Detecting Errors Before Reaching Them*

Luca de Alfaro, Thomas A. Henzinger, and Freddy Y.C. Mang

Department of Electrical Engineering and Computer Sciences
University of California at Berkeley, CA 94720-1770, USA
{dealfaro,tah,fmang}@eecs.berkeley.edu

Abstract. Any formal method or tool is almost certainly more often applied in situations where the outcome is failure (a counterexample) rather than success (a correctness proof). We present a method for symbolic model checking that can lead to significant time and memory savings for model-checking runs that fail, while occurring only a small overhead for model-checking runs that succeed. Our method discovers an error as soon as it cannot be prevented, which can be long before it actually occurs; for example, the violation of an invariant may become unpreventable many transitions before the invariant is violated.
The key observation is that "unpreventability" is a local property of a single module: an error is unpreventable in a module state if no environment can prevent it. Therefore, unpreventability is inexpensive to compute for each module, yet can save much work in the state exploration of the global, compound system. Based on different degrees of information available about the environment, we define and implement several notions of "unpreventability," including the standard notion of uncontrollability from discrete-event control. We present experimental results for two examples, a distributed database protocol and a wireless communication protocol.

1 Introduction

It has been argued repeatedly that the main benefit of formal methods is *falsification*, not verification; that formal analysis can only demonstrate the *presence* of errors, not their absence. The fundamental reason for this is, of course, that mathematics can be applied, inherently, only to an abstract formal model of a computing system, not to the actual artifact. Furthermore, even when a formal model is verified, the successful verification attempt is typically preceded by many iterations of unsuccessful verification attempts followed by model revisions. Therefore, in practice, every formal method and tool is much more often applied in situations where the outcome is failure (a counterexample), rather than success (a correctness proof).

Yet most optimizations in formal methods and tools are tuned towards success. For example, consider the use of BDDs and similar data structures in

* In *Proceedings of CAV 2000,* LNCS, Springer-Verlag, 2000. This research was supported in part by the DARPA (NASA) grant NAG2-1214, the SRC contract 99-TJ-683.003, the MARCO grant 98-DT-660, the DARPA (MARCO) grant MDA972-99-1-0001, and the NSF CAREER award CCR-9501708.

model checking. Because of their canonicity, BDDs are often most effective in applications that involve equivalence checking between complex boolean functions. Successful model checking is such an application: when the set of reachable states is computed by iterating image computations, successful termination is detected by an equivalence check (between the newly explored and the previously explored states). By contrast, when model checking fails, a counterexample is detected before the image iteration terminates, and other data structures, perhaps noncanonical ones, may be more efficient [BCCZ99]. To point out a second example, much ink has been spent discussing whether "forward" or "backward" state exploration is preferable (see, e.g., [HKQ98]). If we expect to find a counterexample, then the answer seems clear but rarely practiced: the simultaneous, dove-tailed iteration of images and pre-images is likely to find the counterexample by looking at fewer states than either unidirectional method. Third, in compositional methods, the emphasis is almost invariably on how to decompose correctness proofs (see, e.g., [HQR98]), not on how to find counterexamples by looking at individual system components instead of their product. In this paper, we address this third issue.

Consider a system with two processes. The first process waits on a binary input from the second process; if the input is 0, it proceeds correctly; if the input is 1, it proceeds for n transitions before entering an error state. Suppose the second process may indeed output 1. By global state exploration (forward or backward), $n + 1$ iterations are necessary to encounter the error and return a counterexample. This is despite the fact that things may go terribly wrong, without chance of recovery, already in the first transition. We propose to instead proceed in two steps. First, we compute on each individual process (i.e., typically on a small state space) the states that are *controllable* to satisfy the requirements. In our example, the initial state is controllable (because the environment may output 0 and avoid an error), but the state following a single 1 input is not (no environment can avoid the error). Second, on the global state space, we restrict search to the controllable states, and report an error as soon as they are left. In our example, the error is reported after a single image (or pre-image) computation on the global state space. (A counterexample can be produced from this and the precomputed controllability information of the first process.) Note that both steps are fully automatic. Moreover, the lower number of global iterations usually translates into lower memory requirements, because BDD size often grows with the number of iterations. Finally, when no counterexample is found, the overhead of our method is mostly in performing step 1, which does not involve the global state space and therefore is usually uncritical.

We present several refinements of this basic idea, and demonstrate the efficiency of the method with two examples, a distributed database protocol and a wireless communication protocol. In the first example, there are two sites that can sell and buy back seats on the same airplane [BGM92]. The protocol aims at ensuring that no more seats are sold than the total available, while enabling the two sites to exchange unsold seats, in case one site wishes to sell more seats than initially allotted. The second example is from the *Two-Chip Intercom* (TCI)

project of the Berkeley Wireless Research Center [BWR]. The TCI network is a wireless local network which allows approximately 40 remotes, one for each user, to transmit voice with point-to-point and broadcast communication. The operation of the network is coordinated by a base station, which assigns channels to the users through a TDMA scheme. In both examples, we first found errors that occurred in our initial formulation of the models, and then seeded bugs at random. Our methods succeeded in reducing the number of global image computation steps required for finding the errors, often reducing the maximum number of BDD nodes used in the verification process. The methods are particularly effective when the BDDs representing the controllable states are small in comparison to the BDD representing the set of reachable states.

To explain several fine points about our method, we need to be more formal. To study the controllability of a module P, we consider a game between P and its environment: the moves of P consist in choosing new values for the variables controlled by P; the moves of the environment of P consist in choosing new values for the input variables of P. A state s of P is *controllable with respect to the invariant* $\Box\varphi$ if the environment has a strategy that ensures that φ always holds. Hence, if a state s is not controllable, we know that P from s can reach a $\neg\varphi$-state, regardless of how the environment behaves. The set C_P of controllable states of P can be computed iteratively, using the standard algorithm for solving safety games, which differs from backward reachability only in the definition of the pre-image operator. Symmetrically, we can compute the set C_Q of controllable states of Q w.r.t. $\Box\varphi$. Then, instead of checking that $P \,\|\, Q$ stays within the invariant $\Box\varphi$, we check whether $P \,\|\, Q$ stays within the stronger invariant $\Box(C_P \wedge C_Q)$. As soon as $P \,\|\, Q$ reaches a state s that violates a controllability predicate, say, C_P, by retracing the computation of C_P, taking into account also Q, we can construct a path of $P \,\|\, Q$ from s to a state t that violates the specification φ. Together with a path from an initial state to s, this provides a counterexample to $\Box\varphi$. While the error occurs only at t, we detect it already at s, as soon as it cannot be prevented. The method can be extended to arbitrary LTL requirements.

