Powerful Techniques for the Automatic Generation of Invariants

Saddek Bensalem[1*], Yassine Lakhnech[2 **], and Hassen Saidi [1***]

[1] VERIMAG, Miniparc-Zirst, Rue Lavoisier 38330 Montbonnot St-Martin, France.
[2] Institut für Informatik und Praktische Mathematik Christian-Albrechts-Universität zu Kiel, Preußerstr. 1-9, D-24105 Kiel, Germany.

Abstract. When proving invariance properties of programs one is faced with two problems. The first problem is related to the necessity of proving tautologies of the considered assertion language, whereas the second manifests in the need of finding sufficiently strong invariants. This paper focuses on the second problem and describes techniques for the automatic generation of invariants. The first set of these techniques is applicable on sequential transition systems and allows to derive so-called local invariants, i.e. predicates which are invariant at some control location. The second is applicable on networks of transition systems and allows to combine local invariants of the sequential components to obtain local invariants of the global systems. Furthermore, a refined strengthening technique is presented that allows to avoid the problem of size-increase of the considered predicates which is the main drawback of the usual strengthening technique. The proposed techniques are illustrated by examples.

1 Introduction

Model checking [17, 4, 13, 20] is by now a well-known method for proving properties of reactive programs. The main reason for its success is that it works fully automatically, i.e. without any intervention of the user. The price to pay for this feature is that it can only be applied on finite-state, or restricted classes of infinite state, programs.

On the other hand, there exist deductive methods to prove safety properties of reactive programs. These methods are based on a proof rule which can be formulated as follows. To prove that some given predicate P is an *invariant* of a given program S, i.e. that every reachable state of S satisfies P, it is necessary and sufficient to find a predicate Q with the following properties: 1.) Q is stronger than P, 2.) Q is preserved by every transition of S, i.e. for every states s and s', if s satisfies Q and s' is reachable from s by a transition, then also s' satisfies Q, and 3.) Q is satisfied by every initial state of S. The predicate Q is called *auxiliary* predicate.

---

[*] Bensalem@imag.fr, Currently visiting the Computer Science Laboratory, SRI International

[**] yl@informatik.uni-kiel.de

[***] Saidi@imag.fr

Although, this rule is sound and (relatively) complete, it provides only a partial answer to the verification problem of safety properties. For it leaves open (i) how to find the auxiliary predicate Q and (ii) how to prove that Q is preserved by every transition of S and satisfied by the initial states. Problem (ii) is related to the problem of proving tautologies of the underlying assertion language.

In this work, we describe techniques for automatically generating auxiliary predicates. We present the following strategies:

- *Generalized reaffirmed invariance:* This applies to transitions for which the value of the guard and of the expressions occurring on the right hand side of its assignment are not changed by the transition itself, i.e. they have the same value before and after the transition. This is more general than the one called reaffirmed invariants in [15, 14].
- *Propagation of invariants:* This technique allows to propagate an assertion that holds whenever control is at some fixed control location to other control locations. We consider two instances of this technique. The most general one allows to propagate even in the presence of loops. Again our technique is applicable in cases not covered by the propagation techniques presented in e.g. [15, 14].
- *Refined strengthening:* One of the most used techniques for strengthening invariants is by calculating the weakest (liberal) precondition [6] w.r.t. the considered invariant and taking it as a conjunct. A drawback of this method is that it increases the complexity of the considered predicate, and hence, after few steps its application leads in many cases to unmanageable predicates. We present a refined version of this method that allows to attenuate the blow up caused by applying this useful strengthening method.
- *Combining Invariants:* This method allows to combine invariants developed separately for the components of a given network $S_1 \parallel \cdots \parallel S_n$ of transition systems to an invariant of the global system.

All predicates that can be generated by these strategies are proved to be invariant by construction. The use of these techniques for various mutual exclusion algorithms shows that they are promising. For instance, in case of the Bakery algorithm [12, 15], which is an infinite-state program, we generate an invariant that is sufficiently strong to prove the required property.

It is also important to note that these techniques are local in the sense that, in order to apply them, they do not require the full transition system to satisfy some restrictions, but rather subsets of control locations and variables are required to satisfy some condition.

The problem of automatically constructing invariants from program description has been intensively investigated in the seventieth leading to results reported in e.g. [11, 9, 3, 7][3]. Here, we present results which are to our knowledge new or extensions of existing ones. Other interesting recent results are reported in [2].

These techniques represent an important component of a tool which is being developed to support the computer-aided verification of safety properties of

---

[3] This list of references is far from being exhaustive. See [15] for other references.

reactive programs. Here, we give a brief description of this tool (See [10] for a detailed discussion). It consists of the following components:

- **Front-end:** The front-end takes as input a description of a transition system written as a program in a simple programming language and a predicate to be proved as invariant of the described transition system. Then, it produces a PVS-theory [16] that mainly contains the verification conditions to be proved. The front-end analyses also the program and generates a file containing information needed to decide, for each control location, whether some invariant generation procedure can be applied.
- **Automatic Invariant Generation:** This is a module that contains procedures implementing several invariant generation techniques. In this paper, we present some of these techniques.
- **Proof Manager:** The user can try to prove that P is an invariant fully automatically. In this case, the system tries to prove that P is inductive, that is, P is preserved by each transition of the program. In case of success, this is reported to the user. Otherwise, the system tries to prove the invariance of P using predicates which are obtained by calling some invariant generating procedures. These predicates are guaranteed to be invariant by construction. In case the system is unable to prove the invariance of P, it may either do some strengthening or enter the interactive modus and requires the user's guidance. This choice is made by the user.
- **PVS** is the theorem prover developed at SRI [16]. It is used during the automatic- as well as interactive proof procedure to discharge the verification conditions.

