1970s for sequential programs [12, 8] and extended to concurrent systems by Owicki and Gries [16], Lamport [13], and recently by others [3] and [2].
- Our method is compositional in the sense that global invariants are associated control locations in each component and not to global control configurations of the entire system. Thus, invariants associated to global control configurations can be obtained by a simple *conjunction* of the invariants

*This research was supported by the National Science Foundation under Grant No. CCR-9509931 and DARPA contract F30602-97-C-0040. I would like to thank my colleagues Natarajan Shankar and John Rushby for their helpful comments on earlier versions of this paper.

collected for each control location of each component.
- The generated invariants allow us to obtain, almost for free, finite abstractions of the analyzed system. An abstract state graph such as the ones we construct using techniques described in [10], can be obtained easily by simple evaluation of the invariants generated for all global control configurations. If the invariant associated to a control configuration is false, the control configuration is considered as non-reachable. In all the classical mutual exclusion algorithms we have treated in previous work using various techniques [9, 10], the generated invariants using our techniques for the control configuration corresponding to the violation of the mutual exclusion property are equivalent to false, which means that the property of mutual exclusion holds.
- If the obtained abstract state graph is not precise enough to prove that a safety property holds, we propose a novel refinement algorithm that refines automatically the abstract state graph until the property is proved or a counterexample is exhibited. In contrast with other recent works on automatic construction of abstractions [10, 4, 1], we give a sufficient condition that ensures that an error trace of the abstract state graph corresponds to an error trace in the original system. This gives us a powerful validation/refutation technique for safety properties of infinite state systems.
- Our methodology is incremental. First, an approximation of the reachable states is obtained using the data flow analysis techniques. The generated invariants define an initial abstraction that is refined until the property of interest is proved or a counterexample is found. At any moment, the refined abstraction defines an invariant of the system.
- Our method is implemented in a tool [18] built on the top of a theorem prover PVS [17]. Our implementation allows us to use the generated invariants to prove properties using deduction as well as to generate finite abstractions using decision procedures.

The paper is organized as follows: in Section 2, we present the model in which systems are described, we give some basic definitions, and we present our algorithm for the automatic generation of invariants for one component. In Section 3, we extend the algorithm to the case of parallel composition. In Section 4, we show how finite abstractions can be constructed using the generated invariants. In Section 5 we present our refinement technique. In Section 6, we report some experimental results.

## 2 Compositional data flow analysis

We consider systems that are parallel compositions of sequential processes, where we consider parallel composition by interleaving and synchronization by shared variables. Each component is described as a set of transitions or guarded commands. 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 the program counter $pc$) ranging over arbitrary domains, such as booleans, integers, lists, and abstract data types, $\mathcal{T}$ is a set of transitions or guarded commands, $\mathcal{L}$ is a set of control locations or control points, and $Init$ is a predicate characterizing the set of initial states. Each transition $\tau$ is a guarded command

$$l: \; guard \longrightarrow X_1 := Exp_1, \cdots, X_k := Exp_k \quad : k$$

where $\{X_1, \cdots, X_k\} \subseteq \mathcal{V}$ and $\{l, k\} \subseteq \mathcal{L}$. Locations $l$ and $k$ are respectively the source and target location of the transition $\tau$. When a program is given by a set of transitions $\mathcal{T} = \{\tau_1, \cdots, \tau_n\}$, where each transition is defined as a guarded command, the predicate transformers $post$ and $\widetilde{pre}$ expressing respectively the strongest postcondition and the weakest precondition are defined as follows:

$$post[\tau](P) = \exists \mathcal{V}'.action_\tau(\mathcal{V}', \mathcal{V}) \wedge P(\mathcal{V}')$$
$$\widetilde{pre}[\tau](P) = \forall \mathcal{V}'.action_\tau(\mathcal{V}, \mathcal{V}') \Rightarrow P(\mathcal{V}')$$

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

$$pc = l \wedge guard \wedge \bigwedge_{i=1}^{k} X_i' = Exp_i \wedge pc' = k$$

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. An invariant is a predicate $P$ over the program variables satisfying one of the following equivalent conditions:

$$post[\tau](P) \Rightarrow P \qquad\qquad P \Rightarrow \widetilde{pre}[\tau](P)$$

That is, $P$ is preserved by every transition of the program. Any predicate $Q$ such that $P \Rightarrow Q$ is also an invariant, that is, a superset of the reachable state of the system. $P$ is then *stronger* (implies) $Q$. We present an algorithm for the generation of invariants for concurrent systems based on data flow analysis. The algorithm is based on the propagation of expressions through the control structure of each component of the system. The algorithm is an improvement of the algorithm presented in [2], which is restricted to propagations through particular loops of each component. First, we consider the following definitions. Let $l$ be a control point of $\mathcal{L}$. The set of transitions incident to $l$ we note $incident(l)$ is the set of transitions $\tau$ such that $target(\tau) = l$. For each transition $\tau$, we note $\mathbf{A}(\tau)$ the set of variables affected in $\tau$, that is, the left-hand side of the assignments expressions $X_1 := Exp_1; \cdots; X_k := Exp_k$. For a set $E = \{p_1, \cdots, p_n\}$ of predicates, we note $\bigwedge E$ the predicate $\bigwedge_{i=1}^{n} p_i$, with the convention that $\bigwedge \emptyset = true$.

