Automating Modular Verification[⋆,⋆⋆]

Rajeev Alur[1] Luca de Alfaro[2] Thomas A. Henzinger[3] Freddy Y.C. Mang[2]

[1] Department of Computer and Information Science, University of Pennsylvania, and Bell Laboratories, Lucent Technologies. Email: alur@cis.upenn.edu

[2] Department of Electrical Engineering and Computer Sciences, University of California at Berkeley. Email: {dealfaro,fmang}@eecs.berkeley.edu

[3] Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, and Max-Planck Institute for Computer Science, Saarbrücken. Email: tah@eecs.berkeley.edu

Abstract. Modular techniques for automatic verification attempt to overcome the state-explosion problem by exploiting the modular structure naturally present in many system designs. Unlike other tasks in the verification of finite-state systems, current modular techniques rely heavily on user guidance. In particular, the user is typically required to construct module abstractions that are neither too detailed as to render insufficient benefits in state exploration, nor too coarse as to invalidate the desired system properties. In this paper, we construct abstract modules automatically, using reachability and controllability information about the concrete modules. This allows us to leverage automatic verification techniques by applying them in layers: first we compute on the state spaces of system components, then we use the results for constructing abstractions, and finally we compute on the abstract state space of the system. Our experimental results indicate that if reachability and controllability information is used in the construction of abstractions, the resulting abstract modules are often significantly smaller than the concrete modules and can drastically reduce the space and time requirements for verification.

1 Introduction

The single largest obstacle to the use of automatic methods in system verification is the state-explosion problem, which is the exponential increase in the number of system states caused by a linear increase in the number of system components

⋆ An abbreviated version of this paper appeared in the *Proceedings of the Tenth International Conference on Concurrency Theory* (CONCUR), Lecture Notes in Computer Science 1664, Springer-Verlag, 1999, pp. 82–97.

⋆⋆ This research was supported in part by the NSF CAREER award CCR-9734115, by the NSF CAREER award CCR-9501708, by the DARPA (NASA Ames) grant NAG2-1214, by the DARPA (Wright-Patterson AFB) grant F33615-98-C-3614, by the ARO MURI grant DAAH-04-96-1-0341, and by the Gigascale Silicon Research Center.

or variables. Modular verification techniques attempt to overcome the state-explosion problem by exploiting the modular structure naturally present in most system designs. The basic idea is to analyze each module of the system separately, perhaps together with an environment that represents a simplified model of the rest of the system; the results obtained for the individual modules are then combined into a single result about the compound system. Unlike other tasks in the verification of finite-state systems, which have been largely automated, current modular verification techniques still rely heavily on user guidance. Aside from deciding how to break up a system into modules, the user also has to specify the environment in which to study each module, which is usually a difficult task. In this paper, we present an approach to modular verification that is almost entirely automatic, leaving to the user only the task of specifying which variables of a module should be relevant to the other modules.

For each concrete module, we erase some variables to construct an abstract module, which has a smaller state space; the abstract module is then used to replace the concrete module in the verification process. If this approach is pursued naively, typically one of two things happens. Either one abstracts only variables that do not influence the property to be verified, which is certainly prudent but more often than not leads to insufficient savings, or one abstracts variables that do influence the desired property, in which case the abstract module may violate the property even though the concrete module does not. We take the second route, but use additional information about the concrete module in order to construct more useful abstractions than could be achieved by simply erasing variables. In the most basic variation of our method, we use reachability information about the concrete module when erasing variables to construct an abstraction. In a more advanced variation, we also use controllability information about the concrete module with respect to the desired property. In all cases, the additional information we use can be obtained fully automatically by looking only at individual modules and the property to be verified —there is no need to involve the compound system. Our experimental results indicate that the use of reachability and controllability information can lead to dramatic improvements in verification: the resulting module abstractions are often much smaller than the concrete modules yet still preserve the desired property.

Our model of computation is that of transition systems defined over finite sets of state variables. We describe systems as the parallel composition of one or more modules. A *module* $P = (\mathcal{V}_P, I_P, T_P)$ consists of a set $\mathcal{V}_P$ of variables, partitioned into *input* and *output variables*, an *initial predicate* I_P over $\mathcal{V}_P$ defining the initial states of P, and a *transition predicate* T_P over $\mathcal{V}_P \cup \mathcal{V}'_P$ defining the possible state transitions of P in terms of their source states (over $\mathcal{V}_P$) and destination states (over $\mathcal{V}'_P = \{x' \mid x \in \mathcal{V}_P\}$). We consider systems consisting of *non-blocking* modules, in which every state has a successor, regardless of the inputs to the module. The semantics of parallel composition is conjunction: $P \,||\, Q = (\mathcal{V}_P \cup \mathcal{V}_Q, I_P \wedge I_Q, T_P \wedge T_Q)$. For the sake of simplicity, in this paper we focus on Moore modules, for which the outputs during a transition depend only on the source state of the transition. Our approach can be adapted with only minor modifications to Mealy-type modules, such as the Reactive Modules of [AH96]. We consider the verification of invariance properties. An invariance property for the module P is specified by an *invariant predicate* φ over $\mathcal{V}_P$. The module P *satisfies* the invariant predicate φ, written $P \models \Box\varphi$, if P never leaves the set of states defined by φ.

Consider a system $P \,\|\, Q$ consisting of two modules P and Q, and a desired invariant predicate φ for $P \,\|\, Q$. To check if $P \,\|\, Q \models \Box\varphi$ without constructing the global state space of $P \,\|\, Q$, we can remove a subset $\mathcal{W}_P \subseteq \mathcal{V}_P$ of the variables of P and a subset $\mathcal{W}_Q \subseteq \mathcal{V}_Q$ of the variables of Q. Formally, the abstract module $(\exists\, \mathcal{W}_P.P) = (\mathcal{V}_P \backslash \mathcal{W}_P, \exists \mathcal{W}_P \,.\, I_P, \exists \mathcal{W}_P \exists \mathcal{W}'_P \,.\, T_P)$ is constructed by existentially quantifying the removed variables in the initial and transition predicates; we say that $(\exists\, \mathcal{W}_P.P)$ is obtained by *erasing* from P the variables in $\mathcal{W}_P$. Then we can attempt to use the following standard inference rule:

$$\frac{(\exists\, \mathcal{W}_P.P) \,\|\, (\exists\, \mathcal{W}_Q.Q) \models \Box\varphi}{P \,\|\, Q \models \Box\varphi} \tag{1}$$

This rule is sound, because every reachable state of the concrete system $P \,\|\, Q$ corresponds to a reachable state of the abstract system $(\exists\, \mathcal{W}_P.P) \,\|\, (\exists\, \mathcal{W}_Q.Q)$. The efficiency advantage of the rule stems from the fact that the premise involves fewer variables than the conclusion, reducing the size of the state space to be explored. However, the premise may fail even though the conclusion holds, because there may be many reachable states of the abstract system that do not correspond to reachable states of the concrete system. In fact, it is often impossible to choose suitable, reasonable large sets $\mathcal{W}_P$ and $\mathcal{W}_Q$, because modular designs aggregate naturally within each module only closely interdependent variables. By erasing such dependencies between variables, the number of transitions of the abstract system grows quickly to the point of violating all but trivial invariants. Our goal is to confine this growth in abstract transitions by utilizing additional information about the component modules P and Q.

