Construction of abstract state graphs with PVS

Susanne Graf and Hassen Saidi
VERIMAG
Centre Equation, 2, Avenue de la Vignate, 38610 Grenoble-Gières
{graf,saidi}@imag.fr
contact author : Susanne Graf

appeared in the Proceedings of CAV'97, LNCS 1254

Abstract: We describe in this paper a method based on abstract interpretation which, from a theoretical point of view, is similar to the splitting methods proposed in [DGG93, Dam96] but the weaker abstract transition relation we use, allows us to construct automatically abstract state graphs paying a reasonable price.

We consider a particular set of abstract states: the set of the monomials on a set of state predicates $\varphi_1, ..., \varphi_\ell$. The successor of an abstract state m for a transition τ of the program is the *least* monomial satisfied by all successors via τ of concrete states satisfying m. This successor m' can be determined exactly if for each predicate φ_i it can be determined if φ_i or $\neg\varphi_i$ is a postcondition of m for τ. In order to do this, we use the PVS theorem prover [SOR93] and our PVS-interface defined in [GS96]. If the tactic used for the proof of the verification conditions is not powerful enough, only an upper approximation of the abstract successor m is constructed.

This allows us to compute upper approximations of the set of reachable states which is sufficient for the verification of invariants. Also, for almost the same price, an abstract *state graph* can be constructed: the expensive part of the algorithm is the computation of an abstract successor as it requires several validity checks. Therefore, only relatively small state graphs can be constructed and the additional cost for the storage of the transition relation is almost negligible. An abstract state graph can be used for the verification of any property expressible as a temporal logic formula without existential quantification over paths, due to the results on property preservation [CGL94, LGS$^+$95] using a model checker.

An abstract state graph represents also a relatively precise global control graph of the system (the guards of the system are used for the construction of the abstract state graph) which can be used for a backwards verification of invariants as described in [GS96]. A global control graph allows us to generate much stronger structural invariants using the tool described in [**?**, BBC$^+$96] than the initial presentation as a parallel composition of processes. In the case that the control of the system is completely independent of the data part, a control graph is obtained much easier by partial evaluation as proposed in [HGD95]; our method allows to mechanize the construction of the global control graph also if some control variables depend on data, as for example in the protocol studied in Section 4.

We have implemented a particular case of this method in our tool [GS96] where only successors of complete monomials are constructed: if a successor is not a complete monomial, it is split into its complete monomials. We have also interfaced our tool

with the state space analysis tools ALDÉBARAN [FGK+96].

We have verified a bounded retransmission protocol developed by Philips which has been proven correct before using the Coq theorem prover [GvdP93, HSV94] and on PVS [HS96]. But for all these proofs powerful auxiliary invariants had to be given by the user. Using our tool, the correctness of this protocol can be proved almost without user intervention.

A Web page, concerning our experience with the verification of the Bounded Retransmission Protocol, can be found at the address

`http:://www.imag.fr/VERIMAG/PEOPLE/Hassen.Saidi/BRP`

Construction of abstract state graphs of infinite systems with PVS

Susanne Graf and Hassen Saidi
VERIMAG[1]

Abstract

In this paper, we propose a method for the automatic construction of an abstract state graph of an infinite state system using the Pvs theorem prover.

Given a system and a partition of the state space induced by n predicates $\varphi_1, ..., \varphi_n$ on the concrete program variables which defines an abstract state space, we construct an abstract state graph, starting in the abstract initial state. The possible successors of a state are computed using the Pvs theorem prover by verifying for each index i if φ_i or $\neg\varphi_i$ is a postcondition of it. This allows an abstract state space exploration for arbitrary systems.

Using this method, we have automatically verified a bounded retransmission protocol which cannot be proved using backward analysis without providing strong auxiliary invariants.

keywords: *abstract interpretation, state graph exploration, theorem proving*

1 Introduction

It is now widely accepted that abstraction techniques are useful, and even necessary for a successful verification [Kur94, CGL94, GL93, LGS$^+$95, Gra95, Dam96] [DF95]. However, in case that the system has an infinite state space, it is difficult to mechanize the construction of an abstract system or state graph. In [GL93, KDG95] tools are described which, given a system (with variables on finite domains), a set of abstract (boolean) variables, and an abstraction relation relating the concrete and the abstract variables, construct automatically a corresponding abstract system, which then may be analyzed by any model-checker. For the analysis of real-time and particular hybrid systems, there exist tools for the abstract analysis by means of abstract interpretation methods based on the use of polyhedra [HH95, DOTY96, HPR94] but they are restricted to systems with linear assignments. In [Gra95, DF95], methods for the construction of abstract state graphs of more general infinite state systems are proposed, but they require an important amount of user intervention, as it is necessary to give for any atomic operation of the system a corresponding abstract operation which must be proven to be correct. The definition of abstract operations and the corresponding correctness proofs are in general rather time consuming, and in case of modification of the system or non satisfaction of the desired properties on the abstract system, some of them need to be modified.

We describe a method based on abstract interpretation which, from a theoretical point of view, is similar to the splitting method proposed in [DGG93, Dam96] but the weaker abstract transition relation we use, allows us to construct automatically abstract state graphs paying a reasonable price.