2 Transition Systems and Invariance Properties

We assume an underlying assertion language $\mathcal{A}$ that includes first-order predicate logic and interpreted symbols for expressing the standard operations and relations over some concrete domains. We assume to have the set of integers among these domains. Assertions (we also say predicates) in $\mathcal{A}$ are interpreted in states that assign values to the variables of $\mathcal{A}$. Let Σ denote the set of states. Given a state s and a predicate P, we use the notation $s \models P$ to denote that s satisfies P, and use $[\![P]\!]$ to denote the set of states that satisfy P. Henceforth, we identify P and its characteristic set $[\![P]\!]$.

Definition 1. A transition system is a structure $S = \langle X, pc : DC, T, \text{Init}\rangle$, where

- X is a finite set $\{x_1 : D_1, \ldots, x_n : D_n\}$ of typed data variables. Each variable x_i ranges over data domain D_i. We assume that the variables in X form a subset of those in $\mathcal{A}$.
- pc is a control variable (or program instruction counter). It ranges over the finite domain DC. We assume that $pc \notin X$.
- T is a finite set of transitions. A transition t is characterized by a quadruple $(pc = d, g(\mathbf{Y}), \mathbf{Z}' = \mathrm{e}(\mathbf{U}), pc' = d')^4$, where $\mathbf{Y}, \mathbf{Z}, \mathbf{U} \subseteq X$. The variables in

---

[4] $\mathbf{Z}'$ can be empty; this is the case when no variable is affected

$\mathbf{Z}$ are called the *variables affected* by transition t, and we denote by $sour(t)$ (resp. $tar(t)$) the value d (resp. d'). These definitions are easily generalized to sets of transitions. Given a transition $t = (pc = d, g(\mathbf{Y}), \mathbf{Z}' = \mathrm{e}(\mathbf{U}), pc' = d')$, and states s and s', s' is called *t-successor of* s, denoted by $s \rightarrow_t s'$, if the following conditions are satisfied: *1.) s satisfies the* enabledness condition $pc = d \wedge g(\mathbf{Y})$ *of transition t and 2.)* s' *satisfies* $s'(z_i) = e_i(s(\mathbf{U}))$, *for each* $z_i \in \mathbf{Z}$, $s'(x) = s(x)$, *for each* x *with* $x \notin \mathbf{Z}$, *and* $s'(pc) = d'$.
- Init is of the form $I(X) \wedge pc = d_0$. The conjunct $I(X)$ specifies the initial condition on data variables, whereas $pc = d_0$ specifies the initial value of the control variable. We call I *the initial predicate of* S and d_0 its *initial control location.*

A transition system generates a set of sequences of states. Since we are only interested in invariance properties, we only consider finite sequences. A finite sequence $\sigma = s_0, \cdots, s_n$ of states is called *computation of* S, if s_0 satisfies Init and, for every $i \in \{0, \ldots, n-1\}$, there exists a transition t in T with $s_i \rightarrow_t s_{i+1}$.

To define the semantics of the parallel construct, we define the product of two transition systems. Let $S_i = \langle X_i, pc_i : DC_i, T_i, \mathrm{Init}_i \rangle$, for $i = 1, 2$, be transition systems. The product of S_1 and S_2, denoted $S_1 \bigotimes S_2$, is a transition system $\langle X, pc : DC, T, \mathrm{Init} \rangle$, where

- $X = X_1 \cup X_2$ is the set of program variables.
- pc ranges over $DC = DC_1 \times DC_2$.
- A transition $(pc = (d_1, d_2), g(\mathbf{Y}), \mathbf{Z}' = \mathrm{e}(\mathbf{U}), pc = (d_1', d_2'))$ is in T iff either
  - $(pc_1 = d_1, g(\mathbf{Y}), \mathbf{Z}' = \mathrm{e}(\mathbf{U}), pc_1' = d_1') \in T_1$ and $d_2' = d_2$ or
  - $(pc_2 = d_2, g(\mathbf{Y}), \mathbf{Z}' = \mathrm{e}(\mathbf{U}), pc_2' = d_2') \in T_2$ and $d_1' = d_1$.
- $\mathrm{Init} = I_1 \wedge I_2 \wedge pc = (d_{1,0}, d_{2,0})$, where $\mathrm{Init}_i = I_i \wedge pc_i = d_{i,0}$, for $i = 1, 2$.

Then, the set of computations of $S_1 \parallel S_2$ is defined to be that of $S_1 \bigotimes S_2$.

Invariance Properties We consider a class of properties, named *invariance properties* (cf. [15]). Intuitively, a property P is an invariant of a transition system S, if in each state of the system S this property holds. In other words, each state that is reached during a computation of S satisfies P.

Definition 2. A state s is called *reachable* (accessible) in the transition system S, if there exists a computation $s_0, \cdots, s_n$ of S such that $s_n = s$. We denote the set of reachable states by *Reach*(S). A predicate P is called *invariance property of* S (or *invariant* of S) iff *Reach*$(S) \subseteq [\![P]\!]$. For $d \in DC$, we say that P is *an invariant of* S *at* d, if $P \vee \neg(pc = d)$ is an invariant of S.

Next, we briefly recall the basic idea for proving invariance properties of programs. This idea underlies many proof rules formulated in different settings (e.g. [8, 1, 15]). To do so, we recall the definition of some predicate transformers.

