Computing Abstractions of Infinite State Systems Compositionally and Automatically *

S. Bensalem† Y. Lakhnech‡ S. Owre§

June 10, 1998

Abstract

We present a method for computing abstractions of infinite state systems *compositionally* and *automatically*. Given a concrete system $S = S_1 \parallel \cdots \parallel S_n$ of programs and given an abstraction function α, using our method one can compute an abstract system $S^a = S_1^a \parallel \cdots \parallel S_n^a$ such that S simulates S^a, i.e. every computation of S can be mapped to a computation of S^a. Thus, if α maps concrete states to abstract states taken from a finite set, we obtain a finite state system S^a which can be analyzed algorithmically. A distinguishing feature of our method is that it does not produce a single abstract state graph but rather preserves the structure of the concrete system. This feature is a prerequisite to benefit from the techniques developed in the context of model-checking for mitigating the state explosion. Moreover, our method has the advantage that the process of constructing the abstract system does *not* depend on whether the computational model is synchronous or asynchronous, i.e., interleaving-based.

We also present an implementation of our method which is based on the theorem prover PVS. The produced abstract system is represented optionally in the specification language of PVS or in that of SMV. Hence, our implementation provides a bridge between the theorem prover PVS and the model-checker SMV. To give evidence to the usefulness and scalability of our method we consider two different applications in which the concrete systems are infinite but in which the sources of infinity are of different nature.

*This work has been partly performed while the first two authors were visiting the Computer Science Laboratory, SRI International. Their visits were funded by NSF Grants No. CCR-9712383 and CCR-9509931.

†VERIMAG, Centre Equation – 2, avenue de Vignate, F-38610 Giéres, France. Bensalem@imag.fr

‡Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Preußerstr. 1-9, D-24105 Kiel, Germany. yl@informatik.uni-kiel.de

§Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA. owre@csl.sri.com

1 Introduction

A major task in proving correctness of protocols consists in proving invariance properties. Indeed, every safety property can be reduced to an invariance property and to prove progress properties one needs to establish invariance properties [22]. Proving invariance properties is especially crucial for infinite and large finite state systems which escape algorithmic methods.

The standard way to prove invariance properties of infinite state systems is by induction. To prove that φ is an invariant of S one has to come up with a stronger invariant ψ that is preserved by every transition of S. The invariant ψ is usually called an auxiliary invariant. This deductive method has three drawbacks: 1.) it is often hard to find suitable auxiliary invariants, 2.) when a choice for an auxiliary invariant fails, one has little hint how to strengthen the invariant and 3.) one obtains no counterexample in the form of a trace when the considered program does not satisfy the invariance property. Besides being automatic, one of the main reasons for the success of model-checking approach, is its ability to produce counterexamples.[1] Techniques for generating and strengthening invariants (cf. [23, 3, 2, 28]) seem to give limited results when applied to protocols where the control is partly encoded within the data part, in particular when shared variables are extensively used for synchronization.

On the other hand, verification by abstraction appears to be promising for reasoning about control intensive protocols in which control is finite but the data part is infinite or very large. The use of abstraction techniques to model-check finite state reactive systems is by now a well-established approach [4, 21, 9, 19, 20, 6, 18]. There are methods/tools that compute an abstract system from the text of a finite state program and an abstraction relation [4, 8, 13, 7, 10]. It should be realized that it is important to avoid the construction of the concrete model which represents the semantics of the considered program before generating the abstract system. Otherwise, one would have to store the concrete system which might be too large. The produced abstract system is usually smaller than the concrete one, and hence is much simpler to model-check.

Verification by abstraction can also be applied to infinite state systems as shown in [11, 12, 26, 16]. However, in all these approaches the verifier has to provide the abstract system and an important amount of user intervention is required to prove that the concrete system simulates the abstract one. What is needed is a method to automatically compute an abstract system for a given infinite state system and an abstraction relation. A method that achieves this for a restricted form of abstraction functions, namely those induced by a set of predicates on the concrete states, is given in [14]. This method has, however, the drawback that it generates an abstract graph rather than the text of an abstract program with the consequence that one can neither apply further abstractions nor techniques for avoiding the state explosion problem as, for example, partial-order techniques.

We present a method that computes an abstract system $S^a = S_1^a \parallel \cdots \parallel S_n^a$, for a given system $S = S_1 \parallel \cdots \parallel S_n$ and abstraction function α, such that S simulates S^a is guaranteed by the construction. Hence, by known preservation results, if S^a satisfies an invariant φ then S satisfies the invariant $\alpha^{-1}(\varphi)$. Since the produced abstract system is not given by a graph but in a programming language, one still can 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. Furthermore, since the process of generation of the abstract system does not depend on the assumed semantics of the parallel operator, our method works for both the synchronous as well as the asynchronous computation model.