[1]Centre Equation, 2, Avenue de la Vignate, 38610 Grenoble-Gières{graf,saidi}@imag.fr

We consider a particular set of abstract states: the set of valuations of a tuple of boolean variables $B_1, ..., B_\ell$ representing a tuple of state predicates $\varphi_1, ..., \varphi_\ell$. As abstraction introduces non-determinism a successor of an abstract state via an abstract transition may not be a single abstract state, but a set of abstract states represented by a boolean expression $\mathsf{exp}^A(B_1, ..., B_\ell)$. We consider only particular successors which are representable by monomials on $B_1, ..., B_\ell$: the successor of a (set of) abstract state(s) $\mathsf{exp}^A(B_1, ..., B_\ell)$ for a transition τ of the program is the *least* monomial representing all successors via τ of concrete states represented by $\mathsf{exp}^A(\varphi_1, ..., \varphi_\ell)$. This successor can be determined exactly if for each predicate φ_i it can be determined if φ_i or $\neg\varphi_i$ is a postcondition of $\mathsf{exp}^A(\varphi_1, ..., \varphi_\ell)$ for τ. In order to do this, we use the PVS theorem prover [SOR93] and our PVS-interface defined in [GS96]. If the tactic used for the proof of the verification conditions is not powerful enough, an upper approximation of the abstract successor is constructed.

This allows us to compute upper approximations of the set of reachable states which is sufficient for the verification of invariants. Also, for almost the same price, an abstract *state graph* can be constructed: the expensive part of the algorithm is the computation of an abstract successor as it requires several validity checks. Therefore, only relatively small state graphs can be constructed and the additional cost for the storage of the transition relation is almost negligible. An abstract state graph can be used for the verification of any property expressible as a temporal logic formula without existential quantification over paths, due to the results on property preservation [CGL94, LGS+95] using a model checker.

An abstract state graph represents also a relatively precise global control graph of the system (the guards of the system are used for the construction of the abstract state graph) which can be used for a backwards verification of invariants as described e.g. in [GS96, BLO98b]. A global control graph allows us to obtain much stronger structural invariants using the tool described in [BBC+96, BL98] than the initial presentation as a parallel composition of processes.

We have implemented a particular case of this method in the tool described in [GS96] where the successors of canonical monomials are constructed. We have also interfaced the tool with the state space analysis tool ALDÉBARAN [FGK+96].

We have verified a bounded retransmission protocol developed by Philips which has already been proven correct before using theorem provers [GvdP93] [HSV94, HS96]. But for all these proofs powerful auxiliary invariants had to be given by the user. Using our tool, this protocol can be verified without user intervention.

2 Construction of abstract state graphs

2.1 Preliminary definitions

We consider systems which are parallel compositions of processes of the following form, where we consider parallel composition by interleaving and synchronization by shared variables as in Unity [CM88]:

Definition 2.1 (Processes)

Name :	P
Declarations :	$x_1 : T_1, ..., x_n : T_n$
Transitions :	$\tau_1, ..., \tau_p$
Initial States :	*init*

where P is a name, x_i are variables of type T_i (which may be any type definable in PVS). The list of variables declared in one process indicates which variables are (intended to be) used in this process, but in fact all variable declarations are *global*. Each transition τ_i is a guarded assignment of the form

$$g_i(\overline{x}) \longmapsto \quad \overline{x} := ass_i(\overline{x}) \tag{1}$$

where $g_i(\overline{x})$ is a boolean PVS-expression and $ass_i(\overline{x})$ a tuple of PVS-expressions ass_{i_j} of type T_j. In order the structural invariant generation to be effective, each process has a finite domain *control* variable, which is always tested for and assigned with a constant expression.

Semantics: As parallel composition is as in Unity, the state graph associated with a parallel composition of processes is the state graph associated with a *single* process having, as variables the union of the variables of all processes, as transitions the union of the transitions of all processes, and as initial predicate the intersection of the initial predicates of all processes. That means, parallel composition is only useful for better readability and for the generation of structural invariants [BL98]. Therefore, we consider here without loss of generality only systems with a single process P. P defines a state graph $\mathcal{S}_P = (Q_P, R_P, I_P)$, where

- $Q_P = T_1 \times ... \times T_n$
- $R_P = \bigcup_{i=1}^{p} \tau_i$, where $\tau_i(q) = \begin{cases} \perp & \text{if } g_i(q) \equiv \mathit{false} \\ ass_i(q) & \mathit{otherwise} \end{cases}$
  denotes also the (partial) transition function associated with transition τ_i.
- $I_P = \{q \mid \mathit{init}(q) \equiv \mathit{true}\}$ is the set of initial states.

Predicate transformers: Let us first recall briefly the notion of predicate transformers associated with relations and their well-known characterization for guarded command programs. In the sequel, we always consider sets of states to be represented by predicates φ (hence the name predicate transformer).

Definition 2.2 (predicate transformers) *Let R be a binary relation on a set Q and $\varphi \in \mathcal{P}(Q)$ represent a subset of Q. Then,*

- $\mathbf{post}[R](\varphi) = \exists q' \,.\, R(q', q) \wedge \varphi(q')$
- $\widetilde{\mathbf{pre}}[R](\varphi) = \forall q' \,.\, (R(q, q') \Rightarrow \varphi(q'))$

$\mathbf{post}[R](\varphi)$ defines the set of successors of φ by R (strongest postcondition). $\widetilde{\mathbf{pre}}[R](\varphi)$ represents the largest set of states such that all its successors satisfy φ (weakest precondition). Preconditions for guarded commands τ_i of the form (1) can be expressed without quantifiers:

$$\widetilde{\mathbf{pre}}[\tau_i](\varphi) \;\equiv\; (g_i(\overline{x}) \Rightarrow \varphi[ass_i(\overline{x})/\overline{x}]) \tag{2.1}$$

whereas the quantifiers in the postconditions can in general not systematically be eliminated. For this reason, symbolic forward analysis is more difficult than backward analysis and it is difficult to compute effectively an abstract state graph by forward analysis: the successor(s) of an abstract state (representing a predicate φ on concrete states) represents an upper approximation of the postcondition of φ by the concrete transition relation.