**Structural invariants** Our invariant generation technique associates to each control location a predicate that is always true at this location. We call these invariants *structural invariants*. Our goal is to generate an invariant associated to the location $l$ no matter via which transition $l$ is reached. We call such a predicate a *reaffirmed invariant*.

**Definition 1 (Reaffirmed invariant)** *Let $l$ be a control location of a program $Prog$. A reaffirmed invariant $\mathtt{Reaf}(l)$ is a predicate that is true at location $l$ when $l$ is reached via any transition of the program. That is, $\mathtt{Reaf}(l)$ verifies $\forall P \,.\, post[\tau](P) \Rightarrow \mathtt{Reaf}(l)$ for any predicate $P$ and for all transitions $\tau$ such that $\tau \in incident(l)$.*

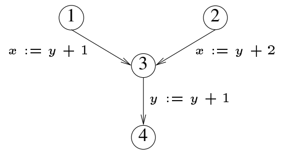


**Figure 1. Example of reaffirmed invariants**

In particular, the predicate $\mathcal{P}red(\tau)$ defined as follows:

$$\mathcal{P}red(\tau) = post[\tau](true)$$

is a reaffirmed invariant at location $l$ when $l$ is reached via the transition $\tau$. Consider the example of Figure 1, where $x$ and $y$ are integers. For locations 3 and 4, we obtain the following reaffirmed invariants: $\mathtt{Reaf}(3) \equiv pc{=}3 \Rightarrow (x = y+1) \vee (x = y+2)$, and $\mathtt{Reaf}(4) \equiv pc = 4 \Rightarrow true$. For a given control location $l$ of a program $Prog$ such that $incident(l) = \{\tau_1, \cdots, \tau_j\}$, the following predicates are invariants of $Prog$:

- $(pc{=}l) \Rightarrow \bigvee_{i=1}^{j} \mathcal{P}red(\tau_i)$ if $l$ is not the initial control location, and
- $(pc = l) \Rightarrow Init \vee \bigvee_{i=1}^{j} \mathcal{P}red(\tau_i)$ if $l$ is the initial control location.

We note $\mathtt{Inv}(l)$ the set of predicates that are true at control location $l$. Predicates such as reaffirmed invariants can be propagated from one location to another if their free variables are not affected.

**Proposition 1** *Let $l$ be a control location and $incident(l) = \{\tau_1, \cdots, \tau_m\}$, and $\{Q_1, \cdots, Q_m\}$ the set of invariants at the source locations of transitions $\{\tau_1, \cdots, \tau_m\}$, that is, $Q_i \in \mathtt{Inv}(source(\tau_i))$. The predicate $(\mathcal{P}red(\tau_1) \wedge Q'_1) \vee \cdots \vee (\mathcal{P}red(\tau_m) \wedge Q'_m)$ is an invariant at location l, and*

$$Q'_i = \begin{cases} Q_i & if \qquad FV(Q_i) \cap \mathbf{A}(\tau_i) = \emptyset \\ true & otherwise \end{cases}$$

In the example of Figure 1, if respectively at location 1 and 2, the predicates $y \geq 1$ and $y \geq 0$ hold, the invariant obtained at location 3 is then $(x = y + 1 \wedge y \geq 1) \vee (x = y + 2 \wedge y \geq 0)$. This predicate is an invariant at location 3 but cannot be propagated to location 4 since the variable $y$ is affected by the transition between locations 3 and 4. However, it is possible to propagate a weaker predicate that does not depend on the variable $y$. This weaker predicate is obtained by *hiding* the variable $y$ using existential quantification.

**Definition 2 (abstraction of an invariant)** *Let $\varphi$ be a predicate, and let $V$ be a set of variables. The abstraction of $\varphi$ with respect to $V$ is the predicate $\exists V \,.\, \varphi$. We note this predicate $\mathtt{Abs}(\varphi, V)$.*

It is now possible to propagate the predicate $\mathtt{Abs}(I, \{y\})$, that is, the predicate $\exists y \,.\, (x = y + 1 \wedge y \geq 1) \vee (x = y + 2 \wedge y \geq 0)$. In the case where $y$ is a positive integer, this predicate can be simplified to $x \geq 2$.

**Proposition 2** *Let $\varphi$ be a predicate, $l$ a control location, and $V$ a set of variables. The following implication is valid: $\varphi \in \mathtt{Inv}(l) \Rightarrow \forall V \,.\, \mathtt{Abs}(\varphi, V) \in \mathtt{Inv}(l)$.*

We can now reformulate proposition 1 as follows:

**Proposition 3** *Let $l$ be a control location and $incident(l) = \{\tau_1, \cdots, \tau_m\}$, and let $\{Q_1, \cdots, Q_m\}$ be the invariants at the source locations of the transitions $\{\tau_1, \cdots, \tau_m\}$, that is, $Q_i \in \mathtt{Inv}(source(\tau_i))$. The following predicate is an invariant at location l*

$$\bigvee_{i=1}^{m} (\mathcal{P}red(\tau_i) \wedge \mathtt{Abs}(Q_i, \mathbf{A}(\tau_i)))$$

**An algorithm for the generation of invariants** The algorithm we propose in Figure 2 combines the propagation and the abstraction of reaffirmed invariants computed for each location. The algorithm consists in computing for the current location $l$ the expression $\mathtt{Reaf(l)}$. This expression is propagated through the control structure of the program. The algorithm terminates when all the locations are stable, that is, the propagation of any structural invariant associated to a location does not provide a stronger invariant. The correction and termination of the algorithm are established by theorem 1.