The basic idea behind our method is simple. In order to construct an abstraction of S, we construct for each concrete transition τ_c an abstract transition τ_a. To construct τ_a we

[1]Notice that this does not contradict the fact that in case the considered system satisfies the property to prove, a proof by induction leads to more insight about why the system is correct than a proof by a model-checker. Indeed, in this case most model-checkers would simply answer that the property is satisfied.

proceed by elimination starting from the universal relation, which relates every abstract state to every abstract state, and eliminate pairs of abstract states in a conservative way, i.e. it is guaranteed that after elimination of a pair the obtained transition is still an abstraction of τ_c. To check whether a pair (a, a') of abstract states can be eliminated we have to check that the concrete transition τ_c does not lead from any state c with $\alpha(c) = a$ to any state c' with $\alpha(c') = a'$. This amounts to proving a Hoare triple. The elimination method is in general too complex. Therefore, we combine it with three techniques that allow to check many fewer Hoare triples. These techniques are based on partitioning the set of abstract variables, using substitutions, and a new preservation result which allows to use the invariant to be proved during the construction process of the abstract system. A partitioning of the set of abstract variables allows to consider a small group of abstract variables at a time. This reduces the number of Hoare triples to check, as the number of transitions of the universal relation is exponential in the number of variables. However, in general, such a partitioning leads to a more non-deterministic abstract system. We give sufficient conditions under which a partitioning of the abstract variables does not increase the non-determinism of the abstract system. We also identify cases in which an abstraction of a transition can be computed solely by applying syntactic substitutions without using the elimination method. The basic idea is as follows: suppose that the concrete transition is given by a guarded multiple assignment. We substitute the expressions occurring on the right hand side of the assignments in the definition of the abstraction function. Then, the latter defines an abstraction of the concrete transition, if we can eliminate all concrete variables from the obtained expression. In many examples, a major part of the transitions can be handled by syntactic substitutions or by a combination of the elimination method with syntactic substitutions. Finally, our new preservation result allows us to consider only Hoare triples whose precondition implies the invariant to prove. Obviously, this reduces the number of Hoare triples to check.

We implemented our method using the theorem prover PVS [27] to check the Hoare triples generated by the elimination method. The first-order formulas corresponding to these Hoare triples are constructed automatically and a strategy that is given by the user is applied.

The produced abstract system is optionally represented in the specification language of PVS or in that of SMV [25]. Thus, our implementation provides a bridge between PVS and SMV. We applied our method and its implementation on a number of examples. In this paper, we report on the verification of the Bounded Retransmission protocol and on the computation of a finite abstraction of the Alternating Bit protocol. The Bounded Retransmission protocol has been verified using theorem proving in [15, 17, 16]. An automatic verification of some of the correctness aspects of the protocol is reported in [14]. We achieved an automatic verification of all correctness aspects of the protocol. A clever abstraction of the Alternating Bit protocol is given in [26] where it is shown using Isabelle/HOL that the given abstract system is an abstraction of the concrete one. We revisit the same exercise and *automatically compute* the abstract system. In addition to these protocols we applied our method to the Bakery and the Peterson mutual exclusion algorithms and to a simplified version of the Futurebus+ protocol. For all these examples an abstract system that satisfies the desired invariance property was computed automatically.

Related Work

As mentioned above a method for computing abstractions of infinite state systems is presented in [14]. In contrast to [14], our method produces an abstract system which has the same structure as the concrete one. This allows for further application of abstractions and other techniques for avoiding the state explosion problem. Moreover, this gives a clear correspondence between concrete and abstract transitions, which is useful for debugging the concrete system. Our method can also deal with abstraction functions that only abstract some of the variables which range over infinite domains. This is not the case for the method in [14], since it generates a global control graph. One other advantage of our method is that it does not depend on the computation model, whether it is synchronous or asynchronous.

The basic idea behind our method for constructing an abstract system is related to the splitting algorithm of [8, 6, 7]. The purpose of the splitting algorithm is to refine an abstract structure in order to preserve properties in two directions, i.e., such that the concrete and the abstract system satisfy the same properties. We are, however, only interested in preservation of invariance properties in one direction, since this allows for more efficient methods for computing abstractions. Moreover, the splitting algorithm is based on splitting abstract states while our method is based on elimination of transitions.

The basic idea underlying the methods of [4, 8, 10] for computing abstractions of finite state systems is that of abstract interpretation [5]. Here, the abstract system is completely determined by abstract versions of the primitive operators. This is not the case in our method, since we do not consider the abstraction of the primitive operators in separation but we compute the abstraction of a complete transition. In general, the methods based on abstract interpretation are efficient but yield abstract systems which are more non-deterministic than the abstract systems computed by our method. See, for example, [8, 10] for a more detailed discussion of the relationship between abstract interpretation and the verification by abstraction approach.