These predicate transformers have many interesting properties (see for example [Sif82]), but here we need only the following:

$$\mathbf{post}[R](\varphi) \Rightarrow \varphi' \;\text{ iff }\; \varphi \Rightarrow \widetilde{\mathbf{pre}}[R](\varphi') \tag{2.2}$$

Abstract semantics of programs

All the results presented in this section are an application of abstract interpretation [CC77]. However, we do not suppose the reader to be familiar with abstract interpretation. Here, we limit ourselves to abstractions representing supersets of the concrete system (in fact, its execution sequences) as we are interested in verifying properties such that whenever a system satisfies property p, then also any system with a smaller transition relation (and therefore set of reachable states).

Definition 2.3 (abstract state graphs) *Let $\mathcal{S} = (Q, R_P = \cup\tau_i, I)$ be the state graph of a program, $\mathbf{Q}^A$ a lattice of abstract states and $(\alpha : \mathcal{P}(Q) \mapsto \mathbf{Q}^A, \gamma : \mathbf{Q}^A \mapsto \mathcal{P}(Q))$ a Galois connection*[2]. *$\mathcal{S}^A = (\mathbf{Q}^A, \cup\tau_i^A, I^A)$ is an* abstraction *of $\mathcal{S}$ iff*

- $I \subseteq \gamma(I^A)$
- $\forall i\ \forall \mathrm{q}^A \in \mathbf{Q}^A\ .\ \mathbf{post}[\tau_i](\gamma(\mathrm{q}^A)) \subseteq \gamma(\tau_i^A(\mathrm{q}^A))$

The *abstraction function* α associates with any set of concrete states a corresponding abstract state (the abstract state space is a lattice where larger abstract states represent larger sets of concrete states). The *concretization function* γ associates with every abstract state the set of concrete state that it represents. The above definition simply expresses that the abstract initial state represents (at least) all concrete initial states, and the successor of any abstract state q^A by some abstract transition represents all successors of the set of concrete states represented by q^A by the corresponding concrete transition. Thus, every concrete execution sequence is represented by at least one abstract one. Intuitively, the smaller the represented superset of execution paths is, the more properties are satisfied on the abstract system. The reason why the abstract lattice is not necessarily of the form 2^E where E is a set of abstract states, is that for efficiency reasons one does not want to consider any such set of abstract states, but only some of them. For example, an often used rather coarse approximation is to approximate every non singleton set of abstract states with the top element of the lattice representing the set of all (abstract and therefore also concrete) states.

[2] a Galois connection is a pair of functions (α, γ) satisfying $\alpha(\gamma(\mathrm{q}^A)) = \mathrm{q}^A$ and $\varphi \Rightarrow \gamma(\alpha(\varphi))$. Given γ, α is implicitly defined by $\alpha(\varphi) = \sqcap\{\mathrm{q}^A \in \mathbf{Q}^A \,|\, \varphi \Rightarrow \gamma(\mathrm{q}^A)\}$.

2.2 A particular abstraction scheme

Choice of an abstract state lattice: We consider an abstract state lattice $\mathbf{Q}^A$ induced by a set of ℓ *predicates* $\{\varphi_1, ..., \varphi_\ell\}$ on the variables of the concrete program P[3]. We choose as abstract state space the set of predicates on ℓ boolean variables $B_1, ..., B_\ell$, where each variable B_i represents all concrete states satisfying the predicate φ_i. That means that the set of concrete states represented by any element of the abstract lattice can easily be computed by substituting each occurrence of an abstract variable B_i by the concrete predicate φ_i which it represents:

$$\gamma(\mathsf{exp}^A(B_1, ..., B_\ell)) = \mathsf{exp}^A[\overline{\varphi}/\overline{B}]$$

whereas the implicitly defined abstraction function

$$\alpha(\varphi) = \bigwedge\{\mathsf{exp}^A(B_1, ..., B_\ell) \mid \varphi \Rightarrow \mathsf{exp}^A[\overline{\varphi}/\overline{B}]\}$$

can in general not easily be computed. For this reason, we use an upper approximation of the function which is less expensive to compute and which yields only particular elements of the abstract lattice which are the monomials on $B_1, ..., B_\ell$[4]:

$$\alpha'(\varphi) = \bigwedge_{i=1}^{\ell}\{B_i \mid \varphi \Rightarrow \varphi_i\}$$

Notice that the set $\mathcal{M}$ of monomials on abstract *boolean variables* $B_1, ..., B_\ell$ forms a complete lattice and (α', γ) is a Gaulois connection from the set of concrete predicates to $\mathcal{M}$. The set of atoms of the lattice is the set of the 2^ℓ *canonical* monomials (representing abstract states).

Abstract transitions: For each concrete transition τ_i of the program, we define an abstract transition function τ_i^A associating with any set of abstract states exp^A a set of abstract states representing all successors of the concrete states represented by exp^A. The least such set is $\alpha(post[\tau_i](\gamma(\mathsf{exp}^A)))$, but as we have already seen this is expensive to compute. Therefore we choose the weaker approximation $\alpha'(post[\tau_i](\gamma(\mathsf{exp}^A)))$ which is much easier to compute as it has the form of a monomial:

$$\tau_i^A(\mathsf{exp}^A) = \begin{cases} \mathit{false} & \text{if } \mathsf{exp}^A[\overline{\varphi}/\overline{B}] \Rightarrow \neg g_i \quad (3.0) \\ \bigwedge_{j=1}^{\ell} \left\{ \begin{array}{lll} B_j & \text{if } \mathbf{post}[\tau_i](\mathsf{exp}^A[\overline{\varphi}/\overline{B}]) \Rightarrow \varphi_j & (3.1) \\ \neg B_j & \text{if } \mathbf{post}[\tau_i](\mathsf{exp}^A[\overline{\varphi}/\overline{B}]) \Rightarrow \neg\varphi_j & (3.2) \\ \mathit{true} & \text{otherwise} & (3.3) \end{array} \right\} & \text{otherwise} \end{cases} \quad (3)$$

