Verifying Parameterized Networks

E. M. CLARKE
Carnegie Mellon University
and
O. GRUMBERG
Technion
and
S. JHA
Carnegie Mellon University

This article describes a technique based on network grammars and abstraction to verify families of state-transition systems. The family of state-transition systems is represented by a context-free network grammar. Using the structure of the network grammar our technique constructs a process invariant that *simulates* all the state-transition systems in the family. A novel idea introduced in this article is the use of *regular languages* to express state properties. We have implemented our techniques and verified two nontrivial examples.

Categories and Subject Descriptors: B.6.2 [**Logics Design**]: Reliability and Testing; D.2.4 [**Software Engineering**]: Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Theory, Verification

Additional Key Words and Phrases: Model checking, parameterized systems, process invariants, temporal logic

1. INTRODUCTION

Automatic verification of state-transition systems using temporal logic model checking has been investigated by numerous authors [Burch et al. 1992; Clarke and Emerson 1981 Clarke et al. 1986; Lichtenstein and Pnueli 1985; Quielle and Sifakis 1982]. The basic model-checking problem can be stated as

> *Given a state-transition system P and a temporal formula f, determine whether P satisfies f.*

E.M. Clarke and S. Jha were supported by the National Science Foundation under Grant no. CCR-8722633 and in part by the Semiconductor Research Corporation under contract 92-DJ-294. O. Grumberg was partially supported by grant no. 120-732 from The United States-Israel Binational Science Foundation (BSF), Jerusalem, Israel.

Authors' addresses: E.M. Clarke, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213; O. Grumberg, Computer Science Department, The Technion, Haifa 32000, Israel; S. Jha, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA 15213.

Current model checkers can only verify a single state-transition system at a time. The ability to reason automatically about entire families of similar state-transition systems is an important research goal. Such families arise frequently in the design of reactive systems in both hardware and software. The infinite family of token rings is a simple example. More complicated examples are trees of processes consisting of one root, several internal and leaf nodes, and hierarchical buses with different numbers of processors and caches.

The verification problem for a family of similar state-transition systems can be formulated as follows:

> *Given a family* $F = \{P_i\}_{i=1}^{\infty}$ *of systems* P_i *and a temporal formula* f*, verify that each state-transition system in the family* F *satisfies* f*.*

In general the problem is undecidable [Apt and Kozen 1986]. However, for *specific* families the problem may be solvable. This possibility has been investigated by Browne et al. [1989]. They consider the problem of verifying a family of token rings. In order to verify the entire family, they establish a *bisimulation* relation between a two-process token ring and an n-process token ring for any $n \geq 2$. It follows that the two-process token ring and the n-process token ring satisfy exactly the same temporal formulas. The drawback of their technique is that the bisimulation relation has to be constructed manually.

Two other research groups [Emerson and Namjoshi 1995; German and Sistla 1992] show that it is possible to automatically solve the parameterized model-checking problems for some special cases. They prove that for rings composed of certain kinds of processes there exists a k such that the correctness of the ring with k processes implies the correctness of rings of arbitrary size. In Vernier [1994] an alternative method is proposed for checking properties of parameterized systems. In this framework there are two types of processes: G_s (slave processes) and G_c (control processes). There can be many slave processes with type G_s, but only one control process with type G_c. The slave processes G_s can only communicate with the control process G_c.

Our technique is based on finding *network invariants* [Kurshan and McMillan 1989; Wolper and Lovinfosse 1989]. Given an infinite family $F = \{P_i\}_{i=1}^{\infty}$, this technique involves constructing an invariant I such that $P_i \preceq I$ for all i. The preorder $\preceq$ preserves the property f we are interested in, i.e., if I satisfies f then P_i satisfies f. Once the invariant I is found, traditional model-checking techniques can be used to check that I satisfies f. The original technique in Kurshan and McMillan [1989] and Wolper and Lovinfosse [1989] can handle only networks with one repetitive component. Also, the invariant I has to be explicitly provided by the user.

In Marelly and Grumberg [1991] and Shtadler and Grumberg [1989] context-free network grammars are used to generate infinite families of processes with multiple repetitive components. Using the structure of the grammar they generate an invariant I and then check that I is *equivalent* to every process in the language of the grammar. If the method succeeds, then the property can be checked on the invariant I. The requirement for equivalence between all systems in F is too strong in practice and severely limits the usefulness of the method. Our goal is to replace *equivalence* with a suitable *preorder* while still using network grammars.

