Compositional Reasoning in Model Checking

Sergey Berezin[1] Sérgio Campos[2] Edmund M. Clarke[1]

Feberuary 2, 1998

CMU-CS-98-106

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[1] Carnegie Mellon University — USA
[2] Universidade Federal de Minas Gerais — Brasil

To appear in the *Proceedings of the Workshop COMPOS'97*.

Abstract

The main problem in model checking that prevents it from being used for verification of large systems is the *state explosion problem*. This problem often arises from combining parallel processes together. Many techniques have been proposed to overcome this difficulty and, thus, increase the size of the systems that model checkers can handle. We describe several *compositional model checking* techniques used in practice and show a few examples demonstrating their performance.

This research is sponsored by the the Semiconductor Research Corporation (SRC) under Contract No. 97-DJ-294, the National Science Foundation (NSF) under Grant No. CCR-9505472, and the Defense Advanced Research Projects Agency (DARPA) under Contract No. DABT63-96-C-0071.

Keywords: automatic verification, temporal logic, compositional model checking, bisimulation

1 Introduction

Symbolic model checking is a very successful method for verifying complex finite-state reactive systems [7]. It models a computer system as a state-transition graph. Efficient algorithms are used to traverse this graph and determine whether various properties are satisfied by the model. By using BDDs [5] it is possible to verify extremely large systems having as many as 10^{120} states. Several systems of industrial complexity have been verified using this technique. These systems include parts of the Futurebus+ standard [12, 19], the PCI local bus [10, 20], a robotics systems [8] and an aircraft controller [9].

In spite of such success, symbolic model checking has its limitations. In some cases the BDD representation can be exponential in the size of system description. This behavior is called the *state explosion problem*. The primary cause of this problem is *parallel composition* of interacting processes. The problem occurs because the number of states in the global model is exponential in the number of component processes. Explicit state verifiers suffer from the state explosion problem more severely than symbolic verifiers. However, the problem afflicts symbolic verification systems as well, preventing them from being applied to larger and more complex examples.

The state explosion can be alleviated using special techniques such as *compositional reasoning*. This method verifies each component of the system in isolation and allows global properties to be inferred about the entire system. Efficient algorithms for compositional verification can extend the applicability of formal verification methods to much larger and more interesting examples. In this paper we describe several approaches to compositional reasoning. Some are automatic and are almost completely transparent to the user. Others require more user intervention but can achieve better results. Each is well suited for some applications while not so efficient for others.

For example, *partitioned transition relations* [6] and *lazy parallel composition* [11, 27] are automatic and, therefore, preferred in cases where user intervention is not desired (for example, when the user is not an expert). These techniques provide a way to compute the set of successors (or predecessors) of a state set without constructing the transition relation of the global system. Both use the transition relations of each component separately during traversal of the state graph. The individual results are combined later to give the set of states in the global graph that corresponds to the result of the operation being performed.

Another automatic technique is based on the use of *interface processes*. This technique attempts to minimize the global state transition graph by focusing on the communication among the component processes. The method considers the set of variables used in the interface between two components and minimizes the system by eliminating events that do not relate to the communication variables. In this way, properties that refer to the interface variables are preserved, but the model becomes smaller.

Assume-guarantee reasoning [17] is a manual technique that verifies each component separately. The behavior of each component depends on the behavior of the rest of the system, i.e., its environment. Because of this, the user must specify properties that the environment has to satisfy in order to guarantee the correctness of the component. These properties are *assumed*. If these assumptions are satisfied, the component will satisfy other properties, called *guarantees*. By combining the set of assume/guarantee properties in an appropriate way, it is possible to demonstrate the correctness of the entire system without constructing the global state graph.

All of these methods have been used to verify realistic systems. This shows that compositional reasoning is an effective method for increasing the applicability of model checking tools. Furthermore, it is a necessity for verification of many complex industrial systems.

The remainder of this paper is organized as follows: Section 2 introduces the formal model that we use for finite-state systems and the kinds of parallel composition we consider. Section 3 describes partitioned transition relations, and Section 4 discusses lazy parallel composition. Interface processes and assume-guarantee reasoning are described in Sections 5 and 6, respectively. Finally, the paper concludes in Section 7 with a summary and some directions for future research.

2 The Model

Given the description of the system to be verified, constructing its model involves two important steps. The first is constructing the model for the individual components. The second is composing these submodels into a global model. We start by showing how to represent each component symbolically given its state-transition graph. Then we describe the parallel composition algorithm used to create the global model.

2.1 Representing a Single Component

Representing a state-transition graph symbolically involves determining its set of states and deriving the transition relation of the graph that models the component. Consider a system with a set of variables V. For a synchronous circuit, the set V is typically the outputs of all the registers in the circuit together with the primary inputs. In the case of an asynchronous circuit, V is usually the set of all nodes. For a protocol or software system, V is the set of variables in the program. A state can be described by giving values to all the variables in V. Since the system is finite-state we can encode all states by boolean vectors. Throughout the paper we assume that this encoding has already been done and that all variables in V are boolean. Therefore, a state can be described by a valuation assigning either 0 or 1 to each variable. Given a valuation, we can also write a boolean expression which is true for exactly that valuation. For example, given $V = \{v_0, v_1, v_2\}$ and the valuation $\langle v_0 \leftarrow 1, v_1 \leftarrow 1, v_2 \leftarrow 0\rangle$, we derive the boolean formula $v_0 \wedge v_1 \wedge \neg v_2$. This boolean formula can then be represented using a BDD.

In general, however, a boolean formula may be true for many valuations. If we adopt the convention that a formula represents the set of *all* valuations that make it true, then we can describe sets of states by boolean formulas and, hence, by BDDs. In practice, BDDs are often much more efficient than representing sets of states explicitly. We denote sets of states with the letter S and we denote the BDD representing the set S by $S(V)$, where V is the set of variables that the BDD may depend on. We also use $f, g, \ldots$ for arbitrary boolean functions.

In addition to representing sets of states of a system, we must be able to represent the transitions that the system can make. To do this, we extend the idea used above. Instead of just representing a set of states using a BDD, we represent a set of ordered pairs of states. We cannot do this using just a single copy of the state variables, so we create a second set of variables V'. We think of the variables in V as *current state* variables and the variables in V' as *next state* variables. Each variable v in V has a corresponding next state variable in V', which we denote by v'. A valuation for the variables in V and V' can be viewed as an ordered pair of states, and we represent sets of these valuations using BDDs as above. We write a formula that is true iff there is a transition from the state represented by V to the state represented by V'. For example, if there is a transition from state $\langle v_0 \leftarrow 1, v_1 \leftarrow 1, v_2 \leftarrow 0\rangle$ to state $\langle v_0 \leftarrow 1, v_1 \leftarrow 0, v_2 \leftarrow 1\rangle$ we write the formula $v_0 \wedge v_1 \wedge \neg v_2 \wedge v_0' \wedge \neg v_1' \wedge v_2'$. The disjunction of all such transitions is the transition relation of the model. If N is a transition relation, then we write $N(V, V')$ to denote the BDD that represents it.