More precisely, a state s of P can be written as a pair $s = (s_a, s_w)$, where s_a is a state over the set $\mathcal{V}_P \backslash \mathcal{W}_P$ of variables, and s_w is a state over the set $\mathcal{W}_P$ of erased variables. The abstract module $(\exists\, \mathcal{W}_P.P)$ contains a transition from source state s_a to destination state s'_a iff the concrete module P contains a transition from (s_a, s_w) to (s'_a, s'_w) for some s_w and s'_w. As a first improvement, we can include a transition from s_a to s'_a in the abstract module only if, for some s_w and s'_w, there is a transition from (s_a, s_w) to (s'_a, s'_w) in the concrete module *and* the state (s_a, s_w) is reachable in the concrete module. This is because it is certainly not useful to include abstract transitions that have no reachable concrete counterparts. To this end, we compute a predicate R_P over $\mathcal{V}_P$ that defines the reachable states of P. The predicate R_P can be computed using standard state-space exploration (symbolic or enumerative). Our experiments based on symbolic methods indicate that this computation is efficient, since the module P is considered in isolation. From the predicate R_P we construct the module $(P \,\&\, R_P) = (\mathcal{V}_P, I_P, T_P \wedge R_P)$, which is like P, except that it allows only transitions from reachable states. After erasing the variables in $\mathcal{W}_P$, we obtain the abstract module $(\exists \mathcal{W}_P.(P \,\&\, R_P))$. In a similar way, we compute the reachability predicate R_Q for Q and construct the abstract module $(\exists \mathcal{W}_Q.(Q \,\&\, R_Q))$. To complete the verification process, we then use the following rule:

$$\frac{(\exists \mathcal{W}_P.(P \,\&\, R_P)) \,\|\, (\exists \mathcal{W}_Q.(Q \,\&\, R_Q)) \models \Box\varphi}{P \,\|\, Q \models \Box\varphi} \tag{2}$$

Since the systems $P \,\|\, Q$ and $(P \,\&\, R_P) \,\|\, (Q \,\&\, R_Q)$ have the same reachable states, rule (2) is sound. As we shall see, unlike the simplistic rule (1), the improved rule (2) can often be successfully applied even when the sets $\mathcal{W}_P$ and $\mathcal{W}_Q$ include variables that contribute to ensure the invariant φ. Yet the savings in checking the premise of rule (2) are just as great as those for checking the premise of the earlier rule (1), because the same sets of variables are erased. In other words, $(\exists \mathcal{W}_P.(P \,\&\, R_P)) \,\|\, (\exists \mathcal{W}_Q.(Q \,\&\, R_Q))$ is a more accurate but no more detailed abstraction of $P \,\|\, Q$ than is $(\exists \mathcal{W}_P.P) \,\|\, (\exists \mathcal{W}_Q.Q)$. In our experiments we shall obtain dramatic results by applying rule (2) with the simple heuristics of erasing those variables that are not involved in the communication between P and Q. While reachability information is often used in algorithmic verification, the novelty of rule (2) consists in the use of such information for the modular construction of abstractions.

The effectiveness of a rule such as (1) or (2) is directly related to the number of variables that can be erased in a successful application of the rule. Rule (2) improves on rule (1) by using *reachability* information about the individual modules in the construction of the abstractions, which usually permits the erasure of more variables. It is possible to further improve on the rule (2) by using, in addition to reachability information, also information about the *controllability* of the individual modules with respect to the specification $\Box\varphi$. This improvement is based on the following observation. The predicate R_P used in (2) defines the reachable states of P *when P is in a completely general environment.* However, the module P may exhibit anomalous behaviors in a completely general environment; in particular, more states may be reachable under a completely general environment than under the specific environment provided by Q. Of course, we do not want to compute the reachable states of P when P is composed with Q: doing so would require the exploration of the state space of the global system $P \,\|\, Q$, which is exactly what our modular verification rules try to avoid. To study the module P under a suitable confining environment, while still avoiding the exploration of the global state space, we consider the module *P in the most general environment E that ensures the invariant φ*; that is, E is the least restrictive module such that $P \,\|\, E \models \Box\varphi$. In practice, we need not construct E explicitly, but compute only the predicate D_P that defines the set of reachable states of $P \,\|\, E$. Since E is more restrictive than the completely general environment, the predicate D_P is stronger than R_P, and the implication $D_P \to R_P$ holds. The algorithm for computing D_P follows from the standard game-theoretic algorithm for computing the set of states of the module P that are controllable with respect to the invariant φ; it can be implemented symbolically or enumeratively, with a time complexity that is linear in the size of the state space of P [Bee80].

This leads to the following modular verification rule:

$$\frac{\begin{array}{l}(I_P \wedge I_Q) \rightarrow (D_P \wedge D_Q) \\ P \,||\, (\exists\, \mathcal{W}_Q.(Q \,\&\, D_Q)) \models \Box D_P \\ Q \,||\, (\exists\, \mathcal{W}_P.(P \,\&\, D_P)) \models \Box D_Q\end{array}}{P \,||\, Q \models \Box\varphi} \qquad (3)$$

where $\mathcal{W}_P \subseteq \mathcal{V}_P$ and $\mathcal{W}_Q \subseteq \mathcal{V}_Q$. The soundness of this rule depends on an inductive argument, and it will be proved in detail in the paper. Essentially, the first premise ensures that the modules P and Q are initially in states satisfying $D_P \wedge D_Q$. The second premise shows that, as long as Q does not leave the set defined by D_Q, the module P will not leave the set defined by D_P; the third premise is symmetrical. As the implications $D_P \rightarrow \varphi$ and $D_Q \rightarrow \varphi$ hold, the three premises lead to the conclusion. The rule is in fact closely related to inductive forms of assume-guarantee reasoning [Sta85, AL95, AH96, McM97]. The use of the stronger predicates D_P and D_Q in the second and third premises of the rule (3) potentially enables the erasure of more variables compared to the earlier rule (2). However, in rule (3) this erasure can take place only on one side of the parallel composition operator or, in the case of multi-module systems, for all modules but one.

