Incremental Verification by Abstraction *

Y. Lakhnech1xx, S. Bensalem1, S.Berezin2, and S. Owre3

1 VERIMAG, Centre Equation 2 Av. de Vignate, 38610 Gères, 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. \cite{15, 17, 20}) 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 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. \cite{16, 17} for infinite-state systems. A less studied issue is that of finding/constructing the abstract domain and the abstraction relation. The 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

* 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-9512383 and CCR-9509831.

xx Contact Author.

together with abstraction functions have been designed and extensively studied. 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. 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, e.g. [28, 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 different 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 Kur-

\footnote{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.}
shan’s work \[20\] 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 \[20\], the splitting in \[20\] is done on atomic formulae instead of equivalence classes which correspond to boolean combinations of these. A similar idea is applied in \[20\].

2 Preliminaries

2.1 Invariants

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.

A transition system is given by a triple \((\Sigma, I, R)\), where \(\Sigma\) 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 \(s_0, \ldots, s_n\) 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 \(s_0, \ldots, s_n\) of \(S\) with \(s_n = s\). We denote by \(\mathcal{R}(S)\) the set of states reachable in \(S\).

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.

2.2 Abstractions

Abstraction techniques \[19\] 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 \(S^o\) such the concretization of \(\mathcal{R}(S^o)\) is a super-set of \(\mathcal{R}(S)\). The use of abstractions techniques in the context of model-checking is well-studied \[19\] \[43\]. The theory is based on the notion of simulation (also called 1-simulation, forward-simulation...) and on preservation results which tell us which properties that are satisfied by \(S^o\) are also satisfied by \(S\).

A drawback of this method is that the simulation notion used does not take into account the invariance property we want to prove. To overcome this, we proposed in \[41\] the following invariant-dependent simulation notion.
**Definition 1.** We say that $S^o$ is an abstraction of $S$ with respect to $\alpha \subseteq \Sigma \times \Sigma^o$ and $P \subseteq \Sigma$, denoted by $S \sqsubseteq_{\alpha}^P S^o$, if the following conditions are satisfied:

1. $\alpha$ is a total relation,
2. for every state $s_0, s_1 \in \Sigma$ and $s^0_1 \in \Sigma^o$ with $s_0 \in P$ and $(s_0, s^0_1) \in \alpha$, if $(s_0, s_1) \in R$ then there exists a state $s^0_1 \in \Sigma^o$ such that $(s^0_1, s^0_1) \in R^o$ and $(s_1, s^0_i) \in \alpha$.
3. $I \subseteq P$, and
4. for every state $s$ in $I$ there exists a state $s^o$ in $I^o$ such that $(s, s^o) \in \alpha$. □

Now, it can be proved by induction on $n$ that for every computation $s_0, \ldots, s_n$ of $S$ such that $s_i \in P$, for every $i = 0, \ldots, n - 1$, there exists a computation $s^0_0, \ldots, s^0_n$ of $S^o$ such that $(s_i, s^0_i) \in \alpha$, for every $i \leq n$. Therefore, we can state the following preservation result:

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

### 3 A General Verification Rule

In this section, we present a general rule for verifying invariance properties which combines the two main approaches 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 $P$.

An **uniform rule** While Theorem 1 allows us to deduce $S \models \square P$ in case $S^o \models \square P^o$, they do not tell us whether it is possible to take advantage from $S^o$ in case $S^o \not\models \square P^o$. Rule (Inv-Uni) (see Fig. 1), 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 $S$. In fact, these concretizations are used to weaken the third premise of the rule (Inv-Uni).

**Theorem 2.** The proof Rule (Inv-Uni) (see Figure 1) is sound and complete.

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

There exists $\alpha \subseteq \Sigma \times \Sigma^*$
\[ S \subseteq R.S \]
\[ \alpha^{-1}(R(S^*)) \subseteq Q \]
\[ Q \cap P \cap P' \subseteq \wp(R, P \cap P') \]
\[ I \subseteq P' \]
\[ S \models \Box(P \cap P') \]

Fig. 1. Proof rule (Inv-Uni).