Notice that exp^A has no successor if and only if in all states satisfying $\gamma(\mathsf{exp}^A)$, τ_i is not enabled. The properties (2.1) and (2.2) allow to recognize easily that the involved implications can be expressed without introducing existential quantifiers. E.g. (3.1) is equivalent to

[3] predicates $\varphi_1, ..., \varphi_\ell$ define a partition of Q_P, even if they are not independent

[4] a monomial on $B_1, ..., B_\ell$ is a conjunction of B_i's and $\neg B_i$'s containing each B_i at most once. Furthermore, we consider the predicate *false* as a monomial.

$$\mathsf{exp}^A[\overline{\varphi}/\overline{B}] \wedge g_i \;\Rightarrow\; \varphi_j[ass_i(\overline{x})/\overline{x}] \tag{3.1}$$

That means that the successor of a given abstract state can be "computed" if it is possible to check the validity of the implications in (3). In the case that these implications are in a decidable theory this can be done by means of an appropriate decision procedure. We choose to use an automatic theorem prover which implements also many interesting decision procedures. In this case, we are sure to compute exactly the transition relation defined by (3) if for all indices i either (3.0), (3.1) or (3.2) can be proved. Otherwise, the impossibility to prove either of these implications may have two different causes:

- It may be the case that $\mathbf{post}[\tau_i](\mathsf{exp}^A[\overline{\varphi}/\overline{B}])$ has a non-empty intersection with both φ_j and with $\neg\varphi_j$. In this case, the non-determinism in the abstract transition relation, is due to the fact that
  - either, the set of abstract starting states exp^A has been chosen too big: it could be cut into smaller pieces and their successors computed separately,
  - we consider monomials as successors,
  - or the abstract state space is not fine enough.
- It may simply be the case that the applied proof strategy is not powerful enough.

Abstract initial state: As abstract initial state we choose $I^A = \alpha'(init)$. Notice that in most practical cases the initial state defines exactly one possible value for most variables and can be computed easily by evaluating the predicates φ_i in the initial state.

2.3 Abstract state space exploration methods

This allows us to do a state space exploration, starting in the abstract initial state. Using the above defined abstract transition functions τ_i^A, different upper approximations of the set of reachable states (invariants) can be defined.

First approximation: $\mathcal{I}_1$ is obtained by imposing also on all computed approximations of the set of reachable abstract states the restriction to be a monomial, where we denote by $\sqcup$ the *lub*-operator of the lattice of monomials $\mathcal{M}$:

$$\mathcal{I}_1 = \bigsqcup_{j=0}^{\infty} X_j \quad \text{where} \quad \begin{cases} X_0 & = I^A \\ X_{j+1} & = \bigsqcup_{i=1}^{p} \tau_i^A(X_j) \end{cases}$$

All approximations X_j are monomials. As the longest chains in $\mathcal{M}$ are of length ℓ, $\mathcal{I}_1$ can be computed in at most ℓ iterations.

Second approximation: The strongest invariant that can be obtained using τ_i^A, is obtained by allowing approximations to be arbitrary elements of the abstract lattice (boolean expressions on $B_1, ..., B_\ell$) and by applying τ_i^A only on canonical monomials $\widehat{m}^c(B_1, ..., B_\ell)$ representing a single state:

$$\mathcal{I}_2 = \bigvee_{j=0}^{\infty} X_j \quad \text{where} \quad \begin{cases} X_0 & = I^A \\ X_{j+1} & = \bigvee\{\tau_i^A(\widehat{m}^c) \mid \widehat{m}^c \Rightarrow X_j,\ i = 1..p\} \end{cases}$$

This corresponds to the invariant obtained by an enumerative state exploration where the successor sets are computed individually for each already reached state. Notice that this increases the precision as

$$\tau_i^A(\widehat{m}^c{}_1) \vee \tau_i^A(\widehat{m}^c{}_2) \Rightarrow \tau_i^A(\widehat{m}^c{}_1 \vee \widehat{m}^c{}_2)$$

but the inverse implication does not hold in general. However, if $\widehat{m}^c{}_1$ and $\widehat{m}^c{}_2$ differ only on the values of abstract variables B_j such that the set of successors does not depend on the fact that φ_j holds or not, then the inverse implication holds also and it is not useful to compute the successors of these two states separately.

Complexity issues: It is reasonable to express the complexity of the computation of the above invariants by means of the number of necessary proofs. In order to compute the successor of any set of abstract states exp^A, at most $K = 2 * p * \ell + 1$ proofs (1 for the enabledness and 2 proofs for each predicate φ_i and each transition τ_j) are needed. The computation of the invariant $\mathcal{I}_1$ needs therefore maximally $\ell * K$ proofs, but it is in general too weak. For the second invariant, in the worst case, the successors of (almost) all 2^ℓ abstract states (canonical monomials) have to be computed, leading to maximally $2^\ell * K$ proofs. However, in practice, the number of necessary proofs is much smaller as

1. some transitions τ_j leave some predicates φ_i trivially unchanged or transform φ_i independently of all (or most) other predicates φ_k In this case it is better to compute the successors of a set of abstract states (represented by a monomial) instead of each (reachable) state contained in this set.