The notion of controllability defined above is classical, but it is often not strong enough to enable the early detection of errors. To understand why, consider an invariant that relates a variable x in module P with a variable y in module Q, for example by requiring that $x = y$, and assume that y is an input variable to P. Consider a state s, in which module P is about to change the value of x without synchronizing this change with Q. Intuitively, it seems obvious that such a change can break the invariant, and that the state should not be considered controllable (how can Q possibly know that this is going to happen, and change the value of y correspondingly?). However, according to the classical definition of controllability, the state s is controllable: in fact, the environment has a move (changing the value of y correspondingly) to control P. This example indicates that in order to obtain stronger (and more effective) notions of controllability, we need to compute the set of controllable states by taking into account the real capabilities of the other modules composing the system. We

introduce three such stronger notions of controllability: constrained, lazy, and bounded controllability. Our experimental results demonstrate that there is a distinct advantage in using these stronger notions of controllability.

Lazy controllability can be applied to systems in which all the modules are *lazy,* i.e., if the modules always have the option of leaving unchanged the values of the variables they control [AH99]. Thus, laziness models the assumption of speed independence, and is used heavily in the modeling of asynchronous systems. If the environment is lazy, then there is no way of preventing the environment from always choosing its "stutter" move. Hence, we can strengthen the definition of controllability by requiring that the stutter strategy of the environment, rather than an arbitrary strategy, must be able to control. In the above example, the state s of module P is clearly not *lazily controllable,* since a change of x cannot be controlled by leaving y unchanged. *Constrained controllability* is a notion of controllability that can be used also when the system is not lazy. Constrained controllability takes into account, in computing the sets of controllable states, which moves are possible for the environment. To compute the set of *constrainedly controllable states* of a module P, we construct a transition relation that constrains the moves of the environment. This is done by automatically abstracting away from the transition relations of the other modules the variables that are not shared by P. We then define the controllable states by considering a game between P and a so constrained environment. Finally, *bounded controllability* is a notion that can again be applied to any system, and it generalizes both lazy and constrained controllability. It considers environments that have both a set of *unavoidable moves* (such as the lazy move for lazy systems), and *possible moves* (by considering constraints to the moves, similarly to constrained controllability). We also introduce a technique called *iterative strengthening,* which can be used to strengthen any of these notions of controllability. In essence, it is based on the idea that a module, in order to control another module, cannot use a move that would cause it to leave its own set of controllable states.

It is worth noting that the technqiues developed in this paper can also be used in an informal verification environment: after computing the uncontrollability states for each of the components, one can *simulate* the design and check if any of these uncontrollable states can be reached. This is similar to the techniques *retrograde analysis* [JSAA97], or *target enlargement* [YD98] in simulation. The main idea of retrograde analysis and target enlargement is that the set of states that violate the invariants are "enlarged" with their preimages, and hence the chances of hitting this enlarged set is increased. Our techniques not only add modularity in the computation of target enlargemen, they also allow one to detect the violation of *liveness* properties through simulation.

The algorithmic control of reactive systems has been studied extensively before (see, e.g., [RW89,EJ91,Tho95]). However, the use of controllability in automatic verification is relatively new (see, e.g., [KV96,AHK97,AdAHM99]). The work closest to ours is [ASSSV94], where transition systems for components are minimized by taking into account if a state satisfies or violates a given CTL property under all environments. In [Dil88], autofailure captures the concept

that no environment can prevent failure and is used to compare the equivalence of asynchronous circuits.

2 Preliminaries

Given a set $\mathcal{V}$ of typed variables, 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 indicate with $\mathit{States}(\mathcal{V})$ be the set of all states over $\mathcal{V}$, and with $\mathcal{P}(\mathcal{V})$ the set of predicates over $\mathcal{V}$. Furthermore, we denote by $\mathcal{V}' = \{x' \mid x \in \mathcal{V}\}$ the set obtained by priming each variable in $\mathcal{V}$. Given a predicate $H \in \mathcal{P}(\mathcal{V})$, we denote by $H' \in \mathcal{P}(\mathcal{V}')$ the predicate obtained by replacing in H every $x \in \mathcal{V}$ with $x' \in \mathcal{V}'$. 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* $T_P \in \mathcal{P}(\mathcal{C}_P \cup \mathcal{E}_P \cup \mathcal{C}'_P)$.
4. An *initial predicate* $I_P \in \mathcal{P}(\mathcal{C}_P)$.

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 $s_0 \models I_P$. We denote by $\mathcal{L}(P)$ the set of initial traces of module P. For a module P, we consider specifications expressed by linear-time temporal logic (LTL) formulas whose atomic predicates are in $\mathcal{P}(\mathcal{V}_P)$. As usual, given an LTL formula φ, we write $P \models \varphi$ iff $\sigma \models \varphi$ for all $\sigma \in \mathcal{L}(P)$.

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: $P \,\|\, Q = \big(\mathcal{C}_P \cup \mathcal{C}_Q, (\mathcal{E}_P \cup \mathcal{E}_Q) \setminus (\mathcal{C}_P \cup \mathcal{C}_Q), I_P \wedge I_Q, T_P \wedge T_Q\big)$. Note that composition preserves non-blockingness.

We assume that all predicates are represented in such a way that boolean operations and existential quantification of variables are efficiently computable. Likewise, we assume that satisfiability of all predicates can be checked efficiently. *Binary decision diagrams* (BDDs) provide a suitable representation [Bry86].