2.2 Parallel Composition

The technique above shows how to construct the graph that models one component of the system. But usually systems are described by a set of components that execute concurrently. For synchronous or asynchronous circuits the components are the smaller circuits that are connected together to construct the bigger circuit. For protocols or programs the components are the processes that execute in parallel.

There are two major ways of composing processes or systems: synchronously and asynchronously. In synchronous composition all processes execute at the same time, one step in one process corresponds to exactly one step in all the other processes. In asynchronous composition, on the other hand, only one process executes at any point in time. When one process steps all the others remain unchanged. The choice of which process steps at any time is nondeterministic. There are different algorithms for composing synchronous and asynchronous systems.

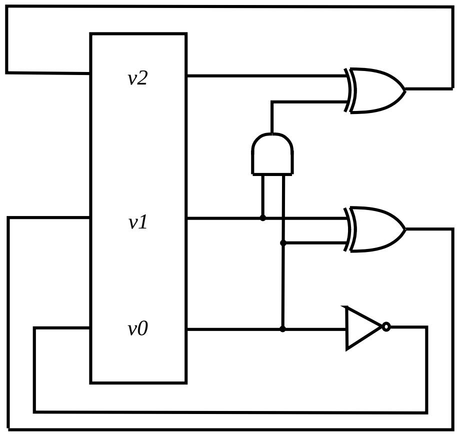


Figure 1: A modulo 8 counter

2.2.1 Synchronous Systems

The method for deriving the transition relation of a synchronous system can be illustrated using a small example. Consider the circuit of a modulo 8 counter on Fig. 1. Let $V = \{v_0, v_1, v_2\}$ be the set of state variables for this circuit, and let $V' = \{v'_0, v'_1, v'_2\}$ be another copy of the state variables. The transitions of the modulo 8 counter are given by

$$\begin{aligned} v'_0 &= \neg v_0 \\ v'_1 &= v_0 \oplus v_1 \\ v'_2 &= (v_0 \wedge v_1) \oplus v_2 \end{aligned}$$

The above equations can be used to define the relations

$$\begin{aligned} N_0(V, V') &= (v'_0 \Leftrightarrow \neg v_0) \\ N_1(V, V') &= (v'_1 \Leftrightarrow v_0 \oplus v_1) \\ N_2(V, V') &= (v'_2 \Leftrightarrow (v_0 \wedge v_1) \oplus v_2) \end{aligned}$$

which describe the constraints each v'_i must satisfy in a legal transition. Each constraint can be seen as a separate component, and their composition generates the counter. These constraints can be combined by taking their conjunction to form the transition relation:

$$N(V, V') = N_0(V, V') \wedge N_1(V, V') \wedge N_2(V, V').$$

In the general case of a synchronous system with n components, we let $\{N_0, \cdots, N_{n-1}\}$ be the set of transition relations for each component. Each transition relation N_i determines the values of a subset of variables in V in the next state. Analogous to the modulo 8 counter, the conjunction of these relations forms the transition relation

$$N(V, V') = N_0(V, V') \wedge \cdots \wedge N_{n-1}(V, V').$$

Thus, the transition relation for a synchronous system can be expressed as a conjunction of relations.

Given a BDD for each transition relation N_i, it is possible to compute the BDD that represents N. We say that such a transition relation is *monolithic* because it is represented by a single BDD. Monolithic transition relations are the primary bottleneck for verification, because their size can be exponential in the number of equations used to define it.

2.2.2 Asynchronous Systems

As with synchronous systems, the transition relation for an asynchronous system can be expressed as a conjunction of relations. Alternatively, it can be expressed as a disjunction. To simplify the description of how such transition relations are obtained, we assume that all the components of the system have exactly one output and have no internal state variables. In this case, it is possible to describe completely each component by a function $f_i(V)$. Given values for the present state variables v, the component drives its output to the value specified by $f_i(V)$. For some components, such as C-elements and flip-flops, the function $f_i(V)$ may depend on the current value of the output of the component, as well as the inputs. Extending the method to handle components with multiple outputs is straightforward.

In speed-independent asynchronous systems, there can be an arbitrary delay between when a transition is enabled and when it actually occurs. We can model this by allowing each component to choose nondeterministically whether to transition or not. This results in a conjunction of n parts, all of the form

$$T_i(V, V') = \big(v_i' \Leftrightarrow f_i(V)\big) \vee (v_i' \Leftrightarrow v_i).$$

This model is similar to the synchronous case discussed above, and allows *more* than one variable to transition concurrently.

Normally, we will use an *interleaving model* for asynchronous composition, in which only one variable is allowed to transition at a time. First, we apply the distributive law to the conjunction of the T_i's, giving a disjunction of 2^n terms:

$$\bigwedge_{i=1}^{n} T_i \equiv \bigvee_{b_1,\ldots b_n} \left(\bigwedge_{i=1}^{n} v_i' \Leftrightarrow g_i^{b_i}(V) \right),$$

where all b_i's are indices over $\{0, 1\}$ and

$$g_i^b(V) = \begin{cases} f_i(V), & \text{if } b = 1 \\ v_i, & \text{if } b = 0. \end{cases}$$

Each of these terms $\bigwedge_{i=1}^{n} v_i' \Leftrightarrow g_i^{b_i}(V)$ corresponds to the simultaneous transitioning of some subset of the n variables in the model for which $b_i = 1$. Second, we keep only those terms that correspond to exactly one variable being allowed to transition (that is, only those disjuncts for which the vector $b_1, \ldots, b_n$ contains exactly one 1). This results in a disjunction of the form

$$N(V, V') = N_0(V, V') \vee \cdots \vee N_{n-1}(V, V'),$$

where

$$N_i(V, V') = \big(v_i' \Leftrightarrow f_i(V)\big) \wedge \bigwedge_{j \neq i} (v_j' \Leftrightarrow v_j).$$

Notice, that using this method asynchronous systems are composed by disjuncting their components, while synchronous systems are composed by conjuncting their components.