Definition 3. Given $\rho \subseteq \Sigma \times \Sigma$, the predicate transformers $pre[\rho]$, $\widetilde{pre}[\rho]$, and $post[\rho]$ are defined by $pre[\rho](P) = \{s \in \Sigma \mid \exists s' \in P \cdot (s, s') \in \rho\}$, $\widetilde{pre}[\rho](P) = \neg pre[\rho](\neg P)$, and $post[\rho](P) = \{s' \in \Sigma \mid \exists s \in P \cdot (s, s') \in \rho\}$

Thus, $pre[\rho](P)$ is the set of predecessors of P by ρ, $post[\rho](P)$ is the set of successors of P, and $\widetilde{pre}[\rho](P)$ is the set of states which either do not have successors by ρ or all their successors are in P. Note that the $\widetilde{pre}[\rho]$ and $post[\rho]$ are the *weakest liberal precondition* and *strongest postcondition* predicate transformers [6].

The main principle used in the literature for proving that a predicate P is an invariant of a system S, consists on finding an *auxiliary* predicate Q such that *1.) Q is stronger than P, 2.) every initial state satisfies Q, and 3.) Q is* inductive, *i.e. for all transitions* $t \in T$, *we have* $[\![Q]\!] \subseteq \widetilde{pre}[\rightarrow_t](Q)$, *or equivalently,* $post[\rightarrow_t](Q) \subseteq [\![Q]\!]$.

This proof rule is unsatisfactory because it does not tell us how to find the auxiliary predicate Q. Finding Q is often the hard part in the proof of invariance properties.

In the next section, we present a set of techniques that, given a transition system S and a predicate P, automatically generate an auxiliary predicate that is by construction an invariant. In some cases, the generated predicate is strong enough to prove that P is an invariant.

3 Automatic Generation of Auxiliary Predicates

In this section we present some of the strategies for deriving auxiliary predicates we implemented in our tool. We concentrate on strategies which are to our knowledge new or extensions of strategies presented in other works (e.g. [9, 11, 15, 14, 2]). The auxiliary predicates derived using our strategies are proved to be invariant by construction.

Generalized Reaffirmed Invariance without Cycles We begin with a strategy that can be applied to a control location d to derive an invariant under the assumption that all transitions that lead to d satisfy some restrictions we define below. This is a generalization of the reaffirmed invariance strategy presented in [15, 14].

Let $S = \langle X, pc : DC, T, I \wedge pc = d_0 \rangle$ be given. For $\alpha \subseteq DC$, let $L(\alpha)$ denote the set of transitions t with $tar(t) \in \alpha$. Thus, $L(\alpha)$ is the set of transitions changing the value of the control variable to a value in α. We write $L(d)$ instead of $L(\{d\})$.

Consider a transition $t = (pc = d_1, g(\mathbf{Y}), \mathbf{Z}' = \mathrm{e}(\mathbf{U}), pc' = d)$, with $\mathbf{Z} \cap \mathbf{U} = \emptyset$. Then, for every states s and s', if $s \rightarrow_t s'$, then $s'(\mathbf{Z}) = \mathrm{e}(s'(\mathbf{U}))$ and $s'(\mathbf{U}) = s(\mathbf{U})$. This suggests to take the predicate $\mathbf{Z} = \mathrm{e}$ as invariant at d.

To formulate the general case, given a transition t as above, we denote by $aff(t)$ the predicate $\mathbf{Z} = \mathrm{e}(\mathbf{U})$ and by $gu(t)$, the guard $g(\mathbf{Y})$. Let, for $d \in DC$, $Ass_S(d) = \bigvee_{t \in L(d)} (gu(t) \wedge aff(t))$, if $d \neq d_0$; and $I \vee \bigvee_{t \in L(d)} (gu(t) \wedge aff(t))$, if $d = d_0$, where I is the initial predicate of S and d_0 its initial control location.

Lemma 4. *Let S be a given transition system with $Init = I \wedge pc = d_0$ and let $D \subseteq DC$ be such that for each $d \in D$ and transition $(pc = d_1, g(\mathbf{Y}), \mathbf{Z}' = \mathrm{e}(\mathbf{U}), pc' = d)$ in $L(d)$ we have $\mathbf{Z} \cap (\mathbf{Y} \cup \mathbf{U}) = \emptyset$. Then, for each $d \in D$, the predicate $Ass_S(d)$ is an invariant of S at d.*

We can actually formulate a strategy that generalizes the one above by relaxing the condition $\mathbf{Z} \cap (\mathbf{Y} \cup \mathbf{U}) = \emptyset$. Let $Ass'_S(d)$ be defined as in Figure 1. Then, for each $d \in DC$, $Ass'_S(d)$ is an invariant of S at d. Henceforth, let **aff-indep** denote

$$\mathcal{A}ss'_S(d) = \begin{cases} \bigvee\limits_{t\in L(d)} (gu(t) \wedge \mathit{aff}(t)) & ; \text{if } d \neq d_0 \text{ and } \mathbf{Z} \cap (\mathbf{Y} \cup \mathbf{U}) = \emptyset \\ I \vee \bigvee\limits_{t\in L(d)} (gu(t) \wedge \mathit{aff}(t)) & ; \text{if } d = d_0 \text{ and } \mathbf{Z} \cap (\mathbf{Y} \cup \mathbf{U}) = \emptyset \\ \bigvee\limits_{t\in L(d)} \mathit{aff}(t) & ; \text{if } d \neq d_0,\ \mathbf{Z} \cap \mathbf{U} = \emptyset \text{ and } \mathbf{Z} \cap \mathbf{Y} \neq \emptyset \\ I \vee \bigvee\limits_{t\in L(d)} \mathit{aff}(t) & ; \text{if } d = d_0,\ \mathbf{Z} \cap \mathbf{U} = \emptyset \text{ and } \mathbf{Z} \cap \mathbf{Y} \neq \emptyset \\ \bigvee\limits_{t\in L(d)} gu(t) & ; \text{if } d \neq d_0,\ \mathbf{Z} \cap \mathbf{Y} = \emptyset \text{ and } \mathbf{Z} \cap \mathbf{U} \neq \emptyset \\ I \vee \bigvee\limits_{t\in L(d)} gu(t) & ; \text{if } d = d_0,\ \mathbf{Z} \cap \mathbf{Y} = \emptyset \text{ and } \mathbf{Z} \cap \mathbf{U} \neq \emptyset \\ \mathit{true} & ; \text{otherwise} \end{cases}$$

Fig. 1. Definition of $\mathcal{A}ss'_S(d)$

the function that for a given transition system S returns as result the predicate $\bigwedge_{d\in D} pc = d \Rightarrow \mathcal{A}ss'_S(d)$.