While automatic approaches to the construction of abstractions for model checking have been proposed, for example, in [Kur94, Dam96, GS97, CC99], these approaches do not exploit reachability and controllability information in a modular fashion. In particular, instead of the standard principle "first abstract, then model check the abstraction," our approach follows the more refined principle "first model check the components, then use this information to abstract, then model check the compound abstraction." In this way, our modular verification rules are doubly geared towards automatic verification methods: state-space exploration is used both to compute the reachability and controllability predicates, and to check all temporal premises (those which contain the $\models$ operator). It is worth pointing out that nontemporal premises would result in rules that are considerably less powerful. For example, suppressing variable erasures, the temporal premise $(P \,\&\, R_P) \,||\, (Q \,\&\, R_Q) \models \Box\varphi$ of rule (2) is weaker than the two nontemporal premises $I_P \wedge I_Q \rightarrow \varphi$ and $\varphi \wedge R_P \wedge T_P \wedge R_Q \wedge T_Q \rightarrow \varphi'$ would be (here, φ' results from φ by replacing all variables with their primed versions). Similarly, the second premise of rule (3) is weaker than the two nontemporal premises $I_P \wedge I_Q \rightarrow D_Q \wedge D_P$ and $D_P \wedge T_P \wedge D_Q \wedge T_Q \rightarrow D'_P$ would be. It is easy to find examples where our temporal premises apply, but their nontemporal counterparts do not.

The outline of the paper is as follows. After introducing preliminary definitions in Section 2, we develop the technical details of the proposed modular verification rules in Section 3. The verification rules have been implemented on top of the Mocha model checker [AHM+98], using BDD-based fixpoint algorithms for the computation of the reachability and controllability predicates. In Section 4 we discuss the implementation of the verification rules, and we describe the *script language* we devised in order to be able to experiment efficiently with various modular verification techniques. In Section 5 we present experimental results for three examples: a demarcation protocol used to maintain the consistency between distributed databases [BGM92], a token-ring arbiter, and a sliding-window protocol for data communication [Hol91]. We conclude the paper with some insights gathered in the course of the experimentation with the proposed verification rules.

2 Modules

Given a set $\mathcal{V}$ of typed variables with finite domain, a state s over $\mathcal{V}$ is an assignment for $\mathcal{V}$ that assigns to each $x \in \mathcal{V}$ a value $s[\![x]\!]$. We also denote by $\mathcal{V}' = \{x' \mid x \in \mathcal{V}\}$ the set obtained by priming each variable in $\mathcal{V}$. Given a predicate H over $\mathcal{V}$, we denote by H' the predicate obtained by replacing in H every $x \in \mathcal{V}$ with $x' \in \mathcal{V}'$. Given a set A and an element x, we often write $A \backslash x$ for $A \backslash \{x\}$, when this generates no confusion. A module $P = (\mathcal{C}_P, \mathcal{E}_P, I_P, T_P)$ consists of the following components:

1. A (finite) set $\mathcal{C}_P$ of *controlled variables,* each with finite domain, consisting of the variables whose values can be accessed and modified by P.
2. A (finite) set $\mathcal{E}_P$ of *external variables,* each with finite domain, consisting of the variables whose values can be accessed, but not modified, by P.
3. A *transition predicate* T_P over $\mathcal{C}_P \cup \mathcal{E}_P \cup \mathcal{C}'_P$.
4. An *initial predicate* I_P over $\mathcal{C}_P$.

We denote by $\mathcal{V}_P = \mathcal{C}_P \cup \mathcal{E}_P$ the set of variables mentioned by the module. Given a state s over $\mathcal{V}_P$, we write $s \models I_P$ if I_P is satisfied under the variable interpretation specified by s. Given two states s, s' over $\mathcal{V}_P$, we write $(s, s') \models T_P$ if predicate T_P is satisfied by the interpretation that assigns to $x \in \mathcal{V}_P$ the value $s[\![x]\!]$, and to $x' \in \mathcal{V}'_P$ the value $s'[\![x]\!]$. A module P is *non-blocking* if the predicate I_P is satisfiable, i.e., if the module has at least one initial state, and if the assertion $\forall \mathcal{V}_P \,.\, \exists \mathcal{C}'_P \,.\, T_P$ holds, so that every state has a successor. A *trace* of module P is a finite sequence of states $s_0, s_1, s_2, \ldots s_n \in \mathit{States}(\mathcal{V}_P)$, where $n \geq 0$ and $(s_k, s_{k+1}) \models T_P$ for all $0 \leq k < n$; the trace is *initial* if $s_0 \models I_P$. Two modules P and Q are *composable* if $\mathcal{C}_P \cap \mathcal{C}_Q = \emptyset$; in this case, their *parallel composition* $P \,||\, Q$ is defined as:

$$P \,||\, Q = \big(\mathcal{C}_P \cup \mathcal{C}_Q, (\mathcal{E}_P \cup \mathcal{E}_Q) \backslash (\mathcal{C}_P \cup \mathcal{C}_Q), I_P \wedge I_Q, T_P \wedge T_Q\big) \;.$$

Given a module P and a predicate H over $\mathcal{V}_P$, we denote by

$$(P \,\&\, H) = \big(\mathcal{C}_P, \mathcal{E}_P, I_P \wedge H, T_P \wedge H\big)$$

the module like P, except that only transitions from states that satisfy H are allowed. Given a module P and a set $\mathcal{W}$ of variables, we let

$$(\exists\, \mathcal{W}.P) = \big(\mathcal{C}_P \backslash \mathcal{W}, \mathcal{E}_P \backslash \mathcal{W}, \exists \mathcal{W} \,.\, I_P, \exists \mathcal{W}, \mathcal{W}' \,.\, T_P\big)$$

be the module obtained by *erasing* the variables $\mathcal{W}$ in P. Note that the module $(P \,\&\, H)$ can be blocking even if module P is non-blocking. On the other hand, the parallel composition of non-blocking modules is non-blocking, and a module obtained from a non-blocking module by erasing variables is also non-blocking.

A state of a module P is *reachable* if it appears in some initial trace of P. We denote by $\mathit{Reach}(P)$ the predicate defining the reachable states of P; this predicate can be compute using standard state-space exploration techniques [CES83]. Given a module P and a predicate φ, the relation $P \models \Box\varphi$ holds iff the implication $\mathit{Reach}(P) \rightarrow \varphi$ is valid. In this paper, we present modular techniques for verifying whether the relation $P_1 \,\|\, \cdots \,\|\, P_n \models \Box\varphi$ holds, where $P_1, P_2, \ldots, P_n$ are composable modules, for $n > 0$, and where φ is defined over the set of variables $\bigcup_{i=1}^{n} \mathcal{V}_{P_i}$. This verification problem is known as the *invariant verification problem,* and it is one of the most basic problems in formal verification.

3 Modular Rules for Invariant Verification

In this section, we present three modular rules for the verification of invariants; the rules are presented in order of increasing sophistication, and of increasing ability of successfully erasing variables. The first rule is a standard rule based on the construction of abstract modules:

$$\frac{(\exists\, \mathcal{W}_1 . P_1) \,\|\, \cdots \,\|\, (\exists\, \mathcal{W}_n . P_n) \models \Box\varphi}{P_1 \,\|\, \cdots \,\|\, P_n \models \Box\varphi} \tag{4}$$

The second rule is derived from the above rule, by using in the construction of the abstract modules also information about the reachable states of the concrete modules. The third rule constructs the abstract modules using both reachability and controllability information about the concrete modules.