3 Partitioned Transition Relations

Computing the image or pre-image of a set of states S under a transition relation N is the most important operation in model checking. A state t is a *successor* of s under N, if there is a transition from s to t or, in other words, $N(s, t)$ holds. The *image* of a set of states S is the set of all successors of S. If the set S and the transition relation N are given by boolean formulas, then the image of S is given by the following formula

$$\exists V \big[S(V) \wedge N(V, V')\big],$$

where $\exists V$ denotes existential quantification over all variables in V. This formula defines the set of successors in terms of free variables V'. Similarly, a state s is a *predecessor* of a state t under N iff $N(s, t)$ is true. The set of predecessors of a state set S is described by the formula

$$\exists V' \big[S(V') \wedge N(V, V')\big].$$

Formulas of this type are called *relational products*.

While it is possible to implement the relational product with one conjunction and a series of existential quantifications, in practice this would be fairly slow. In addition, the OBDD for $S(V') \wedge N(V, V')$ is often much larger than the OBDD for the final result, and we would like to avoid constructing it if possible. For these reasons, we use a special algorithm to compute the OBDD for the relational product in one step from the OBDDs for S and N. Figure 2 gives this algorithm for two arbitrary OBDDs f and g.

Like many OBDD algorithms, *RelProd* uses a result cache. In this case, entries in the cache are of the form (f, g, E, r), where E is a set of variables that are quantified out and f, g and r are OBDDs. If such an entry is in the cache, it means that a previous call to *RelProd*(f, g, E) returned r as its result.

Although the above algorithm works well in practice, it has exponential complexity in the worst case. Most of the situations where this complexity is observed are cases in which the OBDD for the result is exponentially larger than the OBDDs for the arguments $f(\bar{v})$ and $g(\bar{v})$. In such situations, any method of computing the relational product must have exponential complexity.

In the previous section we have described how to construct the global transition relation N from the individual transition relations N_i of the component processes. However, the size of N can be significantly larger than the sum of the sizes of all N_is. Our goal is to be able to compute relational products without constructing the global transition relation explicitly.

3.1 Disjunctive Partitioning

The global transition relation of an asynchronous system may be written as the disjunction of the transition relations for the individual components of the system. In this case, a relational product will have the form

$$\exists V' \big[S(V') \wedge (N_0(V, V') \vee \cdots \vee N_{n-1}(V, V'))\big].$$

```
function RelProd(f, g: OBDD, E: set of variables): OBDD
if f = 0 ∨ g = 0
    return 0
else if f = 1 ∧ g = 1
    return 1
else if (f, g, E, r) is in the result cache
    return r
else
    let x be the top variable of f
    let y be the top variable of g
    let z be the topmost of x and y
    r0 := RelProd(f|z←0, g|z←0, E)
    r1 := RelProd(f|z←1, g|z←1, E)
    if z ∈ E
        r := Or(r0, r1)
            /* OBDD for r0 ∨ r1 */
    else
        r := BDDnode(z, r1, r0)
            /* OBDD for (z ∧ r1) ∨ (¬z ∧ r0) */
    endif
    insert (f, g, E, r) in the result cache
    return r
endif
```

Figure 2: Relational product algorithm

In practice computing the value of a large formula with many quantifiers is usually very expensive. Since the existential quantifier distributes over disjunction we can shrink the scope of the quantifier to the individual components:

$$\exists V' \left[S(V') \wedge N_0(V, V')\right] \vee \cdots \vee$$

$$\exists V' \left[S(V') \wedge N_{n-1}(V, V')\right]$$

When this technique is used it is possible to compute relational products for much larger asynchronous systems.

3.2 Conjunctive Partitioning

For synchronous systems, a relational product will have the form

$$\exists V' \left[S(V') \wedge (N_0(V, V') \wedge \cdots \wedge N_{n-1}(V, V'))\right].$$

Unfortunately, existential quantification does not distribute over conjunction, so we can not directly apply the same transformation as in the asynchronous case. A simple counterexample is

$$\exists a[(a \vee b) \wedge (\neg a \vee c)] \not\equiv \exists a[a \vee b] \wedge \exists a[\neg a \vee c]$$

since it reduces to:

$$[b \vee c] \not\equiv true.$$

Nevertheless, we still can apply partitioning because systems often exhibit locality: most N_is depend only on a small number of variables in V and V'. Subformulas can be moved outside of the scope of existential quantification if they do not depend on any of the variables being quantified:

$$\exists a\big[(a \vee b) \wedge (b \vee c)\big] \equiv \exists a\big[a \vee b\big] \wedge (b \vee c)$$

We can optimize the computation of a relational product by using *early variable elimination* for variables in each N_i. First, pick an order ρ for considering the partitions in the relational product. Then define D_i to be the set of variables process P_i depends on, and E_i to be a subset of D_i consisting of variables that *no* process later in the ordering depends on, i.e.,

$$E_{\rho(i)} = D_{\rho(i)} - \bigcup_{k=i+1}^{n-1} D_{\rho(k)}.$$

We will illustrate this with our example of the modulo 8 counter.

$$\begin{array}{lll}
N_0 = (v_0' \Leftrightarrow \neg v_0) & \text{depends on} & D_0 = \{v_0\} \\
N_1 = (v_1' \Leftrightarrow v_0 \oplus v_1) & \text{depends on} & D_1 = \{v_0, v_1\} \\
N_2 = (v_2' \Leftrightarrow (v_0 \wedge v_1) \oplus v_2) & \text{depends on} & D_2 = \{v_0, v_1, v_2\}
\end{array}$$

If we choose the ordering $\rho = 2, 1, 0$, then $E_2 = \{v_2\}, E_1 = \{v_1\}$ and $E_0 = \{v_0\}$. We now can transform the relational product to:

$$\begin{array}{rcl}
S_1(V, V') & = & \exists_{v \in E_{\rho(0)}} \left[S(V) \wedge N_{\rho(0)}(V, V')\right] \\
S_2(V, V') & = & \exists_{v \in E_{\rho(1)}} \left[S_1(V, V') \wedge N_{\rho(1)}(V, V')\right] \\
 & \vdots & \\
S_n(V') & = & \exists_{v \in E_{\rho(n-1)}} \left[S_{n-1}(V, V') \wedge N_{\rho(n-1)}(V, V')\right].
\end{array}$$

Or putting it all together,

$$\underbrace{\exists V_{\rho(n-1)}\left[\ldots \underbrace{\exists V_{\rho(1)}\left[\underbrace{\exists V_{\rho(0)}\left[S(V) \wedge N_{\rho(0)}(V,V')\right]}_{S_1} \wedge N_{\rho(1)}(V,V')\right]}_{S_2} \wedge \cdots \wedge N_{\rho(n-1)}(V,V')\right]}_{S_n}$$

The ordering ρ has a significant impact on how early in the computation state variables can be quantified out. This affects the size of the BDDs constructed and the efficiency of the verification procedure. Thus, it is important to choose ρ carefully, just as with the BDD variable ordering. For example, a badly chosen ordering $\rho = 0, 1, 2$ for the same modulo 8 counter yields $E_0 = \{\}$, $E_1 = \{\}$ and $E_2 = \{v_0, v_1, v_2\}$, which results in no optimization at all.