2 Preliminaries

Given a set X of typed variables, a *state over* X is a type-consistent mapping that associates with each variable $x \in X$ a value. A *transition system* is given by a triple (Σ, I, R), where Σ is a set of states over a set X of variables, $I \subseteq \Sigma$ is a set of initial states, and $R \subseteq \Sigma^2$ is the transition relation. A *syntactic* transition system is given by a triple $(X, \theta(X), \rho(X, X'))$, where X is a set of typed variables, $\theta(X)$ is a predicate describing the set of initial states and $\rho(X, X')$ is a predicate describing the transition relation. We associate in the usual way a transition system with every syntactic transition system. Given relations $R_i \subseteq \Sigma_i \times \Sigma'_i$, for $i = 1, 2$, we define their *synchronous 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 mapping s to Σ_i. Thus, if $\Sigma_1 = \Sigma_2$ and $\Sigma'_1 = \Sigma'_2$ then $R_1 \otimes R_2 = R_1 \cap R_2$. The *synchronous composition* of transition systems $S_i = (\Sigma_i, I_i, R_i)$, $i = 1, 2$, denoted $S_1 \otimes S_2$, is given by the system $(\Sigma_1 \cup \Sigma_2, \{s \mid s_{|\Sigma_i} \in I_i\}, R_1 \otimes R_2)$. A *computation* of a transition system $S = (\Sigma, I, R)$ is a sequence $s_0, \cdots, s_n$ such that $s_0 \in I$ and $(s_i, s_{i+1}) \in R$, for $i \leq n - 1$. A state $s \in \Sigma$ is called *reachable* in S, if there is a computation $s_0, \cdots, s_n$ of S with $s_n = s$. A set $P \subseteq \Sigma$ is called an *invariant* of S, denoted by $S \models \Box P$, if every state that is reachable in S is in P. Given a set $P \subseteq \Sigma$ of states and a relation $R \subseteq \Sigma^2$ the *weakest liberal precondition* of R with respect to P, denoted by $\mathrm{WP}(R, P)$, is the set consisting of states s such that for every state s', if $(s, s') \in R$ then $s' \in P$. All the semantic notions introduced so far have their syntactic counterparts which we assume as known. Moreover, we will tacitly interchange syntax and semantics, e.g. predicates and sets of states etc., unless there is a necessity to make a distinction.

3 Proving Invariants by Abstraction

In this section we recall the definition of simulation as well as preservation results for invariance properties and present a new preservation result.

Definition 3.1 *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 a total relation,

2.) for all states $s^c_0, s^c_1 \in \Sigma^c$ and $s^a_0 \in \Sigma^a$ with $(s^c_0, s^a_0) \in \alpha$, if $(s^c_0, s^c_1) \in R^c$ then there exists a state $s^a_1 \in \Sigma^a$ such that $(s^a_0, s^a_1) \in R^a$ and $(s^c_1, s^a_1) \in \alpha$.

3.) for every state s^c in I^c there exists a state s^a in I^a such that $(s^c, s^a) \in \alpha$. $\Box$

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 $s_0^c, \cdots, s_n^c$ of S^c there exists a computation $s_0^a, \cdots, s_n^a$ of S^a such that $(s_i^c, s_i^a) \in \alpha$, for every $i \leq n$. This implies the following preservation result (cf. [4, 20]):

Theorem 3.1 *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 $S^c \models \Box\varphi^c$, where $\alpha^{-1}(\varphi^a) = \{s^c \in \Sigma^c \mid \exists s^a \in \varphi^a.(s^c, s^a) \in \alpha\}$.* □

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 $\varphi^a \subseteq \Sigma^a$ with $\alpha^{-1}(\varphi^a) \subseteq \varphi^c$. This method is complete. Indeed, it suffices to take an abstract system with two states s_0^a and s_1^a and a relation α such that $(s^c, s_0^a) \in \alpha$ iff s^c is reachable in S^c; and $(s^c, s_1^a) \in \alpha$ iff s^c 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 $\alpha^{-1}(s_0^a) \subseteq \varphi^c$.

On the other hand, 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 $\alpha^{-1}(\mathcal{R}(S^a))$, where $\mathcal{R}(S^a)$ is the set consisting of the reachable states of S^a.

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

Definition 3.1 can be weakened, allowing more abstractions, while preserving soundness of theorem 3.1. Indeed, if ψ^c is an invariant of S^c then conditions 1.) and 2.) in definition 3.1 can be weakened by restricting the quantification on states in Σ^c to states that satisfy ψ^c.

