Incremental Verification by Abstraction *

Y. Lakhnech[1 **],
S. Bensalem[1], S.Berezin[2], and
S. Owre[3]

[1] Verimag, Centre Equation 2 Av. de Vignate, 38610 Gières, France. {bensalem, lakhnech}@imag.fr
[2] Carnegie Mellon University, Department of Computer Science,5000 Forbes Ave. Pittsburgh, PA 15213. Email: berez+@cs.cmu.edu
[3] Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA. Email : owre@csl.sri.com

Abstract. We present a methodology for constructing abstractions and refining them by analyzing counter-examples. We also present a uniform verification method that combines abstraction, model-checking and deductive verification in a novel way. In particular, it allows and shows how to use the set of reachable states of the abstract system in a deductive proof even when the abstract model does not satisfy the specification and when it simulates the concrete system with respect to a weaker simulation notion than Milner's.

1 Introduction

Verification by abstraction (e.g. [14, 15, 12, 24]) is a major technique for verifying infinite-state and very large systems. This technique consists in finding an abstraction relation and an abstract system that simulates the concrete one and that is amenable to algorithmic verification. One then checks that the abstract system satisfies an abstract version of the property of interest. Well established preservation results allow us then to deduce for a large class of properties that the concrete system satisfies the concrete property, if the abstract system satisfies the abstract one.

In order for this technique to be used more widely, automatic techniques are needed for 1) finding an accurate abstraction relation and 2) automatically generating an abstract property and an abstract system that simulates the concrete one. Several papers have discussed the automatic construction of the abstract system, e.g. [16, 6, 13] for infinite-state. A less studied issue is that of finding/constructing the abstract domain and the abstraction relation. The

* This work was supported in part by the European Commission (FET project ADVANCE, contract No IST-1999-29082). This work has been partly performed while the first two authors were visiting the Computer Science Laboratory, SRI International. Their visits were funded by NSF Grants No. CCR-9712383 and CCR-9509931.

** Contact Author.

situation is somewhat different in the case of program analysis where one is interested in rather generic properties mainly concerning run-time errors. In this case, depending on the programming paradigm (imperative, functional, or logic) and depending on the properties to be checked, several adequate abstract domains together with abstraction functions have been designed and extensively studied [27]. In model-checking, however, as one is interested in verifying properties specific to a given system, one usually needs to generate for every system and property a new abstract domain and abstraction relation. Therefore, it is mandatory to have automatic techniques assisting the user in finding the abstraction.

In this paper, we describe an automatic abstraction technique for invariance properties which is based on the set of atomic formulas appearing in successive applications of the weakest (liberal) predicate transformer on the invariant to be proved. This technique allows us to derive an abstraction function which is then used to construct an abstract system and an abstract property. When the property is true in the abstract system, we can conclude that the concrete system satisfies the invariant. The question arises, however, how to proceed in case the property is not satisfied in the abstract system. There are three possible reasons why the abstract system may not satisfy the abstract property: 1) the abstraction function is not fine enough to prove the property, that is, it identifies concrete states that should be distinguished, 2) the abstract system contains superfluous transitions that can be safely removed, that is, without altering the fact that it is an upper approximation and 3) the concrete system does not satisfy the specification [1]. The main contributions of this paper are on one hand algorithms for analyzing counter-examples that allow either to construct concrete counter-examples when this is possible or to refine the abstraction function. On the other hand, we present a uniform verification method that combines abstraction, model-checking and deductive verification in a novel way. In particular, it allows and shows how to use the set of reachable states of the abstract system in a deductive proof even when it simulates the concrete system in a weaker sense than Milner's notion of simulation.

For analyzing counter-examples, we present an algorithm that allows in many cases to analyze an infinite number of counter-examples at once. That is, the algorithm can deal with counter-examples that contain unfoldings of loops and where each time we unfold the loop we obtain a new counter-example.

Using counter-examples to refine abstract systems has been investigated by a number of other researchers [22, 1, 11]. Closest to our work is Clarke et al's techniques [11]. The main differences are, however, that we focus on infinite-state systems and that our algorithms for analyzing counter-examples work backwards while their algorithms are forward. This difference can lead to completely dif-

[1] In the case of finite-state systems only reason 1) and 3) are relevant as a least non-deterministic abstract system exists and can always be computed, if we consider abstraction functions which is the case here. Computing this abstract system is, in general, not possible for infinite-state system as incomplete decision procedures have to be used for constructing the abstract system.

ferent abstractions. Moreover, our technique allows in many cases to do in one step a refinement that cannot be done in finitely many ones using their method. The key issue here is that our technique incorporates accelerating the analysis of counter-examples that involve the unfolding of loops. On the other hand, we do not consider liveness properties. Also close to our work is Namjoshi and Kurshan's work [28] on computing finite bisimulations/simulations of infinite-state systems. The main idea there is to start from a finite set of atomic formulae and to successively split the abstract state space induced by these formulae until stabilization. However, in contrast to [8, 23, 20], the splitting in [28] is done on atomic formulae instead of equivalence classes which correspond to boolean combinations of these. A similar idea is applied in [29].

2 Preliminaries

Given a set X of typed variables, a *state over* X is a type-consistent mapping that associates with each variable $x \in X$ a value.

Definition 1. *A* transition system *is given by a triple* (Σ, I, R)*, where* Σ *is a set of states over a set* X *of variables,* $I \subseteq \Sigma$ *is a set of initial states, and* $R \subseteq \Sigma^2$ *is the transition relation. A* syntactic *transition system is given by a triple* $(X, \theta(X), \rho(X, X'))$*, where* X *is a set of typed variables,* $\theta(X)$ *is a predicate describing the set of initial states and* $\rho(X, X')$ *is a predicate describing the transition relation.* □

We associate in the usual way a transition system with every syntactic transition system.

A *computation* of a transition system $S = (\Sigma, I, R)$ is a sequence $s_0, \cdots, s_n$ such that $s_0 \in I$ and $(s_i, s_{i+1}) \in R$, for $i \leq n-1$. A state $s \in \Sigma$ is called *reachable* in S, if there is a computation $s_0, \cdots, s_n$ of S with $s_n = s$. We denote by $\mathcal{R}(S)$ the set of states reachable in S.

