Construction of abstract state graphs with PVS

Susanne Graf and Hassen Saidi
VERIMAG[1]
{graf,saidi}@imag.fr

Abstract. In this paper, we propose a method for the automatic construction of an abstract state graph of an arbitrary system using the PVS theorem prover.
Given a parallel composition of sequential processes and a partition of the state space induced by predicates $\varphi_1, ..., \varphi_\ell$ on the 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 programs.

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.

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 $\widehat{m}$ for a

[1] Centre Equation, 2, Avenue de la Vignate, 38610 Grenoble-Gières

transition τ of the program is the *least* monomial satisfied by all successors via τ of concrete states satisfying $\widehat{m}$. 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 $\widehat{m}$ 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, 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 in [GS96]. A global control graph allows us to obtain much stronger structural invariants using the tool described in [BLS96,BBC+96] than the initial presentation as a parallel composition of processes.
We have implemented a particular case of this method in our tool [GS96] where only successors of canonical monomials are constructed: if a successor is not a canonical monomial (that is some non-determinism is introduced by the abstraction), it is split into its canonical monomials. 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 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.

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 [BLS96]. Therefore, we consider here 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 (concrete) states to be represented by predicates φ on the program variables (hence the name predicate transformer).

Definition 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:

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

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

$$\mathbf{post}[R](\varphi) \Rightarrow \varphi' \text{ iff } \varphi \Rightarrow \widetilde{\mathbf{pre}}[R](\varphi')^2 \tag{2.2}$$

Abstract semantics of programs: All the results presented in this section are an application of abstract interpretation [CC77].

[2] this property is due to the fact that $(\widetilde{\mathbf{pre}}[R](), \mathbf{post}[R]())$ forms a Galois connection

Definition 3 (abstract state graphs). Let $\mathcal{S} = (Q, \cup\tau_i, I)$ 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[3]. $\mathcal{S}^A = (\mathbf{Q}^A, \cup\tau_i^A, I^A)$ 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 *concretisation 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.

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 *predicates* $\varphi_1, ..., \varphi_\ell$ on the variables of P[4]. We choose as abstract state space the lattice $\widehat{\mathcal{M}}$ of the $3^\ell + 1$ monomials on abstract *boolean variables* $\widehat{\varphi}_1, ..., \widehat{\varphi}_\ell$[5]. Notice that,

- $\widehat{\mathcal{M}}$ forms a complete lattice with order relation $\Rightarrow$ (implication), *glb* operator $\wedge$ (conjunction,), *lub* operator $\sqcup$ which is weaker than $\vee$ (e.g. $\widehat{\varphi}_1 \wedge \widehat{\varphi}_2 \sqcup \widehat{\varphi}_2 \wedge \neg\widehat{\varphi}_3 = \widehat{\varphi}_2$). The set of atoms of the lattice is the set $\widehat{\mathcal{M}}^c$ of the 2^ℓ *canonical* monomials.
- Each abstract state $\widehat{m} \in \widehat{\mathcal{M}}$ represents a set of concrete states defined by the predicate on concrete program variables obtained by substituting every abstract variable $\widehat{\varphi}_i$ by the concrete predicate φ_i, that is
$$\gamma(\widehat{m}) = \widehat{m}[\varphi_i/\widehat{\varphi}_i]$$
- Sets of abstract states can be represented by arbitrary boolean expressions on $\widehat{\varphi}_1, ..., \widehat{\varphi}_\ell$.

Abstract transitions: For each concrete transition τ_i of the program, we define an abstract transition function τ_i^A associating with any abstract state $\widehat{m}$ the least abstract state representing all successors of the concrete states represented by $\widehat{m}$. The fact that the abstract successor $\widehat{m}'$ is a monomial, allows to determine it as follows: it is "*false*" ($\widehat{m}$ has no successor) if in all concrete states satisfying

[3] 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 $\alpha(\varphi) = \sqcap\{\mathrm{Q}^A \in \mathbf{Q}^A | \varphi \Rightarrow \gamma(\mathrm{Q}^A)\}$.

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

[5] a monomial is a conjunction of $\widehat{\varphi}_i$'s and $\neg\widehat{\varphi}_i$'s containing each $\widehat{\varphi}_i$ at most once. Furthermore, we consider the predicate *false* as a monomial.

