Abstraction as the Key for Invariant Verification

Saddek Bensalem and Susanne Graf and Yassine Lakhnech

VERIMAG [*]
Centre Equation
2, avenue de Vignate
F-38610 Gières
France
{bensalem,graf,lakhnech}@imag.fr

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 particular, it shows how to use 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 notion of simulation than Milner's.

1 Introduction

Verification can always been expressed in terms of fixed point computation over a domain that corresponds to the properties of interest. The properties can be described as sets of states, sets of traces (sequences of states) or sets of trees. A key issue is then the tractability of the computation of these fixed points. In general, in order to be able to compute such fixed points one either has to restrict to finite systems or to be satisfied with approximations of fixed points. Even in the case of finite systems, however, where the effectiveness of the computation of the fixed points is given, one has to rely upon approximations for efficiency reasons. The main issue is that the applied approximations allow to safely deduce properties of the concrete fixed points. One is usually interested in stating whether the concrete fixed point is contained or contains a given property. This is the question of the correctness of the applied abstraction.

There exist two closely related frameworks for developing abstractions and proving their correctness. The framework of simulation [Par81,Mil71] is about structural relation between abstract and concrete transition systems, representing the step relation of programs by means of an abstraction relation between abstract and concrete sets of states, where each concrete (set of transitions) transition must be simulatable by an abstract transition. Thus, non-reachability properties, in particular invariants, of the abstract transition system, hold also on the concrete system. In order to make verification of properties automatically

[*] VERIMAG is a research laboratory associated with CNRS, Université Joseph Fourier and Institut Nationale Polytechnique de Grenoble

verifiable, the abstract transition system is in general chosen finite, even if the concrete transition system is not.

Abstract interpretation [CC77,CC79] replaces the relation between concrete and abstract states by an abstraction function α from concrete sets of states (called properties) to the smallest element of some abstract property lattice, which represents all the elements of the concrete sets. With α is associated a concretisation function γ, which associates with each abstract element the set of all concrete states represented by it, in such a way that the pair (α, γ) forms a Galois connection. With any function f operating on concrete properties is then associated an abstract function f_A such that $\alpha \circ f \circ \gamma \subseteq f_A$, where $\subseteq$ is the approximation ordering. For instance, if we take for f the successor function on concrete sets of states, an invariant can be obtained by computing the fixpoint of f_A applied on the abstract image of the set of initial states [CC80]. In this framework, the abstract domain need not be finite, or more precisely, the finite abstract domain in which the approximated fixed point is computed need not be fixed a priori. It can rather be choosen during computation of the fixed point. This is possible when one can design a *widening* operator which allows to approximate limits of infinite chains.

An important difference between the two frameworks is that the abstract interpretation framework is about the *computation* of abstract properties, whereas simulation is mainly about *abstract systems* whose properties are preserved by concretization. In the simulation framework, nothing is said about how to *compute* a simulation. Only in the context of strong preservation (the property holds on the abstract system if and only if it holds on the concrete one) by bisimulation [Mil80], computation of abstract systems had initially been proposed.

In [BBLS92,LGS+95], we have shown that any simulation relation ρ between a set of concrete and abstract states can be represented in the abstract interpretation framework by taking the pair of associated image functions $(post[\rho], \textsc{wp}[\rho])$ as the corresponding Galois connection, and the other way round, each Galois connection can be put into the form $(post[\rho], \textsc{wp}[\rho])$, as long as the abstract lattice is constructed from a partition of the set of concrete states. Also [DGG93,Dam96] discuss in great detail the relationship between abstract interpretation and the verification by abstraction approach. Both frameworks allow to deal with over- as well as under-approximation of the fixed points. In the framework of simulation a combined framework has been introduced in [Lar89,LSW95] by means of *modal transition systems*, which have *may*-transitions representing supersets of actual transitions and *must*-transitions representing subsets.

The property preservation results of [LGS+95] and [DGG93] lead to an increasing interest in the combination of model-checking and abstraction. Indeed, given a large or infinite state system that has to be verified, one can first compute a tractable finite system that abstracts the given concrete one, then check abstract properties on the obtained systems and finally, in case these properties are satisfied, deduce that the concrete system satisfies the concrete proprties. This is the so-called *verification by abstraction approach.* To apply this approach one needs to first find an accurate abstract space and abstraction relation and then automatically generate 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. [GS97,BLO98b,BLO98a,ABLS01,CU98] for infinite-state systems.

In this paper, we discuss methods for *automatically generating abstract systems*, refining abstractions and analysing the accuracy of the abstractions, in the context of infinite state systems.

When the infiniteness of a systems comes from the presence of variables with infinite domains, abstractions can be computed by means of abstract data types: for each variable to be abstracted define an abstract domain and lift all operations on the sets representable in the abstract domain. Then, an abstract model is obtained by replacing each variable by one of a corresponding abstract domain, and each operation by an apropriate abstract operation. This is the way abstract interpretation is used, and it has also used in combination with model-checking e.g. in the tool Cospan [Kur89,Kur94], in [Gra94] and in the Bandera tool [HDLS98,CDH$^+$00].

These approaches are often based on very general, predefined abstractions and aimed for generic properties. We investigate here more property depending abstractions where better results are obtained when the atomic entity to be abstracted is a "transition relation"; this is meaningful in the context of systems defined as combinations of a set of transition relations. Automatic abstraction of infinite systems based on this principle have been introduced in [GS97] in the form of predicate abstractions, and has been developed in different contexts since then mostly together with counter example based refinement, e.g. [DDP,JHR99].

Parameterized systems, where the source of infinity is the existence of an unbounded number of parallel components can be dealt with by introducing counters for the number of components in each potential system state, and can then be treated using the above mentioned abstraction methods. Nevertheless, a number of specialised approaches for this kind of systems has been proposed, e.g. [ABJN99,BBLS00,BLS01,MP95b].

Abstract transition relations on finite domains can be used directly in any model-checker to verify any properties which are preserved by the abstraction. Whenever, the abstract system does not satisfy the property, the model-checker will come up with a abstract counter example. If this counter example can be transformed into a concrete one, there is a proof that the concrete system does not satisfy the property. Otherwise, the abstract system is not detailed enough and needs to be refined.

Using *counter-examples* to refine abstract systems has been investigated by a number of other researchers e. g. [Kur94,BSV93,CGJ$^+$00]. Closest to our work is Clarke et al's techniques [CGJ$^+$00]. 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 Kurshan's work [NK00] 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 [BFH90,LY92,HHK95], the splitting in [NK00] is done on atomic formulae instead of equivalence classes which correspond to boolean combinations of these. A similar idea is applied in [SUM96].

Abstraction can also be used in combination with other verification techniques, in particular when in the abstract system still some infinite domain variables exist. A typical case is a system with finite but implict control and infinite data, where abstraction is used to define an explicit finite control, but the properties to be verified depend also on the still infinite data part[1]. In this context, either the above mentioned refinement techniques or the combination with techniques working directly on infinite state systems, such as abstract interpretation or deductive verification, can be used. The combination with abstract interpretation techniques has been proposed in particular for parameterized systems [LS97] where the infinite variables are state counters increased or decreased by each transition, in the context of timed systems or linear hybrid systems where the infinite state variables increase in each control state monotonically.

We are interested here in the use of abstraction in the context of deductive verification. From a theoretical point of view, finding a finite abstraction and finding an invariance proof are equivalent. Nevertheless, in practice, the combination of automatic construction of abstractions and deductive invariance proofs can be useful: use automatic abstraction in order to extract information on global control, information which then is used in a deductive invariance proof. In [LBBO01], we have have shown how to exploit the abstract invariant, here we show that sometimes it can be also be interesting to exploit directly the abstract transition relation.

2 Preliminaries

First we define some notations. 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 in its domain. Given X,we denote by $\Sigma(X)$ the set of states on X.

Definition 1. *(Transition system)*
A transition system *is given by a triple* (Σ, I, R)*, where* Σ *is a set of states,* $I \subseteq \Sigma$ *is a set of initial states, and* $R \subseteq \Sigma^2$ *is the transition relation. In the sequel we always consider sets of states of the form* $\Sigma(X)$ *for some set of variables* X*.*

[1] any finite abstract system which does not satisfy a given property, can be considered as suach a system when considering the concrete transition relation in combination with the constructed finite abstraction

Definition 2. *(Syntactic Transition system)*
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 representing the set of initial states and $\rho(X, X')$ is a predicate representing the transition relation, where the unprimed variables $x \in X$ represent the start states and primed variables $x' \in X'$ target states. Often the transition relation is of the form $\rho = \bigcup \tau_i$ where τ_i is the predicate defining the transition relation of an individual transition.*

With a syntactic transition system is associated in the usual way a corresponding transition system.

Definition 3. *(synchronous product)*
Given two binary relations $R_i \subseteq \Sigma_i \times \Sigma'_i$, for $i = 1, 2$, we define their synchronous or conjunctive product *$R_1 \otimes R_2 \subseteq (\Sigma_1 \cup \Sigma_2) \times (\Sigma'_1 \cup \Sigma'_2)$, by $(s, s') \in R_1 \otimes R_2$ iff $(s_{|\Sigma_i}, s'_{|\Sigma'_i}) \in R_i$, for $i = 1, 2$, where $s_{|\Sigma_i}$ denotes the restriction of the state s to Σ_i.*

Thus, if $\Sigma_1 = \Sigma_2$ and $\Sigma'_1 = \Sigma'_2$ then $R_1 \otimes R_2 = R_1 \cap R_2$.