3.1 Reachability-based abstractions

In order to improve the ability of rule (4) to successfully erase variables, we construct the abstract modules using reachability information about the concrete modules. Hence, we formulate the following modular verification rule:

$$\frac{(\exists\, \mathcal{W}_1 . (P_1 \,\&\, \mathit{Reach}(P_1))) \,\|\, \cdots \,\|\, (\exists\, \mathcal{W}_n . (P_n \,\&\, \mathit{Reach}(P_n))) \models \Box\varphi}{P_1 \,\|\, \cdots \,\|\, P_n \models \Box\varphi} \tag{5}$$

This rule is sound. The rule is also complete, since whenever the conclusion holds, the premise also does, with the choice $\mathcal{W}_1 = \cdots = \mathcal{W}_n = \emptyset$. Our experiments indicated that rule (5) is often surprisingly effective in enabling the successful erasure of variables, leading to dramatic savings in the space and time requirements of verification. We illustrate this with an example.

Example 1 This example is a simplified version of the token-ring example presented in Section 5. Consider a system composed of two modules P and Q that circulate a token through a 4-phase handshake protocol. The module P has controlled variables $\mathcal{C}_P = \{\mathit{grant}_1, \mathit{ack}_1, x_1, y_1, c_1\}$ and external variables

$\mathcal{E}_P = \{grant_2, ack_2\}$. All variables are boolean, except for c_1 that has domain $\{0,1,2,3\}$. The module Q is defined similarly, except that the subscripts 1 and 2 are exchanged. Intuitively, $grant_2$ and ack_1 form the handshake that passes a token from Q to P. Once the token arrives into P, it is stored first in x_1, then in y_1. The handshake variables $grant_1$ and ack_2 are used to pass the token back to Q. The variable c_1 is an auxiliary variable that records the number of tokens in P. The initial condition of P is $I_P : \neg ack_1 \wedge \neg grant_1 \wedge x_1 \wedge \neg y_1 \wedge (c_1 = 0)$; the initial condition of Q is $I_Q : \neg ack_2 \wedge \neg grant_2 \wedge \neg x_2 \wedge \neg y_2 \wedge (c_2 = 0)$, so that the token is initially in x_1. We present the transition predicate of P in guarded-commands notation, with the convention that the values of the variables not mentioned in the assignments are not modified, and that the command to be executed is chosen nondeterministically among those whose guards are true:

$$
\begin{array}{lll}
[\!] \ grant_2 \wedge \neg ack_1 \wedge \neg x_1 & \longrightarrow & ack_1' := \mathrm{T};\ x_1' := \mathrm{T};\ c_1' := (c_1 + 1) \mod 4 \\
[\!] \ \neg grant_2 \wedge ack_1 & \longrightarrow & ack_1' := \mathrm{F} \\
[\!] \ x_1 \wedge \neg y_1 & \longrightarrow & x_1' := \mathrm{F};\ y_1' := \mathrm{T} \\
[\!] \ \neg grant_1 \wedge \neg ack_2 \wedge y_1 & \longrightarrow & grant_1' := \mathrm{T};\ y_1' := \mathrm{F};\ c_1' := (c_1 - 1) \mod 4 \\
[\!] \ grant_1 \wedge ack_2 & \longrightarrow & grant_1' := \mathrm{F} \\
[\!] \ \mathrm{T} & \longrightarrow &
\end{array}
$$

The transition predicate of Q is identical, except that the subscripts 1 and 2 are exchanged. The invariant is $\varphi : [(c_1 + c_2) \mod 4 < 2]$, and states that there is at most one token. To verify that $P \,||\, Q \models \Box\varphi$, we can apply rule (5) with sets of erased variables $\mathcal{W}_P = \{x_1, y_1\}$ and $\mathcal{W}_Q = \{x_2, y_2\}$. Hence, we are able to erase all the variables that are not used for communication, and that do not appear in the invariant. The intuition is that, once the value of c_1 is known, the predicate

$$Reach(P) : \big(c_1 = 0 \wedge \neg x_1 \wedge \neg y_1\big) \vee \big(c_1 = 1 \wedge (x_1 \not\equiv y_1)\big) \vee \big(c_1 = 2 \wedge x_1 \wedge x_2\big)$$

provides sufficient information about the possible values of the erased variables x_1 and y_1 to enable an accurate computation of the successor states. In contrast, rule (4) does not enable the erasure of any variables. ∎

3.2 Controllability and reachability-based abstractions

Consider an instance $P_1 \,||\, \cdots \,||\, P_n \models \Box\varphi$ of the invariant verification problem, for $n \geq 1$. As mentioned in the introduction, the predicate $Reach(P_i)$ defines the reachable states of module P_i *when the module P_i is in a completely arbitrary environment,* for $1 \leq i \leq n$. However, a module may have many more reachable states when composed with a completely arbitrary environment, than when composed with the other modules of the system. To obtain more precise predicates, we consider the states of P_i that are reachable under the *most general environment under which P_i satisfies the specification $\Box\varphi$,* for $1 \leq i \leq n$. The idea is that, if the system has been properly designed, then the actual environment of P_i is a special case of this most general environment.

An *environment* for a module P is a non-blocking module E composable with P. Given a module P and a predicate φ, we denote by $\mathit{Envs}(P)$ the set of all environments of P, and we let $\mathit{Envs}_\varphi(P) = \{E \in \mathit{Envs}(P) \mid P \,\|\, E \models \Box\varphi\}$ the set of environments of P under which the specification $\Box\varphi$ holds. We define

$$\mathit{CR}(P, \varphi) = \bigvee_{E \in \mathit{Envs}_\varphi(P)} \exists(\mathcal{V}_E \backslash \mathcal{V}_P) \,.\, \mathit{Reach}(P \,\|\, E)$$

with the convention that $\mathit{CR}(P, \varphi) = \mathrm{F}$ if $\mathit{Envs}_\varphi(P) = \emptyset$. The predicate $\mathit{CR}(P, \varphi)$ defines the set of states of P that can be reached when P is composed with an environment under which $\Box\varphi$ holds. Denote by $\mathcal{V}_\varphi$ the variables occurring in φ. The following proposition gives some additional properties of the predicate $\mathit{CR}(P, \varphi)$.