$\gamma(\widehat{m})$, τ_i is not enabled; $\widehat{m}'$ has a conjunct $\widehat{\varphi}_j$ (resp. $\neg\widehat{\varphi}_j$) if all successors of states satisfying $\gamma(\widehat{m})$ satisfy φ_j (resp. $\neg\varphi_j$); otherwise, $\widehat{m}'$ depends not on $\widehat{\varphi}_j$:

$$\tau_i^A(\widehat{m}) = \begin{cases} \mathit{false} & \text{if } \gamma(\widehat{m}) \Rightarrow \neg g_i \quad (3.0) \\ \bigwedge \widehat{m}'_j, \widehat{m}'_j = \left\{ \begin{array}{lll} \widehat{\varphi}_j & \text{if } \mathbf{post}[\tau_i](\gamma(\widehat{m})) \Rightarrow \varphi_j & (3.1) \\ \neg\widehat{\varphi}_j & \text{if } \mathbf{post}[\tau_i](\gamma(\widehat{m})) \Rightarrow \neg\varphi_j & (3.2) \\ \mathit{true} & \text{otherwise} & (3.3) \end{array} \right\} & \text{otherwise} \end{cases} \tag{3}$$

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

$$\gamma(\widehat{m}) \wedge g_i \;\Rightarrow\; \varphi_j[ass_i(\overline{x})/\overline{x}] \tag{3.1}$$

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 order to prove these implications, one can use a theorem prover. In this case, we are sure to compute the "exact" result defined by (3) if for all indices i either (3.0), (3.1) or (3.2) can be proved. Otherwise, the negative results can either be due to the fact that $\mathbf{post}[\tau_i](\widehat{m})$ has a non-empty intersection with both φ_i and with $\neg\varphi_i$ or simply to the fact that the applied proof strategy is not powerful enough. This allows us to do a state space exploration, starting in the abstract initial state which can be computed analogously.

2.3 Abstract state space exploration methods

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 identifying the abstract state and property lattices:

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

All approximations X_j are abstract states (elements of $\widehat{\mathcal{M}}$). Thus, $\mathcal{I}_1$ can be computed in at most ℓ iterations, as the longest chains in $\widehat{\mathcal{M}}$ are of length ℓ.

Second approximation: The strongest invariant that can be obtained using τ_i^A, is obtained by allowing abstract properties to be arbitrary disjunctions of abstract states (the abstract property lattice is the lattice of boolean expressions on $\widehat{\varphi}_1, ..., \widehat{\varphi}_\ell$) and by applying τ_i^A only on canonical monomials $\widehat{m}^c \in \widehat{\mathcal{M}^c}$:

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

$\mathcal{I}_2$ can be obtained by a state space exploration, where only canonical monomials are treated as "states", and each constructed abstract successor is split into its set of canonical monomials.

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 an arbitrary abstract state $\widehat{m}$, at most $K = 2*p*\ell+1$ proofs (1 proof 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 minimal 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
2. only a subset of states are reachable
3. we have not required the predicates $\varphi_1, ..., \varphi_\ell$ to be independent. If they are not, not all 2^ℓ canonical monomials are consistent (that is states). In this case, a *dependency predicate* allows us to eliminate inconsistent 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

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

Improved versions of this backward analysis which use theorem proving to discharge verification conditions are implemented in [BBC$^+$96,GS96]. 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, e.g., the weakest monomial completely contained in $\widetilde{\mathbf{pre}}[\tau_i](Y_j)$ which can be obtained with at most $2 * \ell$ proofs.

Construction of state graphs: 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 the representation of the transition relation is almost negligible. The construction of a complete state graph has at least two advantages:

– any property representable as a temporal logic property without existential quantification over executions can be verified using a model checker.
– It represents a relatively precise global control graph, especially if all abstract states represent a set of concrete states enabling exactly the same transitions. The method and tool described in [BLS96] 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. For the computation of a successor of $\widehat{m} \wedge \widehat{m}_{new}$ by the refined transition relation, not all implications of Definition (3) — already checked during the construction of the successor of $\widehat{m}$ — have to be checked, but only those which could *not* be proved valid in the previous check (and this information can be deduced from the so far constructed transition relation; 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.

3 An implementation

In [GS96] and [Sa97], we presented a tool implementing the backward computation of inductive invariants (4) and also the methods described in [BLS96]and [BBC+96] for the generation of structural invariants. We have also implemented an abstract state graph generation corresponding to the second approximation. 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 our tool also deals with explicit control, an abstract state $\widehat{m}$ consists of the concrete control configuration c and a valuation of a set of boolean variables $\widehat{\varphi}_1, ..., \widehat{\varphi}_\ell$ as defined in the preceding section.