In practice, we have found it fairly easy to come up with orderings which give good results. We search for a good ordering ρ by using a greedy algorithm to find a good ordering on the variables v_i to be eliminated. For each ordering on the variables, there is an obvious ordering on the relations N_i such that when this relation ordering is used, the variables can be eliminated in the order given by the greedy algorithm.

The algorithm on fig. 3 gives our basic greedy technique. We start with the set of variables V to be eliminated and a collection $\mathcal{C}$ of sets where every $D_i \in \mathcal{C}$ is the set of variables on which N_i depends. We then eliminate the variables one at a time by always choosing the variable with the least cost and then updating V and $\mathcal{C}$ appropriately.

> **while** $(V \neq \phi)$ **do**
> **begin**
>   For each $v \in V$ compute the cost of eliminating v;
>   Eliminate variable with lowest cost by updating $\mathcal{C}$ and V;
> **end**;

Figure 3: Algorithm for variable elimination.

All that remains is to determine the cost metric to use. We will consider three different cost measures. To simplify our discussion, we will use N_v to refer to the relation created when eliminating variable v by taking the conjunction of all the N_i that depend on v and then quantifying out v. We will use D_v to refer to the set of variables on which this N_v depends.

minimum size The cost of eliminating a variable v is simply $|D_v|$. With this cost function, we always try to insure that the new relation we create depends on the fewest number of variables.

minimum increase The cost of eliminating variable v is

$$|D_v| - \max_{A \in \mathcal{C}, v \in A} |A| + 1$$

which is the difference between the size of D_v and the size of the largest D_i containing v. The idea is that if we have a lot of small relations that all share one variable, then we do not want to eliminate that variable, since this may result in a big N_v. But this is what the previous heuristic would suggest. Instead, the minimum increase cost will favor eliminating variables that are shared by a small number of relations, thus, keeping the resulting relation smaller. In other words, we prefer to make a small increase in the size of an already large relation than to create a new large relation.

minimum sum The cost of eliminating variable v is

$$\sum_{A \in \mathcal{C}, v \in A} |A|$$

which is simply the sum of the sizes of all the D_i containing v. Since the cost of conjunction depends on the sizes of the arguments, we approximate this cost by the number of variables on which each of the argument N_i depends.

The overall goal is to minimize the size of the largest BDD created during the elimination process. In our abstraction, this translates to finding an ordering that minimizes the size of the largest set D_v created during the process. Always making a locally optimal choice does not guarantee an optimal solution and there are counterexamples for each of the three cost functions. In fact, the problem of finding an optimal ordering can be shown to be NP-complete. However, the minimum sum cost function seems to provide the best approximation of the cost of the actual BDD operations and in practice has the best performance on most examples.

4 Lazy Parallel Composition

Lazy parallel composition is an alternative method for compositional reasoning that can be related to partitioned transition relations. As in the case of the partitioned transition relations, the global transition relation is never constructed. However, in contrast to the previous method, a restricted transition relation for all processes is created. The restricted transition relation agrees with the global transition relation for 'important' states, but it may behave in a different way for other states. The advantage comes from the fact that in many cases it is possible to construct a restricted transition relation that is significantly smaller than the global transition relation.

There are many possible ways of constructing a restricted transition relation that would produce correct results. Given an original global transition relation N and a state set S, the computation of the set of successors of S can use any restricted transition relation N' that satisfies the following condition:

$$N'|_S = N|_S$$

The formula above means that N and N' agree on transitions that start from states in S. It is possible to represent N' with significantly fewer nodes than N in some cases by using the constrain operator from [14, 27]. For two boolean formulas f and g, $f' = \mathit{constrain}(f, g)$ is a formula that has the same truth value as f for variable assignments that satisfy g. If the variable assignment does not satisfy g, the value of f' can be arbitrary. In other words:

$$f'(x) = \begin{cases} f(x) & \text{if } g(x) \\ \text{don't care} & \text{otherwise} \end{cases}$$

In many cases the size of f' is significantly smaller than the size of f.

The lazy composition algorithm uses the constrain operator to simplify the transition relation of each process before generating the global restricted transition relation. When computing the set of successors of a state set S (represented by a boolean formula) the algorithm computes

$$N' = \bigwedge_{i=0..n} \mathit{constrain}(N_i, S).$$

Each transition $N_i' = \mathit{constrain}(N_i, S)$ agrees with N_i on transitions that start in S by the definition of the constrain operator. As a consequence, the transition relation N' agrees with the global transition relation N on transitions that start in S as well. Therefore, computing the set of successors of S using N' produces the same result as using N. The same method can be applied when computing the set of predecessors of a state set S. Only in this case the constrain operator has to maintain those transitions in N that *end* in S.

4.1 Partitioning vs. Lazy Composition

Lazy parallel composition is less sensitive to the order in which variables are eliminated than partitioned transition relations. This is because step i in the partitioned transition relation depends on step $i-1$, as shown below

$$\underbrace{\exists v_1 \big[\underbrace{\exists v_0 \big[S(V') \wedge N_0(V,V') \big]}_{\text{step1}} \wedge N_1(V,V') \big]}_{\text{step2}}.$$

As a consequence, the final degree of partitioning heavily depends on the order in which we quantify the variables out. We have already seen an example of such dependency in section 3.2.

The lazy parallel composition, on the other hand, processes each component independently, and thus, does not depend on the order in which the constrain operators are applied:

$$\exists V' \big[S(V') \wedge (\underbrace{N_1(V,V') \mid_S}_{\text{step1}} \wedge \underbrace{N_2(V,V') \mid_S}_{\text{step2}}) \big].$$

We have implemented the lazy composition algorithm and obtained significant gains in both space and time. The verification of one example took 18 seconds and 1 MB of memory when lazy composition was used. The same example took about the same amount of time but twice as much memory when partitioned transition relations were used. If neither method was used, verification required more than 40 seconds and 12 MB. A significant part of the savings in both methods results from not constructing the global transition relation. However, lazy parallel composition often requires much less memory. The reason seems to be that partitioned transition relations are heavily influenced by the order in which partitions are processed, because this order determines which variables can or cannot be quantified out early. In lazy parallel composition this does not happen, since all of the variables are quantified out at the same time. This makes it less susceptible to the order in which partitions are processed, and more suitable to be used in the cases in which determining the processing order can be difficult. It also makes the new technique easier to automate.