Proposition 1 *Given a non-blocking module P and a predicate φ, the following assertions hold.*

1. *There is an environment $E \in \mathit{Envs}_\varphi(P)$ with $\mathcal{V}_E = \mathcal{V}_P \cup \mathcal{V}_\varphi$ such that $\mathit{CR}(P, \varphi) \equiv \exists(\mathcal{V}_\varphi \backslash \mathcal{V}_P) \,.\, \mathit{Reach}(P \,\|\, E)$.*
2. *The implications $\mathit{CR}(P, \varphi) \to \exists(\mathcal{V}_\varphi \backslash \mathcal{V}_P) \,.\, \varphi$ and $\mathit{CR}(P, \varphi) \to \mathit{Reach}(P)$ hold.*

Regarding the second assertion, note that in the introduction we implicitly assumed $\mathcal{V}_\varphi \subseteq \mathcal{V}_{P_i}$ for $1 \leq i \leq n$ for the sake of simplicity, while here we are only assuming the weaker $\mathcal{V}_\varphi \subseteq \bigcup_{i=1}^n \mathcal{V}_{P_i}$. We can then formulate the verification rule:

$$\frac{\begin{array}{l} \bigwedge_{i=1}^n I_{P_i} \;\;\to\;\; \bigwedge_{i=1}^n \mathit{CR}(P_i, \varphi) \\ P_i \,\|\, \left(\|_{j \in \{1,\ldots,n\} \backslash i} \left(\exists \mathcal{W}_j . (P_j \,\&\, \mathit{CR}(P_j, \varphi))\right)\right) \models \Box\, \mathit{CR}(P_i, \varphi) \qquad 1 \leq i \leq n \end{array}}{P_1 \,\|\, \cdots \,\|\, P_n \models \Box\varphi} \tag{6}$$

In the second premise of this rule, for $1 \leq i \leq n$, we cannot erase variables of P_i. In fact, the predicate $\mathit{CR}(P_i, \varphi)$ on the right hand side of $\models$ involves most of the variables in P_i, preventing their erasure. In the experiments described in Section 5, the systems were composed of two modules, and rule (5) performed better than rule (6), since in rule (5) the variables could be erased in both the composing modules. In systems composed of many modules, it is conceivable that the advantage derived from using the stronger predicates of rule (6) in all modules but one, thus possibly erasing more variables, outweighs the disadvantage of not being able to erase variables in one of the modules.

Proposition 2 *Rule (6) is sound. If $P_1, \ldots, P_n$ are non-blocking, rule (6) is also complete: if the conclusion holds, then the premises also hold for $\mathcal{W}_1 = \cdots = \mathcal{W}_n = \emptyset$.*

Proof. It suffices to consider the case $\mathcal{W}_1 = \cdots = \mathcal{W}_n = \emptyset$. To show that the rule is sound, we assume that its premises hold, and we prove by induction on $k \geq 0$ that, if $s_0, s_1, \ldots, s_k$ is an initial trace of $P_1 \,\|\, \cdots \,\|\, P_n$, then $s_i \models \mathit{CR}(P_j, \varphi)$ for all $0 \leq i \leq k$ and $1 \leq j \leq n$. The base case follows from the first premise of (6). For the induction step, assume that the assertion holds for k, and consider the assertion for $k+1$ for any j, with $1 \leq j \leq n$. The trace $s_0, s_1, \ldots, s_k, s_{k+1}$ is an initial trace of $P_j \,\|\, \left(\|_{l \in \{1,\ldots,n\} \setminus j} \,(P_j \,\&\, CR(P_j, \varphi))\right)$ Hence, we have that $s_{k+1} \models CR(P_j, \varphi)$, completing the induction step. From $\mathcal{V}_\varphi \subseteq \bigcup_{i=1}^n \mathcal{V}_{P_i}$ and from Proposition 1, part 2, we have that the implication $(\bigwedge_{i=1}^n CR(P_i, \varphi)) \rightarrow \varphi$ holds. This implication, together with the conclusion of the induction proof, leads to the desired result. The completeness of the rule follows by noticing that if $P_1 \,\|\, \cdots \,\|\, P_n \models \Box\varphi$, then by definition of $CR(\cdot, \varphi)$ we have $P_1 \,\|\, \cdots \,\|\, P_n \models \Box(CR(P_1, \varphi) \wedge \cdots \wedge CR(P_n, \varphi))$. ∎

To compute the predicate $CR(P, \varphi)$ given P and φ, we proceed in two steps. First, we compute the predicate $Ctr(P, \varphi)$ defining the set of states from which P is *controllable* with respect to the safety property $\Box\varphi$. The predicate $Ctr(P, \varphi)$ can be computed with a standard controllability algorithm [TW68, Bee80, RW87].

Algorithm 1
Input: Module P and predicate φ.
Output: Predicate $Ctr(P, \varphi)$ over $\mathcal{V}_P$.

Initialization: Let $\mathcal{F} = \mathcal{V}_\varphi \setminus \mathcal{V}_P$ and $U_0 = \exists \mathcal{F} \,.\, \varphi$.
Repeat: For $k \geq 0$, let $U_{k+1} = U_k \wedge \exists(\mathcal{E}'_P \cup \mathcal{F}') \,.\, \forall \mathcal{C}'_P \,.\, (T_P \rightarrow (U'_k \wedge \varphi'))$.
Until: $U_{k+1} \equiv U_k$.
Return: U_k.

The algorithm computes a sequence $U_0, U_1, U_2, \ldots$ of increasingly strong predicates. For $k \geq 0$, predicate U_k defines the states from which it is possible to control P to satisfy predicate φ for at least $k+1$ steps; note that the implication $U_k \rightarrow \exists \mathcal{F} \,.\, \varphi$ holds for $k \geq 0$. At each iteration $k \geq 0$, the algorithm lets U_{k+1} define the set of states from which the environment can choose the next value for the external variables, so that for all choice of the controlled variables, the successor states of the transitions satisfy U_k. The following algorithm computes the predicate $CR(P, \varphi)$, using the previous algorithm as a subroutine.

Algorithm 2
Input: Module P and predicate φ.
Output: Predicate $CR(P, \varphi)$ over $\mathcal{V}_P$.

Initialization: Let $\mathcal{F} = \mathcal{V}_\varphi \setminus \mathcal{V}_P$, and $V_0 = I_P \wedge \exists \mathcal{F} . \forall \mathcal{C}_P . \left(I_P \rightarrow (Ctr(P, \varphi) \wedge \varphi)\right)$.
Repeat: For $k \geq 0$, let

$$V'_{k+1} = V'_k \vee \exists \mathcal{V}_P \,.\, \left[V_k \wedge T_P \wedge \exists \mathcal{F}' \,.\, \forall \mathcal{C}'_P \,.\, \left(T_P \rightarrow (Ctr'(P, \varphi) \wedge \varphi)\right)\right] .$$

Until: $V_{k+1} \equiv V_k$.
Return: V_k.