Generalized Reaffirmed Invariance with Cycles Consider the situation described in Figure 2. Then, function **aff-indep** yields the predicate $x = 2 \vee y = 1$ as invariant at d. It is easy to see, however, that the stronger predicate $x = 2$ is also invariant at d. We develop a technique that extends the previous one and covers situations similar to that of Figure 2.

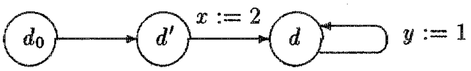

Fig. 2. Generalized Reaffirmed Invariance

A *path* from d to d' in S is a sequence $d_1, t_1, \cdots, t_{n-1}, d_n$ with $n \geq 2$, $d_1 = d$, and $d_n = d$. We say that a path $d_1, t_1, \cdots, t_{n-1}, d_n$ from d to d' *goes through* d'', if $d_i = d''$, for some $i \in \{1, \cdots, n\}$.

Definition 5. Given a transition system S, a control location d of S, and a set α of control locations of S with $d \in \alpha$. We say that α is *guarded by* d, if the following conditions are satisfied:

- The initial control location of S is not in α or it is d.
- For every transition $t \in L(\alpha) \setminus \{d\}$, $sour(t) \in \alpha$.
- Each path from d to $d' \in \alpha$ goes only through control locations in α.

Let $Tr(S, \alpha, d)$ denote the set $L(\alpha) \setminus \{t \mid t \in L(d), sour(t) \notin \alpha\}$.

Example 1. Consider the system S given in Figure 3, where d_0 is the initial control location. Then, $\alpha_1 = \{d_1, d_2, d_3, d_4, d_5\}$ and $\alpha_2 = \{d_1, d_4, d_5\}$ are guarded by d_1, while $\alpha_3 = \{d_1, d_2, d_4, d_5\}$ and $\alpha_4 = \{d_1, d_2, d_3\}$ are not because the second respectively third condition are violated. We have $Tr(S, \alpha_1, d_1) = \{t_1, t_2, t_3, t_4, t_5, t_6\}$ and $Tr(S, \alpha_2, d_1) = \{t_4, t_5, t_6\}$.

Definition 6. Given a transition system S and $d \in DC$. We say that d is *safe* with respect to a set V of variables and a set α of control locations, if α is guarded by d and for every $t \in Tr(S, \alpha, d)$, t does not affect any variable in V.

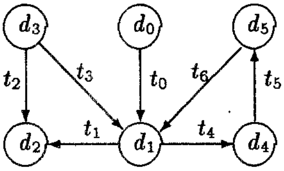

Fig. 3.

Then we have the following lemma.

Lemma 7. *Consider a transition system S, a control location d, a set α of control locations, and a set V of variables such that d is safe w.r.t. V and α. Let S' denote the transition system obtained from S by removing the transitions in $Tr(S, \alpha, d)$. For every predicate Q with free variables V, if Q is an invariant of S' at d, then Q is an invariant of S at every $d' \in \alpha$.*

The lemma above suggests a procedure to derive an invariant **aff-cyc**(S) from the description of the transition system S: For each $d \in DC$, determine a maximal set α of control locations for which d is safe with respect to the variables affected by transitions in $\{t \mid t \in L(d), sour(t) \notin \alpha\}$; in case d is the initial control location, we have to check also w.r.t. the free variables of I. If this is the case, record $\mathcal{A}ss'_{S'}(d)$, where S' is as above, as an invariant of S at d' for each $d' \in \alpha$, otherwise, record $\mathcal{A}ss'_S(d)$ as an invariant of S at d.

Remark. 1. A possible variant of the algorithm **aff-cyc** concerns the case where the initial control location is considered. Instead of requiring that d is safe w.r.t. the free variables of I, we hide those which could be affected by some transition in $Tr(S, \alpha, d)$ by existential quantification.

2. Clearly, determining the maximal set α which is guarded by d and then checking whether d is safe w.r.t. this set and the variables affected by transitions in $\{t \mid t \in L(d), sour(t) \notin \alpha\}$ does not always allow to derive the strongest possible predicate. One can, however, have a procedure which depends on some given set V of variables and which computes the maximal set α such that d is safe w.r.t. V and α.
3. Until now we considered a single transition system S and **aff-cyc** has been formulated for this case. When n transition systems $S_1 \| \cdots \| S_n$ in parallel are considered, we have to strengthen the notion of d being safe w.r.t. a set V of variables and a set α of control locations; and require that all variables in V are only written by the system S_i to which d belongs. Henceforth, whenever we refer to **aff-cyc** when a parallel program is considered, we mean the algorithm obtained by strengthening this notion and taking into account the variation suggested in 1.

Next, we present a technique that allows to propagate predicates that have been proved to be invariant at some control points of the system, i.e. for some value of *pc*. We first start with the basic idea.

Propagation without cycles Given a transition system S, a predicate Q with $\mathbf{V}$ as free variables and a transition t of S, we say that transition t *does not affect* Q, if $\mathbf{Z} \cap \mathbf{V} = \emptyset$, where $\mathbf{Z}$ are the variables affected by t.