5 Interface Processes

An important observation leads to another approach to compositional verification. The state explosion problem is usually most severe for *loosely coupled* processes which communicate using a small number of shared variables.

5.1 Cone of Influence Reduction

Suppose we are given a set of variables σ that we are interested in with respect to the process P. We can simplify the process P using the *cone of influence* reduction. Assume that the system is specified by a set of equations:

$$v_i' = f_i(V).$$

Define the cone of influence C_i of v_i for each variable v_i as the minimal set of variables such that

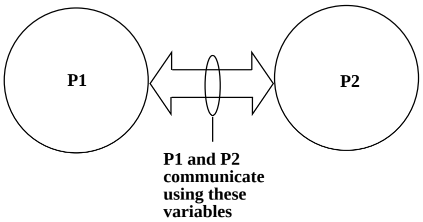
- $v_i \in C_i$,
- if for some $v_l \in C_i$ its f_l depends on v_j, then $v_j \in C_i$.

Construct a new (reduced) process P' from P by removing all the equations whose left hand side variables do not appear in any of the C_i's for $v_i \in \sigma$. It can be easily shown that $P \models \varphi$ iff $P' \models \varphi$, whenever φ contains only variables from σ.

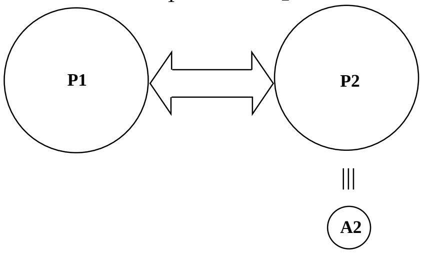
Again, consider our example of the modulo 8 counter (fig. 1). Its set of equations is

$$\begin{aligned} v_0' &= \neg v_0 \\ v_1' &= v_0 \oplus v_1 \\ v_2' &= (v_0 \wedge v_1) \oplus v_2 \end{aligned}$$

Clearly, $C_0 = \{v_0\}$, since f_0 does not depend on any variable other than v_0. We have $C_1 = \{v_0, v_1\}$, since f_1 depends on both of the variables, but $v_2 \notin C_1$ because no variable in C_1 depends on v_2. And C_2 is the set of all the variables.



Assume two processes P_1 and P_2 communicate using a set of variables σ. Then P_1 can only observe the behavior of P_2 through σ. It means that we can replace P_2 by any equivalent process A_2 which is indistinguishable from P_2 with respect to σ and this will completely preserve the behavior of P_1. The idea is to find a smaller process A_2 that hides all events irrelevant to σ.



The following *interface rule* guarantees the correctness of the abstraction A_2 with respect to P_1. Let $P|_\sigma$ be the restriction of P to the cone of influence of variables in σ, and $\mathcal{L}(\sigma)$ be the set of all CTL formulas with free variables from σ. The *interface rule* states that if the following conditions are satisfied:

- $P_2|_\sigma \equiv A_2$,
- $P_1||A_2 \models \varphi$,
- φ is a CTL formula such that $\varphi \in \mathcal{L}(\sigma)$,

then φ is also true in $P_1||P_2$. In fact, it is sufficient for φ to be in $\mathcal{L}(\Sigma_{P_1})$ for this rule to be sound, where Σ_{P_1} is the set of variables of P_1.

In the remainder of this section we describe how this strategy can be made precise and show how it can be used to reduce the state explosion problem for loosely coupled processes.

5.2 Soundness of the interface rule

In order for the interface rule to be sound we need to specify some properties that the process equivalence '$\equiv$' has to satisfy. For a process P let Σ_P be the set of atomic propositions (or state variables) in P, and let $\mathcal{L}(\Sigma)$ be the language of temporal formulas over the alphabet Σ. For any two processes P_1 and P_2 with sets of variables Σ_{P_1} and Σ_{P_2}, the following axioms have to be satisfied:

1. $P_1 \equiv P_2$ implies $\forall \varphi \in \mathcal{L}(\Sigma_{P_1})[P_1 \models \varphi \leftrightarrow P_2 \models \varphi]$
2. If $P_1 \equiv P_2$ then $P_1||Q \equiv P_2||Q$ and $Q||P_1 \equiv Q||P_2$
3. $(P_1||P_2)|_{\Sigma_{P_1}} \equiv P_1||(P_2|_{\Sigma_{P_1}})$ and $(P_1||P_2)|_{\Sigma_{P_2}} \equiv (P_1|_{\Sigma_{P_2}})||P_2$
4. If $\varphi \in \mathcal{L}(\Sigma_\varphi)$ and $\Sigma_\varphi \subseteq \Sigma_P$, then $P \models \varphi$ iff $P|_{\Sigma_\varphi} \models \varphi$

Theorem 1. (Soundness) *The Interface Rule is sound.*
To remind the reader, the interface rule states that

- $P_2|_{\Sigma_{P_1}} \equiv A_2$,
- $P_1||A_2 \models \varphi$,
- φ is a CTL formula such that $\varphi \in \mathcal{L}(\Sigma_{P_1})$,

imply $P_1||P_2 \models \varphi$. Notice, that restricting P_2 to Σ_{P_1} produces the same result as $P_2|_\sigma$, where $\sigma = \Sigma_{P_1} \cap \Sigma_{P_2}$.

PROOF. Since $P_2|_{\Sigma_{P_1}} \equiv A_2$, then by 2 $P_1\|A_2 \equiv P_1\|(P_2|_{\Sigma_{P_1}})$. By 3, $P_1\|(P_2|_{\Sigma_{P_1}}) \equiv (P_1\|P_2)|_{\Sigma_{P_1}}$, hence we also have $P_1||A_2 \equiv (P_1||P_2)|_{\Sigma_{P_1}}$. And since $P_1\|A_2 \models \varphi$ and $\varphi \in \mathcal{L}(\Sigma_{P_1})$, by 1 we derive $(P_1\|P_2)|_{\Sigma_{P_1}} \models \varphi$, and from 4 we immediately get $P_1||P_2 \models \varphi$ as required.