Definition 2. *A set* $P \subseteq \Sigma$ *is called an* invariant *of* S*, denoted by* $S \models \Box P$*, if every state that is reachable in* S *is in* P*.* □

Given a set $P \subseteq \Sigma$ of states and a relation $R \subseteq \Sigma^2$ the *weakest liberal precondition* of R with respect to P, denoted by $wp(R, P)$ or $wp_R(P)$, is the set consisting of states s such that for every state s', if $(s, s') \in R$ then $s' \in P$. The *precondition* of R with respect to P, denoted by $pre_R(P)$, is the pre-image of P by R. We also sometimes write $pre(R)(P)$ instead of $pre_R(P)$.

All the semantic notions introduced so far have their syntactic counterparts which we assume as known. Moreover, we will tacitly interchange syntax and semantics, e.g. predicates and sets of states etc., unless there is a necessity to make a distinction.

3 A General Verification Rule

In this section, we present a general rule for verifying invariance properties which combines the two main approach to the verification of invariance properties of infinite-state systems:

1. the *deductive approach* which consists in applying a rule that allows to reduce the verification to proving a set of 1st-order formulas, and
2. the *verification by abstraction approach* which consists in abstracting the system in hand to a finite system which is then analyzed algorithmically using model-checking techniques.

To do so, we fix throughout this section a transition system $S = (\Sigma, I, R)$ and a set $P \subseteq \Sigma$ of states. We then consider the problem of showing that S satisfies the invariant P.

Deductive verification As shown, e.g. in [25], Rule (Inv), is a sound and complete for proving invariance properties of transitions systems. A predicate Q which

$$\begin{array}{l} \text{There exists } Q \subseteq \Sigma \\ Q \subseteq P \\ I \subseteq Q \\ Q \subseteq wp(R, Q) \\ \hline S \models \Box P \end{array}$$

Fig. 1. Proof rule (Inv).

satisfies the premises of the rule above is called *auxiliary invariant* for S and P. The (semantic) completeness proof of the rule states that for every transition system S and predicate P, if $S \models \Box P$ then there exists an auxiliary predicate Q for S and P. Indeed, it is not difficult to check that the set $\mathcal{R}(S)$ of reachable states of S is an auxiliary predicate for Q and S provided $S \models \Box P$. Alternatively, one could choose the set $\bigcap wp^i(R, P)$ as auxiliary invariant. In fact, this set corresponds to the weakest auxiliary invariant for S and P.

It is very important to understand that the completeness result/proof of Rule (Inv) does not give a clue of how to find an auxiliary invariant. Indeed, choosing $\mathcal{R}(S)$ as auxiliary invariant reduces the first premise to proving the original problem, namely $S \models \Box P$. Moreover, taking $\bigcap wp^i(R, P)$ as auxiliary predicate reduces the second premise to $S \models \Box P$. In case, S is a finite-state system each of these predicates can be expressed in propositional logic and checking the premises can be done algorithmically. However, in general, one needs an assertion language which is at least as expressive Presburger Arithmetic to express either of these predicates which makes checking the premises of the rule undecidable. Even worse, there are systems for which $\mathcal{R}(S)$ is expressible without using quantifiers over natural numbers yet computing such a representation cannot be done effectively.

Verification by abstraction Abstraction techniques [14, 10] can be used to compute an over-approximation of $\mathcal{R}(S)$. Basically, the idea consists in abstracting the considered system S to a finite system S^a such the concretization of $\mathcal{R}(S^a)$ is a super-set of $\mathcal{R}(S)$. The use of abstractions techniques in the context of model-checking is well-studied [12, 24]. The theory is based on the notion of simulation (also called L-simulation, forward-simulation,...) and on preservation results which tell us which properties that are satisfied by S^a are also satisfied by S.

Definition 3. *Let $S = (\Sigma, I, R)$ and $S^a = (\Sigma^a, I^a, R^a)$ be two transition systems. We say that S^a is an* abstraction *of S with respect to a relation $\alpha \subseteq \Sigma \times \Sigma^a$, denoted by $S \sqsubseteq_\alpha S^a$, if the following conditions are satisfied:*

1. *α is a total relation,*
2. *for all states $s_0, s_1 \in \Sigma$ and $s_0^a \in \Sigma^a$ with $(s_0, s_0^a) \in \alpha$, if $(s_0, s_1) \in R$ then there exists a state $s_1^a \in \Sigma^a$ such that $(s_0^a, s_1^a) \in R^a$ and $(s_1, s_1^a) \in \alpha$.*
3. *for every state s in I there exists a state s^a in I^a such that $(s, s^a) \in \alpha$.* □

When Σ^a is finite we call S^a a *finite* abstraction of S with respect to α.

It can be easily proved that if $S \sqsubseteq_\alpha S^a$ then for every computation $s_0, \cdots, s_n$ of S there exists a computation $s_0^a, \cdots, s_n^a$ of S^a such that $(s_i, s_i^a) \in \alpha$, for every $i \leq n$. This implies the following preservation result (cf. [12, 24]):

Theorem 1. *Let S and S^a be transition systems such that $S \sqsubseteq_\alpha S^a$. Let $P \subseteq \Sigma$ and $P^a \subseteq \Sigma^a$. If $\alpha^{-1}(P^a) \subseteq P$ and $S^a \models \Box P^a$ then $S \models \Box P$, where $\alpha^{-1}(P^a) = \{s \in \Sigma \mid \exists s^a \in P^a.(s, s^a) \in \alpha\}$.* □

Thus, to prove that a transition system S satisfies an invariance property P it suffices to find a *finite* abstraction S^a of S with respect to some relation α such that $S^a \models \Box P^a$ for some $P^a \subseteq \Sigma^a$ with $\alpha^{-1}(P^a) \subseteq P$.