**Theorem 1** *The invariants generation algorithm always terminates and for every location l, the predicate $pc = l \Rightarrow \bigwedge \mathtt{Inv}(l)$ is an invariant of $Prog$.*

```
input : Program Prog = < V, T, L, Init >
Begin
Initialization    /* Compute Reaf */
  For all l ∈ L do
  | compute Reaf(l) ; Inv(l) := { Reaf(l) }
 non_stable := L
Iteration    /* Propagation */
  While non_stable ≠ ∅ Do
    choose l ∈ L
      /* Propagating and updating Inv */
      For all transition τi ∈ T
      such that target(τi) = l do
          Ii := Abs(⋀Inv(l), A(τi))

          I := ⋁Ii

          If I ∉ Inv(l) Then
            Insert I in Inv(l)
            non_stable := non_stable ∪ l
      If  Inv(l) is not modified
       Then non_stable := non_stable \ l
End
∀l ∈ L . pc = l ⇒ ⋀Inv(l)    is an invariant of Prog.
```

**Figure 2. Invariants generation algorithm for a single component**

**Proof:** The fact that the above algorithm associates to each location a predicate that is an invariant can be easily justified by propositions 1, 2, and 3. The proof of termination can be established by showing that for each location $l$, the set $\texttt{Inv}(l)$ is a finite set of expressions. $\texttt{Inv}(l)$ is initialized with the expression $\texttt{Reaf}(l)$. It is updated only with expressions of the form $\texttt{Reaf}(l')$ corresponding to any other location $l'$, or with expressions of the form $\texttt{Abs}(\bigwedge \texttt{Inv}(l'), V)$ for a location $l'$ and a subset $V$ of program variables. Since the numbers of program variables and control locations are finite, the set $\texttt{Inv}(l)$ is finite.
□

## 3 Structural invariants of concurrent programs

We consider systems that are parallel compositions of sequential processes, where we consider parallel composition by interleaving and synchronization by shared variables. A trivial way of generating invariants of a parallel system is to construct the asynchronous product of the control structures of all the components, and to apply the previous algorithm. This solution can be used only when we consider a small number of parallel components, each of them with a small number of control locations. For large examples, only compositional techniques can be applied in practice. Most of the work presented on the generation of invariants is based on the compositional approach. However, compositionality is considered in a very restrictive way. In [3], structural invariants of a component depend only on variables that are modified only in the component. Thus, such invariants are trivially invariants of the global system since they are preserved trivially by all transitions in the other components. Techniques described in [2] are more powerful and are based on the combination of local invariants to generate global ones. The idea is based on the abstraction of local invariants in order to have invariants depending only on local variables, which are then trivially preserved by all transitions in the other components. This is done using the definition of abstraction of invariants we gave in Section 2. Another approach proposed in [2] consists of *composing* local invariants to generate global invariants associated with global control configurations. The composition of invariants is defined as follows :

**Proposition 4 (Composition of invariants)** *Let $P_1$ and $P_2$ be two processes. Let $Q_1$ and $Q_2$ be respectively reaffirmed invariants at locations $l_1$ and $l_2$ of $P_1$ and $P_2$. The predicate $Q_1 \vee Q_2$ is an invariant of $P_1 || P_2$ at the global control configuration $(l_1, l_2)$. That is, $pc1{=}l_1 \wedge pc2{=}l_2 \Rightarrow Q_1 \vee Q_2$ is an invariant of $P_1 || P_2$.*

The composition of invariants is restricted in this case to reaffirmed invariants, which are generated by considering only the last transitions in each component that lead to the considered control configuration. Moreover, the invariants constructed by composition are not propagated to other control configurations. One of the main contributions of this paper is to propose new techniques allowing the generation of stronger invariants. Our techniques consist of

- proposing a new definition of abstraction of invariants. Quantifying invariants is too restrictive. Instead of hiding a variable using existential quantification, it is necessary to consider its different values in the different components to have a less restrictive definition.
- proposing a composition technique that is not restricted to the composition of reaffirmed invariants such as described in [2], but which allows composing predicates that are propagated through the control structure of each component. The invariant for a global control configuration $(l_1, l_2)$ is simply the conjunction of the invariants generated compositionally for $l_1$ and $l_2$ by our method.

**Definition 3 (Context expressions)** *Let $x$ be a shared variable in a parallel system $S = P_1 || \cdots || P_n$. The context expression associated to $x$ in $S$ noted $\mathcal{CE}_{\{S\}}(x)$ is the disjunction*

$$\bigvee_{i=1}^{k} assign(x,\tau_i) \bigvee_{y} \mathcal{CE}_{\{S\}}(y)$$

*where*

- $\tau_i$ *is any transition of any component of* $S$.
- $assign(x,\tau_i)$ *is defined for each transition* $\tau_i$ *as follows:*

  $$\bigwedge_{j=1}^{l} e_j \text{ such that, } e_j \in \mathcal{A}\!f\!f(\tau_i), \text{ and } x \in FV(e_j)$$

- $\mathcal{A}\!f\!f(\tau_i)$ *is the set of expressions* $x = Exp_i$ *such that* $x$ *does not appear free in the expression* $Exp_i$. *When this set is empty, the context expression* $\mathcal{CE}_{\{S\}}(x)$ *is equal to true.*
- $y$ *is any variable such that* $FV(e_j) \cap \{y\} \neq \emptyset$.