Controllability. We can view the interaction between a module P and its environment as a game. At each round of the game, the module P chooses the next values for controlled variables $\mathcal{C}_P$, while the environment chooses the next values for the external variables $\mathcal{E}_P$. Given an LTL specification φ, we say that a state s of P is *controllable* with respect to φ if the environment can ensure that all

traces from s satisfy φ. To formalize this definition, we use the notion of *strategy*. A module strategy π for P is a mapping $\pi : \mathit{States}(\mathcal{V}_P)^+ \mapsto \mathit{States}(\mathcal{C}_P)$ that maps each finite sequence $s_0, s_1, \ldots, s_k$ of module states into a state $\pi(s_0, s_1, \ldots, s_k)$ such that $(s_k, \pi(s_0, s_1, \ldots, s_k)) \models T_P$. Similarly, an environment strategy η for P is a mapping $\eta : \mathit{States}(\mathcal{V}_P)^+ \mapsto \mathit{States}(\mathcal{E}_P)$ that maps each finite sequence of module states into a state specifying the next values of the external variables. Given two states s_1 and s_2 over two disjoint sets of variables $\mathcal{V}_1$ and $\mathcal{V}_2$, we denote by $s_1 \bowtie s_2$ the state over $\mathcal{V}_1 \cup \mathcal{V}_2$ that agrees with s_1 and s_2 over the common variables. With this notation, for all $s \in \mathit{States}(\mathcal{V}_P)$ and all module strategies π and environment strategies η, we define $\mathit{Outcome}(s, \pi, \eta) \in \mathit{States}(\mathcal{V}_P)^\omega$ to be the trace $s_0, s_1, s_2, \ldots$ defined by $s_0 = s$ and by $s_{k+1} = \pi(s_0, s_1, \ldots, s_k) \bowtie \eta(s_0, s_1, \ldots, s_k)$. Given an LTL formula φ over $\mathcal{V}_P$, we say that a state $s \in \mathit{States}(\mathcal{V}_P)$ is *controllable with respect to* φ iff there is an environment strategy η such that, for every module strategy π, we have $\mathit{Outcome}(s, \pi, \eta) \models \varphi$. We let $\mathit{Ctr}(P, \varphi)$ be the predicate over $\mathcal{V}_P$ defining the set of states of P controllable with respect to φ.

Roughly, a state of P is controllable w.r.t. φ exactly when there is an environment E for P such that all paths from s in $P \parallel E$ satisfy φ. Since in general E can contain variables not in P, to make the above statement precise we need to introduce the notion of *extension* of a state. Given a state s over $\mathcal{V}$ and a state t over $\mathcal{U}$, with $\mathcal{V} \subseteq \mathcal{U}$, we say that t is an *extension* of s if $s[\![x]\!] = t[\![x]\!]$ for all $x \in \mathcal{V}$. Then, there is module E composable with P such that all paths from extensions of s in $P \parallel E$ satisfy φ iff $s \in \mathit{Ctr}(P, \varphi)$ [AdAHM99].

3 Early Detection of Invariant Violation

Forward and backward state exploration. Given a module R and a predicate φ over $\mathcal{V}_R$, the problem of invariant verification consists in checking whether $R \models \Box\varphi$. We can solve this problem using classic forward or backward state exploration. Forward exploration starts with the set of initial states of R, and iterates a post-image computation, terminating when a state satisfying $\neg\varphi$ has been reached, or when the set of reachable states of R has been computed. In the first case we conclude $R \not\models \Box\varphi$; in the second, $R \models \Box\varphi$. Backward exploration starts with the set $\neg\varphi$ of states violating the invariant, and iterates a pre-image computation, terminating when a state satisfying I_R has been reached, or when the set of all states that can reach $\neg\varphi$ has been computed. Again, in the first case we conclude $R \not\models \Box\varphi$ and in the second $R \models \Box\varphi$. If the answer to the invariant verification question is negative, these algorithms can also construct a counterexample $s_0, \ldots, s_m$ of minimal length leading from $s_0 \models I_R$ to $s_m \models \neg\varphi$, and such that for $0 \leq i < m$ we have $(s_i, s_{i+1}) \models T_R$. If our aim is to find counterexamples quickly, an algorithm that alternates forward and backward reachability is likely to explore fewer states than the two unidirectional algorithms. The algorithm alternates post-image computations starting from I_R with pre-image computations starting from $\neg\varphi$, terminating as soon as the post and pre-images intersect, or as soon as a fixpoint is reached. We denote any of these three algorithms (or

variations thereof) by $\mathit{InvCheck}(R, \varphi)$. We assume that $\mathit{InvCheck}(R, \varphi)$ returns answer YES or NO, depending on whether $R \models \Box\varphi$ or $R \not\models \Box\varphi$, along with a counterexample in the latter case.

Controllability and early error detection. Given $n > 1$ modules P_1,P_2,...,P_n and a predicate $\varphi \in \mathcal{P}(\bigcup_{i=1}^{n} \mathcal{V}_{P_i})$, the modular version of the invariant verification problem consists in checking whether $P_1 \parallel \cdots \parallel P_n \models \Box\varphi$. We can use the notion of controllability to try to detect a violation of the invariant φ in fewer iterations of post or pre-image computation than the forward and backward exploration algorithms described above. The idea is to pre-compute the states of each module $P_1, \ldots, P_n$ that are controllable w.r.t. $\Box\varphi$. We can then detect a violation of the invariant as soon as we reach a state s that is not controllable for some of the modules, rather than waiting until we reach a state actually satisfying $\neg\varphi$. In fact, we know that from s there is a path leading to $\neg\varphi$ in the global system: for this reason, if a state is not controllable for some of the modules, we say that the state is *doomed.*

To implement this idea, let $R = P_1 \parallel \cdots \parallel P_n$, and for $1 \leq i \leq n$, let $\mathit{abs}_i(\varphi) = \exists(\mathcal{V}_R \setminus \mathcal{V}_{P_i}) \,.\, \varphi$ be an approximation of φ that involves only the variables of P_i; note that $\varphi \rightarrow \mathit{abs}_i(\varphi)$. For each $1 \leq i \leq n$, we can compute the set $\mathit{Ctr}(P_i, \Box\mathit{abs}_i(\varphi))$ of controllable states of P_i w.r.t. $\Box\mathit{abs}_i(\varphi)$ using a classical algorithm for safety games. For a module P, the algorithm uses the *uncontrollable predecessor operator* $\mathrm{UPre}_P : \mathcal{P}(\mathcal{V}_P) \mapsto \mathcal{P}(\mathcal{V}_P)$, defined by

$$\mathrm{UPre}_P(X) = \forall\mathcal{E}'_P \,.\, \exists\mathcal{C}'_P \,.\, (T_P \wedge X') \,.$$