Definition 4. *(synchronous composition)*
The synchronous composition *of transition systems $S_i = (\Sigma_i, I_i, R_i)$, $i = 1, 2$, denoted $S_1 \otimes S_2$, is the transition system $(\Sigma_1 \cup \Sigma_2, \{s \mid s_{|\Sigma_1} \in I_1 \wedge s_{|\Sigma_2} \in I_2\}, R_1 \otimes R_2)$.*

Definition 5. *(computation and reachable states)*

- *A* computation *of a transition system $S = (\Sigma, I, R)$ is a sequence of states $s_0, \cdots, s_n$ such that $s_0 \in I$ and $(s_i, s_{i+1}) \in R$, for $i \leq n - 1$. As we are interested in safety properties only and abstraction, we never consider infinite computations.*
- *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* Reach(S) *the set of states reachable in S.*

Definition 6. *A set $P \subseteq \Sigma$ is called an* invariant *of S, denoted by $S \models \Box P$, if* Reach$(S) \subseteq P$*, that is, every reachable state in S is in P.*

Definition 7. *(The predicate transformers post, wp and pre)*
Given a relation $R \subseteq \Sigma \times \Sigma'$, we denote the predicate transformers or image and inverse image functions associated with R as follows. Let be $P \subseteq \Sigma$ and $P' \subseteq \Sigma'$. Then

- *The* postcondition *associated with R with respect to P, denoted by $\textsc{post}[R](P)$, is the image function $2^{\Sigma} \rightsquigarrow 2^{\Sigma'}$ associated with the relation R applied to P,n that is $\textsc{post}[R](P) = \{s' \in \Sigma' \mid \exists s \in \Sigma \,.\, (s, s') \in R\}$.*
- *The* precondition *of R with respect to P', denoted by $\textsc{pre}[R](P')$, is the pre-image of P' by R, that is $\textsc{pre}[R](P') = \{s \in \Sigma \mid \exists s' \in \Sigma' \,.\, (s, s') \in R\}$ We also sometimes write $\textsc{pre}_R(P')$ instead of $\textsc{pre}[R](P')$.*

– *the* weakest liberal precondition *of* R *with respect to* P*, denoted by* $\text{WP}[R](P')$ *or* $\text{WP}_R(P')$*, is the dual of* $\text{PRE}[R](P')$*, that is the set consisting of states* s *such that for every state* s'*, if* $(s, s') \in R$ *then* $s' \in P'$*.*
$\text{WP}[R](P') = \{s \in \Sigma \mid \forall s' \in \Sigma'[(s, s') \in R \Rightarrow s' \in P']\} = \overline{\text{PRE}}(R)(\overline{P'})$

Definition 8. *(Inductive invariant) A set* $P \subseteq \Sigma$ *is called an* inductive invariant *of* S*, if*

– $I \subseteq P$*, that the set of initial states is included in* P*.*
– $P \subseteq wp(R, P)$*, that is the transition relation* R *does not allow to "leave" the set* P

Proposition 1. *Un inductive invariant of* S *is also an invariant of* S*.* □

Notice that we often use semantic notations such as "set of states", interchangably with the corresponding syntactic notations "predicate on X". Also the notions of preconditions and invariant are supposed to be defined likewise semantically as well as syntactically. In the sequel, we will freely switch between semantic and syntactice notations, unless explicitly stated.

3 Deductive verification of invariance properties

There are basically two approaches to the verification of reactive systems, the *algorithmic* approach and the *deductive* approach. The algorithmic approach is based on the computation of fix-points, on effective representations of sets of states, and on decision procedures for solving the inclusion problem of sets of states. For example the *backward procedure* is an instance of this approach. To prove that a set of states P is an invariant of a system S, the backward procedure computes the largest set of states Q satisfying $Q \subseteq P$ and $Q \subseteq \text{WP}(\tau, Q)$, for every transition where $R = \bigcup \tau_i$; this is done iteratively, by eliminating at each iteration the set of states which can leave the current approximation X by computing $X^{i+1} = X^i \cap \bigcap_{\tau \in R} \text{WP}(\tau, X^i)$. Then, P is an invariant of S if and only if every initial state of S satisfies the fixpoint Q, which is the greatest inductive invariant of S included in P, and when P can be expressed as a temporal logic formula φ, Q is the semantic of the temporal formula $\Box\varphi$ or $\forall\Box\varphi$. In general, the algorithmic approach is based on :

1. an effective representation $\mathcal{R}ep$ for sets of states,
2. effective boolean operations and predicate transformers in $\mathcal{R}ep$
3. a procedure for deciding inclusion in $\mathcal{R}ep$,
4. and convergence of fix-points to guarantee completeness.

In general, in case of infinite state systems, first-order logic with Peano arithmetic is considered as representation $\mathcal{R}ep$. In fact, any weaker logic is not expressive enough (e.g. [dB80]), when the considered system contains variables that range over infinite domains. Thus, one has effective boolean operations and can define predicate transformers, but inclusion is undecidable. Moreover, convergence of fix-points is not guaranteed. Consequently, the algorithmic approach cannot be applied in general to infinite state systems.

3.1 Deductive verification

The deductive approach is very powerful and gives a complete method even for infinite state systems. It relies upon finding *auxiliary invariants* and proving validity of first-order formulas, called *verification conditions*. The deductive approach is, however, in contrast to the algorithmic approach, difficult to apply in practice. It is in general hard to find suitable auxiliary invariants and time consuming to discharge all generated verification conditions. One means, for making deductive verification more efficient is, as we will see later on, abstraction.

To prove, by induction, that φ is an invariant of of a system S one has to come up with a stronger inductive invariant ψ that is preserved by every transition of S. As shown, e.g. in [MP95a], Rule (Inv), is a sound and complete for proving invariance properties of transitions systems. A predicate ψ which satisfies the

$$\boxed{\dfrac{\begin{array}{c}\text{There exists a predicate } \psi \text{ s. t.}\\ \psi \Rightarrow \varphi\\ I \Rightarrow \psi\\ \psi \Rightarrow \textsc{wp}(R, \psi)\end{array}}{S \models \Box\varphi}}$$

Fig. 1. Proof rule (Inv).

premises of the rule above is called *auxiliary invariant* for a transition system S and φ. The (semantic) completeness proof of the rule states that for every transition system S and predicate φ, if $S \models \Box\varphi$ then there exists an auxiliary predicate ψ for S and φ.

It is very important to understand that the completeness result/proof of Rule (Inv) does not say anything about how to find an auxiliary invariant. The creative task in deductive invariant verification is to come up with a predicate ψ which satisfies $\mathit{Reach}(()S) \Rightarrow \psi \Rightarrow \Box\varphi$ and which is an inductive invariant. Indeed, $\mathit{Reach}(S)$ is the smallest inductive invariant of S, but it is in general not a very helpful invariant in the deductive approach, as to come up with a predicate representing $\mathit{Reach}(S)$, makes necessary a, possibly infinite, fixpoint computation. Alternatively, one could choose the set $\varphi \cap \bigcap \textsc{wp}^i(R, \varphi)$ as a guess for an auxiliary invariant. And if this does not allow to prove the invariance of φ, try $\varphi \cap \textsc{wp}(R, \bigcap \textsc{wp}^i(R, \varphi))$, and so on. In fact, this corresponds to the inductive computation of the weakest auxiliary invariant $\Box\varphi$ for S and φ. Notice that this algorithm does not necessarily converge, and only the fixpoint is an inductive invariant.

When S is a finite-state system each of these predicates can be expressed in propositional logic, checking the premises can be done algorithmically and convergence of fixpoint computations is guaranteed. When Presburger Arithmetic is sufficient to express all these predicates, then checking the premises of the rule is still decidable, but convergence of fixpoint computations is not guaranteed. When general first order logic is necessary, even the decidability of implications is lost. Our examples, will often be taken from the group of those where decidability of implication is given, but reaching of fixpoints cannot be guaranteed.

Notice that generic techniques for generating and strengthening invariants (cf. [MP95a,BBM97,BLS96,SDB96,BL99]) seem to give limited results, except when the property to be proven invariant is in some sense "almost" an invariant or when the system is relatively simple.

To summarize, the deductive method has three drawbacks:

1. it is often hard to find suitable auxiliary invariants,
2. when a guess for an auxiliary invariant fails, one has little hint how to strengthen it and finally
3. contrary to algorithmic methods, where for a failed invariant proof, one can always construct effectively a counter-example, here a failed proof doesn't even prove the existence of a counter-example.

4 Abstraction

Property preserving abstractions is a crucial technique for pushing further the limits of model-checking: given a program and a property to be verified, find a (simpler) abstract program such that the satisfaction on the abstract program implies the satisfaction on the initial program, called concrete program. Simulation as defined by Milner [Mil71] yields an appropriate preorder. In section 4.2, we introduce the notions of abstraction and property preservation and show that, given some invariant, a weaker notion of simulation, imposing no requirement on the successors of states outside the invariant, preserves the same set of properties.

An important point is, given a concrete program, how to *construct* an abstract program that is both, simple enough in order to be verified by available tools, and that still contains enough relevant details for the satisfaction of the considered properties. Several alternative methods for constructing abstractions are discussed in detail in section 4.3.

4.1 An introductive example

In order to give an intuitive idea about abstraction, let us consider the following small reader-writer example :

```
system  $RW$
vars
     $nr, nw$ : nat;  %representing the number of readers resp. writers
init
     $nr = 0 \wedge nw = 0$;
do
    $nw = 0 \wedge nr = 0 \longrightarrow nw := 1$;          %nobody active; a writer can become active
  [] $nw > 0$              $\longrightarrow nw := nw - 1$;  % writer terminates
  [] $nw = 0$              $\longrightarrow nr := nr + 1$;  %no writer active; readers read at any time
  [] $nr > 0$              $\longrightarrow nr := nr - 1$;  %a reader terminates
od
```

In this example, the set of states is the set $I\!N \times I\!N$ of all values (v_1, v_2) for variables nw, nr. There is only one initial state, in which no reader nor writer is active ($I = \{(0,0)\}$) and the transition relation is given by the guarded commands.