Intuitively, a context expression associated to a variable $x$ is the disjunction of assignment expressions involving the variable $x$ in any other component. Thus, instead of quantifying existentially over the variable $x$, one can replace this quantification by using the context expression that captures all the possible values of the variable $x$ in the other components. Notice that the definition of context expression is recursive. But since the number of variables is finite, context expressions can always be computed.

**Definition 4 (Abstraction of invariants)** *Let* $I$ *be a predicate and* $x$ *a variable appearing free in* $I$. *The abstraction* $[x]_{\{S\}}.I$ *of the variable* $x$ *in* $I$ *with respect to the system* $S$ *is defined by the expression*

$$Abs(I,\{x\}) \ \wedge\ (I \vee \mathcal{CE}_{\{S\}}(x))$$

The abstraction of invariant is the conjunction of the abstraction defined using existential quantification, and our new abstraction definition using context expressions. We can now formulate our result allowing us to generate global invariants in a compositional way.

**Theorem 2** *Let* $S = P_1 \| \cdots \| P_n$ *be a parallel system, and let* $I$ *be an invariant of* $P_i$ *at location* $l$. *The predicate*

$$pc_i{=}l \Rightarrow [\vec{x}]_{\{S'\}}.I$$

*where* $\vec{x}$ *is a set of variables appearing free in* $I$ *and that are modified in the subsystem*[1] $S'$, *is an invariant of* $S$.

[1]A subsystem $S'$ of $S$ is the parallel composition of a subset of processes of $S$.

**Proof:** $Abs(I,\{x\})$ is trivially an invariant of $S$ since it depends only on the variables that are modified in $P_i$, and $I$ is already an invariant of $P_i$. It is also easy to see that $(I \vee \mathcal{CE}_{\{S\}}(x))$ is an invariant of $S$. The disjunct $I$ is preserved by the process $P_i$, and each disjunct $assign(x,\tau_i)$ of $\mathcal{CE}_{\{S\}}(x)$ is preserved by the process to which $\tau_i$ belongs. $\square$

This proposition allows us to generate global invariants by simple conjunction of local invariants. Local invariants in this case depend on shared variables. This is not the case in [2] and [3]. The algorithm we use is the one described in Figure 2 where we use the new definition of abstraction of invariants. To illustrate the effectiveness of our method, we apply our algorithm to the well-known mutual exclusion example of *Bakery*. The algorithm is called the Bakery algorithm, since it is based on the idea that customers, as they enter, 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 the control location 3. Each process process_i modifies its local variable $y$i. The application of our algorithm to process_1 generates the following local invariants (the invariants for process_2 are obtained in a symmetric way):

$$\begin{array}{lcl}
pc1{=}1 & \Rightarrow & y1{=}0 \ \wedge\ [y2]_{\{P_2\}}.y2{=}0 \\
pc1{=}2 & \Rightarrow & [y2]_{\{P_2\}}.y1{=}y2+1 \ \wedge\ [y2]_{\{P_2\}}.y2{=}0 \\
pc1{=}3 & \Rightarrow & [y2]_{\{P_2\}}.y1{=}y2+1 \ \wedge\ [y2]_{\{P_2\}}.y2{=}0 \\
 & & \quad \wedge\ [y2]_{\{P_2\}}.y2{=}0 \vee y1 \leq y2
\end{array}$$

where

$$\begin{array}{lcl}
\mathcal{CE}_{\{P_1\}}(y1) & \equiv & y1{=}0 \vee y1{=}y2+1 \\
\mathcal{CE}_{\{P_2\}}(y2) & \equiv & y2{=}0 \vee y2{=}y1+1
\end{array}$$

The invariant associated to the global control configuration $(3,3)$ is then the following:

$$\begin{array}{lll}
I(3,3) \equiv & & \exists y2.y1{=}y2+1 \\
 & & \wedge((y1{=}y2+1 \vee y2{=}0 \vee y2{=}y1+1) \\
 & & \ \ \wedge\ (y1{=}0 \vee y2{=}0 \vee y2{=}y1+1)) \\
 & \wedge & \exists y1.y2{=}y1+1 \\
 & & \wedge((y2{=}y1+1 \vee y1{=}0 \vee y1{=}y2+1) \\
 & & \ \ \wedge\ (y2{=}0 \vee y1{=}0 \vee y1{=}y2+1))
\end{array}$$

By construction, the predicate $pc1{=}3 \wedge pc2{=}3 \Rightarrow I(3,3)$ is an invariant of $P_1 \| P_2$. However, it easy to check, using decision procedures, that $I(3,3)$ is equivalent to *false*. That is, the predicate $\neg(pc1 = 3 \wedge pc2 = 3)$ is an invariant of Bakery.