The predicate $\mathrm{UPre}_P(X)$ defines the set of states from which, regardless of the move of the environment, the module P can resolve its internal nondeterminism to make X true. Note that a quantifier switch is required to to compute the uncontrollable predecessors, as opposed to the computations of pre-images and post-images, where where only existential quantification is required. For a module P and an invariant $\Box\varphi$, we can compute the set $\mathit{Ctr}(P, \Box\varphi)$ of controllable states of P with respect to $\Box\varphi$ by letting $U_0 = \neg\varphi$, and for $k > 0$, by letting

$$U_k = \neg\varphi \vee \mathrm{UPre}_P(U_{k-1}), \tag{1}$$

until we have $U_k \equiv U_{k-1}$, at which point we have $\mathit{Ctr}(P, \Box\varphi) = \neg U_k$. For $k \geq 0$ the set U_k consists of the states from which the environment cannot prevent module P from reaching $\neg\varphi$ in at most k steps. Note that for all $1 \leq i \leq n$, the computation of $\mathit{Ctr}(P_i, \Box\mathit{abs}_i(\varphi))$ is carried out on the state space of module P_i, rather than on the (larger) state space of the complete system. We can then solve the invariant checking problem $P_1 \parallel \cdots \parallel P_n \models \Box\varphi$ by executing

$$\mathit{InvCheck}\big(P_1 \parallel \cdots \parallel P_n,\ \varphi \wedge \bigwedge_{i=0}^{n} \mathit{Ctr}(P_i, \Box\mathit{abs}_i(\varphi))\big) \,. \tag{2}$$

It is necessary to conjoin φ to the set of controllable states in the above check, because for $1 \leq i \leq n$, predicate $\mathit{abs}_i(\varphi)$ (and thus, possibly, $\mathit{Ctr}(P_i, \Box\mathit{abs}_i(\varphi))$)

may be weaker than φ. If check (2) returns answer YES, then we have immediately that $P_1 \parallel \cdots \parallel P_n \models \Box\varphi$. If the check returns answer NO, we can conclude that $P_1 \parallel \cdots \parallel P_n \not\models \Box\varphi$. In this latter case, the check (2) also returns a partial counterexample $s_0, s_1, \ldots, s_m$, with $s_m \not\models \mathit{Ctr}(P_j, \Box\varphi_j)$ for some $1 \leq j \leq n$. If $s_m \models \neg\varphi$, this counterexample is also a counterexample to $\Box\varphi$. Otherwise, to obtain a counterexample $s_0, \ldots, s_m, s_{m+1}, \ldots, s_{m+r}$ with $s_{m+r} \not\models \varphi$, we proceed as follows. Let $U_0, U_1, \ldots, U_k$ be the predicates computed by Algorithm 1 during the computation of $\mathit{Ctr}(P_j, \Box\varphi_j)$; note that $s_m \models U_k$. For $l > 0$, given s_{m+l-1}, we pick s_{m+l} such that $s_{m+l} \models U_{k-l}$ and $(s_{m+l-1}, s_{m+l}) \models \bigwedge_{i=1}^{n} T_{P_i}$. The process terminates as soon as we reach an l such that $s_{m+l} \models \neg\varphi$: since the implication $U_0 \rightarrow \neg\varphi$ holds, this will occur in at most k steps.

4 Lazy and Constrained Controllability

In the previous section, we have used the notion of controllability to compute sets of *doomed states,* from which we know that there is a path violating the invariant. In order to detect errors early, we should compute the largest possible sets of doomed states. To this end, we introduce two notions of controllability that can be stronger than the classical definition of the previous section. The first notion, *lazy controllability,* can be applied to systems that are composed only of *lazy modules,* i.e. of modules that need not react to their inputs. Several communication protocols can be modeled as the composition of lazy modules. The second notion, *constrained controllability,* can be applied to any system.

Lazy controllability. A module is *lazy* if it always has the option of leaving its controlled variables unchanged. Formally, a module P is *lazy* if we have $(s, s) \models T_P$ for every state s over $\mathcal{V}_P$. If all the modules composing the system are lazy, then we can re-examine the notion of controllability described in Section 3 to take into account this fact. Precisely, we defined a state to be controllable w.r.t. an LTL property φ if there is a strategy for the environment to ensure that the resulting trace satisfies φ, regardless of the strategy used by the system. But if the environment is lazy, we must always account for the possibility that the environment plays according to its *lazy strategy,* in which the values of the external variables of the module never change. Hence, if all modules are lazy, there is a second condition that has to be satisfied for a state to be controllable: for every strategy of the module, the lazy environment strategy should lead to a trace that satisfies φ. It is easy to see, however, that this second condition for controllability subsumes the first. We can summarize these considerations with the following definition. For $1 \leq i \leq n$, denote by η^{ℓ} the lazy environment strategy of module P_i, which leaves the values of the external variables of P_i always unchanged. We say that a state $s \in \mathit{States}(\mathcal{V}_{P_i})$ is *lazily controllable with respect to a LTL formula* ψ iff, for every module strategy π, we have $\mathit{Outcome}(s, \pi, \eta^{\ell}) \models \varphi$. We let $\mathit{LCtr}(P, \varphi)$ be the predicate over $\mathcal{V}_P$ defining the set of states of P that are lazily controllable with respect to φ.

We can compute for the invariant $\Box\varphi$ the predicate $\mathit{LCtr}(P, \Box\varphi)$ by replacing the operator UPre in Algorithm 1 with the operator LUPre : $\mathcal{P}(\mathcal{V}_P) \mapsto \mathcal{P}(\mathcal{V}_P)$,

the *lazily uncontrollable predecessor operator,* defined by:

$$\mathrm{LUPre}_P(X) = \exists \mathcal{C}'_P \,.\, (T_P \wedge X')[\mathcal{E}_P/\mathcal{E}'_P] \,.$$

where $(T_P \wedge X')[\mathcal{E}_P/\mathcal{E}'_P]$ is obtained from $T_P \wedge X'$ by replacing each variable $x' \in \mathcal{E}'_P$ with $x \in \mathcal{E}_P$. Note that $\mathrm{LUPre}_P X$ computes a superset of $\mathrm{UPre}_P X$, and therefore the set $LCtr(P, \Box\varphi)$ of lazily controllable states is always a subset of the controllable states $Ctr(P, \Box\varphi)$.