Let $s_0, \ldots, s_n$ 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 the standard rule for proving invariants, e.g. [40]. \hfill \square

3.1 Concretizing OBDD’s

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

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

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 propositional 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, $b = x \cdot b_{x \rightarrow \text{true}} + \overline{x} \cdot b_{x \rightarrow \text{false}}$. Equivalently, this can be written as a formula
\[
(x = \text{false} \rightarrow \text{formula}(b|_{x \rightarrow \text{false}})) \land (x = \text{true} \rightarrow \text{formula}(b|_{x \rightarrow \text{true}})),
\]
where \text{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
\[
\bigwedge_{v \in \text{type}(x)} (x = v \rightarrow \text{formula}(b|_{x \rightarrow v})).
\]
\( \text{bdd2f}(b; \text{BDD}, \text{var list}; \text{lst of variables}): \text{formula} = \)
\begin{enumerate}
\item if \( b \in H \) then return \( H(b) \);
\item if \( b = \text{true} \) then \( \text{res} := \text{TRUE} \);
\item else if \( b = \text{false} \) then \( \text{res} := \text{FALSE} \);
\item else \( x := \text{car}(\text{var list}) \);
\item \( \text{res} := \text{TRUE} \);
\item for every \( v \in \text{type}(x) \) do
\item \( \text{tmp} := \text{bdd2f}\left( b[+v, \text{cdr}(\text{var list})] \right) \);
\item \( \text{res} := \text{res} \land (x = v \rightarrow \text{tmp}) \);
\item end;
\item \( H \rightarrow (b, \text{res}) \);
\item end if
\item return \( \text{res} \);
\end{enumerate}
end \text{bdd2f}

Fig. 2. Basic algorithm converting BDD to a formula.

The basic algorithm is shown on Figure 2. 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 occurrences 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 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.

\textit{Example 1.} Let us consider the abstract system of the Bakery example (see e.g. 1). If we apply the basic algorithm (see Figure 2) to the obdd that char-

\(^2\) We assume that the types are always finite.
acterizes the reachable states of this abstract system, we obtain the following formula:

\[
\begin{align*}
(a1 & \Rightarrow (a2 \Rightarrow a3 \land pc1 = 111 \land pc2 = 121) \land \\
& (\neg a2 \Rightarrow a3 \land pc1 = 111 \land pc2 \neq 121)) \land \\
& (\neg a1 \Rightarrow (a2 \Rightarrow \neg a3 \land pc1 \neq 111 \land pc2 = 121) \land \\
& (\neg a2 \Rightarrow (a5 \Rightarrow pc1 \neq 111 \land pc2 = 122) \land \\
& (\neg a3 \Rightarrow pc1 = 112 \land pc2 \neq 121))
\end{align*}
\]

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

\[
\begin{align*}
(y1 = 0 \land y2 = 0) & \Rightarrow pc1 = 111 \land pc1 = 121 \\
(y1 = 0 \land y2 > 0) & \Rightarrow pc1 = 111 \land pc1 \neq 121 \\
(y1 \neq 0 \land y2 = 0) & \Rightarrow pc1 \neq 111 \land pc1 = 121 \\
(y1 \neq 0 \land y2 \neq 0 \land y1 \leq y2) & \Rightarrow pc1 \neq 111 \land pc1 = 122 \\
(y1 \neq 0 \land y2 \neq 0 \land y1 > y2) & \Rightarrow pc1 = 112 \land pc1 \neq 121
\end{align*}
\]

In this example, the concrete invariant obtained by this approach is stronger than the invariant generated by the method presented in \textit{4.3}. 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 4, respectively Rule (Inv-Uni), is finding a suitable abstraction relation \( \alpha \). 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 \( \rho \) is given as a finite disjunction of transitions \( \tau_1, \cdots, \tau_n \), where each \( \tau_i \) 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 \( \bigwedge_{i \in N} \wp_i^\alpha(P) \). Then, \( \bigwedge_{i \in N} \wp_i^\alpha(P) \) is also a quantifier-free formula. Let \( \mathcal{F} = \{f_1, \cdots, f_m\} \) be the set of atomic formulas that appear in \( \bigwedge_{i \in N} \wp_i^\alpha(P) \) in the predicate describing the initial states or in the property. (Notice that one can choose \( N \) sufficiently large to include the atomic formulas in the guards.) Then, we introduce for every formula \( f_i \) an abstract variable \( a_i \) and define the abstraction function \( \alpha \) defined by \( a_i \equiv f_i \). In \textit{4.3}, we show how given a
transition system $S$, a predicate $P$ and an abstraction function $\alpha$, we compute a system $S^\alpha$ such that $S \sqsubseteq^\alpha P$ and a predicate $P^\alpha$ such that $\alpha^{-1}(P^\alpha) \subseteq P$.

Rule (Inv-Uni) addresses the question of how to benefit from computing the set of reachable states of $S^\alpha$ even when $S^\alpha$ does not satisfy $\Box P^\alpha$. In this section we address the following questions:

1. given a counter-example for $S^\alpha \models P^\alpha$ 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 $\sigma^\alpha = s_0^\alpha \tau_1^\alpha s_1^\alpha \cdots s_n^\alpha$ be a counter-example for $S^\alpha \models P^\alpha$. The concretization $\alpha^{-1}(\sigma^\alpha)$ of $\sigma^\alpha$ is the sequence $\alpha^{-1}(s_0^\alpha) \tau_1^\alpha \cdots \alpha^{-1}(s_n^\alpha)$.