5.3 Equivalence of Processes

We define concrete equivalence relations over the processes that fulfil our requirements and are the most suitable in our framework. We use *bisimulation equivalence* and *stuttering equivalence* with synchronous parallel composition. We also give an "efficient" polynomial algorithm to determine bisimulation equivalence between processes and a sketch of the algorithm for stuttering equivalence.

Definition 1. A *model* is a triple $M = (S, N, L)$, where S is a set of states, $N \subseteq S \times S$ is a transition relation and L is a *labeling function* mapping each state into a set of *atomic propositions* that are true in that state.

5.3.1 Bisimulation Equivalence.

Consider two models $M = (S, N, L)$ and $M' = (S', N', L')$ with the same set of atomic propositions.

Definition 2. A binary relation $E \subseteq S \times S'$ is called a *bisimulation relation* if for any $s \in S$ and $s' \in S'$, $E(s, s')$ implies $L(s) = L'(s')$ and

$$(i) \quad \forall r \in S.N(s,r) \Rightarrow \exists r' \in S' : N'(s',r') \wedge E(r,r')$$

$$(ii) \quad \forall r' \in S'.N'(s',r') \Rightarrow \exists r \in S : N(s,r) \wedge E(r,r').$$

Definition 3. A *bisimulation equivalence* is the maximum bisimulation relation in the subset inclusion preorder.

Notice that the definition of a bisimulation relation can be viewed as a fixpoint equation. Hence, the bisimulation equivalence is just the greatest fixpoint of that equation. This gives rise to a simple polynomial algorithm for computing the bisimulation equivalence using the well known iterative procedure. We compute a (decreasing) sequence of relations $E_0, E_1, \ldots$ until this sequence converges to a fixpoint at the n-th step. This convergence is guaranteed in finite-state case, since the subset inclusion preorder is well-founded in both directions. Choosing an appropriate E_0 guarantees that this fixpoint is the greatest fixpoint, therefore E_n is the required bisimulation equivalence. The sequence of relations is defined inductively as follows:

1. sE_0s' iff $L(s) = L'(s')$,
2. $sE_{n+1}s'$ iff $L(s) = L'(s')$ and
   - $\forall s_1[N(s, s_1)$ implies $\exists s_1'[N'(s', s_1') \wedge s_1 E_n s_1']]$
   - $\forall s_1'[N'(s', s_1')$ implies $\exists s_1[N(s, s_1) \wedge s_1 E_n s_1']]$

The complexity of this algorithm is $O(m^2)$, where m in the sum of the sizes of the transition relations. There are more efficient algorithms for computing bisimulation equivalence, for example the Paige-Tarjan algorithm [24]. It's complexity is $O(m \log n)$ in time and $O(m+n)$ in space, where n is the sum of the numbers of states in both models, and m in the sum of the sizes of the transition relations. However, it is unclear if this algorithm can employ BDDs as well.

5.3.2 Stuttering Equivalence.

Unlike bisimulation, the *stuttering equivalence* [4, 16] is usually defined over the *computation paths* of the models. Intuitively, two paths π and π' are considered stuttering equivalent if they can be partitioned into finite blocks of repeated, or *stuttered* states, and corresponding blocks are equivalent in the two paths relative to the labeling functions L and L' of the models. Thus, we do not distinguish between two executions that differ only in the number of idle cycles between transitions. The stuttering equivalence also has a definition in terms of the greatest fixpoint.

Definition 4. A binary relation $E \subseteq S \times S'$ is called a *stuttering relation* if for any $s \in S$ and $s' \in S'$, $s\ E\ s'$ implies $L(s) = L'(s')$ and

$$\begin{array}{ll} (i) & \forall r.\ N(s,r) \Rightarrow \exists s_0', \ldots, s_n'\ (n \geq 0).\ s_0' = s' \text{ and } r\ E\ s_n' \text{ and} \\ & \quad \forall 0 \leq i < n.\ N'(s_i', s_{i+1}') \text{ and } s\ E\ s_i'; \\ \\ (ii) & \forall r'.\ N'(s', r') \Rightarrow \exists s_0, \ldots, s_m\ (m \geq 0).\ s_0 = s \text{ and } s_m\ E\ r' \text{ and} \\ & \quad \forall 0 \leq i < m.\ N(s_i, s_{i+1}) \text{ and } s_i\ E\ s'. \end{array}$$

Definition 5. A *stuttering equivalence* is the maximum stuttering relation in the subset inclusion preorder.

Stuttering equivalence preserves the truth of CTL* formulas that do not involve the next time operator $\mathbf{X}$ [4]. As in the case of bisimulation, we define inductively a sequence of relations $E_0, E_1, \ldots$ (that also converges in finite state case) and the stuttering equivalence is the intersection of all the E_i's. However, instead of computing the direct pre-image at each iteration as we did for bisimulation, we compute the set of states from which there is a path to the current state along which the current labeling $L(s)$ changes exactly once. This involves computing another least fixpoint. The details of the algorithm are described in [3]. A more efficient algorithm based on the Paige-Tarjan algorithm was found by Groote and Vaandrager [16] that runs in $O(mn)$ time. It is unknown, however, if this algorithm can use BDDs as well.

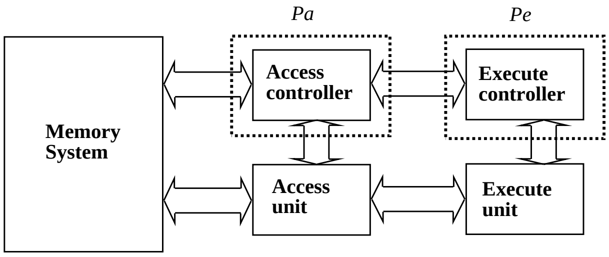


Figure 4: A CPU controller.

5.4 Interface Processes Example

As a simple example, we consider a model of the CPU controller [13] (fig. 4). The model comprises two parallel processes P_a and P_e called the access unit and the execution unit. The access unit P_a fetches instructions and stores them in an instruction queue and maintains a cache of the top location of the CPU stack in a special register. The execution unit P_e pops out the instructions from the queue and interprets them. A major part of the temporal logic specification for CPU's controller defines correct behavior for the access unit and consists of formulas on the set of signals which are inputs or outputs of the unit. These signals constitute Σ_{P_a}. An example of such a formula is the following

$$\mathbf{AG}\,\mathbf{AF}\,fetch$$

This formula is a liveness property which states that instructions are fetched from the access unit to the execution unit infinitely often. *Fetch* is actually a propositional formula defined in terms of request and acknowledge signals between the two units.