This method is complete. Indeed, it suffices to take an abstract system with two states s_0^a and s_1^a and a relation α such that $(s, s_0^a) \in \alpha$ iff s is reachable in S; and $(s, s_1^a) \in \alpha$ iff s is not reachable in S. The abstract system S^a has s_0^a as unique initial state. Obviously, $S \sqsubseteq_\alpha S^a$ and $S^a \models \Box\{s_0^a\}$. Moreover, since $S \models \Box P$, we have $\alpha^{-1}(s_0^a) \subseteq P$.

On the other hand, if $S \models \Box P$ can be proved using an abstraction S^a of S with respect to α, then it can also be proved using the auxiliary invariant $\alpha^{-1}(\mathcal{R}(S^a))$, where $\mathcal{R}(S^a)$ is the set consisting of the reachable states of S^a.

Thus, from a theoretical point of view proving invariance properties using abstractions is as difficult as using auxiliary invariants. Still in practice it is often the case that the method based on abstractions is easier.

A drawback of Definition 3 is that the simulation notion used there does not take into account the invariance property we want to prove. To overcome this, we proposed in [7], the following invariant-dependent simulation notion.

Definition 4. *We say that S^a is an abstraction of S with respect to $\alpha \subseteq \Sigma \times \Sigma^a$ and $P \subseteq \Sigma$, denoted by $S \sqsubseteq_\alpha^P S^a$, if the following conditions are satisfied:*

1. *α is a total relation,*
2. *for every state $s_0, s_1 \in \Sigma$ and $s_0^a \in \Sigma^a$ with $s_0 \in P$ and $(s_0, s_0^a) \in \alpha$, if $(s_0, s_1) \in R$ then there exists a state $s_1^a \in \Sigma^a$ such that $(s_0^a, s_1^a) \in R^a$ and $(s_1, s_1^a) \in \alpha$,*
3. *$I \subseteq P$, and*
4. *for every state s in I there exists a state s^a in I^a such that $(s, s^a) \in \alpha$.* □

Figure 2 shows the difference between both simulation notions.

$$
\begin{tikzcd}
\forall s^a \arrow[r, "R^a"] & \exists s^a \\
\forall s \arrow[u, "\alpha"] \arrow[r, "R"'] & \forall s \arrow[u, "\alpha"']
\end{tikzcd}
\qquad
\begin{tikzcd}
\forall s^a \arrow[r, "R^a"] & \exists s^a \\
\forall s \in P \arrow[u, "\alpha"] \arrow[r, "R"'] & \forall s \arrow[u, "\alpha"']
\end{tikzcd}
$$

Fig. 2. Simulation notions.

Now, it can be proved by induction on n that for every computation $s_0, \cdots, s_n$ of S such that $s_i \in P$, for every $i = 0, \cdots, n-1$, there exists a computation $s_0^a, \cdots, s_n^a$ of S^a such that $(s_i, s_i^a) \in \alpha$, for every $i \leq n$. Therefore, we can state the following preservation result:

Theorem 2. *Let S and S^a be transition systems such that $S \sqsubseteq_\alpha^P S^a$. Let $P^a \subseteq \Sigma^a$ and $P' \subseteq \Sigma$. If $\alpha^{-1}(P^a) \subseteq P \cap P'$, and $S^a \models \Box P^a$, then $S \models \Box(P \cap P')$.* □

Example 1. To illustrate Definition 2, let us consider the Bakery mutual exclusion algorithm, which has an infinite state space.
Transition system S_1:

$$
\begin{array}{lll}
\tau_1 : pc_1 = l_{11} & \longrightarrow & y_1 := y_2 + 1, pc_1 := l_{12} \\
\tau_2 : pc_1 = l_{12} \wedge (y_2 = 0 \vee y_1 \leq y_2) & \longrightarrow & pc_1 := l_{13} \\
\tau_3 : pc_1 = l_{13} & \longrightarrow & y_1 := 0, pc_1 := l_{11}
\end{array}
$$

Transition system S_2:

$$
\begin{array}{lll}
\tau_4 : pc_2 = l_{21} & \longrightarrow & y_2 := y_1 + 1, pc_2 := l_{22} \\
\tau_5 : pc_2 = l_{22} \wedge (y_1 = 0 \vee y_2 < y_1) & \longrightarrow & pc_2 := l_{23} \\
\tau_6 : pc_2 = l_{23} & \longrightarrow & y_2 := 0, pc_2 := l_{21}
\end{array}
$$

Here pc_1 and pc_2 range over $\{l_{11}, l_{12}, l_{13}\}$ and $\{l_{21}, l_{22}, l_{23}\}$, respectively, and y_1, y_2 range over the set of natural numbers. As abstract variables we use the boolean variables a_1, a_2, a_3 and the variables pc_1^a and pc_2^a. The abstraction function α is given by the predicate $a_1 \equiv (y_1 = 0) \wedge a_2 \equiv (y_2 = 0) \wedge a_3 \equiv (y_1 \leq y_2) \wedge pc_1^a \equiv pc_1 \wedge pc_2^a \equiv pc_2$.

In [6] we presented a method that given a transition system S and a predicate P allows to automatically compute an abstract system S^a such that $S \sqsubseteq_\alpha^P S^a$. Applied to Bakery's algorithm and the abstraction relation α given above our tool InVeSt computes the following abstract system, which obviously satisfies the invariant $\neg(pc_1^a = l_{13} \wedge pc_2^a = l_{23})$.
Transition system S_1^a:

$$\begin{array}{lll}
\tau_1^a : pc_1^a = l_{11} & \longrightarrow & a_1 := \text{false}, pc_1^a := l_{12} \\
\tau_2 : pc_1^a = l_{12} \wedge (a_2 \vee a_3) & \longrightarrow & pc_1^a := l_{13} \\
\tau_3 : pc_1^a = l_{13} & \longrightarrow & a_1 := \text{true}, pc_1^a := l_{11}
\end{array}$$

Transition system S_2^a:

$$\begin{array}{lll}
\tau_4 : pc_2^a = l_{21} & \longrightarrow & a_2 := \text{false}, pc_2^a := l_{22} \\
\tau_5 : pc_2^a = l_{22} \wedge (a_1 \vee \neg a_3) & \longrightarrow & pc_2^a := l_{23} \\
\tau_6 : pc_2^a = l_{23} & \longrightarrow & a_2 := \text{true}, pc_2^a := l_{21}
\end{array}$$