For each $k \geq 0$, the predicate V_k over $\mathcal{V}_P$ defines the set of states of P that can be reached in k or less steps when P is composed with an environment E such that $P \,\|\, E \models \Box\varphi$. To understand how this predicate is computed, note that the predicate $\forall \mathcal{C}_P \,.\, (I_P \rightarrow (Ctr(P, \varphi) \wedge \varphi))$ defines the set of initial valuations for the variables in $\mathcal{E}_P \cup \mathcal{F}$ that are *safe for the environment:* if one such valuation is chosen by the environment, the system will start in a controllable state that satisfies φ, regardless of the valuation for the controlled variables in $\mathcal{C}_P$ chosen by the module P. The iteration step follows a similar idea. If V_k defines the set of current states, then the formula $K_1 : \exists \mathcal{V}_P \,.\, (V_k \wedge T_P)$ over $\mathcal{C}'_P$ defines the valuations for the controlled variables that can be chosen by P for the following state. The environment must choose a valuation for the variables in $\mathcal{E}'_P \cup \mathcal{F}'$ that ensures that, regardless of the valuation for $\mathcal{C}'_P$ chosen by the module, the successor state satisfies $Ctr'(P, \varphi) \wedge \varphi$. If V_k defines the set of current states, the set of such valuations for $\mathcal{E}'_P \cup \mathcal{F}'$ is defined by the formula

$$K_2 : \exists \mathcal{V}_P \,.\, \forall \mathcal{C}'_P \,.\, \big((V_k \wedge T_P) \rightarrow (Ctr'(P, \varphi) \wedge \varphi) \big).$$

It is then easy to see that the iteration step of Algorithm 2 can be written simply as $V'_{k+1} = K_1 \wedge \exists \mathcal{F}' . K_2$, so that K_1 constrains the next valuation of the controlled variables, and $\exists \mathcal{F}' \,.\, K_2$ constrains the next valuation of the external variables. Algorithms 1 and 2 can be implemented enumeratively or symbolically, and they have running time linear in $|States(\mathcal{V}_P \cup \mathcal{V}_\varphi)|$. In the next example, we see how rule (6) can enable the erasure of variables that could not be erased with rule (5).

Example 2 Consider the verification problem $P_1 \,||\, P_2 \models \Box\varphi$, where the invariant is $\varphi : \neg z_1 \wedge \neg z_2$. The modules have variables $\mathcal{C}_{P_i} = \{x_i, y_i, z_i\}$ and $\mathcal{E}_{P_i} = \{x_{2-i}, z_{2-i}\}$, for $1 \leq i \leq 2$; all the variables are boolean. Module P_1 has initial predicate $I_{P_1} : \neg x_1 \wedge \neg y_1 \wedge \neg z_1$, and has transition predicate $T_{P_1} : [x'_1 \equiv z_2] \wedge [(\neg x_1 \wedge \neg x_2) \rightarrow (y'_1 \equiv y_1)] \wedge [\neg y_1 \rightarrow (z'_1 \equiv z_1)]$. Module P_2 is defined in a symmetrical fashion. Informally, module P_1 behaves as follows. Initially, all variables are false. At each step, the new value for x_1 is the old value of z_2. If $x_1 \vee x_2$ holds, then y_1 can change value; otherwise, it retains its previous value. If y_1 is true, then z_1 can change value; otherwise, it retains its previous value. It is easy to check that $P_1 \,||\, P_2 \models \Box\varphi$ holds.

Consider module P_1. The states where $z_1 = \mathrm{T}$ or $z_2 = \mathrm{T}$ are obviously not controllable. The states where $y_1 = \mathrm{T}$ are also not controllable, since from these states module P_1 can reach a state where $z_1 = \mathrm{T}$ regardless of the values of the external variables x_2 and z_2. Likewise, the states where $x_1 = \mathrm{T}$ or $x_2 = \mathrm{T}$ are not controllable, since from these states the module can reach a state where $y_1 = \mathrm{T}$ regardless of the values of the external variables. The only controllable (and reachable) state of P_1 is thus defined by the predicate $CR(P_1, \varphi) : \neg x_1 \wedge \neg y_1 \wedge \neg z_1 \wedge \neg x_2 \wedge \neg z_2$. Predicate $CR(P_2, \varphi)$ is defined in a symmetrical fashion. The reachability predicates are given simply by $Reach(P_1) : \mathrm{T}$ and $Reach(P_2) : \mathrm{T}$.

Rule (6) can be applied by taking $\mathcal{W}_1 = \mathcal{W}_2 = \{y_1, y_2\}$. In fact, the composite module $P_1 \,||\, (\exists\, \mathcal{W}_2 . (P_2 \,\&\, CR(P_2, \varphi)))$ admits only the initial traces consisting of repetitions of the state $[x_1 = \mathrm{F}, y_1 = \mathrm{F}, z_1 = \mathrm{F}, x_2 = \mathrm{F}, z_2 = \mathrm{F}]$. This shows that the first premise holds; the case for the second premise is symmetrical. On the other hand, no variable can be successfully erased using rule (5). In fact, if we erase variable y_2, then the right hand side exhibits the initial trace s_0, s_1, where $s_0 : [x_1 = \mathrm{F}, y_1 = \mathrm{F}, z_1 = \mathrm{F}, x_2 = \mathrm{F}, z_2 = \mathrm{F}]$ and $s_1 : [x_1 = \mathrm{F}, y_1 = \mathrm{F}, z_1 = \mathrm{F}, x_2 = \mathrm{F}, z_2 = \mathrm{T}]$. This trace is possible because the state $t_0 : [x_1 = \mathrm{F}, z_1 = \mathrm{F}, x_2 = \mathrm{F}, y_2 = \mathrm{T}, z_2 = \mathrm{F}]$ over $\mathcal{V}_{P_2}$ is reachable, and hence it satisfies $Reach(P_2)$, and agrees with s_0 on the shared variables. The trace is then a consequence of the transition from t_0 to $t_1 : [x_1 = \text{F}, z_1 = \text{F}, x_2 = \text{F}, y_2 = \text{T}, z_2 = \text{T}]$ in P_2. A similar argument shows that it is not possible to erase the variable x_2. ∎

4 Implementation of the Verification Rules

We have implemented the algorithms described in this paper in the verification tool Mocha [AHM+98]. Mocha is an interactive verification environment and it enables, among other things, the verification of invariants using both enumerative and symbolic techniques; for the latter, it relies on the BDD package and image computation engine provided by VIS [BHSV+96], which we used in our implementation.

One important technique we use in the implementation of the rules is that, instead of computing the abstract modules explicitly, we compute them *implicitly*. The idea is as follows: suppose we are computing the reachable states of $(\exists\, \mathcal{W}_P . P) \,||\, (\exists\, \mathcal{W}_Q . Q)$. A straight-forward algorithm would be to first compute the two abstract modules, and then compute the reachable states of their composition. This is very inefficient in terms of the usage of space. Transition relations are usually presented as a list of conjuncts rather than as a single, larger conjunct. The explicit computation of the abstract modules would imply conjoining all the transition relations and building a monolithic one: if represented as a BDD, such a monolithic conjunct would often be prohibitively large. Instead, we quantify away the erased variables of the abstract modules only when necessary, as for example in the computation of the reachable states. For instance, we use the following symbolic algorithm to compute the reachable states of the parallel composition of two abstract modules:

Algorithm 3
Input: Modules P and Q, and variables $\mathcal{W}_P \subseteq \mathcal{V}_P \backslash \mathcal{C}_Q$ and $\mathcal{W}_Q \subseteq \mathcal{V}_Q \backslash \mathcal{C}_P$.
Output: $\mathit{Reach}((\exists\, \mathcal{W}_P . P) \,||\, (\exists\, \mathcal{W}_Q . Q))$.