We first address the question of how to specify a property of a global state of a system consisting of many components. Such a state is an n-tuple, $(s_1, \ldots, s_n)$ for some n. Typical properties we may want to describe are "some component is in a state s_i," "at least (at most) k components are in state s_i," or "if some component is in state s_i, then some other component is in state s_j." These properties are conveniently expressed in terms of *regular languages*. Instead of n-tuple $(s_1, \ldots, s_n)$ we represent a global state by the word $s_1 \cdot \ldots \cdot s_n$ that can either belong to a given regular language $\mathcal{L}$, thus having the property $\mathcal{L}$, or not belong to $\mathcal{L}$, thus not having the property. As an example, consider a mutual exclusion algorithm for processes on a ring. Let nc be the state of a process outside the critical section and let cs be the state inside the critical section. The regular language $nc^*\ cs\ nc^*$ specifies the global states of rings with any number of processes in which exactly one process is in its critical section.

After deciding the types of state properties we are interested in, we can partition the set of global states into equivalence classes according to the properties they possess. Using these classes as *abstract states* and defining an abstract transition relation appropriately, we get an *abstract state-transition system* that is greater in the simulation preorder $\preceq$ than any system in the family. Thus, whenever a $\forall CTL^*$ [Clarke et al. 1992] formula is true of the abstract system it is also true of any of the systems in the family.

Following Marelly and Grumberg [1991] and Shtadler and Grumberg [1989] we restrict our attention to families of systems derived by *network grammars*. The advantage of such a grammar is that it is a finite (and usually small) representation of an infinite family of finite-state systems (referred to as the language of the grammar). Whereas Marelly and Grumberg [1991] and Shtadler and Grumberg [1989] use the grammar in order to find a representative that is equivalent to any system derived by the grammar, we find a representative that is greater in the simulation preorder than all of the systems that can be derived using the grammar.

In order to simplify the presentation we first consider the case of an unspecified composition operator. The only required property of this operator is that it must be monotonic with respect to the simulation preorder. At a later stage we apply these ideas to synchronous models (Moore machines) that are particularly suitable for modeling hardware designs. We also demonstrate our techniques on an asynchronous model of computation. To apply our ideas, we use a simple mutual exclusion algorithm as the running example. Two realistic examples are given in a separate section.

Our article is organized as follows. In Section 2 we define the basic notions of network grammars and regular languages as state properties. In Section 3 we define abstract systems. Section 4 presents our verification method. In Section 5 we describe a synchronous model of computation and show that it is suitable for our technique. Section 6 describes an asynchronous model of computation. In Section 7 we apply our method to two nontrivial examples. Section 8 concludes with some directions for future research.

2. DEFINITIONS AND FRAMEWORK

Definition 2.1. A *Labeled Transition System* or an *LTS* is a structure $M = (S, R, ACT, S_0)$ where S is the set of states; $S_0 \subseteq S$ is the set of initial states;

ACT is the set of actions; and $R \subseteq S \times ACT \times S$ is the *total* transition relation, such that for every $s \in S$ there is some action a and some state s' for which $(s, a, s') \in R$. We use $s \xrightarrow{a} s'$ to denote that $(s, a, s') \in R$.

A path π from a state s in an LTS M is a sequence of transitions $s = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \cdots$. The suffix of π starting from the ith state is denoted by π^i. The ith state on the path π is denoted by $\pi[i]$. Let L_{ACT} be the class of LTSs whose set of actions is a subset of ACT. Let $L_{(S,ACT)}$ be the class of LTSs whose state set is a subset of S and whose action set is the subset of ACT.

Definition 2.2. A function $\| : L_{ACT} \times L_{ACT} \mapsto L_{ACT}$ is called a *composition function* iff given two LTSs $M_1 = (S_1, R_1, ACT, S_0^1)$ and $M_2 = (S_2, R_2, ACT, S_0^2)$ in the class L_{ACT}, $M_1 \| M_2$ has the form $(S_1 \times S_2, R', ACT, S_0^1 \times S_0^2)$. The definition of R' depends upon the exact semantics of the composition function. Notice that we write the composition function in infix notation.

Throughout this article S^i denotes the vectors of length i, with components drawn from the set S. Equivalently, we also interpret S^i as words of length i with S as the alphabet. Our verification method handles a set of LTSs referred to as a *network*. Intuitively, a network consists of a set of LTSs obtained by composing any number of LTSs from the set $L_{(S,ACT)}$. Thus, each LTS in a network is defined over the set of actions ACT and over a set of states in S^i.

Definition 2.3. Given a state set S and a set of actions ACT, any subset of $\bigcup_{i=1}^{\infty} L_{(S^i,ACT)}$ is called a *network* on the tuple (S, ACT).