Consider a transition system S and a control location $d \in DC$ which is not the initial one. Let $\{d_1, \cdots, d_n\} = sour(L(d))$ and assume that, for each $i \in \{1, \cdots, n\}$, $Q_i(\mathbf{V_i})$ is an invariant of S at d_i. If for each $t \in L(d)$ and $i \in \{1, \cdots, n\}$, with $sour(t) = d_i$, t does not affect $\mathbf{V_i}$, then $\bigvee_{i=1}^{n} Q(\mathbf{V_i})$ is an invariant at d. For the case where d is the initial control location, $\bigvee_{i=1}^{n} Q(\mathbf{V_i}) \vee I$, where I is the initial predicate, is an invariant at d. The correctness of this observation is guaranteed by the following lemma.

Lemma 8. *Consider a transition system S and a predicate P that is an invariant of S. Let $d \in DC$ be a control location of S with $L(d) = \{t_1, \cdots, t_m\}$ and $d_i = sour(t_i)$. Let also $Q_1, \cdots, Q_m$ be predicates such that $P \wedge pc = d_i$ implies Q_i, with $i = 1, \cdots, m$. If d is not the initial control location of S, then the predicate $P \wedge (l = d \Rightarrow \bigvee_{i=1}^{m} post[\rightarrow_{t_i}](Q_i))$ is an invariant of S, otherwise $P \wedge (l = d \Rightarrow (\bigvee_{i=1}^{m} post[\rightarrow_{t_i}](Q_i) \vee I))$ is an invariant of S.*

Note that in case that transition t does not affect Q, we have $post[\rightarrow_t](Q) \Rightarrow Q$, and therefore, the correctness of our technique is implied by the lemma above and the fact that if P' is an invariant of S and P' implies Q', then Q' is also an invariant of S.

The implementation of this technique is a function, denoted **propg**, that takes as input a transition system S and a predicate P of the form $\bigwedge_{d \in DC} pc = d \Rightarrow Q_d(\mathbf{V}'_d)$. Then, computes for each control location d, the set of variables affected by any transition in $L(d)$. Let $\mathbf{V}_d$ denote the intersection of this set with $\mathbf{V}'_d$. As result, this function yields, for each control location d, as a local invariant at d the predicate $Q_d(\mathbf{V}'_d) \wedge \exists \mathbf{V}_d \cdot \bigvee_{d' \in L(d)} Q_{d'}(\mathbf{V}'_{d'})$.

Propagation with cycles Consider now the situation described in Figure 4. An application of the simple propagation technique does not allow to strengthen the predicate $\bigwedge_{i=1}^{m} pc = d_i \Rightarrow x = i$. For, we would add as a conjunct the predicate $pc = d \Rightarrow true \vee \bigvee_{i=1}^{m} x = i$, which is equivalent to true. Yet, it is clear that $\bigvee_{i=1}^{m} x = i$ is an invariant at d. We develop the next technique which captures similar situations.

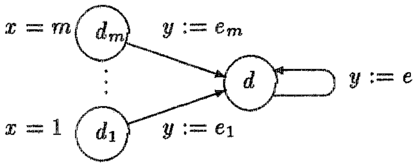

Fig. 4. Propagation with cycles

Consider a control location d and a set α of control locations which is guarded by d. Let $\{d_1, \cdots, d_m\} = sour(L(d)) \setminus \alpha$. Then, if for each $i = 1, \cdots, m$, $Q_i(\mathbf{V_i})$ is an invariant of S at d_i and if d is safe w.r.t. $\bigcup_{i=1}^{m} \mathbf{V_i}$ and α, we can conclude by Lemma 7 and Lemma 8 that $\bigvee_{i=1}^{m} Q_i(\mathbf{V_i})$ is an invariant at each $d' \in \alpha$.

Mixing generalized reaffirmed invariance and propagation Until now we considered propagation and reaffirmed invariance separately. Whereas propagation assumes a given invariant P and propagates local invariants from control locations to others, reaffirmed invariance does not assume such a predicate. We now present a technique that combines propagation and reaffirmed invariance.

Consider a transition system S and an invariant P of S. Let d be a control location of S such that $\{t_1, \cdots t_m\} = L(d)$ and $d_i = \mathit{sour}(t_i)$, for $i = 1, \cdots, m$. Suppose that for each $i = 1, \cdots, m$, $P \wedge pc = d_i$ implies $Q_i(\mathbf{V_i})$. If, for each transition t_i and each j with $d_j = \mathit{sour}(t_i)$, $Q_i(\mathbf{V_i})$ implies $\mathrm{e}(\mathbf{U_i}) = \mathbf{C_i}$ and $\mathbf{Z_i} \cap \mathbf{V_j} = \emptyset$, where $\mathbf{Z_i}' = \mathrm{e}(\mathbf{U_i})$ is $\mathit{aff}(t_i)$ and $\mathbf{C}$ is a list of constants, then we can conclude that $\bigvee_{i=1}^{m}(Q_i(\mathbf{V_i}) \wedge \mathbf{Z_i} = \mathbf{C_i})$ is invariant at d. Correctness of this observation is again a consequence of Lemma 8 and Lemma 8.

Refined Strengthening Suppose we are given a proposed invariant P for transition system S with transitions T. Suppose also that the proof of $P \Rightarrow \widetilde{pre}[\rightarrow_t](P)$ fails for $t_1, \cdots, t_m$. The method of strengthening invariants (e.g. [15]) proposes to try as next invariant $P_1 = P \wedge \bigwedge_{i=1}^{m} \widetilde{pre}[\rightarrow_{t_i}](P)$. Thus, one has to try to prove for each transition t the implication $P \wedge Q \Rightarrow \widetilde{pre}[\rightarrow_t](P \wedge Q)$, where $Q = \bigwedge_{i=1}^{m} \widetilde{pre}[\rightarrow_{t_i}](P)$. The main drawback of this method is that, in general, each strengthening step increases the size of the considered invariant which in some cases leads to unreadable predicates.