Assume we want to formally reason about this transition system. For example that it satisfies mutual exclusion between readers and writers. In order to do so, it is enough to show that $(nr > 0) \Rightarrow (nw = 0)$ (whenever there is a reader active, then there is no writer) is an invariant. As the S_{wr} example has an infinite state space (cf. figure 2) and we cannot directly apply the algorithmic approach[2]

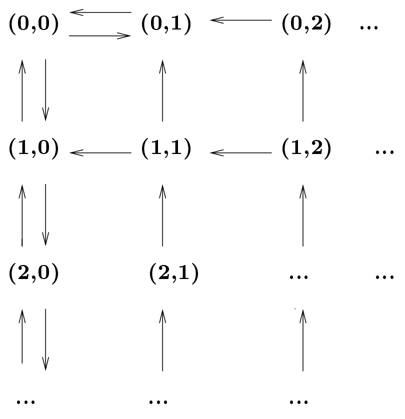

Fig. 2. The system S_{rw}

As already motivated earlier, the construction of an abstract transition system, which *simulates* the concrete transition system of Figure 2 is a possible way to overcome the complexity problems induced by large or infinite state spaces.

[2] Notice that for this simple example, having only integer variables with the + and - operation, we could use tools using polyhedra representations [HPR94] to directly compute the set of reachable states. On the other hand, we know that the deductive approach generates only decidable verification conditions.

The key elements are:

1. Defining a set of *abstract states* and a *mapping from concrete to abstract states*
2. construct an abstract transition system by constructing the *abstract initial states* and an *abstract transition relation*, and
3. finally *reason* on the initial system by examining the abstracted system which has "less states" or "a simpler representation"

We present these elements for the small example in order.

The graph of Figure 2 suggests to use the partition of the set of concrete states induced by the predicates $nw = 0$ and $nr = 0$ to define the set of abstract states (and the corresponding mapping. Notice that htes predicates (or there negations) are used in the guards and they define four abstract states $\Sigma_1, \ldots, \Sigma_4$. There is

– the initial state $(0, 0)$ ($\Sigma_1 = I$) in which there are whether readers nor writers, $(nw = 0) \wedge (nr = 0)$
– Σ_2 represents the states where only readings are being performed that is $\neg(nr = 0) \wedge (nw = 0)$
– Σ_3 represents the states where only writings are being performed, that is $(nr = 0) \wedge \neg(nw = 0)$ and
– and Σ_4 represents all the remaining states where both write and read are performed, that is $\neg(nr = 0) \wedge \neg(nw = 0)$

The mutual exclusion property states that no state Σ_4 can be reached from the initial state.

The abstract transition relation contains at last all those transitions $(\Sigma_i\ ,\ \Sigma_j)$ such that there exists a least one pair of concrete states $s_i \in \Sigma_i$, $s_j \in \Sigma_j$ and the $(s_i, s_j) \in R_{rw}$.

This definition of the abstract transition relation preserves abviously "non-reachability": if Σ_4 is not reachable in this abstract system, then by definition, there exists no concrete transition reaching the set Σ_4 from any state in Σ_1, Σ_2 or Σ_3. But in the other way round, reaching or non-satisfaction of an invariant is not preserved by abstraction.

In this example the abstract transition is indeed very precise: the abstract transition relations corresponding to *start-w* and *start-r* are detemriministic deterministc and for every abstract transition ($\Sigma_i \longrightarrow \Sigma_j$), for every concrete state σ represented by Σ_i has a concrete transition into the set represented by Σ_j, and the other way round, every concrete state σ' represented by Σ_j can be reached by a concrete transition from a state in the set represented by Σ_i.

The abstract transition relations corresponding to *end-w* and *end-r* are non-deterministic, as for example the abstract state Σ_2 represents two classes, the state with $nw = 1$ for which *end-w* leads to state Σ_1, and all other states for which the ranstion *end-w* leads back to state Σ_2. Therefore, on the abstract transition system, the behaviour *start-w*, *end-w*, *end-w*,.... is a possible behaviour, whereas on the concrete system, it is not. Intuitively, simulation based abstraction preserves properties of *all* executions, and in particular invariants, whereas,

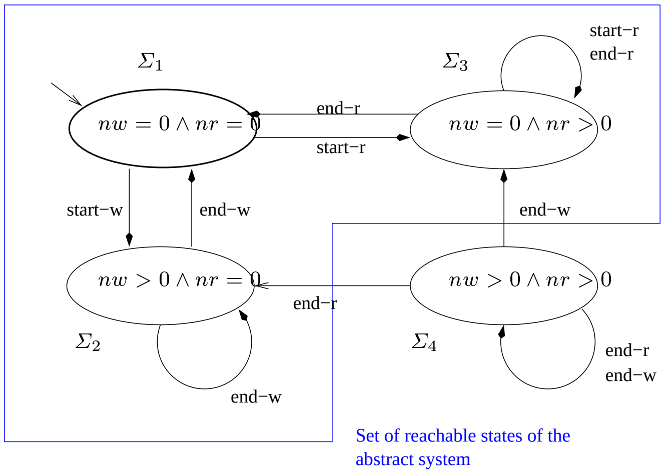

Fig. 3. The abstract system S^A_{rw}

from the existence of a a particular abstract behaviour nothing can be concluded on the concrete system.

In order to prove that $P = nw = 0 \vee (nr = 0) \wedge (nw < 1)$ is an invariant, two possiblities exist.

The first one consists in refining the state Σ_2 which contains both, states in P and states not in P.

When we split Σ_2 into two abstract states, one containing only the value $nw = 1$ and the other all other values of nw, we get, using the same definition of the abstract transition relation, the abstract system of the figure 4 above. The set of reachable states of this refined abstraction is $nw = 0 \vee (nr = 0) \wedge (nw < 1)$. Notice that for the construction of the refined transition relation, only the transitions starting from or leading to the new states Σ_{2a} and Σ_{2b} need to be checked. Moreover as state Σ_4 is known to be non-reachable, thus uninteresting, we need not really care about the transitions starting from Σ_4. Refinement of abstractions are presented in Section 5.3.

A second alternative of making use of an abstract system is

- to use the set of reachable states $nw = 0 \vee nr = 0$ as known auxiliary invariant to prove the invariance of P by the deductive approach.
- to use *transition invariants* of the abstract system in the deductive approach. A transition invariant of the system S^A_{rw} is that *all successor of a transition end-r starting in a reachable state satisfy* $nw = 0$.

Nevertheless, these auxiliary invriants and transition invariants are not helpful in this example, as the new invariant to be proved P is a strict subset of the

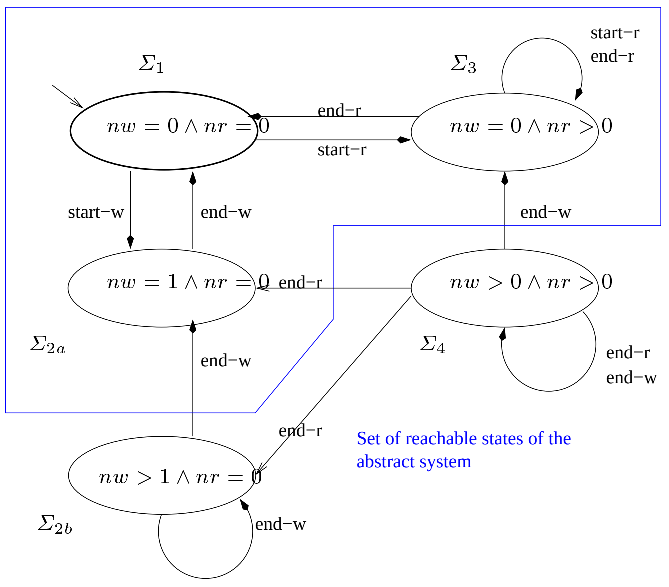

Fig. 4. The refined abstract system $S_{rw}^{A'}$

invariant obtained from the abstraction, and as all verification conditions are very simple. More about the use of abstract invariants in deductive proof rules is given in section 5.1.

4.2 Definition of abstraction and preservation results

We consider here abstraction based on the notion of simulation [Par81,Mil71] and on preservation results which tell for which properties, we can conclude true the fact that they hold on the abstract system S^a, then they hold also on the concrete system S^c.

In this section, we recall the definition of simulation as well as preservation results for invariance properties.

Notation 41 *Let be $\alpha \subseteq \Sigma^c \times \Sigma^a$ an abstraction relation between a concrete set of states Σ^c and an abstract set of states Σ^a. Whenever, no confusion is possible, we dentote*

- *the image or abstraction function associated with α, by writing $\alpha(\varphi^c)$ instead of $\text{POST}[\alpha](\varphi^c)$*
- *the inverse image, precondition or concretisation function associated with α, by writing $\alpha^{-1}(\varphi^a)$ instead of $\text{PRE}[\alpha](\varphi^a)$. Indeed, $\alpha^{-1}(\varphi^a)$ represents the set of all concrete states represented by the set of abstract states φ^a*

Notice that for any concrete set of states φ^c, the result of an abstraction followed by a re concretisation is a superset of φ^c, that is $\varphi^c \subseteq \alpha^{-1}(\alpha(\varphi^c))$[3]. We say that φ^c is abstracted *exactly*, if $\varphi^c = \alpha^{-1}(\alpha(\varphi^c))$ holds.

Definition 9. *Let $S^c = (\Sigma^c, I^c, R^c)$ and $S^a = (\Sigma^a, I^a, R^a)$ be two transition systems. We say that S^a is an* abstraction *of S^c with respect to a relation $\alpha \subseteq \Sigma^c \times \Sigma^a$, denoted by $S^c \sqsubseteq_\alpha S^a$, if the following conditions are satisfied:*

1. *α is total on Σ^c,*
2. *for all states $s_0, s_1 \in \Sigma^c$ and $s_0^a \in \Sigma^a$ with $(s_0, s_0^a) \in \alpha$, if $(s_0, s_1) \in R^c$ 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^c 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^c with respect to α.