Given $n > 1$ lazy modules $P_1, P_2, \ldots, P_n$ and a predicate $\varphi \in \mathcal{P}(\bigcup_{i=1}^n \mathcal{V}_{P_i})$, let $R = P_1 \parallel \cdots \parallel P_n$, and for all $1 \leq i \leq n$. We can check whether $P_1 \parallel \cdots \parallel P_n \models \Box\varphi$ by executing $InvCheck(R,\, \varphi \wedge \bigwedge_{i=1}^n LCtr(P_i, \Box abs_i(\varphi)))$. If this check returns answer No, we can construct a counterexample to $\Box\varphi$ as in Section 3.

Constrained controllability. Consider again $n > 1$ modules $P_1, P_2, \ldots, P_n$, together with a predicate $\varphi \in \mathcal{P}(\bigcup_{i=1}^n \mathcal{V}_{P_i})$. In Section 3, we defined a state to be controllable if it can be controlled by an unconstrained environment, which can update the external variables of the module in an arbitrary way. However, in the system under consideration, the environment of a module P_i is $Q_i = P_1 \parallel \cdots \parallel P_{i-1} \parallel P_{i+1} \parallel \cdots \parallel P_n$, for $1 \leq i \leq n$. This environment cannot update the external variables of P_i in an arbitrary way, but is constrained in doing so by the transition predicates of modules P_j, for $1 \leq j \leq n$, $j \neq i$. If we compute the controllability predicate with respect to the most general environment, instead of Q_i, we are giving to the environment in charge of controlling P_i more freedom than it really has. To model this restriction, we can consider games in which the environment of P_i is constrained by a transition predicate over $\mathcal{V}_{P_i} \cup \mathcal{E}'_{P_i}$ that over-approximates the transition predicate of Q_i. We rely on an over-approximation to avoid mentioning all the variables in $\bigcup_{j=1}^n \mathcal{V}_{P_j}$, since this would enlarge the state space on which the controllability predicate is computed.

These considerations motivate the following definitions. Consider a module P together with a transition predicate H over $\mathcal{V}_P \cup \mathcal{E}'_P$. An *$H$-constrained strategy* for the environment of P is a strategy $\eta : States(\mathcal{V}_P)^+ \mapsto States(\mathcal{E}_P)$ such that, for all $s_0, s_1, \ldots, s_k \in States(\mathcal{V}_P)^+$, we have $(s_k, \eta(s_0, s_1, \ldots, s_k)) \models H$. Given an LTL formula φ over $\mathcal{V}_P$, we say that a state $s \in States(\mathcal{V}_P)$ is *H-controllable* if there is an H-constrained environment strategy η such that, for every module strategy π, we have $Outcome(s, \pi, \eta) \models \varphi$. We let $CCtr(P, \langle\langle H \rangle\rangle \varphi)$ be the predicate over $\mathcal{V}_P$ defining the set of H-controllable states of P w.r.t. φ.[1] For invariant properties, the predicate $CCtr(P, \langle\langle H \rangle\rangle \Box\varphi)$ can be computed by replacing in Algorithm 1 the operator UPre with the operator $\mathrm{CUPre}_P[H] : \mathcal{P}(\mathcal{V}_P) \mapsto \mathcal{P}(\mathcal{V}_P)$, defined by:

$$\mathrm{CUPre}_P[H](X) = \forall \mathcal{E}'_P \,.\, (H \rightarrow \exists \mathcal{C}'_P \,.\, (T_P \wedge X')) \,.$$

When $H = true$, $\mathrm{CUPre}_P[H](X) = \mathrm{UPre}_P(X)$; for all other stronger predicates H, the H-uncontrollable predecessor operator $\mathrm{CUPre}_P[H](X)$ will be a superset

[1] If E_H is a module composable with P having transition relation H, the predicate $CCtr(P, \langle\langle H \rangle\rangle \varphi)$ defines exactly the same set of states as the ATL formula $\langle\langle E \rangle\rangle \Box\varphi$ interpreted over $P \parallel E_H$ [AHK97].

of $\mathrm{UPre}_P(X)$, and therefore the set $CCtr(P, \langle\langle H \rangle\rangle \varphi)$ of H-controllable states will be a subset of the controllable states $Ctr(P, \Box\varphi)$.

Given a system $R = P_1 \| P_2 \| \ldots \| P_n$ and a predicate $\varphi \in \mathcal{P}(\mathcal{V}_R)$, for $1 \leq i \leq n$ we let

$$H_i = \bigwedge_{j \in \{1,\ldots,n\} \setminus \{i\}} \exists \mathcal{U}_j \,.\, \exists \mathcal{U}'_j \,.\, T_{P_j}$$

where $\mathcal{U}_j = \mathcal{V}_{P_j} \setminus \mathcal{V}_{P_i}$. We can then check whether $R \models \Box\varphi$ by executing $InvCheck(R, \varphi \wedge \bigwedge_{i=1}^{n} CCtr(P_i, \langle\langle H_i \rangle\rangle \Box\, abs_i(\varphi)))$. If this check returns answer NO, we can construct a counterexample proceeding as in Section 3.

5 Experimental Results

We applied our methods for early error detection to two examples: a distributed database protocol and a wireless communication protocol. We implemented all algorithms on top of the model checker MOCHA [AHM+98], which relies on the BDD package and image computation engine provided by VIS [BHSV+96].

Demarcation protocol. The *demarcation protocol* is a distributed protocol for maintaining numerical constraints between distributed copies of a database [BGM92]. We considered an instance of the protocol that manages two sites that sell and buy back seats on the same airplane; each site is modeled by a module. In order to minimize communication, each site maintains a *demarcation* variable indicating the maximum number of seats it can sell autonomously; if the site wishes to sell more seats than this limit, it enters a negotiation phase with the other site. The invariant states that the total number of seats sold is always less than the total available.