$\square$

A uniform rule While Theorem 1 and Theorem 2 allow us to deduce $S \models \Box P$ in case $S^a \models \Box P^a$, they do not tell us whether it is possible to take advantage from S^a in case $S^a \not\models \Box P^a$. Rule (Inv-Uni) (see Fig. 3), which can be seen as a uniform presentation of the deductive and the verification by abstraction approaches, addresses this question. Indeed, the proof rule shows how concretizations of invariants of the abstract system can be used to prove that the predicate P is preserved by the transition relation of S.

$$\begin{array}{l}
\text{There exists } \alpha \subseteq \Sigma \times \Sigma^a \\
S \sqsubseteq_\alpha^P S^a \\
\alpha^{-1}(\mathcal{R}(S^a)) \subseteq Q \\
Q \cap P \cap P' \subseteq wp(R, P \cap P') \\
I \subseteq P' \\
\hline
S \models \Box(P \cap P')
\end{array}$$

Fig. 3. Proof rule (Inv-Uni).

Theorem 3. *The proof Rule (Inv-Uni) (see Figure 3) is sound and complete.*

Proof. Let us first show soundness. Let S and S^a be transition systems such that (P1) $S \sqsubseteq_\alpha^P S^a$, (P2): $\alpha^{-1}(\mathcal{R}(S^a)) \subseteq Q$, (P3): $Q \cap P \cap P' \subseteq wp(R, P \cap P')$, and (P4): $I \subseteq P'$.

Let $s_0, \cdots, s_n$ be a computation of S. We prove by induction on n that $s_n \in P \cap P'$. Now, since (P1) implies that all initial states of S satisfy P, and since $I \subseteq P'$, we have $s_0 \in P \cap P'$. Moreover, from (P1) and (P2), we have $s_{n-1} \in Q$, and hence by induction hypothesis, $s_{n-1} \in Q \cap P \cap P'$. Therefore, from (P3), $s_n \in P \cap P'$.

Completeness of Rule (Inv-Uni) is easily proved along the same lines as for the completeness of Rule (Inv). □

We now argue that Rule (Inv) can be easily derived from Rule (Inv-Uni), that is, Rule (Inv) is an instance of Rule (Inv-Uni). To do so, given a predicate Q, let S_Q denote the transition system that consists of two states, denoted by $[Q]$ and $[\neg Q]$, with $[Q]$ as initial state and whose transition relation is described in Figure 3. Then, Rule (Inv) is an instance of Rule (Inv-Uni) with $Q = P'$, $P = \Sigma$ and α the unique total relation defined by $(s, [Q]) \in \alpha$ iff $s \in Q$.

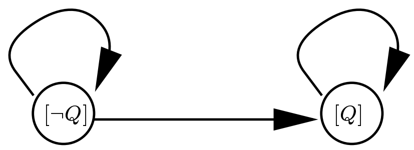

Fig. 4. Transition system S_Q

3.1 Concretizing OBDD's

To be able to apply Rule (Inv-Uni) we need:

1. a finite representation of the set R^a of abstract reachable states and
2. to be able to transform this representation into a finite representation of its concretisation, that is, $\alpha^{-1}(R^a)$.

Symbolic model checkers like SMV use ordered binary decision diagrams (OBDDs) to represent sets of states. To do so, all abstract variables are encoded by boolean variables. An OBDD is very easily transformed into a proposional formula in disjunctive normal form. Such representation can, however, be unnecessarily cumbersome.

In this section, we describe an algorithm for converting an OBDD into a propositional formula over the original state variables (not necessarily boolean), which is often almost as compact as the original OBDD.

Consider first a simple case when the top variable x of an OBDD b is boolean. Then, by the Shannon-Boole expansion law, $b = x \cdot b|_{x=\text{true}} + \bar{x} \cdot b|_{x=\text{false}}$. Equivalently, this can be written as a formula

$$(x = \text{false} \rightarrow \text{formula}(b|_{x=\text{false}})) \wedge (x = \text{true} \rightarrow \text{formula}(b|_{x=\text{true}})),$$

where formula(b) is a formula corresponding to the BDD b. Generalizing this to program variables with arbitrary number of possible values represented as a vector of boolean variables $x = (x_1, \ldots, x_n)$, and assuming that x_i's are the n top variables in b, we can recursively construct a formula

$$\bigwedge_{v \in \mathrm{type}(x)} (x = v \rightarrow \mathrm{formula}(b|_{x=\mathrm{v}})).$$

```
bdd2f(b: BDD, var_list: list of variables): formula =
if b ∈ H then return H(b);
if b = true_bdd then res := TRUE;
else if b = false_bdd then res := FALSE;
else
x := car(var_list);
res := TRUE;
for every v ∈ type(x) do
tmp := bdd2f(b|x=v, cdr(var_list));
res := res ∧ (x = v → tmp);
end;
H → (b, res);
end if
return res;
end bdd2f
```

Fig. 5. Basic algorithm converting BDD to a formula.

The basic algorithm is shown on Figure 5. It takes a BDD b and the list of program state variables (not necessarily boolean), and returns a formula equivalent to b. It uses a hash table H that hashes pairs of the form (b, f), where f is a formula previously constructed for a BDD b. At the very beginning the algorithm checks whether b is already in the table, and if it is, it simply returns the associated formula. If the formula has not been constructed yet, it checks for the trivial base cases (TRUE or FALSE). If we are not at the base case, then it constructs a formula recursively on the BDD structure. For every value in the type of the first variable[2] we restrict b to that value, remove the variable from the list, construct the formula recursively for that restricted BDD, and add the result into the final formula. Finally, the result is included into the hash table before it is returned.