It can be easily proved that if $S^c \sqsubseteq_\alpha S^a$ 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. [CGL94,LGS+95]):

Theorem 1. *Let S^c and S^a be transition systems such that $S^c \sqsubseteq_\alpha S^a$. Let $\varphi^c \subseteq \Sigma^c$ and $\varphi^a \subseteq \Sigma^a$. If $\alpha^{-1}(\varphi^a) \subseteq \varphi^c$ and $S^a \models \Box\varphi^a$ then $S^c \models \Box\varphi^c$*

Thus, to prove that a transition system S^c satisfies an invariance property φ^c it suffices to find a *finite* abstraction S^a of S^c with respect to some relation α such that $S^a \models \Box\varphi^a$ for some abstract predicate $\varphi^a \subseteq \Sigma^a$ whose concretization includes the invariant to be proved.

This method is complete in the sense that, 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^c; and $(s, s_1^a) \in \alpha$ iff s is not reachable in S^c. The abstract system S^a has s_0^a as unique initial state. Obviously, $S^c \sqsubseteq_\alpha S^a$ and $S^a \models \Box\{s_0^a\}$. Moreover, since $S^c \models \Box\varphi^c$, we have $\alpha^{-1}(\{s_0^a\}) \subseteq \varphi^c$.

We can show that, if $S^c \models \Box\varphi^c$ can be proved using an abstraction S^a of S^c with respect to α, then it can also be proved using the auxiliary invariant $\alpha^{-1}(\mathit{Reach}(S^a))$, where $\mathit{Reach}(S^a)$ 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 useful abstractions are easier to find than useful invariants.

Definition 9 can be weakened, allowing more abstractions, while preserving soundness of theorem 1. Indeed, if φ^c is an invariant of S^c then conditions 1 and 2 in definition 9 can be weakened by restricting the quantification on states in Σ^c to states that satisfy φ^c.

We can also weaken definition 9 using a set of states in Σ^c that is *not* necessarily an invariant of S^c leading nevertheless to a (weaker) preservation result. To explain this, let us introduce the following definition:

[3] this result is also true in abstract interpretation, where the concretisation function, call it γ, is chosen as the weakest precondition of the relation α rather than its preimage. In this case, the pair (α, γ) forms a Galois connection, for which we know $\gamma \circ \alpha \leq id$, and for total relations we have $\gamma \leq \alpha^{-1}$

Definition 10. *We say that S^a is an abstraction of S^c with respect to $\alpha \subseteq \Sigma^c \times \Sigma^a$ and $\varphi^c \subseteq \Sigma^c$, denoted by $S^c \sqsubseteq_\alpha^{\varphi^c} S^a$, if the following conditions are satisfied:*

1. *α is total on φ^c,*
2. *for every state $s_0, s_1 \in \Sigma^c$ and $s_0^a \in \Sigma^a$ with $s_0 \in \varphi^c$ and $(s_0, s_0^a) \in \alpha$, if $(s_0, s_1) \in R^c$ 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. *$I^c \subseteq \varphi^c$, and*
4. *for every state s in I^c there exists a state s^a in I^a such that $(s, s^a) \in \alpha$.*

Figure 5 shows the difference between these notions of simulation.

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

Fig. 5. Simulation notions.

In [BLO98a] we have proved by induction on n that for every computation $s_0, \cdots, s_n$ of S^c such that $s_i \in \varphi^c$, 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 $S \sqsubseteq_\alpha^{\varphi^c} S^a$. Let $\varphi^a \subseteq \Sigma^a$ and $\varphi \subseteq \Sigma^c$. If $\alpha^{-1}(\varphi^a) \subseteq \varphi^c \cap \varphi$, and $S^a \models \Box\varphi^a$, then $S^c \models \Box(\varphi^c \cap \varphi)$.* $\Box$

Notice that definition 9 and theorem 1 can be obtained from definition 10 and theorem 2 by taking $\varphi^c = \Sigma^c$.

The advantage of definition 10 is that it allows to define an abstract transition relation in which we do not care about the transitions starting from states from which want to prove that they aren't reachable, similar as for the refinement of the abstract transition relation constructed in section 4.1. This is particularly important when we are seeking a method that automatically computes finite abstractions for analysis by model-checking techniques. Indeed, this result is used in many concrete applications, in the large litterature of abstraction but its correction is mostly shown only informally.

4.3 Computing Abstractions

Verification by means of abstract transition systems can also be applied when the concrete system is infinite state, as shown in [DF95,Gra99,MN95,HS96]. However, in all these approaches the verifier has either to fully provide the abstract system, and only the check that it is indeed an abstraction is tool supported, or, like in [Gra99], the user provides abstract data types which can then be used to compute an abstract transition system. In all cases a relatively important amount of user intervention is required to prove that the abstract system is indeed an abstraction of the concrete one.

What is needed is a method to automatically compute an abstract system for a given infinite state system, an abstract domain and a mapping from concrete to abstract states. We present a method that computes an abstract system $S^a = S_1^a \parallel \cdots \parallel S_n^a$, for a given system $S^c = S_1^c \parallel \cdots \parallel S_n^c$ — where each S_n^c is given by a set of transition relations — and abstraction relation α, such that S^c simulates S^a is guaranteed by the construction. Hence, by the previously established preservation results, if S^a satisfies an invariant φ^a then S^c satisfies its concretisation, $\alpha^{-1}(\varphi^c)$.

We do not want to fix a parallel operator here, as the method builds abstractions of the individual transition relations, and in [GL93a] has been shown that abstraction is preserved by most useful notions of parallel composition. An important point is that the produced abstract system S^a is given in a symbolic manner, which still allows to apply all the known methods for avoiding the state explosion problem, while analyzing S^a. Moreover, there is a clear correspondence between concrete and abstract transitions. This allows for debugging the concrete system, since it can be checked whether a given trace of the abstract system corresponds to a concrete trace.

We consider the problem of computing an abstraction of a transition system S^c with respect to an abstraction relation α. Thus, consider a syntactic transition system $S^c = (C, \theta^c, \rho^c)$, where ρ^c is of the form $\cup_i \tau_i$. Let the abstraction relation α be given by a predicate on $C \cup A$, where A are the variables defining the abstract domain.

The basic idea underlying the methods of [CGL94,GL93b,DGG93,Dam96] for computing abstractions of finite state systems is based on abstract interpretation of individual operators or individual transition relations: the abstract transition relation is completely determined by abstract versions of the primitive operators or of individual transition relations. If the concrete and abstract state space is finite, the abstract transition relation τ_i^a associated with the concrete transition relation τ_i can simply be represented by the relation $\alpha^{-1} \circ \tau \circ \alpha$ on the abstract state space[4]. In [CGL94] abstract transition systems are obtaineded by computing abstractions of primitive operators as defined precedingly (where the abstraction relation is always a function), and in [GL93b] the same principle is applied to symbolic transition systems and abstract domains with boolean variables only.

Here, we follow the approach of abstracting entire individual transition relations rather than primitive operations. We want to compute an abstract transition system of the form $\alpha(S^c) = (A, \alpha(\theta^c), \alpha(\rho^c))$. $\alpha(\theta^c)$ can be represented by the expression $\exists C \cdot (\theta^c \wedge \alpha)$ and $\alpha(\rho^c)$ by $\exists C \exists C' \cdot (\alpha \wedge \alpha' \wedge \rho^c)$[5]. $\alpha(S^c)$ is indeed an

[4] or by its corresponding function on sets, denoted in the same way

[5] as defined earlier, α' is obtained from α by substituting every variable $c \in C$ by c' and every variable $a \in A$ by a'

abstraction of S^c. In case α is a function[6], $\alpha(S^c)$ is the least abstraction of S^c with respect to α.

Unfortunately, it is not possible in general to analyze $\alpha(S^c)$ directly by model-checking even when all the variables in A range over finite domains. The reason is that the description of $\alpha(S^c)$ involves quantification over the variables in C which includes variables ranging over infinite domains and quantifier elimination is not possible in general[7].

The elimination method We present a method for computing abstractions which avoids the direct elimination of quantifiers. Consider again a transition relation given by a predicate $\rho(C, C')$ and an abstraction relation given by a predicate $\alpha(C, A)$. There is a trivial abstraction of ρ with respect to α which is the universal relation on Σ^a. Let us denote it by U_A. Of course one cannot use the universal abstract transition relation U_A to prove any interesting invariant. One can, however, obtain a more interesting abstraction of $\rho(C, C')$ by eliminating transitions from U_A. The following lemma states which transitions can be safely eliminated:

Lemma 1. *Let $S^c = (C, \theta^c, \rho^c)$, $S^a = (A, \theta^a, \rho^a)$ be transition systems such that $S^c \sqsubseteq_\alpha^{\varphi^c} S^a$. Let s_0^a, s_1^a be sets of abstract states. If $\alpha^{-1}(s_0^a) \Rightarrow \textsc{wp}[\rho^c](\neg\alpha^{-1}(s_1^a))$ then $S^c \sqsubseteq_\alpha^{\varphi^c} S'^a$, where S'^a consists of the same components as S^a except that its transition relation is $\rho^a \setminus \{(s_0^a, s_1^a)\}$.* □

In other words, if the concrete transition does not lead from a concrete state s_0^c with $\alpha(s_0^c, s_0^a)$ to a concrete state s_1^c with $\alpha(s_1^c, s_1^a)$, then we can safely eliminate the transition(s) (s_0^a, s_1^a) from S^a. When ρ is given as union of individual transition relations, this can be done individually for each of them. Notice that since the concrete system in general is infinite state the condition $\alpha^{-1}(s_0^a) \Rightarrow \textsc{wp}[\rho^c](\neg\alpha^{-1}(s_1^a))$ can not always be checked algorithmiquely. In our tool, we use the theorem prover PVS [SRSS96] to discharge this kind of verification conditions. Notice also that if we eliminate all the pairs (s_0^a, s_1^a) for which this condition is satisfied, we obtain as result the smallest abstract system $\alpha(S^c)$.