2.1 Network Grammars

Following Marelly and Grumberg [1991] and Shtadler and Grumberg [1989] we use context-free network grammars as a formalism to describe networks. The set of all LTSs derived by a network grammar (as "words" in its language) constitutes a network. Let S be a state set and ACT be a set of actions. Then, $G = \langle T, N, \mathcal{P}, \mathcal{S} \rangle$ is a grammar where

—T is a set of terminals, each of which is an LTS in $L_{(S,ACT)}$; these LTSs are sometimes referred to as *basic processes*;

—N is a set of nonterminals; each nonterminal defines a network;

—$\mathcal{P}$ is a set of production rules of the form

$$A \rightarrow B \|_i C$$

where $A \in N$, and $B, C \in T \cup N$, and $\|_i$ is a composition function. Notice that each rule may have a different composition function;

—$\mathcal{S} \in N$ is the start symbol that represents the network generated by the grammar.

Example 2.1.1. We clarify the definitions using a network consisting of LTSs that perform a simple mutual exclusion using a token ring algorithm. The production rules of a grammar that produces rings with one process Q and at least two processes P are given below. P and Q are terminals, and A and $\mathcal{S}$ are nonterminals where $\mathcal{S}$ is the start symbol.

$$\mathcal{S} \rightarrow Q \| A$$

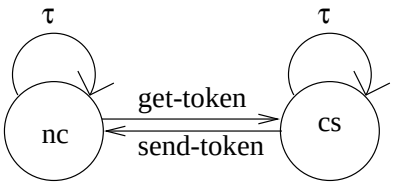

Fig. 1. Process Q, if $S_0 = \{cs\}$; process P if $S_0 = \{nc\}$.

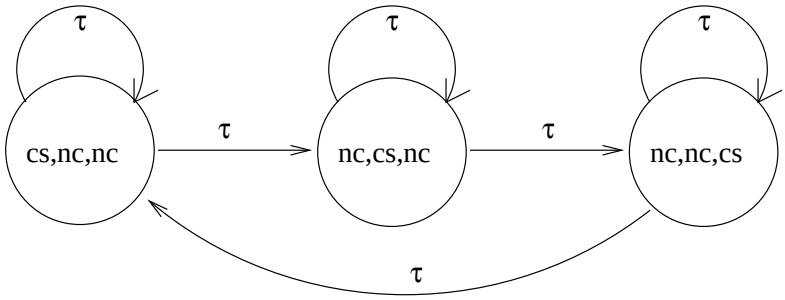

Fig. 2. Reachable states in LTS $Q\|P\|P$.

$$A \;\rightarrow\; P\|A$$

$$A \;\rightarrow\; P\|P$$

P and Q are LTSs defined over the set of states $\{nc, cs\}$ and the set of actions $ACT = \{\tau,\ \texttt{get-token}, \texttt{send-token}\}$. They are identical, except for their initial state, which is cs for Q and nc for P. Their transition relation is shown in Figure 1.

For this example we assume a synchronous model of computation in which at any moment each process takes a step. We do not give a formal definition of the model here. In Sections 5 and 6 we suggest suitable definitions for synchronous and asynchronous models. Informally, a process can always perform a τ action. However, it can perform a `get-token` action if and only if the process to its left is ready to perform a `send-token` action. In the composed LTS, the two actions `get-token` and `send-token` are replaced by τ.

We can apply the following derivation

$$\mathcal{S} \;\Rightarrow\; Q\|A \;\Rightarrow\; Q\|P\|P$$

to obtain the LTS $Q\|P\|P$. Figure 2 shows the reachable states (with corresponding transitions) for the LTS $Q\|P\|P$.

2.2 Specification Language

Let S be a set of states. From now on we assume that we have a network defined by a grammar G on the tuple (S, ACT). The automaton defined in the following has S as its alphabet. Thus, it accepts words that are sequences of state names.

Definition 2.2.1. $\mathcal{D} = (Q, q_0, \delta, F)$ is a *deterministic automaton* over S, where

(1) Q is the set of automaton states,

(2) $q_0 \in Q$ is the initial state,
(3) $\delta \subseteq Q \times S \times Q$ is the transition relation,
(4) $F \subseteq Q$ is the set of accepting states, and
(5) $\mathcal{L}(\mathcal{D}) \subseteq S^*$ is the set of words accepted by $\mathcal{D}$.

Our goal is to specify a network of LTSs composed of any number of basic processes. We use finite automata over S in order to specify atomic state properties. Since a state of an LTS is a tuple from S^i, for some i, we can view such a state as a word in S^*. Let $\mathcal{D}$ be an automaton over S. We say that s satisfies $\mathcal{D}$, denoted $s \models \mathcal{D}$, iff $s \in \mathcal{L}(\mathcal{D})$. Our specification language is a *universal branching temporal logic* (e.g., $\forall CTL$, $\forall CTL^*$ [Grumberg and Long 1994]) with finite automata over S as the atomic formula. In this article we only define $\forall CTL$, but the theorems hold for $\forall CTL^*$ as well.