2. only a small subset of all abstract states is reachable (otherwise $\varphi_1, ..., \varphi_\ell$ has probably not been well chosen)

3. we have not required the predicates $\varphi_1, ..., \varphi_\ell$ to be independent. If they are not, not all 2^ℓ canonical monomials represent a non-empty set of concrete states. In this case, a *dependency predicate* allows us to consider only non-spurious abstract states.

Improvement of the computed invariants: The invariants $\mathcal{I}_K$ can be improved by using them as the starting point of a backward analysis as it has been suggested, e.g., in [CC77]:

$$\mathcal{I}_K^+ = \bigwedge_{j=0}^{\infty} Y_j \quad \text{where} \quad \begin{cases} Y_0 & = \mathcal{I}_K \\ Y_{j+1} & = Y_j \wedge \bigwedge_{i=1}^{p} \widetilde{\mathbf{pre}}[\tau_i](Y_j) \end{cases} \tag{4}$$

Improved versions of this backward analysis which use theorem proving to discharge verification conditions are implemented in [BBC+96, GS96, BLO98b]. Notice that the approximations Y_i are arbitrary predicates of the concrete property lattice and not necessarily boolean combinations of $\varphi_1, ..., \varphi_\ell$. In order to do an abstract backward analysis (cf. [CC77]) a *lower* approximation of $\widetilde{\mathbf{pre}}[\tau_i](Y_j)$ is needed.

Construction of a state graph: As the computation of a successor requires several proofs, only relatively small abstract state spaces (a few thousand successor computations) can reasonably be explored. Under these circumstances, the additional cost for storing not only the set of reachable states but also the transition relation is almost negligible. This has at least two advantages:

- Any property representable as a temporal logic formula on atomic propositions in $\{B_1, ..., B_\ell\}$ without existential quantification over executions can be verified on the abstract state graph using a model checker.
- The obtained abstract state graph represents a relatively precise global control graph, especially if all abstract states represent a set of concrete states enabling exactly the same transitions (this is the case if the guards of the program are boolean combinations of predicates in $\{\varphi_1, ..., \varphi_\ell\}$). The method and tool described in [BL98] generate stronger structural invariants for this control graph than for the initial control structure. These invariants can be used to improve the result of the backward analysis defined by (4).

Refinement of an abstract state graph: If the abstract state space exploration by means of τ_i^A does not allow some property to be verified, one can try to construct a more precise abstraction by adding more predicates to $\varphi_1, ..., \varphi_\ell$, that is, to consider a finer partition of the concrete state space. E.g., for the computation of a successor of $\mathsf{exp}^A \wedge B_{new}$ by the refined transition relation, not all implications of Definition (3) have to checked, but only the new ones and those which could *not* be proved valid during the computation of the successors of exp^A. Notice that this information can be deduced from the so far constructed transition relation and it is not necessary to keep a list of valid assertions. That means the construction of a sufficiently precise state graph can be obtained in an incremental manner. This does not mean however, that it is a good idea to start with a single predicate φ_1, compute an abstract state graph, and add incrementally more and more predicates. In general it is better to start directly with a "reasonable" set of predicates, and to refine it only if it turns out to be not sufficient.

3 An implementation

In the tool Invariant checker [GS96, Saï97] which implements the backward computation of inductive invariants (4) and also the methods described in [BBC+96, BL98] for the generation of structural invariants, we have also implemented an abstract state graph generation. We have achieved an integration with the PVS theorem prover where all the implications necessary to compute the successors of an already reached state are submitted to the PVS prover. A proof strategy combining decision procedures, rewriting and boolean simplification using BDDs, is systematically applied. This proof strategy is often sufficient to prove all valid implications that are generated.

As the tool can handle programs with explicit control locations, an abstract state is a tuple $(ctrl, \widehat{m}^c)$ where $ctrl$ is a concrete control configuration and $\widehat{m}^c$ is a valuation of a set of boolean variables $B_1, ..., B_\ell$ as defined in the preceding section.

1. Given a set of concrete predicates $\{\varphi_1, ..., \varphi_\ell\}$, an upper approximation of a dependency predicate is computed and used in order to generate only consistent successors. The exact dependency predicate can be computed if $\{\varphi_1, ..., \varphi_\ell\}$ can be divided using syntactical independency into a set of small sets of potentially dependent predicates.

2. Auxiliary invariants are generated using the initial control structure where all control configurations of a system consisting of several parallel components are considered reachable.

3. An abstract state graph is generated. The invariant $\mathcal{I}$, which is a conjunction of already known invariants of the system relevant for the transition under study is used to construct smaller successors for each abstract state by replacing the implications of (3) by weaker ones. For example the implication (3.1) becomes:

$$\mathcal{I} \wedge \gamma(\mathsf{exp}^A) \wedge g_i \;\Rightarrow\; \varphi_j[ass_i(\overline{x})/\overline{x}] \tag{3.1'}$$

Also, not all the implications of (3) are generated, but only those compatible with the generated dependency predicate and those which cannot be directly obtained from the already computed abstract transition relation. (3.1') considers only successors of states in $\mathcal{I}$. We could also take care to add only abstract successors representing a non-empty set of concrete states in $\mathcal{I}$, but experimentation showed that this represents a lot of effort for a small number of states that could be eliminated.

An algorithm: We present here a version of the algorithm for systems without explicit control locations. It is based on the representation of abstract state and transition sets by means of boolean expressions on $B_1, ..., B_\ell$[5] respectively BDDs. However, we had problems to interface PVS with an external BDD package and in the actual implementation sets are represented explicitly by lists.