1. Given a set of predicates $\varphi_1, ..., \varphi_\ell$, an upper approximation of a dependency predicate is computed and used in order to generate only consistent successors.
2. Auxiliary invariants described in [BLS96] are generated using the initial control structure where all control configurations of a system consisting of several parallel components are considered reachable.
3. A state graph is generated. The conjunction $\mathcal{I}$ of already known invariants of the system 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:
$$\mathcal{I} \wedge \gamma(\widehat{m}) \wedge g_i \;\Rightarrow\; \varphi_j[ass_i(\overline{x})/\overline{x}] \qquad (3.1')$$

Also, not all the implications of (3) are generated, but only those compatible with the generated dependency predicate. (3.1') considers only successors of states in $\mathcal{I}$. Furthermore, only successors having a non-empty intersection with $\mathcal{I}$ should be added to the set of reachable states. If $\mathcal{I}$ is (provably) inductive, only such successors are constructed. Otherwise, for each abstract state $\widehat{m}$ obtained by (4), it should be verified if
$$\gamma(\widehat{m}) \Rightarrow \neg\mathcal{I} \qquad (5)$$
holds. If the proof of (5) succeeds, no state in $\gamma(\widehat{m})$ is reachable, otherwise, we don't know, and $\widehat{m}$ must be considered as reachable. If an abstract state is reached, such that for some enabled transition no (consistent) successor in $\mathcal{I}$ is constructed, this state is itself not in $\mathcal{I}$, and is eliminated.

The generation of the abstract graph is *completely automatic* 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 generated in the format of the ALDÉBARAN tool [FGK+96], and can therefore be analyzed by all the techniques available in ALDÉBARAN, such as minimization, model-checking and automatic 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.

Choice of the predicates φ_i: In order to prove that ψ is an invariant of the system (or any other property involving ψ), we can try to use ψ for the definition of the abstract state space. But it is 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. Furthermore, each predicate is split into its 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.

Example: We have applied this method for the verification of an 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 only the already implemented backward method to prove (6), the computation of the appropriate inductive invariant[6] does not terminate (also using the generated structural invariants).

Using the predicates appearing in the guards, the abstract state graph of Figure 1 is constructed. These two predicates express the fact that the bit joint with the message (respectively the bit representing the acknowledgment) is of the expected value. 34 implications are submitted to the prover, 5 abstract states are created, and the construction takes 68 seconds.

Using the literals φ_1 and φ_2 of (6) for the construction of the abstract state graph does not result in more precision. We have used two methods to obtain a better approximation:

1. We have refined the so far obtained state graph by using also the internal predicate $message(message\_channel) = head(IN)$ — expressing that the last sent message is the head of IN. The constructed state graph is again the one of Figure 1, but all its states satisfy either $IN = OUT$ or $OUT = tail(IN)$.
2. We have used the state graph of Figure 1 as a control graph which allows to generate more structural invariants using the methods of [BLS96,BBC+96]. 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 protocol, the control depends only on finite domain variables, and it would be much faster to construct the control graph using the expansion

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

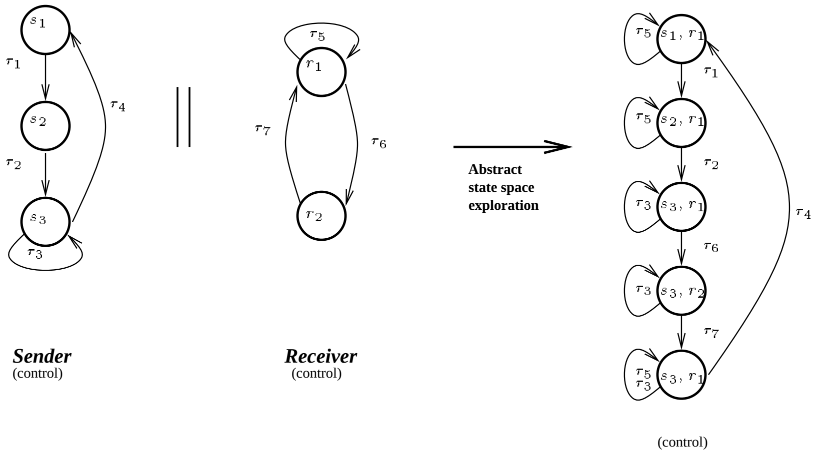

Fig. 1. Abstract control graph for the alternating bit protocol

method described in [HGD95]. In the example of the next section however, the control depends partially on infinite domain variables, and the expansion method does not work.