We propose a variant of this method that is theoretically equivalent, i.e. it leads to logically equivalent verification conditions, but which allows to reduce the number of applications of $\widetilde{pre}$ and to save redoing proofs.

Suppose that the attempt of proving $\forall t \in T \cdot (P \Rightarrow \widetilde{pre}[\rightarrow_t](P))$ fails for the transitions $t_1, \cdots, t_m$, and that one gets subgoals $Q_1, \cdots, Q_m$, which are logically equivalent to $P \Rightarrow \widetilde{pre}[\rightarrow_{t_i}](P)$, $i = 1, \cdots, m$. We propose to take in the next step the predicate $P_1' = P \wedge \bigwedge_{i=1}^{m} Q_i$ instead of P_1. The next lemma implies soundness of our method but also proves that if P_1 is inductive, then also P_1'.

Lemma 9. *Let $P_1 = P \wedge \bigwedge_{i=1}^{m} \widetilde{pre}[\rightarrow_{t_i}](P)$, Q_i be equivalent to $P \Rightarrow \widetilde{pre}[\rightarrow_{t_i}](P)$, and let $P_1' = P \wedge \bigwedge_{i=1}^{m} Q_i$. Then, P_1 and P_1' are equivalent.*

It is worth to note that soundness of our method does not depend on the fact that Q_i is equivalent to $P \Rightarrow \widetilde{pre}[\rightarrow_{t_i}](P)$ but it suffices, if it is stronger.

To see that our method indeed avoids the blow-up of the considered predicates which is due to the repeated application of the predicate transformer $\widetilde{pre}$, let us look at the predicates to be considered at step i when each of the strengthening and refined strengthening methods are applied in turn. In case of the strengthening method, one has to consider at step i the predicate $P_i = P_0 \wedge \widetilde{pre}(P_0) \wedge \cdots \widetilde{pre}^{i}(P_{i-1})$ and to prove $P_i \Rightarrow \widetilde{pre}(P_i)$. In case of the refined strengthening method, however, one has to consider the predicate Q_i which is obtained as a subgoal in step i, and then, to prove $Q_0 \wedge \cdots \wedge Q_i \Rightarrow \widetilde{pre}(Q_i)$. Thus, in the refined strengthening method, at each step $\widetilde{pre}$ has to be applied only once. Another advantage of this method is that Q_i is usually of the form $pc = d \Rightarrow Q$ which can be explained by the fact that Q_i is the predicate that is obtained when the proof of $Q_0 \wedge \cdots \wedge Q_{i-1} \Rightarrow \widetilde{pre}(Q_{i-1})$ for some fixed transition with $pc = d$ as part of the enabling condition has been attempted. Now, when a predicate Q of the form $pc = d \Rightarrow Q'$ is considered in order to prove that Q is preserved by all transitions, it suffices to consider only those in $L(d)$.

Combining Invariants Consider a network $S = S_1 \parallel \cdots \parallel S_n$ of transition systems. Given a predicate P, in order to prove that P is an invariant of S, one can calculate the product $S_1 \bigotimes \cdots \bigotimes S_n$ and then prove that P is an invariant of the resulting sequential transition system. This method is, however, not applicable for large transition systems because of the big size of the obtained system. Indeed, the resulting transition system mainly codes all possible interleaving of the transition steps in the network S. In this section, we present techniques we use to prove invariance properties of networks without calculating the product. These techniques have been successfully applied to many mutual exclusion algorithms, e.g. the Bakery mutual exclusion algorithm [12, 15] in three different versions and Szymanski's mutual exclusion algorithm [18, 19] both parameterized and for two processes.

Definition 10. Given a transition system S, a predicate P is called *history-independent assertion at* $d \in DC$, if $post[t](true) \subseteq [\![P]\!]$ holds for each $t \in L(d)$, and moreover, if d is the initial control location of S, then $Init$ implies P.

An history-independent assertion at d is true whenever computation reaches d independently on how this happens, in particular it does not dependent on the state in which the transition is taken.

Consider transition systems S_1 and S_2 with $S_i = (X_i, pc_i : DC_i, T_i, I_i \wedge pc_i = d_{i,0})$, for $i = 1, 2$. Moreover, consider predicates Q_i, for $i = 1, 2$, and $(d_1, d_2) \in DC_1 \times DC_2$. Assume we know that Q_i is an history-independent assertion at d_i. Then, we can conclude that $Q_1 \vee Q_2$ is an invariant of $S_1 \parallel S_2$ at (d_1, d_2). This leads to the following heuristic formulated in the next lemma.

Lemma 11. *Let* $S_i = (X_i, pc_i : DC_i, T_i, I_i \wedge pc_i = d_{i,0})$, *for* $i = 1, 2$, *be transition systems and let* Q_i *be predicates. Then, for each* $(d_1, d_2) \in DC_1 \times DC_2$ *such that* Q_i *is an history-independent assertion of* S_i *at* d_i, *for* $i = 1, 2$, *the predicate* $Q_1 \vee Q_2$ *is an invariant of* $S_1 \parallel S_2$ *at* (d_1, d_2).

If the predicates Q_1 and Q_2 constraint only variables which are affected only in S_1, respectively, S_2, then we can even conclude that the stronger predicate $Q_1 \wedge Q_2$ is an invariant at (d_1, d_2).