We call $\alpha^{-1}(\sigma^\alpha)$ a symbolic computation of $S$, if there exists a computation $s_0 s_1 \cdots s_n$ of $S$ such that $s_i \in \alpha^{-1}(s_i^\alpha)$, for $i = 0, \ldots, n$. Clearly, this definition can be generalized to arbitrary sequences $Q_0 \tau_1 \cdots \tau_n Q_n$, with $Q_i \subseteq \Sigma$. Then, we have the following:

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

**Lemma 1** suggests the procedure ConAnal given in Figure 3 for checking whether an abstract counter-example is a false negative or whether it corresponds to a behavior of the concrete system.

Input: An abstract counter-example $\sigma^\alpha = s_0^\alpha \tau_1^\alpha s_1^\alpha \cdots s_n^\alpha$.

1. $X := \alpha^{-1}(s_n^\alpha)$;
2. $i := n$;
3. While ($X \neq \emptyset$ and $i > 0$) do
4. $Y := X$;
5. $X := \text{pre}_{\tau_i}(X) \cap \alpha^{-1}(s_{i-1}^\alpha)$;
6. $i := i - 1$;
7. End
8. If $i = 0$ and $\emptyset \cap X \neq \emptyset$ then return "the following is a counter-example:
   - Take any $s \in \emptyset \cap X \neq \emptyset$
   - Let $s_0 := s$, $s_1 := \tau_1(s_0)$, \ldots, $s_n := \tau_n(s_{n-1})$
   - Write $s_0 \cdots s_n$.
9. Else return $i, Y$;

**Fig. 3.** Counter-example Analyzer: ConAnal
Refining the abstraction function First, we consider a simple refinement strategy of the abstraction function. Thus, let \( \sigma^c = s_1^c \pi_1^c s_2^c \cdots \pi_n^c s_n^c \) be a counter-example for \( S^c \models \Box P^c \), that is not a symbolic computation of \( S \). By Lemma 4, procedure ConAnal returns some \( i \leq n \) and a set \( Y = X_i \subseteq \Sigma \) such that \( X_{i-1} = \emptyset \). Now, since \( X_{i-1} = \emptyset \), \( Q_{i-1} \subseteq wp_{\pi_i}(\neg X_i) \) 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 \( \alpha \), a corresponding new abstract variable \( a_f \) with \( a_f \equiv f \). Let \( \alpha_c \) denote the so-obtained new abstraction function. Moreover, let \( S^c_{\alpha} \) be the abstract system with \( (s_i^1, s_i^2) \in \rho_{\alpha} \) iff there exist concrete states \( s_1, s_2 \) such that \( (s_i, s_i^2) \in \alpha_c \), for \( i = 1, 2 \), and \( (s_1, s_2) \in \rho \). Then, \( \sigma^c \) is not a computation of \( S^c_{\alpha} \).

Speeding-up refinement of abstraction functions The simple illustrative example given in Figure 4 shows that in general applying finitely many times the procedure ConAnal 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 ConAnal we will have the abstraction function defined by \( a_1 \equiv x = y, \ldots, a_i \equiv x + i = y, \ldots, a_n \equiv x + n = y \). However, the abstraction function we need is \( a \equiv x = y \). 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 ConAnal to cope with this situation. Let us first explain the main idea.

![Figure 4](image_url)

**Fig. 4.** 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 4 where \( l_0, l_1, l_2 \) are the control locations and \( x \) and \( y \) are the data variables. Let \( \sigma^c = s_1^c \pi_1^c s_2^c \cdots \pi_n^c s_n^c \) be a counter-example for \( S^c \models \Box P^c \). Assume that \( \pi_6, \ldots, \pi_i \) is a loop in the control graph of the concrete system. In the procedure ConAnal we apply one time \( pre_{\pi_i} \) on each \( X_i \). However, since \( \pi_i \) is a loop, it is more interesting to apply an arbitrary number of times \( pre(\pi_6, \ldots, \pi_i) \) on \( X_i \), that is, to consider \( \bigvee_{i \in \omega} pre^i(\pi_6, \ldots, \pi_i) \) on \( X_i \).