If the internal representation of the formula being constructed is done using pointers, then multiple occurences of the same subformula in the final formula does not cause the formula to grow exponentially in the size of b. In fact, its size is only linear. However, the formula cannot be easily printed without losing this structure sharing. A simple solution to that would be to print the subformulas collected in the hash table with names assigned to them, and then print the final formula that has the names instead of these subformulas. However, the formula

[2] We assume that the types are always finite.

will be ugly and hardly manageable both for a human and for a mechanical tool reading it. We designed a set of simplifications that make the formula look a lot more understandable and even more compact. These transformations are applied for each program variable before the function returns from the recursive call.

Example 2. Let us consider the abstract system of the Bakery example 1. If we apply the basic algorithm (see Figure 5) to the obdd that characterizes the reachable states of this abstract system, we obtain the following formula:

$$
\begin{array}{l}
(a1 \;\Rightarrow (a2 \;\;\Rightarrow a3 \wedge pc1 = l11 \wedge pc2 = l21)\wedge \\
\qquad\quad (\neg a2 \Rightarrow a3 \wedge pc1 = l11 \wedge pc2 \neq l21))\wedge \\
(\neg a1 \Rightarrow (a2 \;\;\Rightarrow \neg a3 \wedge pc1 \neq l11 \wedge pc2 = l21)\wedge \\
\qquad\quad (\neg a2 \Rightarrow (a3 \Rightarrow pc1 \neq l11 \wedge pc2 = l22)\wedge \\
\qquad\qquad\quad (\neg a3 \Rightarrow pc1 = l12 \wedge pc2 \neq l21)))
\end{array}
$$

The concretization of the above formula yields the conjunction of following formulae:

$$(y1 = 0 \wedge y2 = 0) \Rightarrow pc1 = l11 \wedge pc1 = l21 \tag{1}$$
$$(y1 = 0 \wedge y2 > 0) \Rightarrow pc1 = l11 \wedge pc1 \neq l21 \tag{2}$$
$$(y1 \neq 0 \wedge y2 = 0) \Rightarrow pc1 \neq l11 \wedge pc1 = l21 \tag{3}$$
$$y1 \neq 0 \wedge y2 \neq 0 \wedge y1 \leq y2 \Rightarrow pc1 \neq l11 \wedge pc1 = l22 \tag{4}$$
$$y1 \neq 0 \wedge y2 \neq 0 \wedge y1 > y2 \Rightarrow pc1 = l12 \wedge pc1 \neq l21 \tag{5}$$

In this example, the concrete invariant obtained by this approach is stronger than the invariant generated by the method presented in [9, 5]. The invariants (4) and (5) cannot be immediately obtained by these methods. Indeed, these methods cannot easily generate invariants relating the variables of different processes.

4 Analyzing Counter-examples and Refining Abstraction Relations

A key issue in applying the verification method described by Theorem 2, respectively Rule (Inv-Uni), is finding a suitable abstraction relation α. In this section, we discuss a heuristic for finding an initial abstraction relation and present a method for refining it by analyzing abstract counter-examples, that is, counter-examples of the abstract system.

4.1 Initial abstraction relation

Assume that we are given a syntactic transition system $S = (X, \theta, \rho)$ and a quantifier-free formula P with free variables in X. Henceforth, we assume that ρ is given as a finite disjunction of transitions $\tau_1, \cdots, \tau_n$, where each τ_i is given by a guard g_i that is quantifier-free formula and a multiple-assignment $x_1, \cdots, x_n := e_1, \cdots, e_n$.

We want to prove that P is an invariant of S. To do so, we choose a constant $N \in \omega$ and compute $\bigwedge_{i \leq N} wp_\rho^i(P)$. Then, $\bigwedge_{i \leq N} wp_\rho^i(P)$ is also a quantifier-free formula. Let $F = \{f_1, \cdots, f_m\}$ be the set of atomic formulas that appear in $\bigwedge_{i \leq N} wp_\rho^i(P)$ in the predicate describing the initial states or in the property. (Notice that one can choose N sufficiently large to include the atomic formulae in the guards.) Then, we introduce for every formula f_i an abstract variable a_i and define the abstraction function α defined by $a_i \equiv f_i$. In [6], we show how given a transition system S, a predicate P and an abstraction function α, we compute a system S^a such that $S \sqsubseteq_\alpha^P S^a$ and a predicate P^a such that $\alpha^{-1}(P^a) \subseteq P$. Rule (Inv-Uni) addresses the question of how to benefit from computing the set of reachable states of S^a even when S^a does not satisfy $\Box P^a$. In this, section we address the following questions:

1. given a counter-example for $S^a \models \Box P^a$ does it correspond to some behavior in the concrete system and
2. in case the answer to the first question is no, how can we use the given counter-example to refine the abstraction function.

Identifying false negatives As in this paper, we focus on invariance properties, counter-examples are finite computations. Let $\sigma^a = s_0^a \tau_1^a s_1^a \cdots \tau_n^a s_n^a$ be a counter-example for $S^a \models \Box P^a$. The concretization $\alpha^{-1}(\sigma^a)$ of σ^a is the sequence $\alpha^{-1}(s_0^a)\tau_0\alpha^{-1}(s_1^a) \cdots \tau_{n-1}\alpha^{-1}(s_n^a)$. We call $\alpha^{-1}(\sigma^a)$ a symbolic computation of S, if there exists a computation $s_0\tau_1 s_1 \cdots \tau_n s_n$ of S such that $s_i \in \alpha^{-1}(s_i^a)$, for $i = 0, \cdots, n$. Clearly, this definition can be generalized to arbitrary sequences $Q_0\tau_1 Q_1 \cdots \tau_n Q_n$, with $Q_i \subseteq \Sigma$. Then, we have the following:

Lemma 1. *A sequence $Q_0\tau_1 Q_1 \cdots \tau_n Q_n$, with $Q_i \subseteq \Sigma$ is a symbolic computation iff $\theta \cap X_0 \neq \emptyset$, where $X_n = Q_n$ and $X_{n-i-1} = Q_{n-i-1} \cap \mathrm{pre}_{\tau_{n-i}}(X_{n-i})$.*
$\Box$

Lemma 1 suggests the procedure CouAnal given in Figure 6 for checking whether an abstract counter-example is a false negative or whether it corresponds to a behavior of the concrete system.

