Automatic Reduction in CTL Compositional Model Checking

Thomas R. Shiple* Massimiliano Chiodo†
Alberto L. Sangiovanni-Vincentelli* Robert K. Brayton*

*Department of EECS, University of California, Berkeley, CA 94720
†Magneti Marelli, Pavia, Italy

Abstract. We describe a method for reducing the complexity of temporal logic model checking of a system of interacting finite state machines, and prove that it yields correct results. The method consists essentially of reducing each component machine with respect to the property we want to verify, and then verifying the property on the composition of the reduced components. We demonstrate the method on a simple example. We assess the potential of our approach on real-world examples.

1 Introduction

Temporal logic model checking procedures are potentially powerful verification tools for finite state systems. However, when the system under examination consists of several communicating parallel machines, the potential arises for an explosion in the size of the representation of the composition. Traditionally, the size of a system is identified with the number of states, and hence the issue is referred to as the *state-explosion* problem. The introduction of symbolic representations, based on binary decision diagrams (BDDs) [1], and symbolic verification procedures [8, 15, 2], made it possible to verify complex systems that could not be handled by techniques based on explicit representations [5]. However, just as with explicit representations, the size of the parallel composition may still be too large to handle.

Methods proposed to avoid the construction of the complete state graph, and therefore to avoid the representation explosion, can be split into two categories, *compositional verification* and *compositional minimization* [9]. In the first category, one tries to deduce properties of a composition of processes by reasoning on the individual components and their interactions, without ever building the composed system. In the second category, one tries to reduce or minimize the components in such a way that their composition yields a smaller yet semantically equivalent model of the total system.

As an example of the first, Wolper [16] inductively verifies complex systems by looking for *network invariants*, that is properties that, if satisfied by a network of n identical processes, will be satisfied by a network of $n + 1$ processes. Kurshan and McMillan have attempted a similar approach [12].

As for compositional minimization, Kurshan [13] uses homomorphic reductions, which "relax" the behavior of the component machines, to produce component machines that have fewer states than the original ones. The ω-regular properties he wants to verify are preserved under such user-provided reductions. Similarly, Clarke *et al.* [10, 7] define subsets of the logics CTL* and CTL, namely $\forall$CTL* and $\forall$CTL,

where only universal path quantification is allowed. For these logics, satisfaction is preserved under composition and homomorphic reduction. However, these logics are strictly less expressive than CTL* and CTL, respectively.

As additional examples of compositional minimization, the compositional model checking algorithm based on the *interface rule*, proposed by Clarke *et al.* [6], is a technique that allows verification of a CTL property [5] on a single (main) machine within a system of interacting machines. Here, the other (side) machines are reduced by hiding those output variables of the side machines which the main machine cannot observe. This reduction is property *independent* in that the reduction is valid for any property on the main machine. The main limitation of this approach is that it cannot handle CTL formulas which specify properties of multiple interacting machines. Burch describes a technique for efficiently computing the existential quantification of variables from a product of component transition relations, a central computation in symbolic model checking [3]. By quantifying out variables from such a product as early as possible, one can avoid forming explicitly the complete product machine, and hence, potentially avoid an explosion in the BDDs. However, the applicability of this technique, and the amount of reduction achieved, depend heavily on the structure of the system and on the user-supplied order in which the component transition relations are processed.

It is our opinion that to make formal verification a usable tool in design, fully automatic techniques whose details are transparent to the user must be developed to attack large, complex problems typical of electronic system design. Our goal is to verify CTL properties on a system of interacting finite state machines.

The approach we take is to extract from the component machines the information relevant to the verification of a given property, and use only this to build the representation of a reduced system that preserves all of the behavior needed to verify the property. In this regard, our approach falls into the compositional minimization category. Our approach is fully automated and returns an exact result; that is the reduced system is verified if and only if the complete system is verified. Our finite state machine model allows non-deterministic transitions and incomplete specification, and thus can be used to represent reduced machines.

In Section 2 we present an overview of our approach, with references to the sections where the topics are addressed in detail. Section 3 gives definitions that will be used in the paper. In Section 4 we fully describe the details of our technique. In Section 5, we apply our technique to a simple example and discuss the results. Finally, in Section 6 we present conclusions and future developments. Detailed proofs of the theorems presented in the body of the paper are given in [4].

2 Overview

Techniques for CTL model checking on a *single* finite state machine are well known: given the transition relation for a single machine, and a CTL property on the machine, a *single machine model checker* will return the set of states of the machine which satisfy the property. The model checker operates on the BDD of the characteristic function of the transition relation.