In order to estimate the sensitivity of our methods to differences in modeling style, we wrote three models of the demarcation protocol; the models differ in minor details, such as the maximum number of seats that can be sold or bought in a single transaction, or the implementation of the communication channels. In all models, each of the two modules controls over 20 variables, and has 8–10 external variables; the diameter of the set of reachable states is between 80 and 120. We present the number of iterations required for finding errors in the three models using the various notions of controllability in Table 1. Some of the errors occurred in the formulation of the models, others were seeded at random.

Two-chip intercom. The second example is from the *Two-Chip Intercom* (TCI) project of the Berkeley Wireless Research Center [BWR]. TCI is a wireless local network which allows approximately 40 remotes to transmit voice with point-to-point and broadcast communication. The operation of the network is coordinated by a base station, which assigns channels to the remotes through a TDMA scheme. Each remote and base station will be implemented in a two-chip solution, one for the digital component and one for the analog. The TCI protocol involves four layers: the functional layer (UI), the transport layer, the medium access control (MAC) layer and the physical layer. The UI provides an interface between the user and the remote. The transport layer accepts service requests from the UI, defines the corresponding messages to be transmitted across the

Error	L	C	R	G
e1	19	19	19	24
e2	31	31	31	36
e3	19	19	19	24
e4	18	23	24	24

(a) Model 1.

Error	L	C	R	G
e1	30	30	35	35
e2	40	40	44	44
e3	28	28	33	33
e4	18	18	25	25

(b) Model 2.

Error	L	C	R	G
e1	14	18	18	18
e2	14	18	18	18
e3	14	18	18	18
e4	12	16	16	16

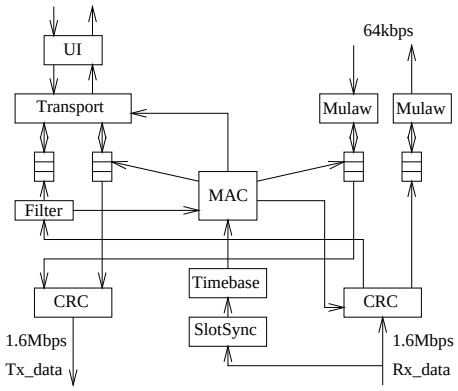
(c) Model 3.

Table 1. Number of iterations required in global state exploration to find errors in 3 models of the demarcation protocol. The errors are e1,...,e4. The columns are L (lazy controllability), C (constrained controllability), R (regular controllability), and G (traditional global state exploration).

network, and transmits the messages in packets. The transport layer also accepts and interprets the incoming packets and sends the messages to the UI. The MAC layer implements the TDMA scheme. The protocol stack for a remote is shown in Figure 1(a). Each of these blocks are described by the designers in Esterel and modeled in *Polis* using *Codesign Finite State Machines* [BCG+97].

There are four main services available to a user: *ConnReq*, *AddReq*, *RemReq* and *DiscReq*. To enter the network, a remote sends a connection request, *ConnReq*, together with the id of the remote, to the base station. The base station checks that the remote is not already registered, and that there is a free time-slot for the remote. It then registers the remote, and sends a connection grant back to the the remote. If a remote wishes to leave the network, it sends *DiscReq* to the base station, which unregisters the remote. If two or more remotes want to start a conference, one of them sends *AddReq* to the base station, together with the id's of the remotes with which it wants to communicate. The base station checks that the remotes are all registered, and sends to each of these remotes an acknowledgment and a time-slot assignment for the conference. When a remote wishes to leave the conference, it sends a *RemReq* request to the base station, which reclaims the time slot allocated to the remote.

We consider a TCI network involving one remote and one base station. The invariant states that if a remote believes that it is connected to the network, then the base station has this remote registered. This property involves the functional and transport layers. In our experiment, we model the network in *reactive modules* [AH99] The modules that model the functional and transport layers for both the remote and the base station are translated directly from the corresponding CFSM models; based on the protocol specification, we provide abstractions for the MAC layer and physical layer as well as the channel between the remote and the base station. Due to the semantics of CFSM, the modules are lazy, and therefore, lazy controllability applies. The final model has 83 variables. The number of iterations required to discover the various errors, some incurred during the modeling and some seeded in, are reported in Figure 1(b).



(a) Protocol Stack.

Error	L	C	R	G
e1	24	25	27	27
e2	39	41	41	41
e3	16	21	23	23
e4	24	25	27	27
e5	57	59	59	59
e6	16	19	21	21

(b) Iterations

Fig. 1. The TCI protocol stack and the number of iterations of global state exploration to discover the error.

Results on BDD sizes and discussion. In order to isolate the unpredictable effect of dynamic variable ordering on the BDD sizes, we conducted, for each error, two sets of experiments. In the first set of experiments, we turned off dynamic variable ordering, but supplied good initial orders. In the second, dynamic variable ordering was turned on, and a random initial order was given. Since the maximum BDD size is often the limiting factor in formal verification, we give results based on the maximum number of BDD nodes encountered during verification process, taking into account the BDDs composing the controllability predicates, the reachability predicate, and the transition relation of the system under consideration. We only compare our results for the verification using lazy controllability and global state exploration, since these are the most significant comparisons. Due to space constraint, we give results for model 3 of the demarcation protocol as well as the TCI protocol.

Without dynamic variable ordering. For each error, we recorded the maximum number of BDD nodes allocated by the BDD manager encountered during verification process. The results given in Table 2(a) and 2(b) are the averages of four experiment runs, each with a different initial variable order. They show that often the computation of the controllability predicates helps reduce the total amount of required memory by about 10–20%. The reason for this savings can be attributed to the fact that fewer iterations in global state exploration avoids the possible BDD blow-up in subsequent post-image computation.

With dynamic variable ordering. The analysis on BDD performance is more difficult if dynamic variable ordering is used. We present the results in Tables 2(c) and 2(d) which show the averages of nine experiment runs on the same models with dynamic variable ordering on. Dynamic variable ordering tries to minimize

Err	Lazy		Global
	Control	Total	Total
e1	4.4 (0.8)	6.6 (0.1)	7.9 (0.8)
e2	4.1 (0.1)	7.2 (0.6)	9.2 (2.4)
e3	4.4 (0.1)	9.0 (0.3)	14.6 (0.3)
e4	7.3 (0.9)	8.7 (0.1)	11.1 (2.1)

(a) Demarcation Protocol (Off).