The elimination method in its rough form is not feasible since it requires too many formulas to be checked for validity. Indeed, if there are n boolean abstract variables then there are 2^{2n} such conditions to be checked. Therefore, we present techniques which make the elimination method feasible.

In [GS97] we have presented a method called predicate abstraction, where the set of abstract states is a set of boolean variables $b_1, \ldots b_n$, representing each one a concrete predicate p_i. The method computes both, a symbolic representation of each individual transition relation and a global reachability graph. Each abstract transition relation is initialized to the trivial relation U_A relating all states, and then stepwise refined by eliminating transitions from states which are reachable.

[6] or a quasi function as shown in [LGS$^+$95]

[7] When C is a finite domain, the abstract transition relation is exactly the on computed for example in the tool presented in [GL93b]

This guarantees that any intermediate result represents an abstraction and the refinement can be stopped at any point of time (with the risk to obtain a too rough overapproximation). The refinement is done as follows:

1. start by some straightforward elimination of transitions (leading to a more precise abstraction) exploiting
   - each of the individual transition relations is in general independent of at least some of the predicates (if R is the current representation of an abstract transition relation, and we find out that it is indepenedent of abstract variables b_1 and b_2, its new representation will be $R \wedge (b_1 = b'_1) \wedge (b_2 = b'_2))$
   - dependences between concrete predicates (which induce dependences between the abstract variables).
   - special cases, where in a given transition relation the "nextvalue" of some prediciate is independent of the values of other predicates (for example, when b_3 stand for the concrete predicate $x > 0$, then for any transition relation with an assignment $x := 0$, the corresponding abstract transition relation has a conjunct $b'_3 = false$.

   Sometimes, these simplifications allow to eliminate a large number of transitions, making the transition relation more deterministic, and thus less work is left for the second step.
2. In the second step a more precise approximation of the successors of an (over approximation of) all reachable states is computed. As computing a a precise abstraction of a transition relation depending on k abstract variables may require up to 2^{2n} tests, and tests are the already mentioned verification condition in terms of concrete variables expressing *s (set of) start states has no successor in a (set of) target states*, we had chosen to test only start states which have been found reachable and only target sets of states of the form *predicate p_i is true* or *predicate p_i is false.*

This method is meant for building a global control abstraction of the system, where the predicates on which the abstraction is based, is a relatively small set of predicates (in our experiments [GS97] we went up to 25 predicates, that is abstract variables), representing guards or their literals. The set of guards of a system in which coordination is done by global variables, are often havily interdependent, the obtained graphs are irregular, and the set of reachable abstract states is small with respect to the overall set of abstract states (for example not more than 1000 reachable states with 25 abstract variables). Only in this context, the simulataneous computation of the set of gloabally reachable states is useful.

In [BLO98a] we have presented a much more compositional method, in the context where the set of conrete variables can be partitioned into a number of small sets of variables, abstracted independently. Moreover we don't restrict ourselves to boolean abstract variables, as the type of abstraction we yield are more of the kind "data abstraction". We present here in more details the second method, as its method for the computation of the abstract transition relation is more evolved. Here, it could also save time to construct only successors of reachable configurations, but in the context of communication through shared variables, local invariants are not necessarily preserved by the global system.

Partitioning the abstract variables A simple and practical way to enhance the elimination method consists of partitioning the set A of abstract variables into subsets $A_1, \cdots, A_m$ and considering the effect of the abstraction of a concrete transition ρ on each set A_i separately. Let us consider this in more detail. We assume that the considered abstraction relation α is a function and we denote by α_i the projection of α onto A_i, i.e. $\alpha_i(s) = \alpha(s)_{|A_i}$, for every concrete state s and $i \leq m$. Then, we have the following lemma:

Lemma 2. *Let $\varphi^c \subseteq \Sigma^c$. For $i = 1, \cdots, m$, let $S_i^a = (A_i, I_i^a, R_i^a)$ be an abstract transition system and let $S^a = \bigotimes_{i \leq m} S_i^a$ (see section 2 for the def. of $\bigotimes$).*

Then, $S^c \sqsubseteq_{\alpha_i}^{\varphi^c} S_i^a$, for $i = 1, \cdots, m$ iff $S^c \sqsubseteq_{\alpha}^{\varphi^c} S^a$. □

Proof. We only consider the implication from left to right and show that for every state $s, s' \in \Sigma^c$ with $s \in \varphi^c$, if $(s, s') \in R^c$ then $(\alpha(s), \alpha(s')) \in R^a$. Therefore, assume that $S^c \sqsubseteq_{\alpha_i}^{\varphi^c} S_i^a$, for $i = 1, \cdots, m$. Consider states $s, s' \in \Sigma^c$ with $s \in \varphi^c$ and $(s, s') \in R^c$. From $S^c \sqsubseteq_{\alpha_i}^{\varphi^c} S_i^a$ and $\alpha_i(s) = \alpha(s)_{|A_i}$, for $i = 1, \cdots, m$, we obtain $(\alpha_i(s), \alpha_i(s')) \in R_i^a$. Hence, by definition of R^a, $(\alpha(s), \alpha(s')) \in R^a$.

For the truth of this statement it suffices to have one of the assumptions that α is a function or $A_1, \cdots, A_m$ is a partition of A. It is, however, in general unsound if we do not have either of these assumptions. The lemma suggests to partition the set of abstract variables and consider each element of the partitioning in isolation. If we have n boolean abstract variables and partition them into two sets of n_1 and n_2 elements then, when applying the elimination method, we have to check for $2^{2n_1} + 2^{2n_2}$ validities instead of for $2^{2(n_1+n_2)}$ validities.

Now, the question arises whether an abstract system that is computed using a partitioning is at most non-deterministic as the system computed without using the partitioning, i.e. whether $\alpha(S^c) = \bigotimes_{i \leq m} \alpha_i(S^c)$ holds. The answer is that in general $\bigotimes_{i \leq m} \alpha_i(S^c)$ has more transitions than $\alpha(S^c)$, because there might be dependencies between the α_i's which are not taken into account during the process of computing $\alpha_i(S^c)$. We can, however, state the following lemma:

Lemma 3. *Assume that the set C of concrete variables can be partitioned into sets $C_1, \cdots C_m$ such that R^c can be written in the form $R_1^c \otimes \cdots \otimes R_m^c$, where each R_i^c is a relation on states over C_i. Assume also that each α_i can be considered as a function of C_i. Then, $\alpha(S^c) = \bigotimes_{i \leq m} \alpha_i(S^c)$.* □

It is often the case, however, that most of the dependencies between the α_i's are captured as an invariant of S^c, which can then be used during the computation of the abstract system.

Given two partitions $P = \{A_1, \cdots, A_m\}$ and $P' = \{A'_1, \cdots, A'_{m'}\}$ of A, we say that P is finer than P', if for every $i \leq m$ there is $j \leq m'$ such that $A_i \subseteq A_{j'}$. In this case, we write $P \leq P'$. The following lemma states that, in general, finer partitions lead to more transitions in the abstract system.

Lemma 4. *Let P and P' be partitions of A such that $P \leq P'$. Moreover, for every $j \leq m'$, let α'_j denote the projection of α on A'_j, i.e., $\alpha'_j(s) = \alpha(s)_{|A'_j}$, for every concrete state s. Then, $\bigotimes_{j \leq m'} \alpha'_j(S^c) \sqsubseteq_{Id_A} \bigotimes_{i \leq m} \alpha_i(S^c)$, where Id_A is the identity on the abstract states.* $\Box$

The proof of the lemma uses the fact that for every concrete state s, every abstract state s^a, every $i \leq m$, and every $j \leq m'$ such that $A_i \subseteq A'_j$, if $s^a{}_{|A'_j} = \alpha'_j(s)$ then $s^a{}_{|A_i} = \alpha_i(s)$.

Using substitutions In many cases we do not need to apply the elimination method to compute the abstraction of a transition τ but we can achieve this using syntactic substitutions. To explain how this goes we assume in this section that transitions are given as guarded simultaneous assignments of the form $g(\boldsymbol{c}) \rightarrow \boldsymbol{c} := \boldsymbol{e}$. Thus, consider a transition τ and an abstraction function α given by $\bigwedge_{a \in A} a \equiv e_a$, i.e., $\alpha(s)(a) = s(e_a)$, for every concrete state s, where $s(e_a)$ denotes the evaluation of e_a in s. To compute the abstraction of τ one can proceed as follows:

1. Determine a list $c_1 = v_1, \cdots, c_n = v_n$ of equations, where $c_i \in C$ and v_i is a constant, such that $c_i = v_i$ follows from the guard g.
2. Substitute each variable c_i with v_i in $\boldsymbol{e}$ obtaining a new concrete transition τ' with $\tau' \equiv g(\boldsymbol{c}) \rightarrow \boldsymbol{c} := \boldsymbol{e'}$ and $\boldsymbol{e'} = \boldsymbol{e}\,[v_1/c_1, \cdots, v_n/c_n]$.
3. Let $\beta(a)$ be $e_a[\boldsymbol{e'}/\boldsymbol{c}]$, for each $a \in A$.
4. We say that an abstract variable a is determined by β, if one of the following conditions is satisfied:
   (a) there is a variable-free expression e such that for every concrete state s, $s(\beta(a)) = s(e)$ holds, or
   (b) there is an abstract variable $\bar{a}$ such that $\beta(a)$ $e_{\bar{a}}$ are syntactically equal.
   Let $\gamma(a)$ be e in the first case and $\bar{a}$ in the second.
5. If all variables in A are determined by β then the transition with guard $\alpha(g)$ and which assigns $\gamma(a)$ to every abstract variable a is an abstraction of τ with respect to α.