Preliminary definitions:

- We construct an abstract invariant $\mathcal{I}_D^A$ obtained by analysis of the dependencies between the predicates $\varphi_1, ..., \varphi_\ell$. Furthermore, we generate a concrete invariant $\mathcal{I}$ using the facilities in our tool. At each successor computation we will only use its useful conjuncts.

- We try to find by static analysis some constraint $\mathsf{Ctau}[i](B_1, ..., B_\ell, B_1', ..., B_\ell')$ for each transition τ_i. For example, we examine which are the predicates φ_j not touched by it which allows intersect $\mathsf{Ctau}[i]$ with the constraint $B_j = B_j'$.

- We define an abstract predicate $\mathsf{Aguard}[i] = \alpha'(g_i)$ for each transition τ_i. In general the predicates $\varphi_1, ..., \varphi_\ell$ are chosen in such a way that $\mathsf{Aguard}[i]$ represents exactly the guard g_i of the transition τ_i.

- AReach represents at each stage of the algorithm the so far computed set of reachable states. At termination it represents an invariant of the program.

- $\mathsf{Atau}[i]$ represents at each stage of the algorithm an upper approximation of the abstract transition relation τ_i^A.

- To_explore is an auxiliary variable representing the set of states for which we have still to compute the successors.

[5] a transition relation is represented by a predicate of the form $trans(B_1, ..., B_\ell, B_1', ..., B_\ell')$ where $B_1, ..., B_\ell$ represents the start and $B_1', ..., B_\ell'$ the target state of each represented transition

Initializations:
$\mathsf{AInit} := \alpha'(init)$;
for all i : $\mathsf{Atau}[i] := \mathcal{I}_D^A \wedge \mathcal{I}_D^{A'} \wedge \mathsf{Ctau}[i]$;
$\mathsf{AReach} := \mathsf{AInit}$;
$\mathsf{To\_explore} := \mathsf{AInit}$;

Iteration:
While $\mathsf{To\_explore} \neq \mathit{false}$

Choose m in $\mathsf{To\_explore}$;
$\mathsf{To\_explore} := \mathsf{To\_explore} \wedge \neg \mathsf{m}$;

% compute the set of successors succ of m and update %
% the abstract transition relation and set of reachable states %
If $\mathsf{m} \Rightarrow \mathsf{Aguard}[i]$ then % this is a boolean decision %

$$\mathsf{succ} := \bigwedge_{j=1}^{\ell} \begin{cases} B_j & \text{if} \quad \mathsf{Atau}[i] \wedge \mathsf{m} \Rightarrow B'_j \quad \%idem\% \\ \neg B_j & \text{if} \quad \mathsf{Atau}[i] \wedge \ \mathsf{m} \ \Rightarrow \neg B'_j \%idem\% \\ B_j & \text{if} \quad \mathcal{I} \wedge post[\tau_i](\gamma(\mathsf{m})) \Rightarrow \varphi'_j \\ \neg B_j & \text{if} \quad \mathcal{I} \wedge post[\tau_i](\gamma(\mathsf{m})) \Rightarrow \neg \varphi'_j \\ \mathit{true} & \text{else} \end{cases}$$

$\mathsf{Atau}[i] := \mathsf{Atau}[i] \wedge (\mathsf{m} \Rightarrow \mathsf{succ}[\overline{B}/\overline{B'}])$;
$\mathsf{To\_explore} := \mathsf{To\_explore} \wedge (\ \mathsf{succ} \wedge \neg\ \mathsf{AReach})$;
$\mathsf{AReach} := \mathsf{AReach} \vee \mathsf{succ}$;

This algorithms allows us to generate a state graph in a totally *automatic* manner as we never try to prove interactively a generated implication: if the proof of a valid implication fails, a weaker successor is obtained. The user guides the verification by (re)defining the predicates $\varphi_1, ..., \varphi_\ell$ for the definition of the abstract state graph and by defining the automatic proof strategy. The constructed abstract state graph is transformed into the format of the ALDÉBARAN tool [FGK+96], and can then be analyzed by all the techniques available in ALDÉBARAN, such as minimization, model-checking and graphical display of graphs. In a near future, it is foreseen to represent abstract state sets and transition relation by BDDs, which is convenient for an incremental construction of the abstract state graph and for the efficient representation of global constraints on $\mathsf{Atau}[i]$.

Choice of the predicates φ_i: In order to obtain good results, it is often essential to use the *guards* appearing in the transitions of the system. This allows us to construct successors only via transitions enabled in all represented concrete states and replaces the enabledness check (3.0) by a boolean test. In order to prove that ψ is an invariant of the system (or any other property involving ψ), we can also try to use ψ for the definition of the abstract state space. Furthermore, each predicate is split into its set of literals. E.g., for the verification of the invariant (6) below we take $\varphi_1 = (OUT = IN)$ and $\varphi_2 = (OUT = tail(IN))$ instead of the disjunction $\varphi_1 \vee \varphi_2$; otherwise, in most cases, too much information is lost. Sometimes, it may be helpful to use φ_1, φ_2 and $\varphi_1 \vee \varphi_2$.

Example: We have applied this method for the verification of a simple alternating bit protocol. The protocol is correct if the list of already received messages OUT is a prefix of the list of so far sent messages IN such that OUT has at most one element less than IN. This can be expressed by

$$\Box(OUT = IN \ \vee \ OUT = tail(IN)) \tag{6}$$

Using the already implemented backward verification method to prove (6), the computation of the appropriate inductive invariant[6] does not terminate and no interesting structural invariants are generated.

Using the two predicates appearing in the guards of the program (they express for both processes that the received bit is the expected one), a deterministic abstract state graph is obtained by the algorithm implemented in our tool. 34 (decidable) implications are submitted to the prover, 5 abstract states are found reachable, and the construction takes 68 seconds.