Err	Lazy		Global
	Control	Total	Total
e1	4.8 (0.4)	6.3 (0.4)	6.7 (0.5)
e2	5.4 (0.6)	8.6 (0.5)	9.0 (0.4)
e3	5.8 (0.4)	6.5 (1.0)	7.7 (1.6)
e4	5.4 (0.1)	10.1 (0.1)	12.0 (0.3)
e5	6.6 (0.5)	40.7 (1.8)	43.8 (0.2)
e6	5.6 (0.6)	6.8 (1.6)	7.7 (1.9)

(b) TCI (Off).

Err	Lazy		Global
	Control	Total	Total
e1	3.0 (0.4)	6.9 (0.7)	7.5 (0.4)
e2	3.5 (1.0)	6.7 (0.4)	8.1 (0.8)
e3	3.6 (0.5)	8.9 (1.3)	12.7 (1.9)
e4	4.4 (0.4)	9.0 (0.9)	11.8 (2.6)

(c) Demarcation Protocol (On).

Err	Lazy		Global
	Control	Total	Total
e1	4.2 (0.9)	7.2 (0.8)	7.3 (0.9)
e2	3.7 (0.6)	10.1 (2.4)	11.0 (2.3)
e3	4.5 (0.5)	7.4 (1.5)	6.4 (0.6)
e4	3.8 (0.3)	11.4 (2.9)	16.9 (7.4)
e5	4.0 (0.4)	60.2 (19.1)	73.7 (29.8)
e6	4.6 (0.5)	7.9 (0.9)	6.8 (0.9)

(d) TCI (On).

Table 2. Average maximum number of BDD nodes required for error detection during the controllability (Control) and reachability computation (Total) phases. Dynamic variable ordering was turned off in (a) and (b), and on in (c) and (d). The results are given for lazy controllability and global state exploration. All data are in thousands of BDD nodes, and the standard deviations are given in parenthesis.

the total size of all the BDDs, taking into account the BDDs representing the controllability and the reachability predicates, as well as the BDDs encoding the transition relation of the system. Hence, if the BDDs for the controllability predicates are a sizeable fraction of the other BDDs, their presence slows down the reordering process, and hampers the ability of the reordering process to reduce the size of the BDD of the reachability predicate. Thus, while our methods consistently reduce the number of iterations required in global state exploration to discover the error, occasionally we do not achieve savings in terms of memory requirements.

When the controllability predicates are small compared to the reachability predicate, they do not interfere with the variable ordering algorithm. This observation suggests the following heuristics: one can alternate the iterations in the computation of the controllability and reachability predicates in the following manner. At each iteration, the iteration in the controllability predicate is computed only when its size is smaller than a threshold fraction (say, 50%) of the reachability predicate. Otherwise, reachability iterations are carried out.

Another possible heuristics to reduce the size of the BDD representation of the the controllability predicates is to allow approximations: our algorithms remain sound and complete as long as we use over-approximations of the controllability predicates.

6 Bounded Controllability and Iterative Strengthening

Bounded controllability. In lazy controllability, we know that there is a move of the environment that is always enabled (the move that leaves all external variables unchanged); therefore, that move must be able to control the module. In constrained controllability, we are given the set of possible environment moves, and we require that one of those moves is able to control the module. We can combine these two notions in the definition of *bounded controllability.* In bounded controllability, unlike in usual games, the environment may have some degree of insuppressible internal nondeterminism. For each state, we are given a (nonempty) set A of possible environment moves, as in usual games. In addition, we are also given a (possibly empty) set $B \subseteq A$ of moves that the environment can take at its discretion, even if they are not the best moves to control the module. We say that a state is *boundedly controllable* if (a) there is a move in A that can control the state, and (b) all the moves in B can control the state. The name *bounded controllability* is derived from the fact that the sets B and A are the lower and upper bounds of the internal nondeterminism of the controller.

Given a module P, we can specify the lower and upper bounds for the environment nondeterminism using two predicates $H^l, H^u \in \mathcal{P}(\mathcal{V}_P \cup \mathcal{E}'_P)$. We can then define the *bounded uncontrollable predecessor operator* $\mathrm{BUPre}[H^l, H^u] : \mathcal{P}(\mathcal{V}_P) \mapsto \mathcal{P}(\mathcal{V}_P)$ by

$$\mathrm{BUPre}[H^l, H^u](X) = \left[\forall \mathcal{E}'_P.(H^u \rightarrow \exists \mathcal{C}'_P.(T_P \wedge X'))\right] \vee \left[\exists \mathcal{E}'_P.(H^l \wedge \exists \mathcal{C}'_P.(T_P \wedge X'))\right] .$$

Note that the quantifiers are the duals of the ones in our informal definition, since this operator computes the uncontrollable states, rather than the controllable ones. Note also that in general we cannot eliminate the first disjunct, unless we know that $\exists \mathcal{E}'_P \,.\, H^l$ holds at all $s \in \mathit{States}(P)$, as was the case for lazy controllability. By substituting this predecessor operator to UPre in Algorithm 1, given a predicate φ over $\mathcal{V}_P$, we can compute the predicate $\mathit{BCtr}[H^l, H^u](P, \Box\varphi)$ defining the states of P that are boundedly controllable w.r.t. $\Box\varphi$. Given a system $R = P_1 \parallel \cdots \parallel P_n$ and a predicate φ over $\mathcal{V}_R$, we can use bounded controllability to compute a set of doomed states as follows. For each $1 \leq i \leq n$, we let as usual $\mathit{abs}_i(\varphi) = \exists(\mathcal{V}_R \setminus \mathcal{V}_{P_i}) \,.\, \varphi$, and we compute the lower and upper bounds by

$$H^l_i = \textstyle\bigwedge_{j \in \{1,\ldots,n\} \setminus \{i\}} \forall \mathcal{U}_{j,i} \,.\, \exists \mathcal{U}'_{j,i} \,.\, T_{P_j} \ , \qquad H^u_i = \textstyle\bigwedge_{j \in \{1,\ldots,n\} \setminus \{i\}} \exists \mathcal{U}_{j,i} \,.\, \exists \mathcal{U}'_{j,i} \,.\, T_{P_j} \ ,$$

where for $1 \leq j \leq n$, the set $\mathcal{U}_{j,i} = \mathcal{V}_{P_j} \setminus \mathcal{V}_{P_i}$ consists of the variable of P_j not present in P_i. We can then check whether $R \models \Box\varphi$ by executing $\mathit{InvCheck}(R, \varphi \wedge \bigwedge_{i=1}^{n} \mathit{BCtr}[H^l_i, H^u_i](P_i, \Box\, \mathit{abs}_i(\varphi)))$. If this check fails, we can construct counterexamples by proceeding as in Section 3.