Our goal is to perform CTL model checking on a *system* of interacting FSMs, where the CTL formulas express properties of the entire system, and not just of a component of the system. This is known as *compositional model checking*. In this setting, we

are given the transition relation for each component machine. Since the interaction between the components affects the behavior of each component, we cannot directly apply the *single* machine model checker to each component. Nonetheless, by forming the complete product of all the component transition relations, we can produce a single transition relation which can be used as input to the model checker. However, this is a naive approach, because there is a distinct possibility that the size of the representation for the product machine will explode. In other words, the size of the representation for each component may be reasonable, but the size of the product may be intractable.

In our approach to compositional model checking, we use the single machine model checker, but not applied to the complete product of the component machines. Instead, we first extract the "interesting" part of each component to yield a "reduced" transition relation for each component. We then take the product of the reduced components, and finally, apply the model checker to the reduced product. In this manner, we hope to avoid an explosion in the size of the product machine.

We have reformulated the single machine model checker computations to better suit our needs for compositional model checking (Section 4.1). Specifically, the output it produces is a transition relation rather than a set of states. To produce the set of states which satisfy a formula, we simply project the transition relation onto the state space. Thus, our model checker takes a CTL formula F and a transition relation T as input, and produces a transition relation T^* as output. The proof of correctness of our compositional model checker relies on two important properties of our single machine model checker. The first is that T contains T^*. The second is that we can delete transitions from T *before* passing it to the model checker, and still get the same final result, as long as we do not delete any transitions that would be in T^*.

A novel aspect of our approach is how we determine the interesting part of each component machine. For a state in the product machine to satisfy a given formula, each state component of the global state must satisfy the formula projected onto the state component's associated machine. Thus, for each component, we project the CTL formula of interest onto the component (Section 4.2), and then apply the model checker to the component machine with the projected formula. The output of the model checker will have the same or fewer transitions than the input transition relation. Hence, by first eliminating some transitions from each component, the hope is that the size of the reduced product machine will be smaller than the complete product machine. Since the reduction of each component depends on the property of interest, we say that the reduction is *property dependent.*

To see why this technique gives the correct result, that is, computes the set of states of the product machine satisfying the input CTL formula, consider Figure 1. We are given a system M composed of interacting machines A and B, and a CTL formula F. The state space of M is the entire box, and the state space of A is the projection of the box onto any horizontal line. Likewise, the state space of B is any vertical line. Let Q_M be the set of states of M satisfying F. Consider the formula F_A produced by projecting the formula F onto machine A. By applying the model checker to machine A with formula F_A, we find the set of states Q_A of A which satisfy the projected formula, *disregarding the interaction of A and B.* Similarly, we produce Q_B by applying the model checker to B with F_B. By this procedure, $Q_A \times Q_B \supseteq Q_M$. Finally, we apply the model checker to the product of the subset of A containing Q_A, and the subset of B containing Q_B, to yield Q_M (Section 4.2).

In addition to the property-dependent reduction of each component machine described above, we apply several other techniques to reduce the size of each component before forming the product machine (Sections 4.2 and 4.2). If we were using an explicit representation (e.g. state transition graph) for the transition relations, then just by removing transitions, we would be decreasing the size of the representation. However, we are using an implicit representation, namely BDDs. Therefore, our actual goal is to find the smallest possible subset, containing the interesting behavior, of the original transition relations. This gives us a large don't care set within which to choose a transition relation with a small BDD representation.

It is important to note that in our work, "reduction" of a machine means reducing the size of the representation of the machine by removing "uninteresting" behavior. This differs, for example, from the usage of the term reduction in COSPAN [13], where reduction means making a machine smaller by adding behavior.

By avoiding the possible explosion in the size of the product machine, we hope to verify complex systems that cannot be verified with present techniques. There are two important features of our method to bear in mind. First, our reduction is fully automatic - it requires no guidance from the user. Second, our approach gives an exact result: it produces exactly the set of states satisfying a given CTL formula.

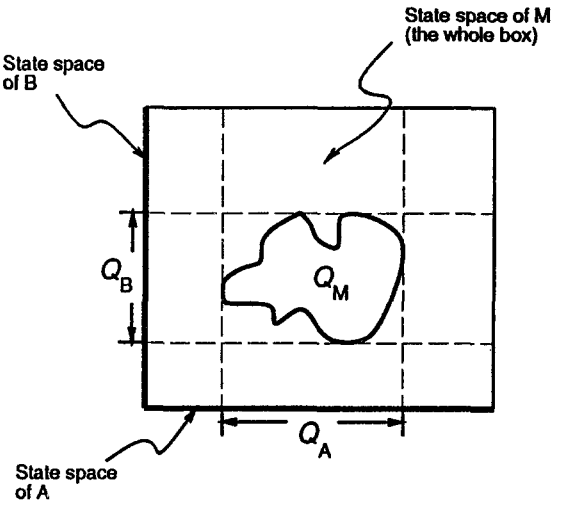

Fig. 1. State spaces in a system of interacting FSMs

3 Background

In the sequel, we use the following notation: The symbol "·" is used for Boolean AND, the symbol "+" for Boolean OR, and the symbols "¬" and overbar for Boolean NOT. $[T]_{x \to y}$ denotes the substitution of each occurrence of the variable x in the function T, by the variable y.