Definition 2.2.2. The logic $\forall CTL$ is inductively defined as follows:

(1) The constant *true* is an atomic formula.
(2) Any specification automaton $\mathcal{D}$ is an atomic formula.
(3) If ϕ is an atomic formula, then $\neg\phi$ is a formula.
(4) If ϕ and ψ are formulas, then $\phi \wedge \psi$ and $\phi \vee \psi$ are formulas.
(5) If ϕ and ψ are formulas, then $\mathbf{AX}\,\phi$, $\mathbf{A}(\phi\,\mathbf{V}\,\psi)$, and $\mathbf{A}(\phi\,\mathbf{U}\,\psi)$ are formulas.

Consider a LTS $M = (S^i, R, ACT, S_0)$. Given $s \in S^i$, the satisfaction relation ($\models$) is inductively defined as follows:

(1) $s \models \mathcal{D} \Leftrightarrow s \in \mathcal{L}(\mathcal{D})$,
(2) $s \models \neg f_1 \Leftrightarrow s \not\models f_1$,
(3) $s \models f_1 \vee f_2 \Leftrightarrow s \models f_1$ or $s \models f_2$,
(4) $s \models f_1 \wedge f_2 \Leftrightarrow s \models f_1$ and $s \models f_2$,
(5) $s \models \mathbf{AX}\,g_1$ iff for all states s' and for all actions α if $s \xrightarrow{\alpha} s'$, then $s' \models g_1$,
(6) $s \models \mathbf{A}(g_1\,\mathbf{U}\,g_2)$ iff for all paths π starting from s there exists $k \geq 0$ such that $\pi[k] \models g_2$ and for all $0 \leq j < k, \pi[j] \models g_1$,
(7) $s \models \mathbf{A}(g_1\,\mathbf{V}\,g_2)$ iff for all paths π starting from s and for every $k \geq 0$, if $\pi[j] \not\models g_1$ for all $0 \leq j < k$, then $\pi[k] \models g_2$.

The operators **AG** and **AF** can be defined as follows:

$$\mathbf{AG}\,f = \mathbf{A}(\text{false}\,\mathbf{V}\,f)$$
$$\mathbf{AF}\,f = \mathbf{A}(\text{true}\,\mathbf{U}\,f)$$

Example 2.2.3. Consider again the network of Example 2.1.1. Let $\mathcal{D}$ be the automaton of Figure 3, defined over $S = \{cs, nc\}$, with $\mathcal{L}(\mathcal{D}) = \{nc\}^* cs \{nc\}^*$. The formula $\mathbf{AG}\,\mathcal{D}$ specifies mutual exclusion, i.e., at any moment there is exactly one process in the critical section. Let $\mathcal{D}'$ be an automaton that accepts the language $cs\{nc\}^*$, then the formula $\mathbf{AG}\,\mathbf{AF}\,\mathcal{D}'$ expresses nonstarvation for process Q. Note that, for our simple example nonstarvation is guaranteed only if some kind of fairness is assumed.

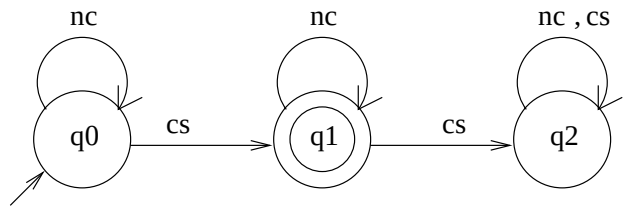

Fig. 3. Automaton $\mathcal{D}$ with $\mathcal{L}(\mathcal{D}) = \{nc\}^* cs \{nc\}^*$.

3. ABSTRACT LTS

In the following sections we define abstract LTSs and abstract composition in order to reduce the state space required for the verification of networks. The abstraction should preserve the logic under consideration. In particular, since we use $\forall CTL$, there must be a *simulation preorder* $\preceq$ such that the given LTS is smaller by $\preceq$ than the abstract LTS. We also require that composing two abstract states will result in an abstraction of their composition. This will allow us to replace the abstraction of a composed LTS by the composition of the abstractions of its components. For the sake of simplicity, we first assume that the specification language contains a single atomic formula $\mathcal{D}$. We later show how to extend the framework to a set of atomic formulas.