Initialization: Let $U_0 = \exists (\mathcal{W}_P \cup \mathcal{W}_Q) \,.\, (I_P \wedge I_Q)$.
Repeat For $k \geq 0$, let $U'_{k+1} = U'_k \vee \exists (\mathcal{V}_P \cup \mathcal{V}_Q \cup \mathcal{W}'_P \cup \mathcal{W}'_Q) \,.\, (U_k \wedge T_P \wedge T_Q)$.
Until $U_{k+1} \equiv U_k$.
Return: U_k.

In the body of the loop, we rely on the early quantification algorithm in VIS to keep the intermediate BDDs small. With this scheme, a monolithic transition relation is never built. In particular, our implementation represents abstract modules as pairs consisting of a concrete module and of a list of variables that have been erased from it; such pairs are called *extended modules.*

In order to experiment with the verification rules proposed in this paper, we implemented a simple script language, called `sl`, built on top of Mocha and based on the Tcl/Tk API. The algorithms and methodologies described in this paper provide the theoretical basis of the commands provided by `sl`. The verification rules proposed in this paper can be implemented as `sl` scripts, and the language `sl` provides invaluable flexibility for experimenting with alternative forms of the rules. An example of script is the following, which verifies the correctness of the *demarcation protocol* using rule (5) (the demarcation protocol is described in Section 5.1).

```
read_module  demarc.rm
sl_em        P Q Spec
sl_reach     phi    em_Spec s
sl_reach     rp     em_P s
sl_restrict  Prest  rp em_P
sl_erase     Pabs   Prest  P/xw P/xr P/req1 P/grant1 P/req2 \
                           P/grant2 P/xlupd1 P/xlupd2 P/busy
sl_reach     rq     em_Q s
sl_restrict  Qrest  rq em_Q
sl_erase     Qabs   Qrest  Q/xw Q/xr Q/req1 Q/grant1 Q/req2 \
                           Q/grant2 Q/xlupd1 Q/xlupd2 Q/busy
sl_compose   Rabs   Pabs Qabs
sl_checkinv  Rabs   phi s
```

The command `read_module` parses the file `demarc.rm`, containing the declarations of the modules `P` and `Q`, composing the protocol, and `Spec`, whose reachable states constitute the invariant. The command `sl_em P Q Spec` builds the extended modules `em_P`, `em_Q`, and `em_Spec` from `P`, `Q`, and `Spec`; of course, these extended modules have empty sets of erased variables. The command `sl_reach phi em_Spec s` computes the predicate `phi` = *Reach*(`em_Spec`). The parameter `s` of this and other commands means "silent", i.e., no diagnostic information is printed. The rest of the script checks that $\texttt{em\_P} \| \texttt{em\_Q} \models \Box\texttt{phi}$ using rule (5). First, the commands `sl_reach` and `sl_restrict` are used to compute `rp` = *Reach*(`em_P`) and `Prest` = (`em_P` & `rp`). Then, the command `sl_erase` erases a specified list of variables from `Prest`, producing the extended module `Pabs`. As discussed earlier, the command `sl_erase` performs no actual computation, but simply adds the specified variables to the list of erased variables. The extended module `Qabs` is constructed in an analogous fashion. Finally, the command `sl_compose` composes `Pabs` and `Qabs` into a single extended module `Rabs`, which is checked against the specification $\Box$`phi` by command `sl_checkinv`.

Apart from these commands, we also have implemented commands including `sl_wcontr` and `sl_contrreach`, which together compute the predicate $CR(P, \varphi)$ given a module P and a predicate φ.

5 Experimental Results

To demonstrate the effectiveness of the proposed approach to modular verification, we compare the time and memory requirements of global state-space exploration with those of rule (5) and rule (6). We do not compare our approach with other modular verification approaches, since these approaches involve user intervention for the construction of the environments. By manually constructing the environments or the abstractions it is possible to improve on our results.

We consider three examples: a demarcation protocol used in distributed databases, a token-ring arbiter, and a sliding-window protocol for data communication. All experiments have been run on a 233 MHz Pentium® II PC with 128MB memory running Linux. We report the memory usage by giving the maximum number of BDD nodes used in any fixpoint computation or predicate; this is essentially the maximum number of BDD nodes used at any single time during verification. We also report the total CPU time; this time does not include swap activity (swap activity was in any case very limited for all examples reported). The automatic variable reordering heuristics of Mocha were enabled during the experiments. We remark that differences in time or memory usage of up to a factor of 2 are not significant, since they can easily be produced by a variation in the automatic choice of variable ordering.

5.1 Demarcation protocol

The *demarcation protocol* is a distributed protocol aimed at maintaining numerical constraints between data residing in distributed copies of a database, while minimizing the communication requirements [BGM92]. We consider an instance of the protocol that ensures that two databases, residing at sites 1 and 2, never sell more than the maximum available number of seats m aboard a plane. The variables x_1 and x_2 indicate the number of seats that have been sold at sites 1 and 2. Each site can both sell seats, and receive seats returned due to cancellations. In order to minimize the communication between two sites, each site $i = 1, 2$ maintains a variable xl_i indicating the maximum number of seats it can sell autonomously. If a site wishes to sell more seats than this limit allows, the site can send a request to the other site for more seats. Depending on the number of unsold seats, the other site has the option of rejecting the request, or of granting it in part or in full.

We model each site $i = 1, 2$ by a module P_i; the specification is $\Box[(x_1 \leq xl_1) \wedge (x_2 \leq xl_2) \wedge (xl_1 + xl_2 \leq m)]$. Each of P_1 and P_2 controls 20 variables, of which 8 are used for communication with the other module or appear in the invariant, and 12 are internal. Rule (5) enable the erasure of 9 of these 12 variables in each of P_1 and P_2; all of these variables are in the cone of influence of the specification. The table below compares the time and space requirements of global state space exploration with those of rules (5) and (6), for various values of m. To check the robustness of rule (5) against changes in the system model, we also wrote an alternative, somewhat more complex model for the demarcation protocol. For $m = 4$, the verification of the alternative model required 136156 BDD nodes and 2009 seconds with the global approach, and 18720 BDD nodes and 211 seconds with rule (5).