S_xT denotes the *smoothing* of the relation T by x, computed as $S_xT = T_{\overline{x}} + T_x$, where T_x is the cofactor of T with respect to x [15]. This is interpreted as the projection of T onto the subspace orthogonal to x, or equivalently as the existential quantification of the variable x on the relation T. We will make use of the following properties of the smoothing operator: (a) S_xf is the smallest function independent of x that contains f, and (b) $S_{x,y}f = S_x(S_yf) = S_y(S_xf)$.

Our model of finite state machines is based on the Moore model. A *synchronous finite state machine* with states X and inputs I is specified by a *transition relation* $T \subseteq X \times I \times X$. Each $(x, i, x') \in T$ is a *transition* from *present state* $x \in X$ to *next state* $x' \in X$ enabled by input i. We require each state to have a next state. Otherwise, we allow non-deterministic and incompletely specified machines. Lastly, in place of the Moore model's explicit outputs, which are functions of the state variables, our model simply allows the environment to directly observe the state variables.

In a *system* of interacting FSMs, the input to each component machine consists of the present states of the other components, plus external inputs. The definition of the parallel composition of two interacting machines follows.

Definition 1. Let $A(x, i, x')$ and $B(y, j, y')$ be transition relations describing two interacting FSMs. Furthermore, let $x, x' \in X$, $i \in Y \times I$, $y, y' \in Y$, and $j \in X \times I$. The transition relation M defining the behavior of the parallel composition of the two synchronous machines A and B is given by a subset of $(X \times Y) \times I \times (X \times Y)$ where

$$((x, y), k, (x', y')) \in M \text{ iff } (x, (y, k), x') \in A \text{ and } (y, (x, k), y') \in B. \quad \blacksquare$$

This definition is easily extended to a system of n interacting machines $A_1, A_2, \ldots, A_n$. Each machine A_i has present state variable $x_i \in X_i$ and next state variable $x_i' \in X_i$, and takes as input $\underline{x}_{\neq i} = [x_1, \cdots, x_{i-1}, x_{i+1}, \cdots, x_n]$, the present state variables of the other machines. The global state of the system is $\underline{x} = [x_1, \ldots, x_n] \in \underline{X} = X_1 \times X_2 \times \ldots \times X_n$. We restrict our attention to *closed* systems, that is systems that do not take inputs from the external environment. Interaction with the external environment is modeled by other state machines that produce (possibly non-deterministically) the inputs to which the system is sensitive. Thus we can represent each A_i by the transition relation $A_i(x_i, \underline{x}_{\neq i}, x_i') = A_i(\underline{x}, x_i')$ (see Figure 2).

In Section 4, we use the notion of path to define the output of the model checker computations.

Definition 2. A *path* on the transition relation T, from state x_0 to state x_k and of length k, is a finite sequence of transitions $< (x_0, i_0, x_1), (x_1, i_1, x_2), \ldots, (x_{k-1}, i_{k-1}, x_k) >$. (A path may have cycles.) If x_k belongs to a cycle, the path is said to be *infinite*. $\blacksquare$

Note that every set of transitions, be it a transition relation or a path, is denoted by its characteristic function implemented as a BDD.

3.1 Computation Tree Logic

In this work we use Computation Tree Logic, or CTL [5], to specify properties of FSMs. The set of all CTL formulas can be defined inductively in terms of a subset of

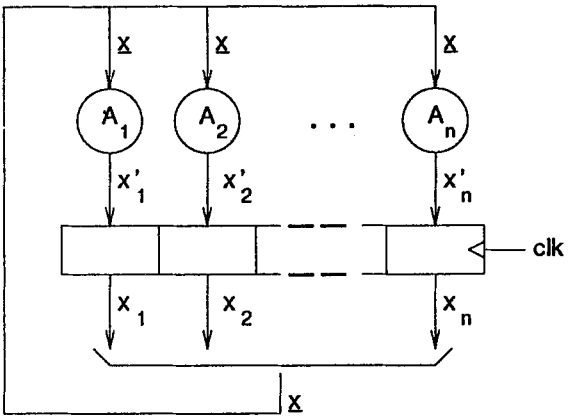

Fig. 2. Interacting Finite State Machines

base CTL formulas. The choice of the subset is not unique. In the following definition, an *atomic proposition* f is a function of the state variables of a FSM. That is, f is true in a state $x \in X$ if $f(x) = 1$.