To see that 5.) is true notice that transitions τ and τ' are semantically equivalent and that for all concrete states s and s' if $(s, s') \in \tau'$ then $\alpha(s')(a) = \alpha(s)(\gamma(a))$, for every $a \in A$.

Thus, in case all abstract variables are determined by β the complete abstraction of τ is determined by substitutions without need for the elimination method. However, in general we can apply the procedure described above followed by the elimination method to determine the assignments to the abstract variables which are not determined by β.

Example 1. To illustrate how we can use syntactic substitution to compute the abstraction of a concrete transition, we 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 $a_1 \equiv (y_1 = 0) \wedge a_2 \equiv (y_2 = 0) \wedge a_3 \equiv (y_1 \leq y_2) \wedge pc_1^a \equiv pc_1 \wedge pc_2^a \equiv pc_2$.

Let us consider transition τ_1 of S_1 and apply step 1.) to 5.) to it. It can be easily seen that we obtain $\beta(pc_1^a) \equiv l_{12}$, $\beta(a_1) \equiv 1 + y_2 = 0$, $\beta(a_3) \equiv 1 + y_2 \leq y_2$, $\beta(pc_2^a) \equiv pc_2$, and $\beta(a_2) \equiv y_2 = 0$. Moreover, $\alpha(pc_1 = l_{11}) \equiv pc_1^a = l_{11}$. Since $1 + y_2 = 0$ and $1 + y_2 \leq y_2$ are equivalent to `false`, we obtain as abstract transition $pc_1^a = l_{11} \rightarrow a_1 := \texttt{false}, a_3 := \texttt{false}, pc_1^a := l_{11}$.

Also the abstraction of transitions τ_2 to τ_5 are computed by substitutions. For transition τ_6, the assignment to variables a_2 and pc_2^a are determined by substitutions, while we need the elimination method to determine the affectation to a_3.

5 Combining abstraction and deductive verification

In the case where the abstract system does not verify the invariant φ to be proved, several alternatives exist depending on the choice of the abstract domain:

1. if φ is exactly represented by the abstraction relation[8], then either the abstraction is not good enough to show invariance of φ and we have to refine it, or we did not try hard enough to prove all verification conditions and have not eliminated some of the bad transitions. The second case is not very interesting and the first case is considered in section 5 where we compute refinements of abstractions.
2. if φ is *not* exactly represented by the abstraction relation, for example because some of the variables occuring in it are totally abstracted away in the abstract domain. Then, the information contained in the constructed abstract system, can be used when trying to prove by any backward verification method, that is symbolic model checking or deductive verification. In both cases, the invariant $\alpha(Reach(S^a))$ obtained from the set of abstract reachable states can be used
   - in symbolic model-checking, we compute a chain of decreasing approximations of a deductive invariant included in φ. The knowlwdge that $\alpha(Reach(S^a))$ is an invariant, allows us to start with the smaller set $\varphi \wedge \alpha(Reach(S^a))$ and possibly to faster convergence

[8] that is when $\alpha^{-1}(\alpha(\varphi)) = \varphi$

  - similarly, if φ contains infinite domain variables and we apply deductive verification, the applied proof rule can exploit the fact that $\alpha(\mathit{Reach}(S^a))$ is an established invariant.

3. finally, we can not only use the invariant represented by the abstract system but also the abstract transition relation itself, that is all the proofs of non reachability which have been made to obtain a given abstraction. This is specially interesting when the abstract transition relation where non trivial to obtain. The invariance proof tries to establish that

   the successors of all states which satisfy φ and $\alpha(\mathit{Reach}(S^a))$, must satisfy also φ.

   For any transition relation τ and given start set X, the abstract transition relation allows to compute an overapproximation $\textsc{post}(\tau)(X)$ of states reachable from X by τ. THis allows to simplify the verification condition above to

   the successors of all states which satisfy φ and $\alpha(\mathit{Reach}(S^a))$, must satisfy also φrestricted to the subset $\textsc{post}(\tau)(X)$.

   We will use the equivalent formulation in terms of preconditions.

5.1 Proof rules exploiting the existence of an abstract transition system

First, we exploit the invariant represented by an abstract transition system.

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

While Theorem 2 allows us to deduce $S \models \Box\varphi^c$ in case $S^a \models \Box\varphi^a$, it does not tell us whether it is possible to take advantage from S^a in case $S^a \not\models \Box\varphi^a$. Rule (Inv-Uni) (see Fig. 6), shows how the invariant represented by the abstract transition system can be exploited. Indeed, the proof rule shows how concretizations of invariants of the abstract system can be used to prove that the predicate φ^c is preserved by the transition relation of S^c. In fact, these concretizations are used to strengthen the inductive hypothesis in the third premise of the rule (Inv-Uni).

$$\boxed{\begin{array}{l} S^c \sqsubseteq_{\alpha}^{\varphi^c} S^a \\ \alpha^{-1}(\mathit{Reach}(S^a)) \Rightarrow Q \\ Q \wedge \varphi^c \wedge \varphi \Rightarrow \textsc{wp}[R^c](\varphi^c \wedge \varphi) \\ I^c \Rightarrow \varphi^c \\ \hline S^c \models \Box(\varphi^c \wedge \varphi) \end{array}}$$

Fig. 6. Proof rule (Inv-Uni).

Theorem 3. *The proof Rule (Inv-Uni) (see Figure 6) is sound and complete.*

Proof. Let us first show soundness. Let S^c and S^a be transition systems such that

– (P1) $S^c \sqsubseteq_{\alpha}^{\varphi^c} S^a$,
– (P2) $\alpha^{-1}(Reach(S^a)) \Rightarrow Q$,
– (P3) $Q \wedge \varphi^c \Rightarrow \text{WP}[R^c](\varphi^c \wedge \varphi)$, and
– (P4) $I^c \Rightarrow \varphi$.

Let $s_0, \cdots, s_n$ be a computation of S^c. We prove by induction on n that $s_n \in \varphi^c \cap \varphi$. Now, since (P1) implies that all initial states of S^c satisfy φ^c, we have $s_0 \in \varphi^c \cap \varphi$. Moreover, from (P1) and (P2), we have $s_{n-1} \in \varphi^{R^A}$, and hence by induction hypothesis, $s_{n-1} \in \varphi^{R^A} \cap \varphi^c \cap \varphi$. From (P3), we get $s_n \in \varphi^c \cap \varphi$. Completeness is obtained from the fact that Rule (Inv-Uni) is a generalization of Rule (Inv) and thus at least as complete as it. □

Rule (Inv) can be easily derived from Rule (Inv-Uni), that is, Rule (Inv) is an instance of Rule (Inv-Uni). To do so, we choose Q equal the auxiliary invariant φ of rule (Inv), and build an abstract transition system S_Q represented in Figure 5.1 with two states, denoted by $[Q]$ and $[\neg Q]$, where $[Q]$ is initial. As Q is an invariant, we have $S^c \sqsubseteq_{\alpha}^{\varphi^c} S^a$ for $\varphi^c = \Sigma^c$ and α the total relation defined by $(s, [Q]) \in \alpha$ iff $s \in Q$, and we can therefore extend rule (Inv) to an instance of rule (Inv-Uni).

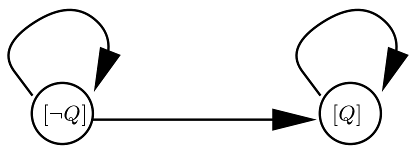

Fig. 7. Transition system S_Q

The rule (Inv-Uni) is stronger than rule (Inv) as it adds the fact that only the set $Q = \alpha(Reach(S^a))$ is indeed reachable, and thus only successors of states in Q need to be tested.

But we can also reuse the "proof" of the "invariance of Q", that is the fact that for every transition τ_i $Q \Rightarrow \text{WP}(\tau_i)(Q)$ which is of the form $Q \wedge \tau_i \Rightarrow Q'$. Indeed, this assertion is a tautology and it has been proved during the construction of the abstract system. Thus, conjoining it with any premiss in a proof rule, does *not* strengthen the proof rule, strictly speaking, but it summarizes a proof done earlier, may be with an important effort.

Now we can reuse the information contained in the abstract transition system at any level of granularity. For any $X^a \subseteq Reach(S^a)$, we can determine easily an abstract set Y^a such that $post(\tau_i)(X^a) \subseteq Y^a$ or equivalently $X^a \subseteq \textsc{wp}(\tau_i)(Y^a)$. Posing $X = \alpha^{-1}(X^a)$ and $Y = \alpha^{-1}(Y^a)$, we get tautologies of the form $X \Rightarrow \textsc{wp}(\tau_i)(Y)$ or $X \wedge \tau_i \Rightarrow Y'$.

In order to prove

$$Q \wedge \varphi^c \wedge \varphi \Rightarrow \textsc{wp}(\tau_i)(\varphi^c \wedge \varphi)$$

as required in rule (Inv-Uni), supposing that X^a is the set $guard(\tau_i) \wedge Reach(S^A)$, we can replace this assertion by

$$(X \Rightarrow \textsc{wp}(\tau_i)(Y)) \Rightarrow Q \wedge \varphi^c \wedge \varphi \Rightarrow \textsc{wp}(\tau_i)(\varphi^c \wedge \varphi)$$

which can be simplified to

$$\varphi^c \wedge \varphi \wedge X \wedge \textsc{wp}(\tau_i)(Y) \Rightarrow \textsc{wp}(\tau_i)(\varphi^c \wedge \varphi)$$

or

$$\varphi^c \wedge \varphi \wedge X \wedge \tau_i \wedge Y' \Rightarrow \varphi^{c\prime} \wedge \varphi'$$

In practice, this means that one needs not to prove again that Y' can be assumed, and Y' can often allow to deduce $\varphi^{c\prime} \wedge \varphi'$ easily.