Iterative strengthening. We can further strengthen the controllability predicates by the process of *iterative strengthening*. This process is based on the following observation. In the system $R = P_1 \parallel \cdots \parallel P_n$, in order to control P_i, the environment of P_i must not only take transitions compatible with the transition relation of the modules P_j, for $j \in \{1, \ldots, n\} \setminus \{i\}$, but these modules must also stay in their own sets of controllable states. This suggests that when we compute the controllable states of P_i, we take into account the controllability predicates already computed for the other modules. For $1 \leq i \leq n$, if δ_i is the controllability predicate of module P_i, we can compute the upper bound to the environment nondeterminism by

$$H_i^u(\bar{\delta}) = \bigwedge_{j \in \{1,\ldots,n\} \setminus \{i\}} \exists \mathcal{U}_{j,i} \,.\, \exists \mathcal{U}'_{j,i} \,.\, (T_{P_j} \wedge \delta_i \wedge \delta'_i) \;,$$

where $\bar{\delta} = \delta_1, \ldots, \delta_n$. For all $1 \leq i \leq n$, we can compute a sequence of increasingly strong controllability predicates by letting $\delta_i^0 = \mathrm{T}$ and, for $k \geq 0$, by $\delta_i^{k+1} = BCtr[H_i^l, H_i^u(\bar{\delta}^k)](P_i, \Box\varphi)$. For all $1 \leq i \leq n$ and all $k \geq 0$, predicate δ_i^{k+1} is at least as strong as δ_i^k. We can terminate the computation at any $k \geq 0$ (reaching a fixpoint is not needed), and we can verify $R \models \Box\varphi$ by executing $InvCheck(R, \varphi \wedge \bigwedge_{i=1}^n \delta_i^k)$. As k increases, so does the cost of computing these predicates. However, this increase may be offset by the faster detection of errors in the global state-exploration phase.

Discussion. The early error detection techniques presented in the previous sections for invariants can be straightforwardly extended to general linear temporal-logic properties. Given a system $R = P_1 \parallel \cdots \parallel P_n$ and a general LTL formula ψ over $\mathcal{V}_R$, we first compute for each $1 \leq i \leq n$ the predicate δ_i, defining the controllable states of P_i with respect to ψ. This computation requires the solution of ω-regular games [EJ91,Tho95]; in the solution, we can use the various notions of controllability developed in this paper, such as lazy, constrained, or bounded controllability. Then, we check whether $R \models \psi \wedge \Box(\bigwedge_{i=1}^n \delta_i)$: as before, if a state that falsifies δ_i for some $1 \leq i \leq n$ is entered, we can immediately conclude that $R \not\models \psi$. For certain classes of properties, such as reachability properties, it is convenient to perform this check in two steps, first checking that $R \models \Box(\bigwedge_{i=1}^n \delta_i)$ (enabling early error detection) and then checking that $R \models \psi$.

Acknowledgements. We thank Andreas Kuehlmann for pointing out the connection of this work with target enlargement.

References

[AdAHM99] R. Alur, L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. Automating modular verification. In *CONCUR 99: Concurrency Theory*, LNCS. Springer-Verlag, 1999.

[AH99] R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[AHK97] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proc. 38th IEEE Symp. Found. of Comp. Sci.*, pages 100–109. IEEE Computer Society Press, 1997.

[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 *CAV 98: Computer Aided Verification*, volume 1427 of *LNCS*, pages 521–525. Springer-Verlag, 1998.

[ASSSV94] A. Aziz, T.R. Shiple, V. Singhal, and Alberto L. Sangiovanni-Vincentelli. Formula-dependent equivalence for CTL model checking. In *CAV 94: Computer Aided Verification*, LNCS. Springer-Verlag, 1994.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of TACAS: Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 193–207. Springer-Verlag, 1999.

[BCG+97] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Press, 1997.

[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*, volume 580 of *LNCS*, 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 *CAV 96: Computer Aided Verification*, volume 1102 of *LNCS*, pages 428–432. Springer-Verlag, 1996.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[BWR] Berkeley wireless research center. http://bwrc.eecs.berkeley.edu.

[Dil88] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. MIT Press, 1988.

[EJ91] E.A. Emerson and C.S. Jutla. Tree automata, mu-calculus and determinacy (extended abstract). In *32nd Symp. on Foundations of Computer Science (FOCS)*, pages 368–377, 1991.

[HKQ98] T.A. Henzinger, O. Kupferman, and S. Qadeer. From *pre*historic to *post*modern symbolic model checking. In *CAV 98: Computer Aided Verification*, volume 1427 of *LNCS*, pages 195–206. Springer-Verlag, 1998.

[HQR98] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. You assume, we guarantee: methodology and case studies. In *CAV 98: Computer Aided Verification*, volume 1427 of *LNCS*, pages 440–451. Springer-Verlag, 1998.

[JSAA97] J. Juan, J. Shen, J. Abraham, and A. Aziz. On combining formal and informal verification. In *CAV 97: Computer Aided Verification*, LNCS. Springer-Verlag, 1997.

[KV96] O. Kupferman and M.Y. Vardi. Module checking. In *CAV 96: Computer Aided Verification*, volume 1102 of *LNCS*, pages 75–86. Springer-Verlag, 1996.

[RW89] P.J.G. Ramadge and W.M. Wonham. The control of discrete event systems. *IEEE Transactions on Control Theory*, 77:81–98, 1989.

[Tho95] W. Thomas. On the synthesis of strategies in infinite games. In *Proc. of 12th Annual Symp. on Theor. Asp. of Comp. Sci.*, volume 900 of *LNCS*, pages 1–13. Springer-Verlag, 1995.

[YD98] C.H. Yang and D.L. Dill. Validation with guided search of the state space. In *Design Automation Conference*, June 1998.