Definition 3. The set of CTL formulas $\mathcal{F}$ is defined inductively as follows:

1. Every atomic proposition f is a CTL formula.
2. If f and g are CTL formulas, then $\neg f$, $f \cdot g$, $\exists X f$, $\exists G f$, and $\exists[f R g]$ are CTL formulas. ∎

The formula $\exists \boldsymbol{X} f$ is true in a state s if f is true in some successor of s. The formula $\exists \boldsymbol{G} f$ is true in a state s if f is true in every state of some infinite path beginning with s. The formula $\exists[f \boldsymbol{R}\, g]$ is true in a state s if f is true in s and there is some path beginning with a successor of s along which f is true in every state until g is true.

The basis we have chosen is the same as the one presented by other researchers [5, 2] except that the *until* operator $\exists[fUg]$ is replaced by the *repeat* operator $\exists[fRg]$ (see Figure 3). The semantics of $\exists[fRg]$ is similar to that of $\exists[fUg]$, but the paths must have length at least one. More formally, $\exists[fUg] \equiv \exists[fRg] + g$.

Syntactic abbreviations, such as $\forall F f$, which is equivalent to $\neg\exists G \neg f$, are often used for notational convenience. We assume the semantics of CTL formulas is known to the reader. For example, the formula $\forall G \forall F$**enabled** specifies that the signal **enabled** holds infinitely often along every computation path.

A non-nested CTL formula, that is one whose arguments f and g are atomic propositions (where g may be NIL, as in the case of $\exists G f$), will be referred to as a *simple* formula.

4 Model Checking

4.1 Single Machine Model Checker

The input to our single machine model checker is a transition relation T and a base CTL formula F. The output is a set of transitions T^* such that F holds at the present state of each transition.

Definition 4. The *model checker* implements a function $mc : 2^{(X \times I \times X)} \times \mathcal{F} \rightarrow 2^{(X \times I \times X)}$. We denote the output of the model checker as T^*, that is, $T^* = mc(T, F)$. ∎

Since the output is a set of transitions, we have chosen for Definition 3 a set of base formulas that expresses all of CTL logic and can be given semantics in terms of sets of transitions. In this way we can easily define which paths, and consequently, which transitions, are relevant to each property.

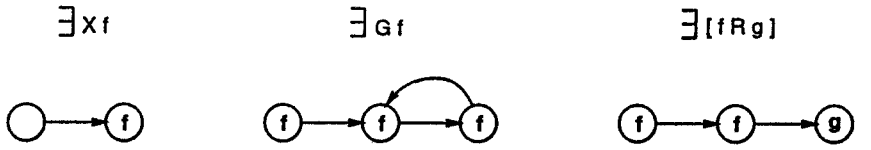

Fig. 3. Base CTL Formulas

Definition 5. ($\exists X f$): $T^* = mc(T, \exists X f)$ contains all transitions $(x, i, x') \in T$ such that $f(x') = 1$. T^* is computed as $T^* = f(x') \cdot T$ ∎

Definition 6. ($\exists G f$): $T^* = mc(T, \exists G f)$ contains all transitions $(x, i, x') \in T$ that belong to some infinite path on which f is true at each state. T^* is computed by the following *greatest* fixed point computation.

$$\begin{aligned} T_0 &= T \cdot f(x) \cdot f(x') \\ T_{n+1} &= T_n \cdot [S_{i,x'}(T_n)]_{x \rightarrow x'} \\ T^* &= T_n, \text{ s.t. } T_{n+1} = T_n \end{aligned}$$ ∎

Definition 7. ($\exists [f R\, g]$): $T^* = mc(T, \exists [f R g])$ contains all transitions $(x, i, x') \in T$ that have present state in f and belong to some path where f is true until g is true. T^* is computed by the following *least* fixed point computation.

$$\begin{aligned} \tilde{T} &= T \cdot f(x) \cdot f(x') \\ T_0 &= T \cdot f(x) \cdot g(x') \\ T_{n+1} &= \tilde{T} \cdot [S_{i,x'}(T_n)]_{x \rightarrow x'} + T_n \\ T^* &= T_n, \text{ s.t. } T_n = T_{n+1} \end{aligned}$$ ∎

Since the set T is finite, the fixed point computations are guaranteed to terminate.

We prove two properties on the relation between the input and output of the model checker.

Proposition 8. Let $T^* = mc(T, F)$. Then $T^* \subseteq T$, for any base CTL formula F.

This result follows since the model checker only removes transitions from T; it never adds transitions. As a consequence, the transition relation returned by the model checker generally describes an incompletely specified machine.