The parallel composition of the access unit and the execution unit in our design has approximately 1100 reachable states. However, by restricting the outputs of the execution unit to those in Σ_{P_a}, and then minimizing it, we obtain an interface process A_{P_e} such that $P_a \| A_{P_e}$ has only 196 reachable states. The reason for this reduction is that, while the execution unit interprets many different instructions, the memory accesses of these instructions fall into a few basic patterns.

6 Assume/Guarantee Reasoning

Assume-guarantee reasoning is a semi-automatic method that verifies each component separately. Ideally, compositional reasoning exploits the natural decomposition of a complex system into simpler components, handling one component at a time. In practice, however, when a component is verified it may be necessary to *assume* that the environment behaves in a certain manner. If the other components in the system *guarantee* this behavior, then we can conclude that the verified properties are true of the entire system. These properties can be used to deduce additional global properties of the system.

The *assume-guarantee paradigm* [17, 21, 23, 25] uses this method. Typically, a formula is a triple $\langle g \rangle M \langle f \rangle$ where g and f are temporal formulas and M is a program. The formula is true if whenever M is part of a system satisfying g, the system must also satisfy f. A typical proof shows that $\langle g \rangle M \langle f \rangle$ and $\langle true \rangle M' \langle g \rangle$ hold and concludes that $\langle true \rangle M \parallel M' \langle f \rangle$ is true. This proof strategy can also be expressed as an inference rule:

$$\frac{\langle true \rangle M' \langle g \rangle \quad \langle g \rangle M \langle f \rangle}{\langle true \rangle M \parallel M' \langle f \rangle}$$

The soundness of this simple assume-guarantee rule is straightforward.

In order to automate this approach, a model checker must be able to check that a property is true of *all* systems which can be built using a given component. More generally, it must be able to restrict to a given class of environments when doing this check. An elegant way to obtain a system with this property is to provide a preorder $\preceq$ on the finite state models that captures the notion of "more behaviors" and to use a logic whose semantics is consistent with the preorder. The order relation should preserve satisfaction of formulas of the logic, i.e. if a formula is true for a model, it should also be true for any model which is smaller in the preorder. Additionally, composition should preserve the preorder, and a system should be smaller in the preorder than its individual components. Finally, satisfaction of a formula should correspond to being smaller than a particular model (a tableau for the formula) in the preorder.

Following Grumberg and Long [17], we use *synchronous process composition*, the *simulation preorder*, and the temporal logic *ACTL* (a subset of CTL without existential path quantifiers). This choice is motivated by the expressiveness of ACTL and the existence of a very efficient model checking algorithm for this logic. The simulation preorder is also a natural choice, since it is simple and intuitive as well as easily automated. We employ tableau construction methods for converting formulas into processes. Informally, a tableau for a formula φ is the greatest process A_φ (in the preorder) such that $A_\varphi \models \varphi$. In the remainder of this section we will not distinguish formulas and processes and will write, for example, $M \preceq \varphi$ to mean $M \preceq A_\varphi$.

It can be easily shown that our choice of formalisms meets all the requirements [17]. In particular, for all M and M' we have $M \| M' \preceq M$, and if $M' \preceq A$ then $M \| M' \preceq M \| A$, because synchronous composition can only restrict possible behaviors. Since M is greater than any system containing M, we can focus on proving properties of M in isolation. This insures that the same properties hold for an arbitrary system containing M.

Using the tableau construction we can verify $M \models \varphi$ by checking the relation $M \preceq \varphi$. In practice, however, we use classical model checking for verifying $M \models \varphi$ for a single component M if φ is given by a formula, and the simulation preorder if φ is an automaton, to increase the efficiency. Assumptions on the model correspond to composition. That is, a model M has the same set of behaviors under assumptions ψ as the model $M \| \psi$ without any assumptions. Thus, our triple $\langle\varphi\rangle M \langle\psi\rangle$ corresponds to $\varphi \| M \preceq \psi$. In other words, discharging assumptions corresponds to checking the preorder. Finally, the rule $M \preceq M \| M$ allows multiple levels of assume-guarantee reasoning.

Earlier we mentioned that the logic must preserve the preorder relation. Now we formalize and state the properties explicitly.

1. For all M, M' and φ, if $M \preceq M'$ and $M' \models \varphi$, then $M \models \varphi$ (removing behaviors cannot change a formula from true to false). Since $M \| M' \preceq M$, it is enough to check $M \models \varphi$ to know that any system containing M also satisfies φ.

2. For every φ, there is a structure T_φ such that $M \models \varphi$ if and only if $M \preceq T_\varphi$. This allows us to use φ as an assumption by composing M with T_φ.

3. Every model of φ is also a model of ψ if $T_\varphi \models \psi$

These lemmas are proved rigorously in [17] for synchronous composition of processes, the simulation preorder and the logic ACTL.

6.1 Implementation of Assume Guarantee Reasoning

Suppose we want to show that $M \| M' \models \psi$. That is, in terms of triples, we need to prove $\langle true\rangle M \| M' \langle\psi\rangle$. We verify that M satisfies some property ϑ by model checking. Next, using ϑ

as an assumption, we show that M' satisfies some other auxiliary property φ. Finally, we show that M satisfies the required property ψ under the assumption φ. Since this extends to any system containing M, we are done. If the intermediate formulas (or processes) φ and ϑ are much smaller than M and M' respectively, then all the transition relations that must be constructed are significantly smaller than the one for $M \| M'$. This strategy for proving $M \| M' \models \psi$ can be summarized in the following assume-guarantee rule:

$$\frac{\langle true \rangle M \langle \vartheta \rangle \quad \langle \vartheta \rangle M' \langle \varphi \rangle \quad \langle \varphi \rangle M \langle \psi \rangle}{\langle true \rangle M \| M' \langle \psi \rangle}$$

In our framework, this corresponds to

$$\frac{M \preceq \vartheta \quad \vartheta \| M' \preceq \varphi \quad \varphi \| M \preceq \psi}{M \| M' \preceq \psi}$$

It is straightforward to show that this rule is sound by using the properties of preorder relation stated earlier.

Theorem 2. *The assume-guarantee rule is sound.*

PROOF. Since $M \preceq \vartheta$, then $M \| M' \preceq \vartheta \| M'$. Since $\vartheta \| M' \preceq \varphi$, by transitivity $M \| M' \preceq \varphi$. Composing both sides with M we get $M \| M' \| M \preceq \varphi \| M$. Since parallel composition is commutative and associative, we can group the left hand side as $M \| M \| M'$. Then using $M \preceq M \| M$ and composing both sides with M' we obtain $M \| M' \preceq \varphi \| M$. Finally, from the last assumption $\varphi \| M \preceq \psi$ and transitivity we draw the conclusion of the rule $M \| M' \preceq \psi$.