Using the literals φ_1 and φ_2 of (6) for the construction of the abstract state graph, leads to a state graph with more states and more non-determinism, but not to a more precise one. We have used two methods to obtain a more precise approximation:

1. We have refined the so far obtained abstract state graph by using also the internal predicate $message(message\_channel) = head(IN)$ — expressing that the last sent message is the head of IN. This refinement does not allow to eliminate any state nor transition and the resulting abstract state graph is exactly the initial one, but all its states satisfy either "$IN = OUT$" or "$OUT = tail(IN)$".

2. We have used the computed abstract state graph as a control graph on which the tool generates much stronger structural invariants than on the original system. Then, we apply the suggested backward analysis to strengthen the already obtained invariant. The Property (6) can be proved with a single iteration.

In this simple example, the control depends only on finite domain variables, and it would be much easier to construct the control graph using partial evaluation as proposed e.g. in [HGD95]. In the example of the next section however, the control depends on non instantiated parameters and partial evaluation is not possible.

4 Case Study : Bounded Retransmission Protocol

We have used this method to verify a Bounded Retransmission Protocol (BRP) developed by Philips [GvdP93]. The BRP protocol is an extension of the alternating bit protocol, where not single messages, but message packets are transmitted and the number of possible retransmissions per message is bounded by some number max. We consider a fully parameterized version of the protocol where the packets can be of any size, and max any positive number. The protocol has already been proved before using a theorem prover [GvdP93, HSV94, HS96], where a large amount of user interaction has been necessary to provide powerful enough auxiliary invariants.

[6] The weakest inductive invariant implying $IN = OUT \ \vee \ OUT = tail(IN)$

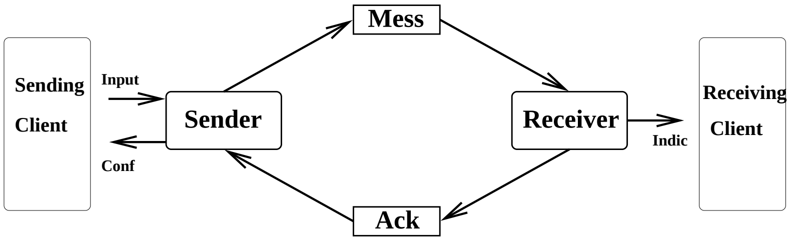

Figure 1: The architecture of the BRP protocol

Description of the protocol: The sender receives from a sending client a message packet to transmit. The sender delivers a confirmation to its client: `OK`, if all messages have been transmitted and acknowledged, `NOT_OK`, if the transmission has been aborted as more than max retransmissions would have been necessary to deliver a message, `DONT_KNOW`, if the last message has not been acknowledged (in this case, it is not possible to know if this message or its acknowledgment has been lost).

The receiver acknowledges each received message, and delivers an indication to the receiving client. The indication is `FIRST` for the first received message of a packet, `INCOMPLETE` for any intermediate message, and `OK` for the last message. If the sender abandons the transmission of a packet after sending successfully at least one message, the receiver delivers a not `NOT_OK` indication.

There are two timers `T1` and `T2`. Timeout of `T1` indicates to the sender that a message (or its acknowledgment) has been lost. Timeout of `T2` indicates to the receiver that the sender has definitively abandoned the transmission of the packet.

Correctness criterion: One must prove that the sequences of received messages and of sent messages are consistent, that is, Property (6) of Section 3. It has also to be proved that for each packet, the indication and the confirmation delivered to the clients are consistent. That means, if the sender delivers a `OK` confirmation, the receiver delivers an `OK` indication. If the receiver delivers a `NOT_OK` indication, the sender delivers the `DONT_KNOW` or `NOT_OK` confirmation. These properties can easily be expressed by temporal logic formulas.

Verification of the protocol: To construct the abstract state graph for the BRP, we have used 19 predicates appearing in the guards of the system. The constructed abstract graph has 475 states and 685 transitions and has been obtained in three hours on a Sparc 10. Of the 24 possible global control configurations, only 9 are found reachable. On this graph, the control properties concerning confirmations and indications could be verified using ALDÉBARAN. Property (6) has been verified on a weaker abstraction where only predicates concerning the transmission of a single message are considered (it can be obtained by ALDÉBARAN from the above mentioned more complex abstraction). The obtained abstract state graph is very similar to the one obtained for the alternating bit protocol, except that at any moment the transmission can be abandoned because the maximal number of retransmissions may have been reached.

5 Conclusions

We have presented and implemented a method allowing to construct abstract state graphs of arbitrary infinite state systems, where abstract states are valuations of a set of predicates $\varphi_1, ..., \varphi_\ell$ on concrete variables. At a first sight, the method may look rather expensive as the construction of a successor requires several proofs, and the construction of an abstract state graph for the BRP with 500 states takes several hours. However, the actual implementation is extremely inefficient as the independence between transitions and predicates is not really exploited. Furthermore, all proofs are done without user interaction using a single tactic, and if this tactic fails to prove some valid statements, a weaker abstraction is obtained. Once the user has provided the predicates $\varphi_1, ..., \varphi_\ell$ (the tool proposes a set consisting of the literals occurring in the guards and properties to be proved), the construction is *completely automatic*. In this case, execution time is less critical. It is always possible to apply this method to get a first approximation of a system which — from the point of view of human effort — is for free. The constructed state graph is always of a reasonable size and can be explored by a model-checker. It can also be used as a finite global control graph which can be used for further invariant generation and backward analysis.