The implementation of both observations above is realized by a single function **comp** which takes as arguments the transition systems S_1 and S_2 as well as two predicates P_1 and P_2 for S_1 and S_2, respectively, which are of the form $\bigwedge_{d_j \in DC_i} pc = d_j \Rightarrow P_i(d_j)$, $i = 1, 2$. The result of the application of this function is a predicate of the form $\bigwedge_{d \in DC} pc = d \Rightarrow Q(d)$, where $DC = DC_1 \times DC_2$ and for $d = (d_1, d_2)$, $Q(d)$ is defined in Figure 3.

Remark. It is worth to note that each invariant Q obtained by applying the function **aff-indep** is history-independent.

In a concrete implementation, the predicate obtained by an application of the function **comp**, can be encoded by adding to each local invariant $P_i(d_i)$ at d_i two bits. The first one encodes whether $P_i(d_i)$ is history-independent and the second whether it refers to a variable affected in S_j with $j \neq i$.

$$Q(d) = \begin{cases} P_1(d_1) \vee P_2(d_2) & ; \text{if for } i = 1,2,\ P_i \text{ is an history-independent assertion at} d_i \\ & \quad \text{and one of the predicates } P_1 \text{ or } P_2 \text{ refers to a variable} \\ & \quad \text{affected in } S_2 \text{ respectively } S_1 \\ P_1(d_1) \wedge P_2(d_2) & ; \text{if for } i = 1,2,\ P_i \text{ is an history-independent assertion at } d_i \\ & \quad \text{and predicate } P_1 \text{ respec. } P_2 \text{ does not refer to any variable} \\ & \quad \text{affected in } S_2 \text{ respec. } S_1 \\ \mathit{true} & ; \text{otherwise} \end{cases}$$

Fig. 5. Definition of **comp**

The next lemma shows how given $d_i' \in DC_i$ and a predicate Q that is history-independent at d_i', we can deduce a predicate Q' which is also history-independent at d_i' and which does not refer to variables affected in S_j with $j \neq i$.

Lemma 12. *Let S_1 and S_2 be transition systems and let $d_1 \in DC_1$ (resp. $d_2 \in DC_2$) be a control location of S_1 (resp. S_2). If Q is a history-independent assertion at d and $\mathbf{Y}$ are the variables occurring in Q which are affected in S_2 (resp, S_1), then $\exists \mathbf{Y} \cdot Q$ is a history-independent assertion at d.*

Clearly, the predicate $\exists \mathbf{Y} : \mathbf{D} \cdot Q$ does not refer to variables affected in S_j. Let **abst** be a function that takes as arguments two transition systems S_1 and S_2 and a predicate P for S_1, and returns a predicate Q for S_1 such that Q is obtained from P by applying the observation above.

Next we present the tactic we apply to synthesize an invariant from a given network $S_1 \parallel S_2$. This is presented by an algorithm written in pseudo-code and which uses the heuristics presented above.

```
Input: $S_1 \parallel S_2$
Output: An invariant
 1. $P_i := \mathbf{aff\text{-}indep}(S_i)$; for $i = 1,2$
 2. $P := \mathbf{comp}(S_1, S_2, P_1, P_2)$
 3. $Q_1 := \mathbf{abst}(S_1, S_2, P_1)$, $Q_2 := \mathbf{abst}(S_2, S_1, P_2)$
 4. $Q_i := Q_i \wedge \mathbf{propg}(S_i, Q_i)$, for $i = 1,2$
 5. return $P \wedge Q_1 \wedge Q_2$
```

4 Example

The example we consider is the Bakery mutual exclusion algorithm [12, 15]. Two processes are competing to enter their respective critical sections represented by location 4. Thus, the invariant we are going to prove is given by the predicate $INV = \neg(pc_1 = 4 \wedge pc_2 = 4)$.

It can easily be checked that this invariant is not inductive. Moreover, calculating the set of reachable states using the *post* operator does not terminate (no fix-point can be reached in a finite number of steps). Calculating the weakest invariance property that is contained in INV does terminate after 8 steps (cf. [14]). We can automatically generate by our techniques an invariant that is inductive and that allows to prove that INV is indeed an invariant.

Transition system S_1	Transition system S_2
$pc_1 = 1 \longrightarrow pc_1' = 2$	$pc_2 = 1 \longrightarrow pc_2' = 2$
$pc_1 = 2 \longrightarrow y_1' = y_2 + 1, pc_1' = 3$	$pc_2 = 2 \longrightarrow y_2' = y_1 + 1, pc_2' = 3$
$pc_1 = 3 \wedge (y_2 = 0 \vee y_1 \leq y_2) \rightarrow pc_1' = 4$	$pc_2 = 3 \wedge (y_1 = 0 \vee y_2 < y_1) \rightarrow pc_2' = 4$
$pc_1 = 4 \longrightarrow pc_1' = 5$	$pc_2 = 4 \longrightarrow pc_2' = 5$
$pc_1 = 5 \longrightarrow y_1' = 0, pc_1' = 1$	$pc_2 = 5 \longrightarrow y_2' = 0, pc_2' = 1$
$Init = (y_1 = y_2 = 0 \wedge pc_1 = pc_2 = 1)$	

Applying generalized reaffirmed invariance without cycles for S_1 (resp. S_2) yields the predicate P_1 (resp. P_2) with:

$P_1 = (pc_1 = 1 \Rightarrow y_1 = 0 \vee y_1 = 0 \wedge y_2 = 0) \;\wedge\; (pc_1 = 3 \Rightarrow y_1 = y_2 + 1) \wedge$
$\quad (pc_1 = 4 \Rightarrow y_2 = 0 \vee y_1 \leq y_2)$

$P_2 = (pc_2 = 1 \Rightarrow y_2 = 0 \vee y_1 = 0 \wedge y_2 = 0) \;\wedge\; (pc_2 = 3 \Rightarrow y_2 = y_1 + 1) \wedge$
$\quad (pc_2 = 4 \Rightarrow y_1 = 0 \vee y_2 < y_1)$