Instead of using X^a which represents all the abstract reachable states in which tau_i is enabled, one can partion X into $\cup X_i^a$, which allows also to reduce the corresponding sets $Y_i^a = \textsc{post}(\tau_i)(X_i^a)$[9]. We will than enforce the above assertion to be proved by the tautaulogy $\cap_i(X_i \Rightarrow \textsc{wp}(\tau_i)(Y_i))$ or $\cap_i(X_i \wedge \tau_i \Rightarrow Y_i')$.

Notice that we do not get a new proof rule, as formally we just enforce some of the assertions with *true* which doesn't change anything from the point of view of validity. Nevertheless, these tautologies can greatly simplify the proofs. In all cases where the construction of the abstract transition relation is very simple, the obtained information is not very "precious" and adding these tautologies can be more an overhead than a help.

5.2 Concretizing BDD's

In order to apply the Rule (Inv-Uni), we need

1. a efficient representation of the set $Reach(S^a)$ of abstract reachable states and
2. to transform it into a finite representation of its concretisation, that is, $\alpha^{-1}(Reach(S^a))$.

The second step is straighforward, as given an expression representing $Reach(S^a)$, a representation of $\alpha^{-1}(Reach(S^a))$ is obtained by simple substitution of each abstract variable by the concrete expression it stands for (the definition of α. In

[9] which need not to represent a partition of Y^a; notice that $\cup X_i^a$ is an interesting partition if $\cup Y_i^a$ is a little overlapping as possible

this section, we give a procedure for computing an efficient representation of the set $\mathit{Reach}(S)^a$.

As the set of abstract states is finite, finiteness of the representation of $\mathit{Reach}(S)^a$ is not a problem, and as application of α^{-1} the substitution, the finiteness of the representation of the set of concrete states is also not a problem.

Invariant verification of the abstract system, is best done by means of a model-checker. Enumerative model-checker like CADP [] or IF [] create an explicite set of concrete states and transitions for which one can extract the required representation of the corresponding concrete states as a disjunction, but this is effecient only if the set of states is relativelly small. Symbolic model-checkers, like SMV [] can directly compute a predicate representing the set of abstract reachable states. As these model-checkers use often BDDs, we have to :

- encode the abstract transition relation by booleans,
- extract from the BDD representation a compact expression. Such representation can, however, be unnecessarily cumbersome.

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

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

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

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

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

The basic algorithm is shown on Figure 8. It takes an BDD b and the list of state variables (not necessarily boolean), and returns an equivalent formula. For better performance, one can use 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). Otherwize, it constructs a formula recursively on the BDD structure. For every value in the domain of the first variable [10] we restrict b to that value, remove the variable from the list, construct the formula recursively for that restricted BDD, and add

[10] We assume that the domains are always finite.

```
bdd2f($b$: BDD, $var\_list$: list of variables): formula =
**if** $b \in H$ **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** $v \in$ domain($x$) **do**
   $tmp$ := bdd2f($b|_{x=v}$, cdr($var\_list$));
   $res := res \wedge (x = v \rightarrow tmp)$;
  **end**;
  $H \rightarrow (b, res)$;
 **end if**
**return** res;
**end** bdd2f
```

Fig. 8. Basic algorithm converting BDD to a formula.

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 variables of type "formula", 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 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.

We assume that the current variable is x, and the original formula is the conjunction of implications, as described above.

1. Collect all the values $\{v_{i_0}, \ldots, v_{i_k}\}$ of x for which the conclusion F of the implication $x = v_{i_j} \rightarrow F$ is the same, and replace all such conjuncts by $x \in \{v_{i_0}, \ldots, v_{i_k}\} \rightarrow F$. Also, if the size of the set $\{v_{i_0}, \ldots, v_{i_k}\}$ contains more than the half of the domain of x, replace it by $x \notin (\text{type}(x) - \{v_{i_0}, \ldots, v_{i_k}\}) \rightarrow F$. If the set is a singleton, replace the set inclusion by equality or disequality respectively.
2. Remove conjuncts of the form TRUE and $g \rightarrow$ TRUE. Replace conjuncts of the form $g \rightarrow$ FALSE by $\neg g$. If the entire conjunction has only two conjuncts, then $(g_1 \rightarrow \text{FALSE}) \wedge (g_2 \rightarrow F)$ is simplified to $(g_2 \wedge F)$. This transformation is sound, since g_i's restrict x to disjoint ranges of values, and together they have to cover the entire domain of x, which implies $g_1 = \neg g_2$.

Splitting on Guards. The shape of the formula for the set of reachable states can be very complicated and does not always reflect the structure of the program. To simplify further, we use program guards to "slice" the set of reachable states into smaller and more manageable pieces.

The idea is very simple. Suppose, $g_1, \ldots, g_n$ are some of the program guards, and the set of reachable states is represented by a predicate R. Construct BDDs for $R \wedge \bigwedge_{1 \leq i \leq n} g_i^{p_i}$, where

$$g^p = \begin{cases} g, \text{ if } p = 0 \\ \neg g, \text{ if } p = 1. \end{cases}$$

Convert each such BDD into a formula $F_{\boldsymbol{p}}$. Then the formula for the set of reachable states can be put in the following form:

$$R \equiv \bigwedge_{\boldsymbol{p} \in \mathcal{B}^n} (\bigwedge_{1 \leq i \leq n} g_i^{p_i} \rightarrow F_{\boldsymbol{p}}).$$

The hope is that the set of reachable states is highly influenced by the guards in the program, and therefore, splitting on the guards could lead to much shorter and simpler formulas

Example 2. Let us consider the abstract system of the Bakery example :
Abstract transition system S_1^a:

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

Abstract transition system S_2^a:

$$\begin{array}{lll}
\tau_4^a : \ pc_2^a = l_{21} & \longrightarrow & a_2 := false, a_3 := true, pc_2^a := l_{22} \\
\tau_5^a : \ pc_2^a = l_{22} \wedge (a_1 \vee \neg a_3) & \longrightarrow & pc_2^a := l_{23} \\
\tau_{6_1}^a : pc_2^a = l_{23} \wedge \neg a_1 \wedge \neg a_2 \wedge \neg a_3 & \longrightarrow & a_2 := true, a_3 := false, pc_2^a := l_{21} \\
\tau_{6_2}^a : pc_2^a = l_{23} \wedge \neg a_1 \wedge \neg a_2 \wedge a_3 & \longrightarrow & a_2 := true, a_3 := false, pc_2^a := l_{21} \\
\tau_{6_3}^a : pc_2^a = l_{23} \wedge a_1 \wedge \neg a_2 \wedge a_3 & \longrightarrow & a_2 := true, a_3 := true, pc_2^a := l_{21} \\
\tau_{6_4}^a : pc_2^a = l_{23} \wedge \neg a_1 \wedge a_2 \wedge \neg a_3 & \longrightarrow & a_2 := true, pc_2^a := l_{21} \\
\tau_{6_5}^a : pc_2^a = l_{23} \wedge a_1 \wedge a_2 \wedge a_3 & \longrightarrow & a_2 := true, a_3 := true, pc_2^a := l_{21}
\end{array}$$

If we apply the basic algorithm (see Figure 8) 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:

$$(y1 = 0 \wedge y2 = 0) \Rightarrow pc1 = l11 \wedge pc1 = l21 \qquad (1)$$

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

In this example, the concrete invariant obtained by this approach is stronger than the invariant generated by the method presented in [BLY96,BL99]. 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.

5.3 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.

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 $\bigwedge_{i \leq N} \mathrm{WP}^i_\rho(P)$. Then, $\bigwedge_{i \leq N} \mathrm{WP}^i_\rho(P)$ is also a quantifier-free formula. Let $F = \{f_1, \cdots, f_m\}$ be the set of atomic formulas that appear in $\bigwedge_{i \leq N} \mathrm{WP}^i_\rho(P)$ 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 [BLO98a], we show how given a transition system S, a predicate P and an abstraction function α, we compute a system S^a such that $S \sqsubseteq^P_\alpha S^a$ 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 $\sigma^a = s^a_0 \tau^a_1 s^a_1 \cdots \tau^a_n s^a_n$ 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 5. *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 $X_{n-i-1} = Q_{n-i-1} \cap \text{PRE}_{\tau_{n-i}}(X_{n-i})$.* □