We can, however, weaken definition 3.1 using a set of states in Σ^c that is *not* necessarily an invariant of S^c leading, however, to a weaker preservation result. To explain this let us introduce the following definition:

Definition 3.2 *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 a total relation,*
2.) *for every state $s_0^c, s_1^c \in \Sigma^c$ and $s_0^a \in \Sigma^a$ with $s_0^c \in \varphi^c$ and $(s_0^c, s_0^a) \in \alpha$, if $(s_0^c, s_1^c) \in R^c$ then there exists a state $s_1^a \in \Sigma^a$ such that $(s_0^a, s_1^a) \in R^a$ and $(s_1^c, s_1^a) \in \alpha$.*
3.) *for every state s^c in I^c there exists a state s^a in I^a such that $(s^c, s^a) \in \alpha$.* □

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

Theorem 3.2 *Let S^c and S^a be transition systems such that $S^c \sqsubseteq_\alpha^{\varphi^c} S^a$. Let $\varphi^a \subseteq \Sigma^a$ and $\varphi \subseteq \Sigma^c$. If $\alpha^{-1}(\varphi^a) \subseteq \varphi^c \cap \varphi$, $S^a \models \Box\varphi^a$, and $I^c \subseteq \varphi^c$, then $S^c \models \Box(\varphi^c \cap \varphi)$.* □

Notice that definition 3.1 and theorem 3.1 can be obtained from definition 3.2 and theorem 3.2 by taking $\varphi^c = \Sigma^c$.

The advantage of definition 3.2 is that it allows abstractions with fewer transitions and less reachable states than it is allowed by definition 3.1. This is particularly important when we are seeking a method that automatically computes finite abstractions for analysis by model-checking techniques.

4 Computing Abstractions

In this section we consider the problem of computing an abstraction of a transition system S^c with respect to a relation α. Thus, consider a syntactic transition system $S^c = (C, \theta^c, \rho^c)$. Let α be a predicate whose set of free variables is $C \cup A$. Let $\alpha[S^c] = (A, \alpha[\theta^c], \alpha[\rho^c])$ where $\alpha[\theta^c]$ is given by $\exists C \cdot (\theta^c \wedge \alpha)$ and $\alpha[\rho^c]$ by $\exists C \exists C' \cdot (\alpha \wedge \alpha' \wedge \rho^c)$ and α' is obtained from α by substituting every variable $c \in C$ by c' and every variable $a \in A$ by a'. It can then be easily proved that $\alpha[S^c]$ is an abstraction of S^c. In case α is a function, $\alpha[S^c]$ is the smallest abstraction of S^c with respect to α.

Unfortunately, it is not possible in general to analyze $\alpha[S^c]$ 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 include in general variables that range over infinite domains.

In the sequel of this section we present a method for computing abstractions which avoids the quantification problem described above.

The elimination method Consider a transition relation given by a predicate $\rho(C, C')$ and consider an abstraction relation given by a predicate $\alpha(C, A)$, where A is a set of abstract variables. There is an obvious abstraction of ρ with respect to α which is the universal relation on Σ^a given by the everywhere true predicate. Let us denote it by $\mathcal{U}_A$. Of course one cannot use the abstract relation $\mathcal{U}_A$ to prove any interesting invariant, i.e., one which is not a tautology. One can, however, obtain a more interesting abstraction of $\rho(C, C')$ by eliminating transitions from $\mathcal{U}_A$. The following lemma states which transitions can be safely eliminated:

Lemma 4.1 *Let S^c, S^a be transition systems such that $S^c \sqsubseteq_\alpha^{\varphi^c} S^a$. Let s_0^a, s_1^a be abstract states. If $\alpha^{-1}(s_0^a) \Rightarrow \mathrm{WP}(R^c, \Sigma^c \setminus \alpha^{-1}(s_1^a))$ 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 $R^a \setminus \{(s_0^a, s_1^a)\}$.* □

In other words, if the concrete transition does not lead from a concrete state s_0^c with $(s_0^c, s_0^a) \in \alpha$ to a concrete state s_1^c with $(s_1^c, s_1^a) \in \alpha$, then we can safely eliminate the transition (s_0^a, s_1^a) from S^a. Notice that since the concrete system in general is infinite state the condition $\alpha^{-1}(s_0^a) \Rightarrow \mathrm{WP}(R^c, \Sigma^c \setminus \alpha^{-1}(s_1^a))$ has to be checked by means of a theorem prover. 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 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 as demonstrated by the case studies presented in section 6.