3.1 State Equivalence

We start by defining an equivalence relation over the state set of an LTS. The equivalence classes are then used as the states of the abstract LTS. Given an LTS M, we define an equivalence relation on the states of M, such that if two states are equivalent then they both either satisfy or falsify the atomic formula. In other words the two states are either both accepted or both rejected by the automaton $\mathcal{D}$. We also require that our equivalence relation be preserved under composition. This means that if s_1 is equivalent to s_1' and s_2 is equivalent to s_2' then (s_1, s_2) is equivalent to (s_1', s_2').

We use $h(M)$ to denote the abstract LTS corresponding to M. The straightforward definition that defines s and s' to be equivalent iff they both are in or not in the language $\mathcal{L}(\mathcal{D})$ has the first property, but does not have the second one. The following example illustrates this point.

Example 3.1.1. Consider LTSs defined by the grammar of Example 2.1.1. Let $\mathcal{D}$ be the automaton in Figure 3, i.e., $\mathcal{L}(\mathcal{D})$ is the set of states that have exactly one component in the critical section. Let s_1, s_1', s_2, s_2' be states such that $s_1, s_1' \in \mathcal{L}(\mathcal{D})$, and $s_2, s_2' \notin \mathcal{L}(\mathcal{D})$. Further assume that the number of components in the critical section are 0 in s_2 and 2 in s_2'. Clearly, $(s_1, s_2) \in \mathcal{L}(\mathcal{D})$ but $(s_1', s_2') \notin \mathcal{L}(\mathcal{D})$. Thus, the equivalence is not preserved under composition.

We therefore need a more refined equivalence relation. Our notion of equivalence is based on the idea that a word $w \in S^*$ can be viewed as a function on the set of states of an automaton. We define two states to be equivalent if and only if they induce the same function on the automaton $\mathcal{D}$. Formally, given an automaton $\mathcal{D} = (Q, q_0, \delta, F)$ and a word $w \in S^\star$ the *function induced* by w on Q, $f_w : Q \mapsto Q$,

is defined by

$$f_w(q) = q' \text{ iff } q \xrightarrow{w} q'.$$

Note that $w \in \mathcal{L}(\mathcal{D})$ if and only if $f_w(q_0) \in F$, i.e., w takes the initial state to a final state. The set of functions associated with an automaton $\mathcal{D}$ is a monoid and has been studied extensively (see Eilenberg [1974]).

Let $\mathcal{D} = (Q, q_0, \delta, F)$ be a deterministic automaton. Let f_w be the function induced by a word w on Q. We define two states s and s' to be *equivalent*, denoted by $s \equiv s'$, iff $f_s = f_{s'}$. It is easy to see that $\equiv$ is an equivalence relation. The function f_s corresponding to the state s is called the *abstraction* of s and is denoted by $h(s)$. Let $h(s) = f_1$ and $h(s') = f_1'$. The abstract state of (s, s') is $h((s, s')) = f_1 \circ f_1'$ where $f_1 \circ f_1'$ denotes composition of functions. We follow the convention that when we write $f_1 \circ f_1'$ we apply f_1 before f_1'. Note that $s \equiv s'$ implies that $s \in \mathcal{L}(\mathcal{D}) \Leftrightarrow s' \in \mathcal{L}(\mathcal{D})$. Thus, we have $s \models \mathcal{D}$ iff $s' \models \mathcal{D}$. We also have the following result.

LEMMA 3.1.2. *If* $h(s_1) = h(s_2)$ *and* $h(s_1') = h(s_2')$ *then* $h((s_1, s_1')) = h((s_2, s_2'))$.

In order to interpret specifications on the abstract LTSs, we extend $\models$ to abstract states so that $h(s) \models \mathcal{D}$ iff $f_s(q_0) \in F$. This guarantees that $s \models \mathcal{D}$ iff $h(s) \models \mathcal{D}$.

Our framework can be easily extended to any set of atomic formulas. The restriction to one atomic formula was made in order to simplify the presentation. However, in practice we may want to have several such formulas, related by boolean and temporal operators. The notion of equivalence is extended to any set of atomic formulas in the following way. Let $\mathcal{AF} = \{\mathcal{D}_1, \ldots, \mathcal{D}_k\}$ be a set of atomic formulas, where $\mathcal{D}_i = (Q_i, q_0^i, \delta_i, F_i)$. Let f_s^i be the function induced by s on Q_i. Then, two states s and s' are *equivalent* if and only if for all i, $f_s^i = f_{s'}^i$. The abstraction of s is now $h(s) =< f_s^1, \ldots, f_s^k >$, and we have that, if $s \equiv s'$ then for every i, $s \in \mathcal{L}(\mathcal{D}_i) \Leftrightarrow s' \in \mathcal{L}(\mathcal{D}_i)$. The relation $\models$ is extended to abstract states by defining $h(s) \models \mathcal{D}_i$ iff $f_s^i(q_0^i) \in F_i$. Thus, we again have that for every $\mathcal{D}_i \in \mathcal{AF}$, $s \models \mathcal{D}_i$ iff $h(s) \models \mathcal{D}_i$. Recall that s and s' are interpreted as words over the alphabet S.