	Global		Rule (5)		Rule (6)	
m	BDD nodes	seconds	BDD nodes	seconds	BDD nodes	seconds
4	20881	97	2847	25	8695	75
6	64345	439	3338	40	20953	218
8	179364	1671	8367	81	43915	517
10	633102	8707	10475	112	65410	1878
12	space-out	—	15923	174	93295	1980
14	space-out	—	22205	300	145676	3913

5.2 Token ring arbiter

The second example is a synchronous token-ring arbiter. It involves a ring of m stations, around which a single *token* is passed unidirectionally through four-phase handshake protocols. The invariant states that there is at most one token present in the stations. A straightforward invariant would involve nearly all the variables in the system, and be rather tedious to write. Hence, we introduce *observer modules* that observe the number of tokens in the system. To enable the decomposition of the ring into two modules P_1 and P_2 representing the half-rings, we introduce two such observers, one for each half. We were able to erase all the variables used for the internal communications and state of the half-rings, even though these variables clearly belong to the cone of influence of the invariant. Each half ring controls $1 + 5m/2$ variables; of these, all but 4 could be erased. Below we compare the performance of global state-space exploration and of rules (5) and (6).

	Global		Rule (5)		Rule (6)	
m	BDD nodes	seconds	BDD nodes	seconds	BDD nodes	seconds
16	657	8	979	7	608	8
20	466	10	1619	9	308	12
24	1138	22	1297	26	473	20
28	1300	39	3486	24	519	29
32	1187	110	3190	143	772	143
36	1323	611	8230	242	1346	195

5.3 Sliding window protocol

Our last example is a classical sliding windows protocol from [Hol91], whose encoding is taken from the MOCHA distribution. The protocol uses send and receive windows of size m, and it is composed of a sender module and a receiver module. Our invariant states essentially that the windows are not over-run by the protocols. In both the sender and the receiver, roughly half of the variables not used for communication with the other module can be erased when applying our modular approach. The comparison between the performance of global state-space exploration and rules (5) and (6) is presented below.

	Global		Rule (5)		Rule (6)	
m	BDD nodes	seconds	BDD nodes	seconds	BDD nodes	seconds
3	8992	35	776	12	2443	33
4	11831	99	1723	41	3740	42
5	36359	1911	3843	84	8503	105
6	94684	4994	7048	156	18316	500
7	95667	2630	8282	513	22289	771
8	space-out	—	26611	1582	47605	6245

5.4 Discussion

The experimental results indicate that the proposed approach leads to a considerable reduction in the time and space requirements for the verification process.

In the examples we considered, we identified which variables could be erased in the application of rule (5) by a simple trial-and-error process. We can automate this process by providing, for each module P, a list $\{x_1, \ldots, x_k\} \subseteq \mathcal{C}_P$ of variables of P that are not part of the specification, and that are not accessed by other modules. We list first the variables that are more likely to be successfully erased: those that are more "internal" to the module, and that interact with fewer other variables. We then apply rule (5) successively with the sets of erased variables $\{x_1, \ldots, x_k\}$, $\{x_1, \ldots, x_{k-1}\}$, $\{x_1, \ldots, x_{k-2}\}$, $\ldots$, until the rule succeeds. This process is efficient in practice. In fact, the more variables are erased, the smaller is the state space of the abstract modules: hence if too many variables are erased, the rule will fail in a fraction of the time required for a successful proof.

In the three examples considered, the stronger reachability predicates used to construct the abstract modules in rule (6) did not enable the erasure of any additional variable. In the demarcation protocol and in the sliding window protocol examples, the ability of rule (5) to erase variables on both sides of the parallel composition operator led to superior results compared with rule (6). In the token ring arbiter example, module P_i has many more reachable states in a completely general environment than in an environment compatible with the specification, for $i = 1, 2$. Hence, the predicates $\mathit{Reach}(P_i)$ are much weaker (and take more time and space to compute) than the predicates $\mathit{CR}(P_i, \varphi)$, for $i = 1, 2$. For this reason, rule (6) performs better than rule (5) in this example.

If the premise of rule (5) does not hold, we can construct automatically a trace over the variables in $\bigcup_{i=1}^{n}(\mathcal{V}_{P_i} \setminus \mathcal{W}_i)$, leading to a state that does not satisfy φ. This trace is a trace over a partial set of system variables, and it does not necessarily correspond to a counterexample to the conclusion. If the first premise of rule (6) does not hold, then using facts about controllability we can reconstruct automatically a counterexample trace over the complete set of system variables. On the other hand, if the second premise of rule (6) does not hold for some $1 \leq i \leq n$, then we obtain a trace over a partial set of system variables that leads to a state t_i where the predicate $\mathit{CR}(P_i, \varphi)$ does not hold. From t_i, using facts about controllability we can again construct a trace over the complete set of system variables that leads to a state where φ does not hold. When confronted with a trace over a partial set of variables, we have taken the naïve approach of selectively un-erasing some variables in the premises, until either the premises became valid, or the design error could be identified.

References

[AH96] R. Alur and T.A. Henzinger. Reactive modules. In *Proc. 11th IEEE Symp. Logic in Comp. Sci.*, 1996.

[AHM+98] R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. Mocha: modularity in model checking. In *Computer Aided Verification*, LNCS 1427, pages 521–525. Springer-Verlag, 1998.

[AL95] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Trans. Prog. Lang. Sys.*, 17(3):507–534, 1995.

[Bee80] C. Beeri. On the membership problem for functional and multivalued dependencies in relational databases. *ACM Transactions on Database Systems*, 5:241–259, 1980.

[BGM92] D. Barbara and H. Garcia-Molina. The demarcation protocol: a technique for maintaining linear arithmetic constraints in distributed database systems. In *EDBT'92: 3rd International Conference on Extending Database Technology*, LNCS 580, pages 373–388. Springer-Verlag, 1992.

[BHSV+96] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. Shiple, G. Swamy, and T. Villa. VIS: A system for verification and synthesis. In *Computer Aided Verification*, LNCS 1102, pages 428–432. Springer-Verlag, 1996.

[CC99] P. Cousot and R. Cousot. Refining model checking by abstract interpretation *Automated Software Engineering Journal*, 6(1):69–95, 1999.

[CES83] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic. In *Proc. 10th ACM Symp. Princ. of Prog. Lang.*, 1983.

[Dam96] D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Technical University of Eindhoven, 1996.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification*, LNCS. Springer-Verlag, 1997.

[Hol91] G.J. Holzman. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[HQRT98] T.A. Henzinger, S. Qadeer, S.K. Rajamani, and S. Tasiran. An assume-guarantee rule for checking simulation. In *Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design (FMCAD 1998)*, LNCS 1522, pages 421–432. Springer-Verlag, 1998.

[Kur94] R.P. Kurshan. *Computer-aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[McM97] K.L. McMillan. A compositional rule for hardware design refinement. In *Computer Aided Verification*, LNCS 1254, pages 24–35. Springer-Verlag, 1997.

[RW87] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete-event processes. *SIAM Journal of Control and Optimization*, 25:206–230, 1987.

[Sta85] E.W. Stark. A proof technique for rely/guarantee properties. In *Proc. of 5th Conference on Foundations of Software Technology and Theoretical Computer Science*, LNCS 206, pages 369–391. Springer-Verlag, 1985.

[TW68] J.W. Thatcher and J.B. Wright. Generalized finite-automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2:57–81, 1968.