Lemma 5 suggests the procedure CouAnal given in Figure 9 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 $\sigma^a = s_0^a\tau_1^a s_1^a\cdots\tau_n^a s_n^a$
$X := \alpha^{-1}(s_n^a)$;
$i := n$;
while ($X \neq \emptyset$ and $i > 0$) do
    $Y := X$;
    $X := \text{PRE}_{\tau_i}(X) \cap \alpha^{-1}(s_{i-1}^a)$;
    $i := i - 1$
 od
if $i = 0$ and $\theta \cap X \neq \emptyset$ then return "the following is a counter-example:"
    Take any $s \in \theta \cap X \neq \emptyset$
    Let $s_0 := s, s_1 := \tau_1(s_0)\cdots, s_n := \tau_n(s_{n-1})$
    write $s_0\cdots s_n$
else return $i, Y$
fi
```

Fig. 9. Counter-example Analyzer: CouAnal

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 5, procedure CouAnal returns some $i \leq n$ and a set $Y = X_i \subseteq \Sigma$ such that $X_{i-1} = \emptyset$[11]. Now, since $X_{i-1} = \emptyset$, $Q_{i-1} \subseteq \text{WP}_{\tau_i}(\neg X_i)$ 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.

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

Speeding-up refinement of abstraction functions The simple illustrative example given in Figure 10 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.

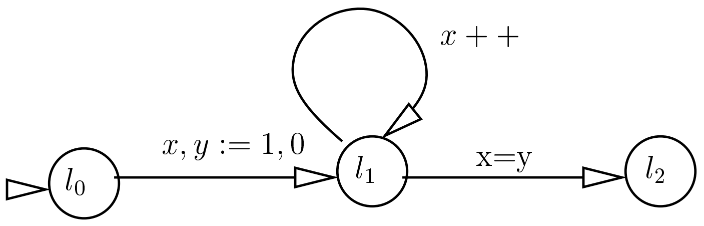

Fig. 10. Example for showing that speeding-up is needed

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 10, 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 $\text{PRE}(\tau_{i_0}, \cdots, \tau_{i_1})$ on X_{i_1}, that is, to consider $\bigvee_{i \in \omega} \text{PRE}^i(\tau_{i_0}, \cdots, \tau_{i_1})$ on X_{i_1}.

For instance, in the example of Figure 10, applying $\bigvee_{i \in \omega} \text{PRE}^i(x{+}{+})$ on $x = y$ gives 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 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 11. There are several remarks to say about procedure 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 sim-

```
Input: An abstract counter-example $\sigma^a = s_0^a \tau_1^a s_1^a \cdots \tau_n^a s_n^a$;
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
$\tau_1, \cdots, \tau_n = \tau_1, \cdots, \tau_{i_1-1} L_1, \tau_{i_1+k_1+1}, \cdots, L_m, \tau_{i_m+k_m+1}, \cdots, \tau_n$;
$X := \alpha^{-1}(s_n^a)$;
$i := n$;
$k := m$;
while ($X \neq \emptyset$ and $i > 0$) do
    $Y := X$;
    if $i = i_k$ then
        $X := \bigvee_{j \in \omega} pre^j_{L_k}(X) \cap \alpha^{-1}(s_{i-1}^a)$;
        $i := i -$ length$(L_k)$
    else $X := \text{PRE}_{\tau_i}(X) \cap \alpha^{-1}(s_{i-1}^a)$
    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. 11. Accelerated Counter-example Analyzer: AccCouAnal

ple 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.

References

[ABJN99] P.A. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson. Handling Global Conditions in Parameterized System Verification. In N. Halbwachs and D. Peled, editors, *CAV '99*, volume 1633 of *LNCS*, pages 134–145. Springer-Verlag, 1999.

[ABLS01] A. Annichini, A. Bouajjani, Y. Lakhnech, and M. Sighireanu. Analyzing fair parametric extended automata. In P. Cousot, editor, *Proceedings of the International Symposium on Static Analysis*, volume 2126 of *lncs*. springer, 2001.

[BBLS92] A. Bouajjani, S. Bensalem, C. Loiseaux, and J. Sifakis. Property preserving simulations. In *Workshop on Computer-Aided Verification (CAV), Montréal*. LNCS 630, June 1992.

[BBLS00] 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 *LNCS*. Springer-Verlag, 2000.

[BBM97] N. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.

[BFH90] 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.

[BL99] S. Bensalem and Y. Lakhnech. Automatic generation of invariants. *Formal Methods in System Design*, 15(1):75–92, July 1999.

[BLO98a] 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 *LNCS*, pages 319–331. Springer-Verlag, 1998.

[BLO98b] 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 *LNCS*, pages 505–510. Springer-Verlag, 1998.

[BLS96] S. Bensalem, Y. Lakhnech, and H. Saidi. Automatic generation of invariants. In *??*, 1996.

[BLS01] K. Baukus, Y. Lakhnech, and K. Stahl. Verification of parameterized networks. *Journal of Universal Computer Science*, 7(2), 2001.

[BLY96] 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 *LNCS*, pages 306–326. Springer-Verlag, 1996.

[BSV93] F. Balarin and A. Sangivanni-Vincentelli. An iterative approach to language containment. In *5th Workshop on Computer-Aided Verification (CAV93)*. LNCS 697, Springer Verlag, June 1993.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL*, January 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis framework. In *Proc. 6th ACM Symp. on Principle of Programming Languages*, 1979.

[CC80] P. Cousot and R. Cousot. Constructing program invariance proof methods. In *Int. Worksh. on Program Construction, Chateau Bonas*. INRIA, France, 1980.

[CDH$^+$00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from java source code. In *22nd International Conference on Software Engineering*, 2000.

[CGJ$^+$00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, LNCS, pages 154–169. Springer-Verlag, 2000.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CU98] 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.

[Dam96] D. Dams. *Abstract interpretation and partition refinement for model checking*. Phd thesis, Technical University of Eindhoven, July 1996.

[dB80] J.W. de Bakker. *Mathematical Theory of Program Cortrectness*. Prentice-Hall, NJ, 1980.

[DDP] S. Das, D. Dill, and S. Park. Experience with predicate abstraction. In *11th International Conference on Computer-Aided Verification,*.

[DF95] J. Dingel and Th. Filkorn. Model checking for infinite state systems using data abstraction, assumption-committment style reasoning and theorem proving. In *Proc. of 7th CAV 95, Liège*. LNCS 939, Springer Verlag, 1995.

[DGG93] D. Dams, O. Grumberg, and R. Gerth. Generation of reduced models for checking fragments of CTL. In *Proceedings of CAV'93, Crete (GR)*, volume 697, pages 479–490. Lecture Notes in Computer Science, 1993.

[DGG97] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions in Programming Languages and Systems*, 19(2), 1997.

[GL93a] S. Graf and C. Loiseaux. Program verification using compositional abstraction. In *TAPSOFT 93, joint conference CAAP/FASE*. LNCS 668, Springer Verlag, April 1993.

[GL93b] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In *Conference on Computer Aided Verification CAV 93, Heraklion Crete*. LNCS 697, Springer Verlag, 1993.

[Gra94] S. Graf. Verification of a distributed cache memory by using abstractions. In *Conference on Computer Aided Verification CAV'94, Stanford*. LNCS 818, Springer Verlag, June 1994. a largely improved and extended version appeared in Distributed Computing.

[Gra99] S. Graf. Characterization of a sequentially consistent memory and verification of a cache memory by abstraction. *Distributed Computing*, 12, 1999. accepted for publication since 1995.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with pvs. In *Conference on Computer Aided Verification CAV'97, Haifa*, volume 1254 of *LNCS*, June 1997.

[HDLS98] John Hatcliff, Matthew B. Dwyer, Shawn Laubach, and David Schmidt. Staging static analyses using abstraction-based program specialization. In *Principles of Declarative Programming: 10th International Symposium, PLILP'98*, LNCS 1490, 1998.

[HHK95] 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.

[HPR94] N. Halbwachs, Y.-E. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In *International Static Analysis Symposium, SAS'94*, Namur (Belgium), September 1994.

[HS96] K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Proceedings of Formal Methods in Europe'96*, 1996.

[JHR99] B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numericalproperties. In *Static Analysis Symposium, SAS'99, Venezia (Italy)*, September 1999.

[Kur89] R.P. Kurshan. Analysis of discrete event coordination. In *REX Workshop on Stepwise Refinement of Distributed Systems, Mook*. LNCS 430, Springer Verlag, 1989.

[Kur94] R.P. Kurshan. *Computer-Aided Verification of Coordinating processes, the automata theoretic approach*. Princeton Series in Computer Science. Princeton University Press, 1994.

[Lar89] K. Larsen. Modal specifications. In *Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*. LNCS 407, Springer Verlag, 1989.

[LBBO01] Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *TACAS 2001*, volume 2031 of *lncs*, 2001.

[LGS$^+$95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design, Vol 6, Iss 1, January 1995*, 1995.

[LS97] David Lesens and Hassen Saïdi. Automatic verification of parameterized networks of processes by abstraction. In *Proceedings of the 2nd International Workshop on the Verification of Infinite State Systems (INFINITY'97, Bologna, Italy)*, July 1997.

[LSW95] Kim G. Larsen, Bernhard Steffen, and Carsten Weise. A constraint oriented proof methodology. In *Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'95*, pages 17–40. LNCS vol. 1019, 1995.

[LY92] 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.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Proc. Second Int. Joint Conf. on Artificial Intelligence*, pages 481–489. BCS, 1971.

[Mil80] R. Milner. A calculus of communication systems. In *LNCS 92*. Springer Verlag, 1980.

[MN95] O. Müller and T. Nipkow. Combining model checking and deduction for I/O-automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *LNCS*, 1995.

[MP95a] Z. Manna and A. Pnueli. *The temporal Logic of reactive and concurrent systems, Volume 2: Verification, 1995*. Springer Verlag, 1995.

[MP95b] Z. Manna and A. Pnueli. Verification of parameterized programs. In E. Börger, editor, *Specification and Validation Methods*, pages 167–230. Oxford University Press, Oxford, 1995.

[NK00] K. S. Namjoshi and R .P. Kurshan. Syntactic program transformations for automatic abstraction. In *Computer Aided Verification*, LNCS, pages 435–449. Springer-Verlag, 2000.

[Par81] D. Park. Concurrency and automata on infinite sequences. In *5th GI-Conference on Theoretical Computer Science*, number 104. LNCS, 1981.

[SDB96] J. X. Su, D. L. Dill, and C. Barrett. Automatic generation of invariants in processor verification. In *FMCAD '96*, volume 1166 of *LNCS*, 1996.

[SRSS96] S.Owre, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking and model-checking. In *CAV'96*, volume 1102 of *LNCS*, 1196.

[SUM96] 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 *LNCS*, pages 208–219. Springer-Verlag, 1996.