Example 3.1.3. Consider again the automaton $\mathcal{D}$ of Figure 3 over $S = \{cs, nc\}$. $\mathcal{D}$ induces functions $f_s : Q \mapsto Q$, for every $s \in S^*$. Actually, there are only three different functions, each identifying an equivalence class over S^*:

$f_1 = \{(q_0, q_0), (q_1, q_1), (q_2, q_2)\}$ represents all $s \in nc^*$ (i.e., $f_s = f_1$ for all $s \in nc^*$);
$f_2 = \{(q_0, q_1), (q_1, q_2), (q_2, q_2)\}$ represents all $s \in nc^*\ cs\ nc^*$; and
$f_3 = \{(q_0, q_2), (q_1, q_2), (q_2, q_2)\}$ represents all $s \in nc^*\ cs\ nc^*\ cs\ \{cs, nc\}^*$.

3.2 Abstract Process and Abstract Composition

Let $\mathcal{F}_\mathcal{D}$ be the set of functions corresponding to the deterministic automaton $\mathcal{D}$. Let Q be the set of states $\mathcal{D}$. In the worst case $|\mathcal{F}_\mathcal{D}| = |Q|^{|Q|}$, but in practice the size is much smaller. Note that $\mathcal{F}_\mathcal{D}$ is also the set of abstract states for $s \in S^*$ with respect to $\mathcal{D}$. Subsequently, we apply abstraction both to states $s \in S^*$ and to abstract states $h(s)$. To unify notation we first extend the abstraction function h to $\mathcal{F}_\mathcal{D}$ by setting $h(f) = f$ for $f \in \mathcal{F}_\mathcal{D}$. We also extend the abstraction function h to $(S \cup \mathcal{F}_\mathcal{D})^\star$ in the natural way, i.e., $h((a_1, a_2, \cdots, a_n)) = h(a_1) \circ \cdots \circ h(a_n)$. From now on we consider LTSs in the network $\mathcal{N}$ on the tuple $(S \cup \mathcal{F}_\mathcal{D}, ACT)$.

We define abstract LTSs. The abstract transition relation is defined in the usual manner (see Dams et al. [1994]). If there is a transition between two states in the original structure, then there is a transition between corresponding states in the abstract structure. Formally, this is expressed as follows:

Definition 3.2.1. Given an LTS $M = (S^i, R, ACT, S_0)$ in the network $\mathcal{N}$, the corresponding *abstract LTS* is defined by $h(M) = (S^h, R^h, ACT, S_0^h)$, where

—$S^h = \{h(s) \mid s \in S^i\}$ is the set of abstract states,
—$S_0^h = \{h(s) \mid s \in S_0\}$, and
—the relation R^h is defined as follows. For any $h_1, h_2 \in S^h$, and $a \in ACT$

$$(h_1, a, h_2) \in R^h \Leftrightarrow \exists s_1, s_2[h_1 = h(s_1) \text{ and } h_2 = h(s_2) \text{ and } (s_1, a, s_2) \in R].$$

We say that M' *simulates* M [Milner 1971] (denoted $M \preceq M'$) if and only if there is a *simulation preorder* $\mathcal{E} \subseteq S \times S'$ that satisfies the following conditions: for every $s_0 \in S_0$ there is $s_0' \in S_0'$ such that $(s_0, s_0') \in \mathcal{E}$. Moreover, for every s, s', if $(s, s') \in \mathcal{E}$ then

(1) $h(s) = h(s')$ and
(2) for every s_1 such that $s \xrightarrow{a} s_1$ there is s_1' such that $s' \xrightarrow{a} s_1'$ and $(s_1, s_1') \in \mathcal{E}$.

Notice that the definition of the simulates relation is using the fact that M and M' are defined on the same set of actions. If $(s, s') \in \mathcal{E}$, we say that $s \preceq s'$. Given a path π in M and a path π' in M', we say that $\pi \preceq \pi'$ iff for all i $\pi[i] \preceq \pi'[i]$.