## 4 Generating finite state abstractions

The generated invariants can be used to obtain, almost for free, finite abstract state graphs (ASGs) similar to the

```
bakery : SYSTEM
BEGIN
   process_1 : PROGRAM                                   process_2 : PROGRAM
   y1 : VAR nat                                          y2 : VAR nat
   BEGIN                                                 BEGIN
     1: TRUE              → y1 := y2+1   : 2      ||       1: TRUE              → y2 := y1+1   : 2
     2 : y2=0 ∨ y1 ≤ y2   →              : 3               2: y1=0 ∨ y2 < y1    →              : 3
     3: TRUE              → y1 := 0      : 1               3: TRUE              → y2 := 0      : 1
   END process_1                                         END process_2
INITIALLY : y1=0 ∧ y2=0 ∧ pc1=1 ∧ pc2=1
END bakery
```

**Figure 3. Bakery transition system**

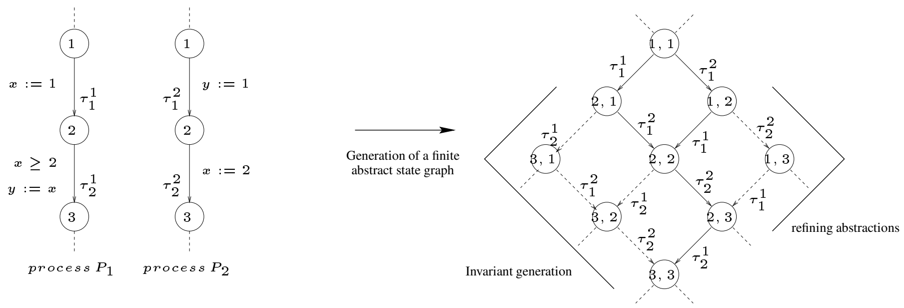


**Figure 4. Abstract state graphs using generated invariants**

ones we construct with the method described in [10] that uses decision procedures. The example of Figure 4 illustrates how the invariant generation techniques are applied compositionally and locally, that is, by considering only a small part of the control flow of each process. We apply our techniques to analyze the data flow between two particular control configurations (1,1) where $x$ and $y$ are equal to 0, and (3,3). The invariant corresponding to configuration (3,2) is $x \geq 2 \wedge y = x \wedge (y = 1 \vee y = 0) \wedge (x = 1 \vee x = 0)$, that is, the predicate *false*. Thus, the control configurations (3,2) and (3,1) are not reachable. One can then generate the abstract state graph of Figure 4 where with each reachable control configuration we associate the collected invariant. To generate stronger invariants, we can apply the algorithm presented in Section 2 on the new refined control structure of the system. This new control structure contains fewer paths than the original one. Thus, propagation is done only through *reachable* paths. A more powerful technique consists of computing boolean abstractions by successively refining the constructed ASG. In this case, it will be possible to deduce that the control configuration (2,3) is a deadlock state if it is reached from location (1,3).

**Definition 5 (Abstract state graph)** *Let $\mathcal{B} = \{B_1, \cdots, B_n\}$ a set of boolean variables. An ASG is a labeled graph where each state is a tuple $(c, B)$, such that $c$ is a global control configuration and $B$ a boolean expression of the form $\bigwedge_{i=1}^{n} b_i$, where $b_i$ is either a boolean variable $B_i$ or its negation $\neg B_i$.*

In [10], each *abstract* variable $B_i$ of the set $\mathcal{B}$ corresponds to a predicate $\varphi_i$ over the variables of the original system. Each property $P$ computed on the abstract level can be concretized using the substitution function $\gamma$ such that $\gamma(P) = P[\varphi_1/B_1 \cdots \varphi_n/B_n]$. An abstract state graph is constructed as follows:
- $(c_0, init^A)$ is the original state such that, $c_0$ is the original control configuration (for instance (1,1) for the Bakery example), and $init^A$ is equal to

$$\bigwedge_{i=1}^{k} \begin{cases} B_i & \text{if} \quad Init \Rightarrow \varphi_i \\ \neg B_i & \text{if} \quad Init \Rightarrow \neg\varphi_i \\ true & \text{otherwise} \end{cases}$$

- a transition $(s_1^A, \tau, s_2^A)$, such that $s_1^A = (c_1, I_1)$ and $s_2^A = (c_2, I_2)$, is added to the ASG if $\tau$ is a guarded command executable from $c_1$ and leads to $c_2$, and such that

$$I_2 = \bigwedge_{i=1}^{k} \begin{cases} B_i & \text{if} \quad \gamma(I_1) \Rightarrow \varphi_i \\ \neg B_i & \text{if} \quad \gamma(I_1) \Rightarrow \neg\varphi_i \\ true & \text{otherwise} \end{cases}$$

If it is not possible to establish that either $\varphi_i$ or $\neg\varphi_i$ holds, two successor states of $s_1^A$ by $\tau$ are created in which respectively $B_i$ and $\neg B_i$ holds. This creates nondeterminism in the ASG. The constructed ASG is a conservative abstraction of the original system that preserves safety properties. That is, when a property holds in the abstract system, it holds in the concrete one. However, when a property does not hold in the abstract system, the exhibited counterexample may not correspond to an execution of the concrete system, since the abstract system is an overapproximation of the concrete one. The generated invariants by our compositional data flow analysis define an abstract state graph where the set $\mathcal{B}$ is the set of literals that form the set $\mathtt{Inv}(l)$ for each location $l$. This set is finite, and contains the collected reaffirmed invariants and their abstract versions that are propagated through the control structure. When an invariant cannot be propagated, nondeterminism is created. The ASG for the Bakery example has, for instance, eight states corresponding to all reachable control configurations and the corresponding invariants. The ASG can be model checked to prove safety properties such as mutual exclusion $\Box\neg(pc1{=}3 \wedge pc2{=}3)$ and progress properties such as $\Box(pc1{=}2 \Rightarrow \Diamond pc1{=}3)$. An ASG can be used as an abstract system that can be used by a model checker to verify any preserved temporal property, such as safety properties on the control or/and where the atomic propositions are included in the set $\mathcal{B}$.