Proposition 9. Let $T^* = mc(T, F)$. If M is another transition relation such that $T^* \subseteq M \subseteq T$, then $mc(M, F) = T^*$.

Think of T as the full transition relation for a machine. As usual, T^* is the output of the model checker when T is the input. Proposition 9 tells us that we can use as input to the model checker any transition relation M that is contained in T and contains T^*, and still get T^* as output. This result will be used in the compositional model checker to justify the removal of "uninteresting" transitions from the component machines.

4.2 Compositional Model Checker

The input to the compositional model checker is an array of transition relations specifying a system of n interacting machines $A_1, A_2, \ldots, A_n$ and a base CTL formula F. The output is the set of states of the global system that satisfy F.

Definition 10. The *compositional model checker* implements a function $cmc : 2^{\underline{X} \times \underline{X}} \times \mathcal{F} \rightarrow 2^{\underline{X}}$. We denote the output of the compositional model checker as $Q(\underline{x}) = cmc(A_1, \cdots, A_n, F)$. ∎

```
    function cmc(array[A_i], F) {
1       for i = 1 to n {      /* project F on A_i and run mc on A_i */
            A_i^* = mc(A_i, F_{A_i});
        }
2       R = ∏(S_{x_i'} A_i^*);        /* compute reducing term */
        for i = 1 to n {      /* find minimum onset of A_i */
            A_i^{*'} = A_i^* · R;
        }
3       for i = 1 to n {      /* use don't cares to minimize BDD for A_i */
            Â_i = min_bdd(A_i, A_i^{*'});
        }
4       M̂ = mc(∏_i Â_i, F);   /* run mc on reduced product */
        Q = S_{x'} M̂;          /* states that satisfy F */
        return Q;
    }
```

Fig. 4. Compositional Model Checker

The compositional model checker procedure *cmc* consists of four phases (Figure 4).

Phase 1 The first phase is to check the components independently. For each machine A_i we compute the reduced component A_i^* by applying the model checker to A_i and the projection of F onto A_i.

Definition 11. Let $A_i(x_i, \underline{x}_{\neq i}, x_i')$ be a transition relation in a system $A_1, A_2, \ldots, A_n$. Let F be a base CTL formula with atomic propositions $f(\underline{x})$ and $g(\underline{x})$. The *projection* F_{A_i} of F onto A_i is obtained by replacing $f(\underline{x})$ with $f(x_i) = S_{\underline{x}_{\neq i}} f(\underline{x})$ and $g(\underline{x})$ with $g(x_i) = S_{\underline{x}_{\neq i}} g(\underline{x})$ in F.

Phase 2 The second phase is to reduce statically each component with respect to the interaction of the other components.

Proposition 12. Consider a system of interacting machines $A_1, A_2, \ldots, A_n$. Let

$$R(\underline{x}) = \prod_{j=1}^{n} (S_{x'_j} A_j)$$

Then $A'_i = A_i \cdot R$ is the smallest subset of A_i which contains $\prod A_j$ and is independent of $\underline{x'}_{\neq i}$.

We apply this proposition in this phase by intersecting each A_i^* with $R(\underline{x}) = \prod(S_{x'_j} A_j^*)$ to obtain $A_i^{*'}$. The transition relations of these machines contain the smallest possible sets of transitions (the minimum onsets) that are needed to verify the given property F.

Phase 3 In the worst case, the size of the AND of two BDDs is the product of the number of nodes of each BDD. Hence, in the third phase, we want to minimize the size of the component BDDs by applying heuristics such as the *restrict operator* [8]. Given an incompletely specified function with onset $f \cdot c$ and don't care set $\overline{c}$, the restrict operator $P_c f$ is a function $f \cdot c \subseteq P_c f \subseteq f + \overline{c}$ that in most cases has a smaller BDD than f. Here, the don't care set derived from phases 1 and 2 is $\overline{c} = A_i - A_i^{*'}$. Thus, we apply the restrict operator to each transition relation with $f = A_i^{*'}$ and $c = \overline{A}_i + A_i^{*'}$, to yield $\hat{A}_i$. Unfortunately, minimizing the sizes of the BDDs for the A_i's does not guarantee that we are minimizing the size of the BDD for the product. Only thorough experimentation will indicate the effectiveness of this heuristic.

Phase 4 The fourth and final phase of the procedure *cmc* is to apply the single machine model checker to the original formula F and the product $\prod \hat{A}_i$ of the minimized components, to determine the states of the product machine that satisfy F. In fact, we can form the product of the minimized components incrementally, applying the model checker after taking the composition with each component. The following theorem states that applying the model checker to the reduced product, gives the same result as if we had taken the naive approach and applied the model checker to the complete product machine.