Lemma 3.2.2. *Let M and M' be two LTSs such that $M \preceq M'$. Let π be a path starting from s in M. If $s \preceq s'$, then there exists a path π' starting from s' in M' such that $\pi \preceq \pi'$.*

Proof. Proof follows from the definition of the simulation relation $\mathcal{E}$. □

Lemma 3.2.3. *We have the following results:*

(1) $M \preceq h(M)$, i.e., $h(M)$ simulates M.
(2) If $M \preceq M'$, then $h(M) \preceq h(M')$.

Proof. For the proof of the first part, consider the simulation relation

$$(s, a) \in \mathcal{E} \Leftrightarrow (h(s) = a).$$

It is easy to prove the $\mathcal{E}$ has all the desired properties. For the second part, let $\mathcal{E}$ be the simulation relation between M and M'. Define the relation $\mathcal{E}_h$ in the following manner:

$$(a, b) \in \mathcal{E}_h \Leftrightarrow \exists s_1, s_2(h(s_1) = a \wedge h(s_2) = b \wedge (s_1, s_2) \in \mathcal{E})$$

It is easy to prove that $\mathcal{E}_h$ is the required simulation relation between $h(M)$ and $h(M')$. □

Recall that the abstraction h guarantees that a state and its abstraction agree on the atomic property corresponding to the automaton $\mathcal{D}$. Based on that and on the previous lemma, the following theorem is obtained. A proof of a similar result appears in Clarke et al. [1992].

THEOREM 3.2.4. *Let ϕ be a formula in $\forall CTL$ over the atomic formula $\mathcal{D}$. Let M and M' be two LTSs such that $M \preceq M'$. Let $s \preceq s'$. Then $s' \models \phi$ implies $s \models \phi$.*

PROOF. The proof is by structural induction on ϕ. Notice that $(s' \models \phi \Rightarrow s \models \phi)$ is equivalent to $(s \not\models \phi \Rightarrow s' \not\models \phi)$. During the course of the proof, we sometimes use the second formulation.

—**Case** $\phi = \mathcal{D}$**:** $h(s) = h(s')$ implies that $s \models \mathcal{D}$ if and only $s' \models \mathcal{D}$.

—**Case** $\phi = f_1 \vee f_2$**:** Assume that $s' \models \phi$. By definition $s' \models f_1$ or $s' \models f_2$. By the inductive hypothesis $s \models f_1$ or $s \models f_2$. Therefore, $s \models \phi$.

—**Case** $\phi = f_1 \wedge f_2$**:** This is very similar to the case given previously.

—**Case** $\phi = \mathbf{AX}\, g_1$**:** Assume that $s \not\models \phi$. Then there exists s_1 such that $s \xrightarrow{\alpha} s_1$, and $s_1 \not\models g_1$. By definition there exists s_1' such that $s' \xrightarrow{\alpha} s_1'$ and $s_1 \preceq s_1'$. By the inductive hypothesis $s_1' \not\models g_1$. Therefore, $s' \not\models \phi$.

—**Case** $\phi = \mathbf{A}(g_1\, \mathbf{U}\, g_2)$**:** Assume that $s \not\models \phi$. This means that there exists a path π starting from s such that for all k, either $\pi[k] \not\models g_2$ or there exists $0 \leq j < k$ such that $\pi[j] \not\models g_1$. Using Lemma 3.2.2 there exists a path π' starting from s' in M' such that $\pi \preceq \pi'$, i.e., for every $i \geq 0$, $\pi[i] \preceq \pi'[i]$. By the induction hypothesis we have that for all k, either $\pi'[k] \not\models g_2$ or there exists $0 \leq j < k$ such that $\pi'[j] \not\models g_1$. Thus, $\pi \not\models g_1\, \mathbf{U}\, g_2$, which implies that $s' \not\models \phi$.

—**Case** $\phi = \mathbf{A}(g_1\, \mathbf{V}\, g_2)$**:** Assume that $s' \models \phi$. We prove that $s \models \phi$. Let π be an arbitrary path starting from s in M. Let π' be a path starting from s' in M' such that $\pi \preceq \pi'$. Since $s' \models \phi$, for all k, either there exists a j such that $0 \leq j < k$, $\pi'[j] \models g_1$ or $\pi'[k] \models g_2$. By the induction hypothesis, we can conclude that for k, $\pi[k] \models g_2$ or there exists j such that $0 \leq j < k$ and $\pi[j] \models g_1$. Therefore, $\pi \models g_1\, \mathbf{V}\, g_2$. Since π is an arbitrary path starting from s, we have that $s \models \phi$.

□

Definition 3.2.5. A composition $\|$ is called *monotonic* with respect to a simulation preorder $\preceq$ if and only if given LTSs such that $M_1 \preceq M_2$ and $M_1' \preceq M_2'$, we have that $M_1 \| M_1' \preceq M_2 \| M_2'$. A network grammar G is called *monotonic* if and only if all rules in the grammar use only monotonic composition operators.