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 4.2 *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 4.3 *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 $R^c_1 \otimes \cdots \otimes R^c_m$, where each R^c_i is a relation on states over C_i. Assume also that each α_i can be considered as a function of C_i. Then, $\alpha[S^c] = \bigotimes_{i\leq m} \alpha_i[S^c]$.* □

In fact, it is often the case 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 4.4 *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.* □

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 $g(\vec{c}) \rightarrow \vec{c} := \vec{e}$. 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 $\vec{e}$ obtaining a new concrete transition τ' with $\tau' \equiv g(\vec{c}) \rightarrow \vec{c} := \vec{e'}$ and $\vec{e'} = \vec{e}[v_1/c_1, \cdots, v_n/c_n]$.

3.) Let $\beta(a)$ be $e_a[\vec{e'}/\vec{c}]$, 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 4.1 *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.*

$$\begin{array}{llll}
\textit{Transition system } S_1: & \tau_1: & pc_1 = l_{11} & \longrightarrow\ y_1 := y_2 + 1, pc_1 := l_{12} \\
& \tau_2: & pc_1 = l_{12} \wedge (y_2 = 0 \vee y_1 \leq y_2) & \longrightarrow\ pc_1 := l_{13} \\
& \tau_3: & pc_1 = l_{13} & \longrightarrow\ y_1 := 0, pc_1 := l_{11} \\
\textit{Transition system } S_2: & \tau_4: & pc_2 = l_{21} & \longrightarrow\ y_2 := y_1 + 1, pc_2 := l_{22} \\
& \tau_5: & pc_2 = l_{22} \wedge (y_1 = 0 \vee y_2 < y_1) & \longrightarrow\ pc_2 := l_{23} \\
& \tau_6: & pc_2 = l_{23} & \longrightarrow\ y_2 := 0, pc_2 := l_{21}
\end{array}$$

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 $pc_1^a = l_{11} \rightarrow a_1 := \text{false}, a_3 := \text{false}, pc_1^a := l_{11}$.

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 effect on a_3.

5 A PVS-based Implementation

We have implemented a tool that computes an abstraction of a network $S_1 \parallel \cdots \parallel S_n$, where $\parallel$ is the synchronous or asynchronous composition of transition systems. As a specification language for concrete systems we use a subset of the specification language of PVS. The produced abstract system, which has the same structure as the concrete one, is optionally described in PVS or SMV. The PVS theorem-prover is used to check the formulas generated by the elimination method. The user supplies a list of proof strategies which are used to check these formulas. Besides the proof strategies the user provides the following components:

1.) A PVS theory describing the concrete system. The user can choose whether to use the invariant to be checked during generation of the abstract system as given by definition 3.2. The user can also give a list of already proved invariants of the concrete system which are then used while constructing the abstract system.

2.) A PVS-theory describing the abstract state space and defining the abstraction function. We implemented a procedure that computes a first abstraction function which associates a boolean variable with every atomic formula of the form $r(x_1, \cdots, x_n)$ which appears in a guard, if there is at least one concrete variable among $x_1, \cdots, x_n$ that ranges over an infinite data domain, and which associates a boolean variable with every expression of the form $x' = \exp$ which appears in a concrete transition, if x ranges over an infinite data type and does not occur in exp.[2]

3.) A partition of the abstract variables. We have also implemented a procedure that computes a first partition of the abstract variables from the abstraction function. For an abstraction function α given by $\bigwedge_{a \in A} a \equiv e_a$, the computed partition is given by the reflexive transitive closure of the relation P_α such that $(a, a') \in P_\alpha$ iff there is a concrete variable that occurs in both e_a and $e_{a'}$. The user can then either refine or coarsen this partition. As shown in the previous section, finer partitions make it easier to compute the abstract system, but lead to more non-determinism.

[2]In [14] it is proposed to take as abstraction the partition of the concrete state space which is induced by the literals appearing in the guards. This abstraction is, however, generally too coarse.

The user can optionally provide a set of concrete variables for which our tool computes for each atomic operation on these variables an abstract operation. The computed abstract operations are then stored and reused each time an abstraction of the concrete system is computed unless the abstraction function has been modified.

The generation of the abstract system is *completely automatic* and compositional as we consider transition by transition. Thus, for each concrete transition we obtain an abstract transition (which might be non-deterministic). This is a very important property of our method, since it enables the debugging of the concrete system or alternatively enhancing the abstraction function. Indeed, the constructed abstract system may not satisfy the desired property, for three possible reasons: 1.) the concrete system does not satisfy the invariant, 2.) the abstraction function is not suitable for proving the invariant, or 3.) the provided proof strategies are too weak. Now, a model-checker such as SMV provides a trace as a counterexample, if the abstract system does not satisfy the abstract invariant. Since we have a clear correspondence between abstract and concrete transitions, we can examine the trace and find out which of the three reasons listed above is the case. In particular if the concrete system does not satisfy the invariant then we can transform the trace given by SMV to a concrete trace, thus generating a concrete counterexample.