Theorem 13. Consider a system of interacting machines $A_1, A_2, \ldots, A_n$. Let F be any base CTL formula. Then

$$mc(\prod_{i=1}^{n} \hat{A}_i, F) = mc(\prod_{i=1}^{n} A_i, F).$$

The correctness of our technique relies on our choice of the base formulas for CTL. By allowing only existential path quantifiers in our base formulas, we can guarantee that each reduced component A_i^* contains every transition that would be present in the output of the model checker when applied to the complete product machine. The intuition behind this is that a component "loses behavior" when composed with the rest of the system, relative to its behavior when viewed independently of the system.

On the other hand, our technique fails if we choose as our base formulas those with universal path quantifiers (e.g. $\forall Xp$). To illustrate this, in the example of Figure 5 we have communicating machines $A(x, y, x')$ and $B(y, x, y')$ and propositions $p(x)$ and $q(y)$. State (0,0) in the product machine $A \cdot B$ satisfies the formula $\forall G(p \cdot q)$, even though no state of A satisfies the projected formula $\forall Gp$. In the notation of Section 2, $Q_A = \emptyset$, and thus, $Q_M \not\subseteq Q_A \times Q_B \subseteq X \times Y$.

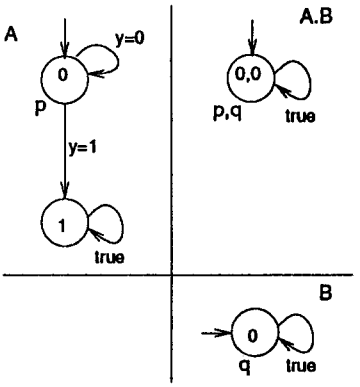

Fig. 5. Interacting FSMs (state (1,0) is not reachable)

4.3 Nested Formulas

A general CTL formula can be nested, that is the propositions can either be explicit sets of states or formulas to be computed. For example, the CTL formula $F = \forall G(\mathbf{req} \rightarrow \forall F\mathbf{ack})$ is a nested formula. For this reason, a general CTL formula is represented as a binary tree. Each node of the tree is a structure composed of a type (e.g. $\exists G$), and two pointers (f and g) to sub-formulas which can either be atomic propositions or other CTL formulas. As in [5], we verify a nested CTL formula by traversing the formula from the leaves to the root. At each level of nesting, we verify one or more simple CTL formulas whose propositions are either given or computed from the previous level. In the example above, the atomic propositions **req**, **ack**, and **reset** are given as BDDs. The formula is then verified as follows. Let $M = array[A_i]$. The formula $F = \forall G(\mathbf{req} \rightarrow \forall F\mathbf{ack})$ can be rewritten as $\neg(\exists[1R\neg(\mathbf{req} \rightarrow \neg\exists G\neg\mathbf{ack})] + \neg(\mathbf{req} \rightarrow \neg\exists G\neg\mathbf{ack}))$, and is satisfied by the set Q of states of M given by:

$$
\begin{aligned}
Q_0 &= cmc(M, F = \neg\mathbf{ack}) \\
Q_1 &= cmc(M, F = \exists G Q_0) \\
Q_2 &= cmc(M, F = \neg Q_1) \\
Q_3 &= cmc(M, F = \mathbf{req} \rightarrow Q_2) \\
Q_4 &= cmc(M, F = \neg Q_3) \\
Q_5 &= cmc(M, F = \exists[1\ R\ Q_4)
\end{aligned}
$$

$$Q_6 = cmc(M, F = Q_4 + Q_5)$$
$$Q = cmc(M, F = \neg Q_6)$$

To verify a nested formula F, we embed the *cmc* function, which verifies a simple CTL formula, in a recursive procedure *rcmc* (Recursive Compositional Model Checker) that traverses the tree of the formula F from the leaves to the root (see Figure 6).

```
function rcmc(array[A_i], F) {
  if (NESTED(F.f)) {     /* if f is nested, apply rcmc to f */
    F.f = rcmc(array[A_i], F.f);
  }
  if (NESTED(F.g)) {     /* if g is nested, apply rcmc to g */
    F.g = rcmc(array[A_i], F.g);
  }
  Q = cmc(array[A_i], F);   /* run cmc on root formula */
  return Q;
}
```

Fig. 6. Recursive Compositional Model Checker

5 Example

In this section, we assess the potential of our compositional technique by verifying a simple system of four interacting machines.

The structure of the system is shown in Figure 7. A resource *server* is shared by two *user* processes. Each user can request the server at any time. If both require it at the same time, the server non-deterministically decides which user to serve. Once acknowledged, a user can release the server at any time, but after a time t_{max} the server is automatically released. The fourth component is an 8-bit *counter* that is started whenever *server* acknowledges a user, and is reset to t_0 when the *server* is released. When the counter reaches the state t_{max} the server is forcefully released. The sizes of the components in BDD nodes are 29 for the server, 14 for each user, and 769 for the counter. The size of the complete product transition relation is 27796. Note that the counter is the component that most contributes to the size of the system.