Refining the abstraction function First, we consider a simple refinement strategy of the abstraction function. Thus, let $\sigma^a = s_0^a \tau_1^a s_1^a \cdots \tau_n^a s_n^a$ be a counter-example for $S^a \models \Box P^a$ that is not a symbolic computation of S. By Lemma 1, procedure CouAnal returns some $i \leq n$ and a set $Y = X_i \subseteq \Sigma$ such that $X_{i-1} = \emptyset$[3]. Now, since $X_{i-1} = \emptyset$, $Q_{i-1} \subseteq wp_{\tau_i}(\neg X_i)$ and abstract transitions from abstractions of states in Q_{i-1} to abstractions of states in X_i are superfluous and should be omitted. To achieve this, we add for every atomic formula f in $\neg X_i$ which is not already in α, a corresponding new abstract variable a_f with $a_f \equiv f$. Let α_e denote the so-obtained new abstraction function. Moreover, let S_e^a be the abstract system with $(s_1^a, s_2^a) \in \rho_e$ iff there exist concrete states s_1, s_2 such that $(s_i, s_i^a) \in \alpha_e$, for $i = 1, 2$, and $(s_1, s_2) \in \rho$. Then, σ^a is not a computation of S_e^a.

[3] We assume that $i > 0$ as the case of $i = 0$ is easily handled

```
Input: An abstract counter-example σ^a
X := Q_n;
i := n;
while (X ≠ ∅ and i > 0) do
      Y := X;
      X := pre_{τ_i}(X) ∩ α^{-1}(Q_{i-1});
      i := i − 1
 od
if i = 0 and θ ∩ X ≠ ∅ then return "the following is a counter-example:"
      Take any s ∈ θ ∩ X ≠ ∅
      Let s_0 := s, s_1 := τ_1(s_0) ···, s_n := τ_n(s_{n-1})
      write s_0 ··· s_n
else return i, Y
fi
```

Fig. 6. Counter-example Analyzer: CouAnal

Speeding-up refinement of abstraction functions The simple illustrative example given in Figure 7 shows that in general applying finitely many times the procedure CouAnal is not sufficient. In this example, we want to show that location l_2 is not reachable and we initially take the abstraction function defined $a \equiv x = y$. After the n-th application of CouAnal we will have the abstraction function defined by $a_1 \equiv x = y, \cdots, a_i \equiv x+i = y, \cdots, a_n \equiv x+n = y$. However, the abstraction function we need is $a \equiv x = y, a_1 \equiv x > y$. The problem here is clearly that the abstract counter-examples contain abstract transitions that correspond to the unfolding of a loop in the concrete system. In the following, we generalize procedure CouAnal to cope with this situation. Let us first explain the main idea.

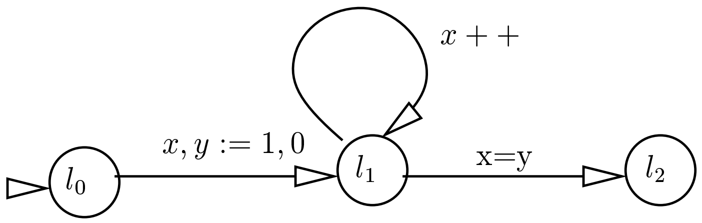

Fig. 7. Example for showing that speeding-up is needed

Henceforth, we assume that the description of the concrete system makes a clear distinction between control and data variables. That is, we assume that the concrete system is given by an extended transition system as in Figure 7, where l_0, l_1, l_2 are the control locations and x and y are the data variables. Let $\sigma^a = s_0^a \tau_1^a s_1^a \cdots \tau_n^a s_n^a$ be a counter-example for $S^a \models \Box P^a$. Assume that $\tau_{i_0}, \cdots, \tau_{i_1}$ is a loop in the control graph of the concrete system. In the procedure CouAnal we apply one time pre_{τ_i} on each X_i. However, since $\tau_{i_0}, \cdots, \tau_{i_1}$ is a loop, it is more interesting to apply an arbitrary number of times $pre(\tau_{i_0}, \cdots, \tau_{i_1})$ on X_{i_1}, that is, to consider $\bigvee_{i \in \omega} pre^i(\tau_{i_0}, \cdots, \tau_{i_1})$ on X_{i_1}.

For instance, in the example of Figure 7, applying $\bigvee_{i \in \omega} pre^i(x{+}{+})$ on $x = y$ gives after quantifier elimination the predicate $x \leq y$. Now, since $pre(x, y := 1, 0)(x \leq y)$ is empty our strategy consists in adding an abstract variable b such that b is true in the abstraction of a state s iff s satisfies $\neg(x \leq y)$ which is $x > y$; what we indeed expect.

This idea of speeding-up counter-example analyzes leads to the procedure AccCouAnal given in Figure 8. There are several remarks to say about pro-

Input: An abstract counter-example σ^a;
Let $L_1, \cdots, L_m$ be loops in the concrete system such that
$L_j = \tau_{i_j}, \cdots, \tau_{i_j + k_j}$ and
$\tau_1, \cdots, \tau_n = \tau_1, \cdots, \tau_{i_1 - 1} L_1, \tau_{i_1 + k_1 + 1}, \cdots, L_m, \tau_{i_m + k_m + 1}, \cdots, \tau_n$;
$X := Q_n$
$i := n$;
$k := m$;
while $(X \neq \emptyset$ and $i > 0)$ do
    $Y := X$;
    if $i = i_k$ then
        $X := \bigvee_{j \in \omega} pre^j_{L_k}(X) \cap \alpha^{-1}(Q_{i-1})$;
        $i := i -$ length(L_k)
    else $X := pre_{\tau_i}(X) \cap \alpha^{-1}(Q_{i-1})$
    fi
    $i := i - 1$
od
if $i = 0$ and $\theta \cap X \neq \emptyset$ then return "S does not satisfy the property"
else return i, Y
fi

Fig. 8. Accelerated Counter-example Analyzer: AccCouAnal

cedure AccCouAnal. The first one is that for a sequence $\tau_1, \cdots, \tau_n$ of transitions there are in general several but finitely many ways to partition it in $\tau_1, \cdots, \tau_{i_1 - 1} L_1, \tau_{i_1 + k_1 + 1}, \cdots, L_m$. The accuracy of the obtained abstraction function depends on this choice. In principle, one could, however, consider all possible choices and combine the obtained abstraction functions into a single one (take their conjunction). An other point is that in order to have reasonably simple abstraction functions one needs to simplify the predicates $\bigvee_{j \in \omega} pre^j_{L_k}(X) \cap$

$\alpha^{-1}(Q_{i-1})$, in particular, when possible, one should eliminate the existential quantification on i.