6 Case Studies

6.1 The Bounded Retransmission Protocol

We consider the verification of the Bounded Retransmission protocol [24], BRP for short. The BRP protocol is an extension of the alternating bit protocol, where files of individual data are transmitted and the number of retransmissions per datum is bounded by a parameter. The protocol has been verified using theorem proving [15, 17, 16], where a large number of auxiliary invariants were needed. In the original formulation of the case study the requirements on the protocol are given by an abstract protocol, BRP-spec, and the task is to prove that the concrete protocol BRP simulates (is a refinement of) BRP-spec. In [14] it has been shown by computing an abstraction of BRP that the concrete protocol satisfies a set of temporal properties which have been extracted from the specification BRP-spec. There is, however, no guarantee that the checked temporal properties exclude all the behaviors excluded by BRP-spec: They do not exclude, for instance, that the protocol cheats both the sending and receiving clients by telling them that the transmission was successful while this is not the case. Using our method and its implementation we have been able to automatically prove that BRP implements BRP-spec.

Description of the protocol The BRP protocol accepts requirements REQ(f) from a producer to transmit a file f of data to a consumer. The protocol consists of a sender at the producer side and a receiver at the consumer side (see Figure 1). The sender transmits data

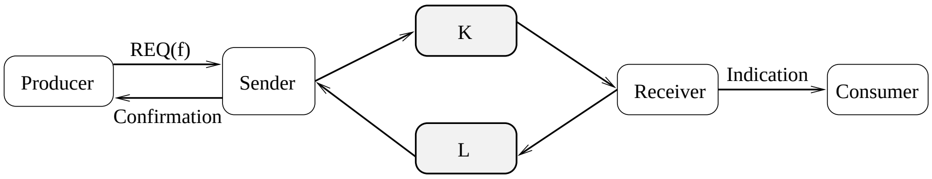

Figure 1: The Bounded Retransmission Protocol.

frames to the receiver via channel **K** and waits for acknowledgment via channel **L**. Since these channels may lose messages timeouts are used to identify a loss of messages. After sending a message, the sender waits for an acknowledgment. When the acknowledgment arrives, the sender either proceeds with the next message in the file, if there is one, or sends a confirmation message to the producer. If a timeout occurs before reception of an acknowledgment, the sender retransmits the same message. This procedure is repeated as often as specified by a parameter max. On its side, the receiver after acknowledging a message that is not the last one waits for further messages. If no new message arrives before a timeout, it concludes that there is a loss of contact to the sender and reports this to the consumer. The protocol is responsible for informing the producer whether the file has been transmitted correctly, whether transmission failed, or whether the last message is possibly lost. On the consumer side, the protocol passes data frames indicating whether the datum is the first one in a file, the last one, or whether it is an intermediate one.

Correctness criterion To reduce the problem of proving that BRP simulates BRP-spec to an invariance problem, we follow the same approach as in [17]. Thus, we consider a superposition of BRP and BRP-spec and prove that the superposed protocol, BRP^+, satisfies the invariance property $\Box \mathit{Safe}$, where *Safe* is a variable that is set to false as soon as BRP makes a transition that is not allowed by BRP-spec. It should be realized that BRP^+ contains for many variables of the protocol two different copies corresponding to the variable in BRP and BRP-spec, respectively. So, for instance there are two variables *file* and *afile* which correspond to the file to be sent and two variables *head* and *ahead* which correspond to the position of the data being processed in *file* and *afile*, respectively.

Verification of the protocol The BRP protocol represents a family of parameterized protocols. The parameters are the number of allowed retransmissions max, the length of a file *Last*, and finally, the data type $Data$. To obtain a finite abstraction of the protocol it is natural to eliminate these parameters by introducing additional nondeterminism. The abstraction we used is essentially obtained by the procedure we proposed in section 5. The only exception concerns an abstract variable that encodes the distance between the position variables *head* and *ahead*. A finite abstract system has been fully automatically produced within one hour and 20 minutes on an Ultra Sparc[3] and has been successfully model-checked by SMV within 2.11 seconds.