Combining the predicates P_1 and P_2 according to function **comp** results in a predicate equivalent to

$$\begin{aligned} P = {} & (pc = (1,1) \Rightarrow y_1 = 0 \vee y_2 = 0) \wedge (pc = (1,3) \Rightarrow y_1 = 0 \vee y_2 = y_1 + 1) \wedge \\ & (pc = (1,4) \Rightarrow y_1 = 0 \vee y_2 < y_1) \wedge (pc(3,1) \Rightarrow y_1 = y_2 + 1 \vee y_2 = 0) \wedge \\ & (pc = (3,3) \Rightarrow y_1 = y_2 + 1 \vee y_2 = y_1 + 1) \wedge (pc = (3,4) \Rightarrow y_1 = 0 \vee y_2 < y_1) \wedge \\ & (pc = (4,1) \Rightarrow y_2 = 0 \vee y_1 \leq y_2) \wedge (pc = (4,3) \Rightarrow y_2 = 0 \vee y_2 < y_1) \end{aligned}$$

In the sequel, we write $pc_1 = d_1 \wedge pc_2 = d_2$ for $pc = (d_1, d_2)$.

Next, we apply the abstraction function **abst** on P_1 and P_2 to obtain:

$Q_1 = (pc_1 = 1 \Rightarrow y_1 = 0) \;\wedge\; (pc_1 = 3 \Rightarrow y_1 \geq 1)$
$Q_2 = (pc_2 = 1 \Rightarrow y_2 = 0) \;\wedge\; (pc_2 = 3 \Rightarrow y_2 \geq 1)$

Then, we apply our propagation technique without cycles. It can easily be checked that we can propagate from control location 1 to 2, from 3 to 4, and from 4 to 5, which yields the following predicates:

$Q_1' = (pc_1 = 1 \vee pc_1 = 2 \Rightarrow y_1 = 0) \;\wedge\; (pc_1 = 3 \vee pc_1 = 4 \vee pc_1 = 5 \Rightarrow y_1 \geq 1)$
$Q_2' = (pc_2 = 1 \vee pc_2 = 1 \Rightarrow y_2 = 0) \;\wedge\; (pc_2 = 3 \vee pc_2 = 4 \vee pc_2 = 5 \Rightarrow y_2 \geq 1)$

Then, we can show $P \wedge Q_1' \wedge Q_2' \wedge INV \Rightarrow \widetilde{pre}[\rightarrow_t](INV)$, for each transition t of $S_1 \parallel S_2$.

5 Discussion and Future Work

This paper provides a set of techniques for the automatic generation of auxiliary predicates to prove invariants of programs. The use of these heuristics for the verification of various mutual exclusion algorithms shows that they are promising. They have been applied to different versions of the Bakery, Dekker, Peterson, and Szymanski algorithms (see [15] for a recent presentation of many of these algorithms and for references). Concerning Szymanski's mutual exclusion algorithm, we verified the parameterized as well as the unparameterized case. We intend to combine our techniques with others as abstract interpretation [5] to discover relationships between program variables that can be used to derive invariants and to investigate heuristics and strategies for the decomposition of large programs.

Acknowledgements We thank J. Sifakis who continuously encouraged and supported this work. Many interesting discussions with S. Graf and A. Pnueli helped clarifying and fixing our ideas. We also thank the anonymous referees for judecious comments.

References

1. K.R. Apt. Ten years of Hoare's logic : a survey, part I. *ACM Trans. on Prog. Lang. and Sys.*, 3(2):431–483, 1981.
2. N. Bjøner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. In U. Montanari, editor, *1st Int. Conf. on Principles and Practice of Constraint Programming*, 1995.
3. M. Caplain. Finding invariant assertions for proving programs. In *Proc. Int. Conf. on Reliable Software*, Los Angeles, CA, 1975.
4. E.M. Clarke, E.A. Emerson, and E. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *POPL'83*. ACM, 1983.
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 ACM symp. of Prog. Lang.*, pages 238–252. ACM Press, 1977.
6. E. W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation. *Comm. ACM*, 18(8):453–457, 1975.
7. B. Elspas. The semiautomatic generation of inductive assertions for proving program correctness. Research report, SRI, Menlo Park, CA, 1974.
8. R. W. Floyd. Assigning meanings to programs. In *In. Proc. Symp. on Appl. Math. 19*, pages 19–32. American Mathematical Society, 1967.
9. S. M. German and B. Wegbreit. A synthesizer of inductive assertions. *IEEE Trans. On Software Engineering*, 1:68–75, March 1975.
10. S. Graf and H. Saidi. Verifying invariants using theorem proving. In *In this volume*, 1996.
11. S. Katz and Z. Manna. A heuristic approach to program verification. In *Proc. 3rd Int. Joint Conf. on Artificial Intelligence*, Stanford,CA, 1976.
12. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Comm. ACM*, 17(8):453–455, 1974.
13. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *POPL*, pages 97–107, 1985.
14. Z. Manna, A. Anuchitanukul, N. Bjøner, A. Browne, E. Chang, M. Colon, L. De Alfaro, H. Devarajan, H. Sipma, and T. Uribe. STeP : The Stanford Temporal Prover. Technical report, Stanford Univ., Stanford, CA, 1995.
15. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, 1995.
16. S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 1995.
17. J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int. Sym. on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1982.
18. B. K. Szymanski. A simple solution to Lamport's concurrent programming problem verification. In *Proc. Intern. Conf. on Supercomputing Sys.*, pages 621–626, 1988.
19. B. K. Szymanski and J. M. Vidal. Automatic verfication of a class of symmetric parallel programs. In *Proc. 13th IFIP World Computer Congress*, 1994.
20. M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS'86*. IEEE, 1986.