5 Example

To illustrate how we can use the procedure CouAnal, we consider the verification of the Bounded Retransmission protocol [26, 17, 19, 18], BRP for short.

The BRP accepts requirements from a producer to transmit a file of data to a consumer. The protocol consists of a sender at the producer side and a receiver at the consumer side (see Figure 9). Sender transmits data frames to the receiver

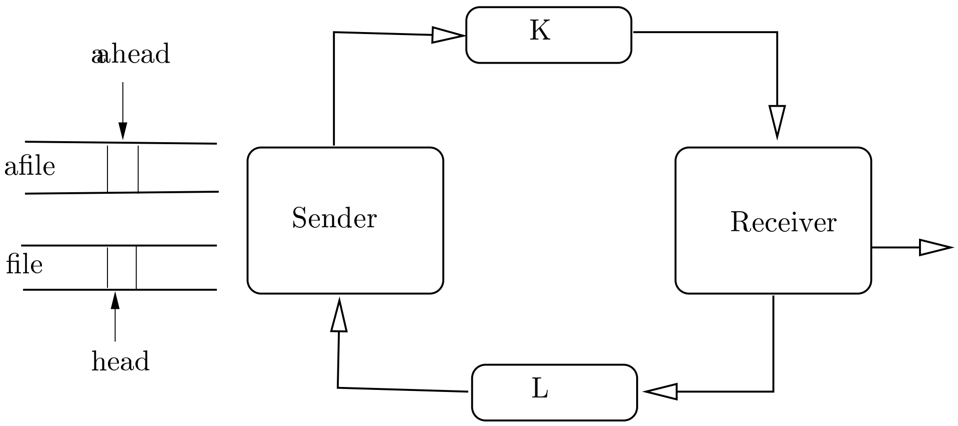

Fig. 9. The Bounded Retransmission Protocol.

via channel K and waits for acknowledgment via channel L. Since these channels may lose messages timeouts are used to identify a loss of messages. After sending a message, the sender waits for an acknowledgment. When the acknowledgment arrives, the sender either proceeds with the next message in the file, if there is one, or sends a confirmation message to the producer. However, if a timeout occurs before reception of an acknowledgment, the sender retransmits the same message. This procedure is repeated as often as specified by a parameter max. On its side, the receiver after acknowledging a message that is not the last one waits for further messages. If no new message arrives before a timeout, it concludes that there is a loss of contact to the sender and reports this to the consumer. Since the same message may be sent several times by the sender, a data frame includes a bit to indicate whether the same datum is resent or not. In fact, the BRP protocol can be seen as an extension of the alternating bit protocol [2]. The protocol is responsible for informing the producer whether the file has been transmitted correctly. On the consumer side, the protocol passes data frames indicating whether the datum is the first one in a file, the last one, or whether it is an intermediate one. Thus, a data frame contains also the information whether the data is the first, the last, or an intermediate. A third timeout is used in case a transmission has been interrupted to ensure that the sender waits enough to be sure that the receiver is prepared to receive new frames.

Correctness Criterion In the original formulation the requirements on the protocol are given by an abstract BRP-spec, and the task is to prove that BRP implements BRP-spec. To reduce the problem of proving that BRP implements BRP-spec to an invariance problem, we consider a superposition of BRP and BRP-spec and prove that the superposed protocol, BRP^+, satisfies the invariance property $\Box Safe$, where $Safe$ is a variable that is set to $false$ as soon as BRP makes a transition that is not allowed by BRP-spec. It should be realized that BRP^+ contains for many variables of the protocol two different copies corresponding to the variable in BRP and BRP-spec, respectively. So, for instance there are two variables *file* and *afile* which correspond to the file to be sent and two variables *head* and *ahead* which correspond to the position of the data being processed in *file* and *afile*, respectively. It is also worth mentioning that the variables *head* and *ahead* are never compared in BRP^+. Any relation between them these variables can not be deduced from the specification of BRP^+. The property that says the data transmitted is the same as the data received is important for the verification of this protocol.

5.1 Verification of the protocol

The BRP protocol represents a family of parameterized protocols. The parameters are the number of allowed retransmissions max, the length of a file *last*, and finally, the data type *Data*. Let us describe now the main steps we followed in the verification of BRP using InVeSt [7].

An initial abstraction function is generated automatically and used to compute an abstract system of the BRP^+. This initial abstraction function is obtained from the predicates describing the initial states, the specification and $\bigwedge_{i \leq N} \text{wp}^i_\rho(P)$ with $i = 1$. The abstraction is the identity on variables ranging over finite domains . The concrete variables that range over an infinite, resp. parameterized domain, and their abstract versions are (partially) given in the following table:

head<last ≡ file = MANY	(afile[ahead]=data(msg)) ≡ msgafile
(head=last) ≡ (file=ONE)	(afile[ahead]=data(k)) ≡ kafile
(head=last+1) ≡ (file=NONE)	rn=0 ≡ ¬Rn
⋮	⋮

It turns out the abstract system obtained by the initial abstraction does not satisfy the specification. The provided trace by the model-checker SMV has 6 states, each state contains 39 variables (not all of them booleans). This trace is concretized and checked using CouAnal. The result of this analysis is that this counter-example is spurious. Moreover, the result of this analyzes is that we have to add a boolean abstract variable $h_a h$ that is true iff $head = ahead$. Then, analyzing the abstract system obtained by this new abstraction shows a new counter-example. In this new counter-example, first *head* is incremented which on the abstract level assigns false to $h_a h$ as initially $head = ahead$. Then, after a few steps, *ahead* is incremented. Thus, though, at the concrete level $head = ahead$, this cannot be inferred at the abstract level. Indeed, applying CouAnal shows us that we have to add a new abstract variable corresponding to $head = ahead + 1$. Again, the abstract system obtained by this new abstraction shows a new counter-example. This time the incrementation of *ahead* precedes *head* and we have to add a new abstract variable corresponding to $head + 1 = ahead$. And now we are done. The new abstraction is fine enough for proving the property; the constructed finite abstract system satisfies the specification.