We first want to check which states satisfy the following property: *if* $\mathbf{req}_1$ *is present and* $\mathbf{req}_2$ *is not present, then* $\mathbf{ack}_1$ *must be present sometime in the future.* In CTL notation this is

$$F = \mathbf{req}_1 \cdot \neg \mathbf{req}_2 \rightarrow \forall F \mathbf{ack}_1.$$

This formula is rewritten in base formulas as

$$F = \mathbf{req}_1 \cdot \neg \mathbf{req}_2 \rightarrow \neg \exists G \neg \mathbf{ack}_1.$$

The largest BDD computed in this verification has a size of 15405 nodes, which is about 55% of the complete product machine. This BDD is computed in verifying the

sub-formula $\exists G \neg \mathbf{ack}_1$, which is the only significant sub-formula in this case. Analyzing the component reductions, we find that only *server* is actually reduced (from 29 to 16 nodes). In fact, the paths in *server* that visit $\mathbf{ack}_1$ are removed. The other components, including the counter, remain unchanged. This is because the variable $\mathbf{ack}_1$ depends on *server* only. The projection of the sub-formula $\exists G \neg \mathbf{ack}_1$ onto the other components is $\exists G 1$, which means: *there exists some infinite path.* Consequently, since all components are completely specified, all transitions belong to some infinite path, and therefore no reduction is achieved. In general, a formula with propositions that depend on only a few machines will yield less reduction than one with propositions on many machines.

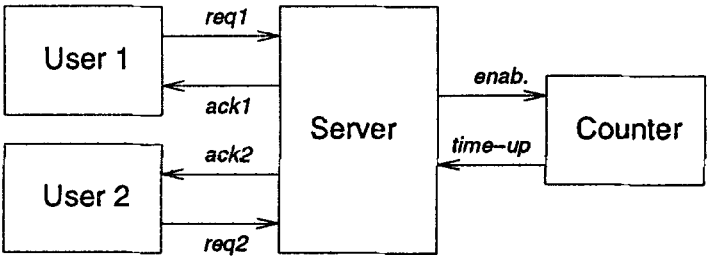

Fig. 7. Four machine system

This result seems promising. However, it is not hard to come up with a case where there is no reduction. For example, let us check for which states the property above is globally true. That is: *for all states of all paths, if* $\mathbf{req}_1$ *is present and* $\mathbf{req}_2$ *is not present, then* ack1 *must be present sometime in the future.* In CTL notation we have

$$H = \forall G(\mathbf{req}_1 \cdot \neg \mathbf{req}_2 \rightarrow \forall F \mathbf{ack}_1)$$

rewritten as

$$H = \neg(\exists[1\ R\ \neg F] + \neg F),$$

where F is defined as above. This property is not satisfied by any state of the system. In fact, from every state of the product machine, there is at least one path to a state where the implication above does not hold. In this case, the complete product machine must be computed. This happens because the sub-formula $\exists[1\ R\ \neg F] + \neg F$, which reads *there exists some path to* $\neg F$, is trivially true for every state of a strongly connected system where $\neg F$ is true in at least one state. Thus, no paths are removed and no reduction is achieved in verifying this sub-formula.

This example shows how the amount of reduction depends on the structure of the machine being checked and on the formula being verified. If the machine is strongly connected, little reduction is expected because the paths tend to span the machine. On the other hand, if the machine is not strongly connected, then more reduction may be possible because the paths may be bound to some regions of the graph. Formulas that have the tautology as a proposition (notably $\forall G p$ rewritten as $\exists[1\ R\ \neg p] + \neg p$) are most sensitive to the structure of the machine.

In this example, if we could compute $\forall GF$ directly, without reexpressing it in terms of a formula with an existential path quantifier, then we would achieve some reduction. However, this does not fit in our method because Theorem 13 does not hold

under the universal path quantifier. Nonetheless, in many cases it may be useful to trade off the exact (i.e. necessary and sufficient) result for a sufficient but computable result.

6 Conclusions

We have described a method for verifying a CTL property on a system of interacting FSMs. It turns out that for some properties, by first applying property-dependent reductions to the component machines before building the product machine, we will be able to verify larger systems than is currently possible. Our method is fully automatic, and produces the exact set of states which satisfy a given CTL property.

The core of the compositional model checker has been implemented. It remains to provide an interface to the verification system being developed at UC Berkeley, a system that will be tightly linked with the Sequential Interactive Synthesis system [14]. Once this interface is completed, we plan to test thoroughly the effectiveness of our method on large examples.