6.2 The Alternating Bit Protocol

We consider the Alternating Bit Protocol [1], whose task is to deliver a list of messages from a sender to a receiver without changing their order and in presence of channels that can lose and duplicate messages. The correct behavior of the protocol is specified by a finite state transition system ABP-spec. The implementing protocol, which we call ABP, consists of two components which communicate via two unbounded channels, *Sch* and *Rch*, which may lose or duplicate messages. Thus, ABP is infinite state for two reasons: 1.) the channels are unbounded and 2.) the message alphabet is infinite. Since one can deal with the second reason using Wolper's results on data-independence [29] we concentrate on the unboundedness of the channels and we assume that we have a finite message alphabet. In order to prove that ABP implements ABP-spec, Müller and Nipkow [26] use a clever finite abstraction which is based on the observation that the content of the channels is always of the form $m_1^* m_2^*$. Hence, one can obtain a finite state abstraction by merging adjacent identical messages.

We consider the same abstraction as Müller and Nipkow [26] with a different goal. Our goal is to compute a finite abstraction for ABP rather than to define an abstract system and to check a posteriori that it is an abstraction of ABP.

Let Mes be the finite set of messages the channels may contain. Moreover, let $Chcont = \{m_1^* m_2^* \mid m_1, m_2 \in Mes\}$. As abstract data type for channels we use the set $Chcont^a$ of sequences of messages that consists of the empty sequence ϵ, sequences of length one, and sequences $m \cdot m'$ of length two such that $m \neq m'$. Let $reduce : Chcont \longrightarrow Chcont^a$ be defined by $reduce(\epsilon) = \epsilon$,

[3]The implementation of [14] takes five hours for a version of the BRP with fewer variables.

$reduce(m{\cdot}u) = reduce(u)$, if $u = m{\cdot}u'$, and finally $reduce(m{\cdot}u) = m{\cdot}reduce(m'{\cdot}u')$, if $u = m'{\cdot}u'$ and $m \neq m'$. Abstract states have the same components as concrete states except that in abstract states the channels Sch and Rch range over $Chcont^a$ instead of over $Chcont$. It is then natural to take as an abstraction function the function α such that for every concrete state s and abstract variable a, $\alpha(s)(a) = s(a)$, if a is neither Sch nor Rch; and otherwise, $\alpha(s)(a) = reduce(s(a))$.

We have specified ABP in the specification language of PVS. We specified the channels as variables whose type is lists over Mes. A perfect sending of a message m over channel Sch is then modeled by concatenating m to Sch. A receipt of a message from channel Sch is modeled by removing the last element in the sequence. The loss and duplication of messages are modeled by send and receive transitions that do not change the content of the channels.

We have computed a finite abstraction of ABP using the abstraction function α for the case where $Mes = \{0, 1\}$. The abstract and concrete systems differ only in the sending and receiving transitions. For a concrete sending transition "$Sch := m \cdot Sch$" via channel Sch, we obtain the following abstract transitions

• $Sch = \epsilon \wedge m = 1 \rightarrow Sch := 1$	• $Sch = \epsilon \wedge m = 0 \rightarrow Sch := 0$
• $Sch = 1 \wedge m = 1 \rightarrow Sch := 1$	• $Sch = 1 \wedge m = 0 \rightarrow Sch := 0{\cdot}1$
• $Sch = 0 \wedge m = 1 \rightarrow Sch := 1{\cdot}0$	• $Sch = 0 \wedge m = 0 \rightarrow Sch := 0$
• $Sch = 1{\cdot}0 \wedge m = 1 \rightarrow Sch := 1{\cdot}0$	• $Sch = 0{\cdot}1 \wedge m = 0 \rightarrow Sch := 0{\cdot}1$

For a concrete receive transition "$Sch(s) \neq \epsilon \rightarrow Sch := reverse(cdr(reverse(Sch(s))))$" from channel Sch we obtain the following abstract transitions:

• $Sch = 1 \rightarrow Sch := \epsilon$	• $Sch = 0 \rightarrow Sch := \epsilon$
• $Sch = 1 \rightarrow Sch := 1$	• $Sch = 0 \rightarrow Sch := 0$
• $Sch = 0{\cdot}1 \rightarrow Sch := 0$	• $Sch = 1{\cdot}0 \rightarrow Sch := 1$
• $Sch = 0{\cdot}1 \rightarrow Sch := 0{\cdot}1$	• $Sch = 1{\cdot}0 \rightarrow Sch := 1{\cdot}0$

It should be realized that our abstraction function α is a total function on $Chcont$ but not on Mes^*. However, since it can checked that the ABP satisfies the invariant $\Box(Sch \in Chcont \wedge Rch \in Chcont)$, we deduce that the computed abstract system is indeed an abstraction of ABP. It is also worth mentioning that the difficulty in this exercise has been to find a suitable proof strategy that handles the involved lists and the recursively defined functions *reduce* and *reverse*. We used lemmata that state some properties relating *reverse* and *reduce* and the list operations *car*, *cdr*, and *cons*.