As a second experimentation we used the proof rule (Inv-Uni) to verify the BRP protocol. We started with the initial abstract system. But rather than going through the refinement process of the abstraction function, we concretized, using the procedure described in section 3, the OBDD that characterizes the abstract reachable states of the first abstract system. One iteration of strengthening was needed to prove the desired property of BRP protocol.

6 Conclusion and Future Work

We have presented a novel verification methodology that combines abstraction, model-checking and deductive methods. To support this methodology, and in particular, the verification by abstraction method we developed techniques for refining abstraction functions. These are based on the analysis of counter-examples and allow in many cases the simultaneous analysis of infinitely many examples by applying acceleration techniques. These techniques have been implemented in summer 1998[4] in the tool InVeSt, which is a tool for verifying invariance properties of infinite-state systems. InVeSt is based on PVS and connected to SMV. Since then we applied these techniques to several interesting examples and the results are promising.

In contrast to [11] we did not consider liveness properties. For infinite-state systems, the key issue is techniques for automatically generating ranking functions and fairness conditions (cf. [3, 4, 21]).

References

1. F. Balarin and A.L. Sangiovanni-Vincentelli. Am iterative approach to language containment. In Costas Courcoubetis, editor, *Computer Aided Verification: 5th International Conference*, volume 697 of *Lecture Notes in Computer Science*, pages 29–40. Springer-Verlag, 1993.

[4] The techniques have been presented in two seminars in summer 1998: one at SRI International and the other at the Dill-Manna joint seminar at Stanford University

2. K.A. Barlett, R.A. Scantlebury, and P.T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5), 1969.
3. K. Baukus, S. Bensalem, Y. Lakhnech, and K. Stahl. Abstracting WS1S Systems to Verify Parameterized Networks. In S. Graf and M. Schwartzbach, editors, *TACAS'00*, volume 1785 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
4. K. Baukus, Y. Lakhnech, and K. Stahl. Verifying universal properties of parameterized networks. In M. Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1926 of *Lecture Notes in Computer Science*, pages 291–304. Springer-Verlag, 2000.
5. S. Bensalem and Y. Lakhnech. Automatic generation of invariants. *Formal Methods in System Design*, 1999. To appear.
6. S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems automatically and compositionally. In Alan J. Hu and Moshe Y. vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 319–331. Springer-Verlag, 1998.
7. S. Bensalem, Y. Lakhnech, and S. Owre. Invest: A tool for the verification of invariants. In Alan J. Hu and Moshe Y. vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 505–510. Springer-Verlag, 1998.
8. A. Bouajjani, J. Cl. Fernandez, and N. Halbwachs. Minimal model generation. In *Workshop on Computer-aided Verification*. Rutgers – American Mathematical Society, Association for Computing Machinery, June 1990.
9. Ahmed Bouajjani, Yassine Lakhnech, and Sergio Yovine. Model checking for extended timed temporal logics. In B. Jonsson and J. Parrow, editors, *4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems FTRTFT'96*, volume 1135 of *Lecture Notes in Computer Science*, pages 306–326. Springer-Verlag, 1996.
10. Bettina Buth, Karl-Heinz Buth, Martin Fränzle, Burghard v. Karger, Yassine Lakhnech, Hans Langmaack, and Markus Müller-Olm. Provably correct compiler development and implementation. Compiler Construction, 1992.
11. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, Lecture Notes in Computer Science, pages 154–169. Springer-Verlag, 2000.
12. E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5), 1994.
13. M. A. Colon and T. E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. *Lecture Notes in Computer Science*, 1427:293–304, 1998.
14. 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.
15. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th ACM POPL*, pages 269–282, 1979.
16. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, 1997.
17. F.F. Groote and J.C. van de Pol. A bounded retransmission protocol for large packets. In *A case study in computer checked verification*, Logic Group Preprint Series 100. Utrecht University, 1993.

18. K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe, FME'96 Symposium*, volume 1051 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
19. L. Helmink, M.P.A. Sellink, and F.W. Vaandrager. Proof-checking a data link protocol. Technical Report CS-R9420, Centrum voor Wiskunde en Informatica (CWI), March 1994.
20. M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *36th Annual Symposium on Foundations of Computer Science (FOCS'95)*, pages 453–462, Los Alamitos, October 1995. IEEE Computer Society Press.
21. Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Information and Computation*, To appear, 2000.
22. R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes, the automata theoretic approach*. Princeton Series in Computer Science. Princeton University Press, 1994.
23. David Lee and Mihalis Yannakakis. Online minimization of transition systems (extended abstract). In *Proceedings of the Twenty-Fourth Annual ACM Symposium on the Theory of Computing*, pages 264–274, Victoria, British Columbia, Canada, 4–6 May 1992.
24. 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*, 6(1), 1995.
25. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.
26. S. Mauw and G.J. Veltink editors. *Algebraic Specification of Communication Protocols*. Number 36 in Cambridge Tracts in Theoretical Computer Science. 1993.
27. S.S. Muchnick and N. D. Jones, editors. *Program Flow Analysis*. Prentice-Hall, 1981.
28. K. S. Namjoshi and R .P. Kurshan. Syntactic program transformations for automatic abstraction. In *Computer Aided Verification*, Lecture Notes in Computer Science, pages 435–449. Springer-Verlag, 2000.
29. H. Sipma, T.E. Uribe, and Z. Manna. Deductive model checking. In R. Alur and T.A. Henzinger, editors, *8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 208–219. Springer-Verlag, 1996.