Several ideas to increase the power of our approach deserve investigation. Clarke's interface rule could be added as a preprocessor to our approach for those properties on a single machine, and Burch's method for computing relational products could be used to handle the product of the reduced components. Another idea is to *add* transitions to each component to reduce the size of the BDDs. Of course, these transitions must be chosen judiciously so that the final result is not altered. We would like to further investigate the general problem of finding a cover, with a minimal BDD representation, of an incompletely specified function. Also, using a formula as a criterion for repartitioning the original system into a new set of FSMs, may lead to smaller BDDs for each component.

We could relax the exact nature of our approach by never taking the product of the reduced components. In this case, we would be limited to stating conclusively that a certain state does *not* satisfy a given property. Conversely, as suggested by the example, we could compute the universally quantified properties directly, as in COSPAN, and be content with knowing that a certain state *does* satisfy a given property, although others may also. Similarly, another interesting development would be to combine automatic reduction with homomorphic abstraction. Finally, we may consider extending our approach to other logics, such as ECTL and ω-regular languages.

7 Acknowledgements

We wish to thank Kolar L. Kodandapani for his contribution in the early stages of this work, and Bob Kurshan for stimulating discussions which sparked our interest in automatic reduction.

References

1. R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. on Computers*, C-35(8), pp. 677-691, Aug. 1986.
2. J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill, "Sequential Circuit Verification Using Symbolic Model Checking," in *Proc. of 27th Design Automation Conference*, pp. 46-51, June 1990.

3. J. R. Burch, E. M. Clarke, and D. E. Long, "Representing Circuits More Efficiently in Symbolic Model Checking," in *Proc. of 28th Design Automation Conference*, pp. 403-407, June 1991.
4. M. Chiodo, T. R. Shiple, A. Sangiovanni-Vincentelli, and R. K. Brayton, "Automatic Reduction in CTL Compositional Model Checking," Memorandum No. UCB/ERL M92/55, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, Jan. 1992.
5. E. M. Clarke, E. A. Emerson, and P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Trans. Prog. Lang. Syst.*, 8(2), pp. 244-263, 1986.
6. E. M. Clarke, D. E. Long, and K. L. McMillan, "Compositional Model Checking," in *Proc. of the 4th Annual Symposium on Logic in Computer Science*, Asilomar, CA, June 1989.
7. E. M. Clarke, O. Grumberg and D. E. Long, "Model Checking and Abstraction," in *Proc. of Principles of Programming Languages*, Jan. 1992.
8. O. Coudert, C. Berthet, and J. C. Madre, "Verification of Synchronous Sequential Machines Based on Symbolic Execution," in *Lecture Notes in Computer Science: Automatic Verification Methods for Finite State Systems*, vol. 407, editor J. Sifakis, Springer-Verlag, pp. 365-373, June 1989.
9. S. Graf and B. Steffen, "Compositional Minimization of Finite State Systems," in *Lecture Notes in Computer Science: Proc. of the 1990 Workshop on Computer-Aided Verification*, vol. 531, editors R. P. Kurshan and E. M. Clarke, Springer-Verlag, pp. 186-196, June 1990.
10. O. Grumberg and D. E. Long, "Model Checking and Modular Verification," in *Lecture Notes in Computer Science: Proc. CONCUR '91: 2nd Inter. Conf. on Concurrency Theory*, vol. 527, editors J. C. M. Baeten and J. F. Groote, Springer-Verlag, Aug. 1991.
11. J. E. Hopcroft, "An $n \log n$ Algorithm for Minimizing the States in a Finite Automaton," in *The Theory of Machines and Computation*, New York: Academic Press, pp. 189-196, 1971.
12. R. P. Kurshan and K. L. McMillan, "A Structural Induction Theorem for Processes," in *Proc. of 8th ACM Symp. on Principles of Distributed Computing*, Aug. 1989.
13. R. P. Kurshan, "Analysis of Discrete Event Coordination," in *Lecture Notes in Computer Science: Proc. REX Workshop on Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness*, vol. 430, editors J. W. de Bakker, W. -P. de Roever, and G. Rozenberg, Springer-Verlag, May 1989.
14. E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Sequential Circuit Design Using Synthesis and Optimization," in *Proc. of International Conference on Computer Design*, Oct. 1992.
15. H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Implicit State Enumeration of Finite State Machines using BDDs," in *Proc. of IEEE International Conference on Computer-Aided Design*, pp. 130-133, Nov. 1990.
16. P. Wolper and V. Lovinfosse, "Verifying Properties of Large Sets of Processes with Network Invariants," in *Lecture Notes in Computer Science: Automatic Verification Methods for Finite State Systems*, vol. 407, editor J. Sifakis, Springer-Verlag, pp. 68-80, June 1989.