7 Conclusion

We have presented a method that automatically and compositionally computes abstractions for infinite state systems. The salient feature of our method, apart from being automatic, is that the generated abstract system has the same structure as the concrete one. This makes our method applicable for synchronous as well as asynchronous computation models. Moreover, this allows for the application of other techniques for reducing the state explosion problem as well as for debugging the concrete system. An other important feature of our method is that it is incremental, in the following sense. Assume that we computed an abstraction S^a of a system S with respect to an abstraction function α. Assume that we want to add new abstract variables to those in α, that is, we consider a new abstraction function α' which agrees with α on the old abstract variables. Then, all transitions which have been eliminated during the generation of S^a need not be considered for the construction of an abstraction of S with respect to α'. Furthermore, it is worth mentioning that, by the preservation results of [9, 20], one can use our method to compute a finite abstract system that can be used to verify every temporal property that does not include an existential quantification over computation paths.

Though our method is based on a rather simple mathematical background, we view it as practically important. We implemented the method using PVS to check the conditions generated by the elimination method. The generated abstract system is optionally described in the specification language of PVS or of SMV. Thus, our implementation presents a bridge between the PVS theorem prover and the SMV model-checker.

We applied our method on several examples. In addition to the BRP and the Alternating bit protocol described in this paper we verified the Bakery and Peterson's mutual exclusion algorithms, the reader-writer example, and a simplified version of the Futurebus+ cache coherence protocol. For all of these examples an abstract system has been fully automatically and efficiently generated.

The PVS specifications of the examples, the definitions of the abstraction functions as well as the produced abstract systems can be down-loaded from "http://www.informatik.uni-kiel.de/~yl/cav98.html".

Currently, we are integrating our implementation with our techniques for generating auxiliary invariants [2]. We are also planning to investigate methods to automate the debugging process of the concrete system. What is needed is a module that transforms a trace of the abstract system into a concrete one and then checks whether this trace corresponds to a computation of the concrete system.

References

[1] K.A. Barlett, R.A. Scantlebury, and P.T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5), 1969.

[2] S. Bensalem, Y. Lakhnech, and H. Saidi. Powerful techniques for the automatic generation of invariants. In R. Alur and T.A. Henzinger, editors, *8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 323–335. Springer-Verlag, 1996.

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

[4] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5), 1994.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM symp. of Prog. Lang.*, pages 238–252. ACM Press, 1977.

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

[7] D. Dams, R. Gerth, G. Döhmen, R. Herrmann, P. Kelb, and H. Pargmann. Model checking using adaptive state and data abstraction. In *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 455–467. Springer-Verlag, 1994.

[8] D. Dams, R. Gerth, and O. Grumberg. Generation of reduced models for checking fragments of CTL. In *Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 479–490. Springer-Verlag, 1993.

[9] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems: Abstractions preserving ACTL*, ECTL* and CTL*. In *Proceedings of the IFIP WG2.1/WG2.2/WG2.3 (PROCOMET)*. IFIP Transactions, North-Holland/Elsevier, 1994.

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

[11] J. Dingel and Th. Filkorn. Model checking for infinite state systems using data abstraction. In P. Wolper, editor, *Computer Aided Verification*, volume 939 of *LNCS*, pages 54–69. Springer-Verlag, 1995.

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

[13] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In *Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[14] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, 1997.

[15] F.F. Groote and J.C. van de Pol. A bounded retransmission protocol for large packets. In *A case study in computer checked verification*, Logic Group Preprint Series 100. Utrecht University, 1993.

[16] K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe, FME'96 Symposium*, volume 1051 of *Lecture Notes in Computer Science*. Springer-verlag, 1996.

[17] L. Helmink, M.P.A. Sellink, and F.W. Vaandrager. Proof-checking a data link protocol. Technical Report CS-R9420, Centrum voor Wiskunde en Informatica (CWI), March 1994.

[18] P. Kelb. *Abstraktionstechniken für Automatische Verifikationsmethoden*. PhD thesis, University of Oldenburg, 1995.

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

[20] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1), 1995.

[21] D. E. Long. *Model Checking, Abstraction, and Compositional Reasoning*. PhD thesis, Carnegie Mellon, 1993.

[22] Z. Manna and A. Pnueli. Completing the temporal picture. *Theoretical Computer Science*, 83(1):97–130, 1991.

[23] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[24] S. Mauw and G.J. Veltink editors. *Algebraic Specification of Communication Protocols*. Number 36 in Cambridge Tracts in Theoretical Computer Science. 1993.

[25] K.L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, Boston, 1993.

[26] 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 *Lecture Notes in Computer Science*, 1995.

[27] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 1995.

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

[29] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. 13th ACM Symp. Principles of Programming Languages*, pages 184–193. ACM Press, 1986.