## 5 Automatic refinement of abstract state graphs

Preservation results concerning abstraction [14, 7] are established via equivalences and preorders between models. For instance, if $S$ is a concrete system and $S^a$ an abstract system and then if the respective models $K_S$ and $K_{S^a}$ of $S$ and $S^a$ are equivalent, then any $CTL^*$ property is strongly preserved. When $K_{S^a}$ simulates $K_S$, only the fragment $\forall CTL$ is preserved – that is, temporal formulas with universal quantification over paths, including safety properties such as invariants. We use the preservation results in a different way. We compute an ASG $G_S$ that is more precise than the model $K_{S^a}$ generated by state exploration techniques of an abstract system $S^a$ computed in a compositional way such as in [4, 1]. This allows us to *concretize* the ASG to a concrete system that can be used for further analysis. Equivalences and preorders can be checked between the guarded command programs instead of their models, which is much more efficient. This is done by lifting the definition of equivalence between models to equivalence between guarded command programs, where states are control states and transitions $\tau$ are guarded commands. This technique is illustrated in Figure 6. The concretization of an ASG consists of considering the abstract graph as the new control graph, where each label $\tau$ is replaced by the corresponding guarded command. We denote the concretization of an ASG $G_S$, $\Gamma(G_S)$. As a simple example, we consider the program of Figure 6, where the concrete variable $x$ is an integer initialized with 0. The abstraction algorithm we use computes a nondeterministic ASG for the abstract variable $B \equiv x \geq 2$. A more refined abstraction using two abstract variables $B_1 \equiv x < 1$ and $B_2 \equiv x{=}1$ produces a deterministic ASG. In this example, the program obtained after refining a first abstraction is equivalent to the original one. The methodology we present in this paper consists of computing invariants in a compositional way, and composing them to compute an abstract state graph that is refined until the property of interest is proved or a real counterexample is generated. When applying this method to the producer-consumer example of Figure 7, which uses a semaphore $b$ and a counter $c$ of produced items, the invariant generation fails in generating useful invariants. In fact, the predicate $true$ is collected at each control location. This occurs when all collected context expressions are equivalent to $true$. The generated ASG is then trivially the asynchronous product of the transition systems corresponding to the consumer and the producer. In this case, we propose to use the algorithm developed in [10] to compute the abstract state graph. In [10], we propose a heuristic that consists of using as set $\mathcal{B}$ of abstract variables the set of literals appearing in the guards of each guarded command of the system. This allows us to perform a useful reachability analysis, and to have an exact abstraction of the guards. We propose to complete this algorithm with an automatic refinement algorithm that computes an ASG using the literals of the guards and then uses model checking to prove any safety property. When the property is violated, an error trace is generated and analyzed. We give sufficient conditions allowing us to conclude that this trace corresponds to an execution of the concrete system. We propose a refinement algorithm that generates a new set of abstract variables. We first consider the following theorem in which case the abstract graph strongly preserves not only safety properties but also liveness properties. That is, any error trace exhibited by a model checker is indeed an error trace in the concrete system.

**Theorem 3** *Let $S$ be a system and $G_s$ an ASG computed for any set $\mathcal{B}$ including the literals appearing in the guards of the transitions of $S$. If the construction of $G_S$ does not introduce nondeterminism then, $S$ and $\Gamma(G_S)$ are equivalent.*

**Proof:** It is easy to show that any abstract state graph $G_s$ simulates the system $S$. This is usually the argument to show that a system is an abstraction of a more concrete system. To show the equivalence between $S$ and $\Gamma(G_S)$, it is sufficient to show that in this particular case the system $S$ simulates the abstract state graph $G_s$. To show this,


**Figure 5. Preservation via equivalences and preorders**

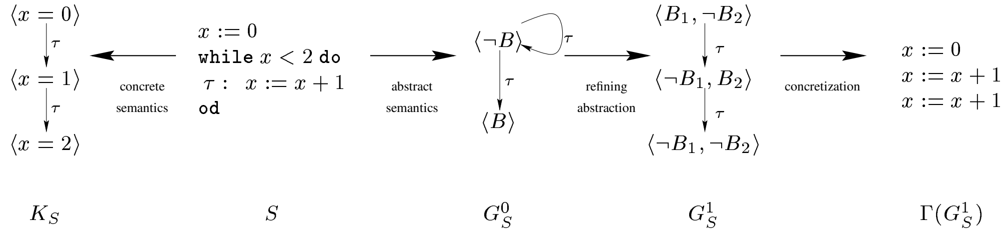


**Figure 6. Simple example of successive refinement**

it is sufficient to prove that for each pair of abstract states $s_1^A = (c_1, I_1)$ and $s_2^A = (c_2, I_2)$, if $(s_1^A, \tau, s_2^A)$ is a transition of the abstract system, then, for every pair $s_1$ and $s_2$ of states in the concretization of $s_1^A$ and $s_2^A$, the transition $(s_1^A, \tau, s_2^A)$ is a transition of the original system. Every concrete state $s_1$ in the concretization of $s_1^A$ satisfies the guard of $\tau$, and every successor $s_2$ of $s_1$ is in the concretization of $s_2^A$. Thus, $S$ simulates $G_s$ .
□