4 Verification of a 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 using a theorem prover [GvdP93,HSV94,HS96], where a large amount of user interaction has been necessary to provide powerful enough auxiliary invariants.

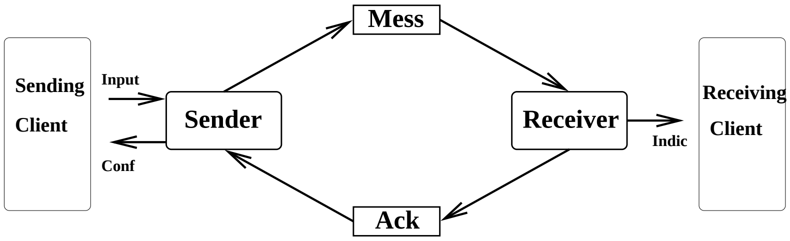

Fig. 2. The architecture of the BRP protocol

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.

Correctness criterion: We have to prove that the sequences of received messages and of sent messages are consistent, that is, Property (6) of Section 3. We have also to prove 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 transitions of the system. The constructed abstract graph has 475 states and 685 transitions and has been obtained in five hours on a Sparc 10. Of the 24 possible global control configurations, only 9 are found reachable. On this graph the properties concerning confirmations and indications have been verified using ALDÉBARAN. Property (6) has been verified on a weaker abstraction where only predicates concerning the transmission of a single message are considered relevant. The obtained abstract state graph is similar to the one obtained for the alternating bit protocol (cf Figure 1), except that at any moment the transmission can be abandoned because the maximal number of retransmissions has 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 5 hours. However, 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, the execution time is not really a problem. One can always 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 invariant generation and backward analysis already implemented [BLS96,GS96].

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).

This method 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. Our method takes the control of the program of the program as a starting point and refines it until it satisfies the property to be verified. The particularity of our method is that it integrates a reachability analysis.

It has also some other interesting characteristics:

- it is *incremental*: a refinement generates new implications and strengthens the left hand side of previously generated implications. 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.
- It is *compositional*: for each component a separate abstract state graph can be constructed, where in each component the predicates involving its variables are used. The obtained global abstraction is in general weaker, and for the examples presented in this paper, the compositional approach turned out not to be interesting.
- 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.

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 *CAV'96*. LNCS 1102, 1996.

[BBL97] K. Baukus, S. Bensalem, and Y. Lakhnech. A PVS based tool for the verification of invariants. submitted to CAV'97.

[BLS96] S. Bensalem, Y. Lakhnech, and H. Saidi. Powerful techniques for the automatic generation of invariants. In *CAV'96*, LNCS 1102, 1996.

[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), 1994.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design*. 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-commitment style reasoning and theorem proving. In *CAV'95*. LNCS 939, 1995.

[DGG93] D. Dams, O. Grumberg, and R. Gerth. Generation of reduced models for checking fragments of CTL. In *CAV'93*, LNCS 697, 1993.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III, Verification and Control*, LNCS 1066, 1996.

[FGK+96] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier and M. Sighireanu. CADP (Cæsar/Aldébaran Development Package): A protocol validation and verification toolbox. In *CAV'96*. LNCS 1102.

[GL93] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In *CAV'93*. LNCS 697, 1993.

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

[GS96] S. Graf and H. Saidi. Verifying invariants using theorem proving. In *CAV'96*, LNCS 1102, 1996.

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

[HGD95] H. Hungar, O. Grumberg, and W. Damm. What if model checking must be truly symbolic. In *TACAS'95*. LNCS 1019, 1995.

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

[HPR94] N. Halbwachs, Y.-E. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In *SAS'94*. LNCS 864.

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

[HSV94] L. Helmink, M. Sellink, and F. Vaandrager. Proof-checking a data link protocol. Technical report CS-R9420, CWI, 1994.

[KDG95] P. Kelb, D. Dams, and R. Gerth. Efficient symbolic model-checking for the full μ-calculus using compositional abstractions. Tech Rep 31, TU Eindhoven, 1995.

[Kur94] R.P. Kurshan. *Computer-Aided Verification of Coordinating processes, the automata theoretic approach*. Princeton Series in Computer Science. 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, 1995*

[Sa97] H. Saïdi. The invariant checker: Automated deductive verification of reactive systems. In *this volume*.

[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.