So far, we have not discussed fairness. Both the preorder and the semantics of the logic should include some type of *fairness*. This is essential for modeling systems (hardware or communication protocols) at the appropriate level of abstraction. Moreover, fairness is necessary for the ACTL tableau construction.

Unfortunately, no efficient technique exists to check or compute *fair preorder* between models. In [17], Grumberg and Long suggest how to check the fair preorder only for a few trivial cases. Kupferman and Vardi showed that the general case is PSPACE-hard to compute [22]. Henzinger, Kupferman, and Rajamani [18] have proposed a new type of fair preorder that can be computed in polynomial time. However, it is not clear that this preorder is appropriate for compositional reasoning.

6.1.1 Example: The Futurebus+ Protocol.

David Long has used this type of reasoning to verify safety and liveness properties for the Futurebus+ standard of cache coherence protocol [12, 19]. The whole design is divided into parallel components that represent single modules like cache, memory, bus, etc. This example requires several levels of assumptions and guarantees.

The first stage of the verification was to check safety properties, since they can be verified using only forward reachability analysis and checking at each iteration that the current set of reachable states satisfies the property. Once a violation is found, the search is terminated immediately and an error trace is generated. The ability to terminate the search early was important since the BDD representing the set of reached states tended to become very large once an erroneous transition had occurred. As soon as all of the basic safety properties were satisfied, more complex formulas were checked in the state space restricted to the set of reachable states. Such a restriction also helped greatly in keeping the BDD from blowing up in size.

Using this technique he found specifications that were satisfied by a single bus configuration but not by multiple bus configurations. The details of the verification can be found in [12].

7 Conclusions

We describe several methods of dealing with the state explosion problem, which arises frequently due to parallel composition of processes. It is clear that compositional reasoning is critical in formal verification. Such techniques dramatically reduce the complexity of model checking and permit the verification of significantly larger systems. We have used compositional methods extensively to verify large complex systems such as the Futurebus+ [12] and the PCI bus [10, 20] protocols.

This paper does not cover all of compositional proof techniques. There are a number of other compositional techniques that can also be successfully used. For example, *partial model checking* [1] encodes one of the processes into the formula, which is being checked, and simplifies the resulting formula. Similar method is described in [2]. Theorem proving techniques are also used to decompose and prove (manually) the property for each of the component [15, 26].

In general, all of the compositional model checking techniques have their limitations and much work remains to be done. The most important problem is the trade-off between efficiency and automation. More powerful methods that can handle enormous complexity usually require an expert user and significant manual effort. These techniques usually rely on a powerful theorem prover under human guidance or careful choice of model checking parameters. On the other hand, completely automatic techniques frequently cannot handle extremely complex systems. The problem with automatic techniques is that they rely heavily on heuristics which may or may not work on different types of examples, and most of the intellectual work still has to be done by the user.

References

[1] Henrik R. Andersen. Partial model checking (extended abstract). Technical Report ID-TR: 1994-148, Department of Computer Science, Technical University of Denmark, October 1994. Accepted for LICS'95.

[2] Henrik R. Andersen, Colin Stirling, and Glynn Winskel. A compositional proof system for the modal μ-calculus. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 144–153, Paris, France, 4–7 July 1994. IEEE Computer Society Press. Also as BRICS Report RS-94-34.

[3] S. Berezin, E. Clarke, S. Jha, and W. Marrero. Model checking algorithms for the mu-calculus. Technical Report TR CMU-CS-96-180, Carnegie Mellon University, September 1996.

[4] M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1–2), July 1988.

[5] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[6] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In *VLSI 91*, Edinburgh, Scotland, 1990.

[7] Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. McMillan, and David L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 13(4):401–424, April 1994.

[8] S. Campos, E. Clarke, W. Marrero, and M. Minea. Verus: a tool for quantitative analysis of finite-state real-time systems. In *Workshop on Languages, Compilers and Tools for Real-Time Systems*, 1995.

[9] S. Campos, E. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. In *IEEE Real-Time Systems Symposium*, 1994.

[10] S. Campos, E. Clarke, and M. Minea. Verifying the performance of the PCI local bus using symbolic techniques. In *Proceedings of the IEEE International Conference on Computer Design*, pages 73–79, 1995.

[11] S. V. Campos. *A Quantitative Approach to the Formal Verification of Real-Time System.* PhD thesis, SCS, Carnegie Mellon University, 1996.

[12] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In L. Claesen, editor, *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and their Applications.* North-Holland, April 1993.

[13] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 353–362. IEEE Computer Society Press, June 1989.

[14] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Proceedings of the 1989 International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France*, volume 407 of *Lecture Notes in Computer Science.* Springer-Verlag, June 1989.

[15] M. Dam. Compositional proof systems for model checking infinite state processes. In *Proceedings of CONCUR'95*, volume 962 of *Lecture Notes in Computer Science*, pages 12–26. Springer-Verlag, 1995.

[16] J.F. Groote and F.W. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In M. Paterson, editor, *Proceedings* 17^{th} *ICALP,* Warwick, volume 443 of *Lecture Notes in Computer Science*, pages 626–638. Springer-Verlag, July 1990.

[17] Orna Grumberg and David Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[18] T. A. Henzinger, O. Kupferman, and S. K. Rajamani. Fair simulation. In *Proc. of the 7th Conference on Concurrency Theory (CONCUR'97)*, volume 1243 of *LNCS*, Warsaw, July 1997.

[19] IEEE Computer Society. *IEEE Standard for Futurebus+—Logical Protocol Specification*, 1994. IEEE Standard 896.1, 1994 Edition.

[20] Intel Corporation. *PCI Local Bus Specification*, 1993.

[21] B. Josko. Verifying the correctness of AADL-modules using model checking. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of the REX Workshop on Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 386–400. Springer-Verlag, May 1989.

[22] O. Kupferman and M. Y. Vardi. Module checking revisited. In O. Grumberg, editor, *Proc. of the 9th conference on Computer-Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 36–47, Haifa, June 1997.

[23] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7(4), July 1981.

[24] R. Paige and R. Tarjan. Three efficient algorithms based on partition refinement. *SIAM Journal on Computing*, 16(6), Dec 1987.

[25] A. Pnueli. In transition for global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI series. Series F, Computer and system sciences*. Springer-Verlag, 1984.

[26] C. Stirling. Modal logics for communicating systems. *Theoretical Computer Science*, 49:311–348, July 1987.

[27] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using bdd's. In *IEEE Int. Conf. Computer-Aided Design*, pages 130–133, 1990.