**A refinement algorithm** Our refinement algorithm consist of reducing the nondeterminism in the constructed ASG. Consider the situation of the producer-consumer example (Figure 8) when we construct the successors via the transition **synchronize** of an abstract state where the variable $B_i \equiv b > 0$ is true. Two verification conditions are generated to compute the value of $B_i$ in the next state:

$$\gamma(I) \Rightarrow \widetilde{pre}[\textbf{synchronize}](\gamma(B_i))$$

$$\gamma(I) \Rightarrow \widetilde{pre}[\textbf{synchronize}](\gamma(B_i))$$

which correspond to the assertions

$$b > 0 \Rightarrow b - 1 > 0 \quad \text{and} \quad b > 0 \Rightarrow b - 1 \leq 0$$

Of course, both assertions are not valid. Thus, nondeterminism is created for the transition **synchronize**. We propose to eliminate nondeterminism by adding for each successor state its pre-image to the guard. For transition **synchronize** and the predicate $b > 0$ the pre-images for the created states are respectively $pre[\textbf{synchronize}](b > 0)$ and $pre[\textbf{synchronize}](\neg\ b > 0)$, that is, the predicates $b > 1$ and $b = 1$. In this case, the evaluation of these predicates in the state where nondeterminism was created shows that $b = 1$ holds. Thus, only the transition on the left is possible. The abstraction algorithm starting with the predicates $b > 0$, $c > 0$, and $c < N$ appearing in the guards, and refined with the new automatically generated predicates $b > 1$ and $b = 1$, computes an ASG showing that the global control configurations (2,2), (2,4), (2,5), (4,2), (4,4), (4,5), (5,2), (5,4), and (5,5) are not reachable. This invariant is strong enough to prove all the safety properties of the producer-consumer system.

# 6 Implementation and analysis methodology

We implemented our techniques in the tool InvariantChecker [18] built on top of the PVS verification system. Programs are described in a simple algorithmic language SPL similar to the one defined by Manna and Pnueli [15]. However, variables types and expressions are PVS types and expressions. Program variables can be of any type definable in PVS, and can be assigned by any definable PVS expression of a compatible type. Our SPL includes common algorithmic constructions such as single and multiple assignment statements, conditionals, and loop statements. We also allow parallel composition by interleaving and synchronization by shared variables.Systems described in SPL are translated automatically into guarded commands with explicit control. Our invariant generation techniques are applied for each process independently from the other process, and independently from the properties we want to verify. Invariants are then composed and a first ASG is generated. The incremental construction and refinement of the ASG then starts. A model checker is used to check temporal properties that are preserved by the abstraction, such as safety properties. In our verification system, the PVS decision procedures are used as a back-end tool to check the validity of predicates without user intervention. Our tech-

```
prod_cons : SYSTEM
b, c : VAR int
N : VAR posnat
BEGIN
  producer : PROGRAM                                            consumer : PROGRAM
  BEGIN                                                         BEGIN
    t1_synchronize            1: b>0     → b := b–1   :2            1: b>0      → b := b–1   :2
    t2_wait                   2: ¬ c < N → b := b+1   :3            2: ¬ c > 0  → b := b+1   :3
    t3_exit_wait              2: c<N     →  SKIP      :4    ||      2: c>0      →  SKIP      :4
    t4_request_b              3: b>0     → b := b–1   :2            3: b>0      → b := b–1   :2
    t5_produce / consume      4: TRUE    → c := c+1   :5            4: TRUE     → c := c–1   :5
    t6_release_b              5: TRUE    → b := b+1   :1            5: TRUE     → b := b+1   :1
  END producer                                                  END consumer
INITIALLY :  c = 0 ∧ w = FALSE ∧ b = 1 ∧ pc1 = 1 ∧ pc2 = 1
END prod_cons
```