\footnote{We assume that \( i > 0 \) as the case of \( i = 0 \) is easily handled}
For instance, in the example of Figure 4 applying $\bigvee_{i \in w} \text{pre}(x + k)$ on $x = y$
will give after quantifier elimination the predicate $x \leq y$. Now, since $\text{pre}(x, y :=
1, 0)(x \leq y)$ is empty our strategy consists in adding an abstract variable $k$ such
that $k$ 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
AccConAnal given in Figure 5. There are several remarks to say about pro-

\textbf{Input:} An abstract counter-example $\sigma^a = s^n_0 s^n_1 \ldots s^n_m$;
we $L_j, \ldots, L_m$ be loops in the concrete system such that
\begin{itemize}
\item $L_0 = \tau_0, \ldots, \tau_k + \delta_k$ and
\item $\tau_1, \ldots, \tau_m = \tau_0, \ldots, \tau_{j - 1}, \tau_{j + \delta_k + 1}, \ldots, \tau_{k + \delta_k + 1}, \ldots, \tau_m, \tau_{m + \delta_k}$
\item $X := \alpha^{-1}(s^n_0)$;
\item $i := n$
\item $k := m$
\end{itemize}
\begin{algorithmic}
\While{$(X \neq \emptyset)$ and $i > 0$ do
\begin{itemize}
\item $Y := X$;
\item if $i = i_k$ then
\begin{itemize}
\item $X := \bigvee_{j \in w} \text{pre}_{L_j}(X) \cap \alpha^{-1}(s^n_{i - 1})$;
\item $i := i - \text{length}(L_k)$
\end{itemize}
\item else $X := \text{pre}_{\theta}(X) \cap \alpha^{-1}(s^n_{i - 1})$
\end{itemize}
\EndIf
\item $i := i - 1$
\EndWhile
\EndIf
\begin{itemize}
\end{itemize}
\item if $i = 0$ and $\emptyset \cap X \neq \emptyset$ then return "S does not satisfy the property"
\item else return $i, Y$
\item fi
\item fi
\item fi
\item fi

\textbf{Fig. 5.} Accelerated Counter-example Analyzer: AccConAnal

\textbf{procedure} AccConAnal. The first one is that for a sequence $\tau_1, \ldots, \tau_n$ of transitions there
are in general several but finitely many ways to partition it in $\tau_1, \ldots, \tau_{j - 1}, L_1, \tau_{j + \delta_k + 1}, \ldots, L_m$.
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 $\bigvee_{j \in w} \text{pre}_{L_j}(X) \cap \alpha^{-1}(Q_{i - 1})$, in particular, when possible, one should eliminate the existential quantification on $i$. 

5 Example

To illustrate how we can use the procedure ConAnal, we consider the verification of the Bounded Retransmission protocol [14,15,16], 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 6). Sender transmits data frames to the receiver 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.
**Concrete 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, $\text{BRP}^+$, satisfies the invariance property $\square \text{Safe}$, where $\text{Safe}$ is a variable that is set to $\text{false}$ as soon as BRP makes a transition that is not allowed by BRP-spec. It should be realized that $\text{BRP}^+$ 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 $\text{file}$ and $\text{afile}$ which correspond to the file to be sent and two variables $\text{head}$ and $\text{ahead}$ which correspond to the position of the data being processed in $\text{file}$ and $\text{afile}$, respectively. It is also worth mentioning that the variables $\text{head}$ and $\text{ahead}$ are never compared in $\text{BRP}^+$. Any relation between them these variables can not be deduced from the specification of $\text{BRP}^+$. 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 $\text{max}$, the length of a file $\text{last}$, and finally, the data type $\text{Data}$. Let us describe now the main steps we followed in the verification of BRP using InVeSt $\texttt{I}$. An initial abstraction function is generated automatically and used to compute an abstract system of the $\text{BRP}^+$. This initial abstraction function is obtained from the predicates describing the initial states, the specification and $\bigwedge_{k \leq N} \text{wp}^k_i(P)$ 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:

| $\text{head}=\text{last} \equiv \text{file} = \text{MANY}$ | $\text{afile}[\text{ahead}]=\text{data}(\text{msg})] \equiv \text{msgafile}$ |
| $\text{head}=\text{last} \equiv \text{file} = \text{ONE}$ | $\text{afile}[\text{ahead}]=\text{data}(\text{k})] \equiv \text{lafile}$ |
| $\text{head}=\text{last+1} \equiv \text{file} = \text{NONE}$ | $\text{rn} = 0 \equiv \neg \text{rn}$ |

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 ConAnal. 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 $h$, that is true iff $\text{head} = \text{ahead}$. Then, analyzing the abstract system obtained by this new abstraction shows a new counter-example. In this new counter-example, first $\text{head}$ is incremented which on the abstract level assigns false to $h$, as initially $\text{head} = \text{ahead}$. Then, after a few steps, $\text{ahead}$ is incremented. Thus, though, at the concrete level
head = ahead, this cannot be inferred at the abstract level. Indeed, applying CoaAnal 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 4 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

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 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 [4] we did not consider liveness properties. For infinite-state systems, the key issue is techniques for automatically generating ranking functions and fairness conditions (cf. [1,2,4]).

References