If the initial set of predicates, defining the abstract state space, does not give a satisfactory abstraction, one can try to add new predicates to obtain a more precise abstraction. To provide a new predicate is similar to providing an auxiliary invariant, which is usually necessary to prove program properties. However, it is easier to provide some predicates leading to a sufficiently refined state graph than the corresponding auxiliary invariant (expressing when these predicates hold and when not). Sometimes, the refining predicates can be obtained from the so far constructed abstract state graph by examining the nondeterminism occurring in sequences leading to states violating the required invariant.

This construction of an abstract state graph is in some sense complementary to the tableau construction implemented in STeP [BBC+96] where the tableau of the property to be proved (or disproved) is taken as the starting point for an abstract state graph construction by expanding it until it fits with the program. We take the control of the program of the program as a starting point and refine it until it satisfies the property to be verified. The particularity of our method is that it integrates a reachability analysis in order to avoid the (costly) computation of successors of unreachable states.

It has also some other interesting characteristics:

- it is *incremental*: a refinement generates new implications which are weakenings of those generated for the previous partition. Hence, all implications valid for a given partition, are also valid for a finer partition. Furthermore, in order to use this fact, it is not necessary to store the already proved implications, but only the corresponding abstract transition relation.

- The abstract state graphs constructed by our method are interesting for debugging. It can be used to guide the search of a concrete execution sequence violating a required property, especially as any transition enabled in some abstract state is enabled in *all* concrete states it represents.

Since the first version of this paper new results in this direction have been obtained. In [BLO98a], Bensalem et al. present a similar abstraction framework which improves the one presented here essentially in two points:

- Successors of sets of abstract states (abstract predicates) and not only of single abstract states are constructed. Also, not only successors of the form of a monomial are constructed. In fact, if the transitions τ of a system,
  - let many predicates trivially unchanged,
  - $\varphi_1, ..., \varphi_\ell$ can be cut into subsets which are transformed by τ independently of each other, and all these subsets are small

  then, one may well consider arbitrary successor sets.
- The computation of the abstract transition relation is not combined with the state space exploration. On one hand, this may lead to many computations of successors of unreachable states. On the other hand, if there are many independencies as described above, the number of unnecessary successor computations may in fact be quite small. And it is an advantage to construct a abstract transition relation for each transition of the concrete system; the obtained set of transition relations can be used directly by an appropriate model-checker for either state graph generation or "on-the-fly" model-checking or compositional verification ...

With this improved method, together with the use of data abstractions of message queues (similar to those proposed in [Gra95]), they obtain a finite abstraction of the Bounded Retransmission protocol on which also data properties can be model-checked.

References

[BBC+96] N. Bjorner, A. Browne, E. Chang, M. Colon, A. Kapur, Z. Manna, H. Sipma, and T. Uribe. Step: Deductive-algorithmic verification of reactive and real-time systems. In *Conference on Computer Aided Verification, CAV'96*. LNCS 1102, Springer Verlag, 1996.

[BL98] S. Bensalem and Y. Lakhnech. Automatic generation of invariants. *Formal Methods in System Design*, 1998. To appear.

[BLO98a] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems automatically and compositionally. In *Accepted in CAV'98*, 1998.

[BLO98b] S. Bensalem, Y. Lakhnech, and S. Owre. Invest : A tool for the verification of invariants. In *Accepted in CAV'98*, 1998.

[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.

[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.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, Massachusetts, 1988.

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

[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.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III, Verification and Control*, volume 1066 of *LNCS*, pages 208–219. Springer Verlag, 1996.

[FGK+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. Cadp (cæsar/aldebaran development package): A protocol validation and verification toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *LNCS*, pages 437–440. Springer Verlag, August 1996.

[GL93] 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.

[Gra95] S. Graf. Characterization of a sequentially consistent memory and verification of a cache memory by abstraction. *accepted to Distributed Computing*, 1995.

[GS96] S. Graf and H. Saidi. Verifying invariants using theorem proving. In *Conference on Computer Aided Verification CAV'96*, volume 1102 of *LNCS*, July 1996.

[GvdP93] J.F Groote and J. van de Pol. A bounded retransmission protocol for large data packets. Technical Report Logic Group Preprint Series 100, Utrecht University, 1993.

[HGD95] H. Hungar, O. Grumberg, and W. Damm. What if model checking must be truly symbolic. In *Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'95*. LNCS 1019, 1995.

[HH95] T. Henzinger and P.H. Ho. Hytech: the cornell hybrid technology tool. In *Hybrid Systems II*. LNCS 999, Springer Verlag, 1995.

[HPR94] N. Halbwachs, Y.-E. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In B. LeCharlier, editor, *International Symposium on Static Analysis, SAS'94*, Namur (Belgique), September 1994. LNCS 864, Springer Verlag.

[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.

[HSV94] L. Helmink, M. Sellink, and F. Vaandrager. Proof-checking a data link protocol. Cs-r9420, Centrum voor Wiskunde en Informatica, 1994.

[KDG95] P. Kelb, D. Dams, and R. Gerth. Efficient symbolic model-checking for the full μ-calculus using compositional abstractions. Technical Report Computing Science Reports 95/31, Eindhoven University of Technology, October 1995.

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

[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.

[Saï97] H. Saïdi. The invariant-checker: Automated deductive verification of reactive systems. In *Proceedings of the 9th Conference on Computer-Aided Verification, CAV'97*, LNCS 1254. Springer Verlag, 1997.

[Sif82] Joseph Sifakis. A unified approach for studying the properties of transition systems. *TCS*, 18, 1982.

[SOR93] N. Shankar, S. Owre, and J.M. Rushby. *A Tutorial on Specification and Verification using PVS.* SRI International, Menlo Park, CA, 1993.