**Figure 7. Producer-consumer transition system**

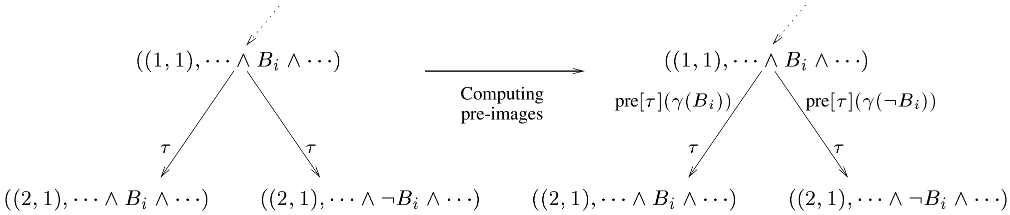


**Figure 8. Reducing nondeterminism in abstract state graphs**

niques can be integrated to any verification environment that provides decision procedures for validity checking. The invariant generation, the abstraction and refinement algorithms, do not require user intervention and are completely automatic. User intervention is needed only to select in which part of an ASG nondeterminism is reduced. That is, for which state the pre-image should be computed and added as an abstract variable. An automatic mode where this is done for all states can also be selected. We applied these techniques to several classical examples, including an alternating and a bounded retransmission protocol that we already verified by computing boolean abstractions [10]. In all these examples, the generated invariants were sufficient to generate a finite abstraction where all the properties of interest were model checkable. In [10], the BRP protocol was proved by computing a boolean abstraction with 19 predicates. The construction takes 1 hour on a 200 MHz Pentium machine, and among the 24 possible control configurations, 15 are found not reachable. Our new invariant generation algorithm takes 15 seconds to generate strong invariants. By simple evaluation of these invariants for the 15 nonreachable control configurations, we were able to find that for 12 of them the collected invariant is equivalent to $false$. Thus, only 3 predicates among the initial 19 predicates were used to refine the abstraction and to prove that the 3 others are not reachable as well. The whole process takes about a minute on the same machine. Notice that previous attempts to mechanize the verification of the protocol using theorem provers [11] require 2 to 6 months. For instance, the proof in [11] requires 57 auxiliary invariants to be given by the user to prove that the protocol can be simulated by a more abstract one that behaves correctly. Our methodology allows us, in a matter of minutes or at most hours, to generate finite abstraction that can be used for model checking, for debugging, or simply as an invariant of the original system that can be exploited in several ways. In our case, the invariant defined by the ASG obtained for the protocol is stronger than the conjunction of the 57 auxiliary invariants derived by hand in [11].

## 7 Conclusion and future work

We presented an incremental methodology for the verification of safety properties of infinite state systems based on abstract interpretations. Our methodology can be viewed as a practical application of the theory of symbolic analysis of programs developed in [6]. Invariant generation and abstraction construction can be viewed as symbolic forward analysis. Our refinement algorithm can be viewed as a use of backward analysis to refine the results of the forward analysis. Our methodology can also be viewed as a practical application of the different abstraction preservation results [14, 7] based on preorder and equivalences of concrete and abstract models. For more efficiency, the computation of an ASG and its refinement can be done on the fly. A safety property can be checked for each state added to the ASG. When the property is violated, the construction is interrupted, and if nondeterminism was created, the refinement algorithm refines according to the transitions where nondeterminism was created and the construction is carried on. If nondeterminism was not created, then the abstract state violating the property is indeed the characterization of a set of concrete states violating the property, and any error trace leading to this abstract state from the initial abstract state is a concrete error trace. We have implemented this technique for invariance checking, where at each new abstract state, we check the invariance property. Our refinement algorithm is a dual to the deductive model checking (DMC) procedure of [19], where the product of the program and the negation of a temporal formula is represented as a finite graph that is refined until a counterexample is found or the property is proved. Neither technique is guarantee to terminate, and at any moment the graph computed in DMC represents an approximation of the states that violate the property. However, since our technique is not tied to a particular property, at any moment an abstract graph represents an overapproximation of the set of reachable states. That is, an invariant that can be used, for instance, in a pure deductive proof. The scalability of such techniques to large programs written in real programming languages relies, for instance, more on the speed than the power of the decision procedures used for validity checking. The speeds obtained decision procedures implemented in the new PVS version are very promising. Hundreds of assertions can be discharged in a few minutes.

## 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 *Computer-Aided Verification, CAV '96*, LNCS 1102. Springer-Verlag, pages 323–335, New Brunswick, NJ, July/August 1996.

[3] Nikolaj Bjorner, Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 1997.

[4] 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.

[5] 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.

[6] Patrick Cousot. Semantics Foundation of Program Analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, 1981.

[7] 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*, 1994.

[8] Steven M. German and Ben Wegbreit. A synthesizer of inductive assertions. *IEEE Transactions on Software Engineering*, 1(1):68–75, March 1975.

[9] S. Graf and H. Saïdi. Verifying invariants using theorem proving. In *Conference on Computer Aided Verification CAV'96*, LNCS 1102. Springer Verlag, 1996.

[10] 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.

[11] Klaus Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe FME '96*, LNCS 1051. Springer Verlag, Oxford, UK, March 1996.

[12] Shmuel M. Katz and Zohar Manna. A heuristic approach to program verification. In Nils J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, Stanford, CA, August 1973. William Kaufmann.

[13] L. Lamport. The 'Hoare logic' of concurrent programs. In Springer-Verlag, editor, *Acta Informatica 14*, pages 21–37, 1980.

[14] 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.

[15] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurent Systems: Specification*. Springer-Verlag, 1993.

[16] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[17] 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.

[18] Hassen Saïdi. The Invariant-Checker: Automated deductive verification of reactive systems. In *Proceedings of the 9th Conference on Computer-Aided Verification, CAV'97*. Springer Verlag, 1997. `www.csl.sri.com/~saidi`.

[19] Henny B. Sipma, Tomás E. Uribe, and Zohar Manna. Deductive model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *LNCS*. Springer Verlag, August 1996.