4. THE VERIFICATION METHOD AND THE UNFOLDING HEURISTIC

Given a network grammar G, we associate with each symbol A of the grammar a *representative process* $rep(A)$. A set of representative processes is said to satisfy the *monotonicity property* if and only if

—for every terminal A, $h(rep(A)) \succeq h(A)$, and

—for every rule $A \rightarrow B \| C$ we have the condition

$$h(rep(A)) \;\succeq\; h(h(rep(B)) \| h(rep(C))).$$

We thus have the following theorem:

THEOREM 4.1. *Let G be a monotonic grammar (see Definition 3.2.5) and suppose we can find representatives for the symbols of G that satisfy the monotonicity property. Let A be a symbol of the grammar G, and let a be an LTS derived from A using the rules of the grammar G. Then, $h(rep(A)) \succeq a$.*

PROOF. We prove that $h(rep(A)) \succeq h(a)$. Since $h(a) \succeq a$, the result follows by transitivity. Let $A \Rightarrow^k a$, i.e., A derives a in k steps. Our result is by induction on k.

—($k = 0$): In this case A is a terminal and $a = A$. The result follows from the monotonicity property.

—($k > 0$): In the derivation of a from A, let the first rule be $A \rightarrow B \| C$. Assume $B \Rightarrow^i b$ and $C \Rightarrow^j c$ such that $i < k$, $j < k$, and $a = b \| c$. By the induction hypothesis $h(rep(B)) \succeq h(b)$ and $h(rep(C)) \succeq h(c)$. We have the following equations:

$$\begin{array}{rll} h(rep(B))\|h(rep(C)) & \succeq h(b)\|h(c) & \text{(monotonicity of } \|) \\ & \succeq b\|c & \text{(Lemma 3.2.3 and monotonicity of } \|) \\ h(h(rep(B))\|h(rep(C))) & \succeq h(b\|c) & \text{(Lemma 3.2.3)} \\ & \succeq h(a) & \end{array}$$

By the monotonicity property we have

$$h(rep(A)) \succeq h(h(rep(B))\|h(rep(C))).$$

The result follows. □

4.1 Verification Method

This section describes our verification method. Assume that we are given a monotonic grammar G and a $\forall CTL$ formula ϕ with atomic formulas $\mathcal{D}_1, \cdots, \mathcal{D}_k$. To check that every LTS derived by the grammar G satisfies ϕ we perform the following steps:

(1) For every symbol A in G choose a representative process $rep(A)$ and construct the abstract LTS $h(rep(A))$ with respect to the atomic formulas $\mathcal{D}_1, \cdots, \mathcal{D}_k$.

(2) Check that the set of representatives satisfies the monotonicity property. Theorem 4.1 implies that for every a derived by the grammar G, $h(rep(\mathcal{S})) \succeq a$.

(3) Perform model checking on $h(rep(\mathcal{S}))$ with ϕ as the specification. If $h(rep(\mathcal{S})) \models \phi$, then by Theorem 3.2.4 we can conclude that for all LTSs M derived by the grammar G, $M \models \phi$.

Next, we describe an unfolding heuristic that might help us find the set of representatives which satisfy the monotonicity property.

4.2 The Unfolding Heuristic

We discuss a heuristic that might be helpful in automatically finding monotonic representatives. Initially, the representative of a symbol A in $G = \langle T, N, \mathcal{P}, \mathcal{S} \rangle$ is the LTS that can be derived from A using the minimum number of rules. A further search for monotonic representatives is based on the observation that often certain behaviors only occur when a process is composed with other processes (these other processes provide the environment). This leads to the idea that by *unfolding* the current set of representatives we will find a larger set of potential representatives that might satisfy the monotonicity property. Given two sets of LTSs X and Y we define $X \| Y$ in the following way:

$$X \| Y \ = \ \{p \| q \mid p \in X \text{ and } q \in Y\}$$

An *association* for a grammar G assigns a set of processes to each symbol A of the grammar. Let $\mathcal{N}$ be a network on the tuple (S, ACT) (see Section 2). Formally, an association is a function $\mathcal{AS} : (T \cup N) \rightarrow 2^{\mathcal{N}}$. Given an association $\mathcal{AS}$ for a grammar G, we define an *unfolding* operator. The unfolding of an association $\mathcal{AS}$ is denoted by $\mathcal{U}[\mathcal{AS}]$. $\mathcal{U}[\mathcal{AS}]$ is defined as follows:

—If A is a terminal, $\mathcal{U}[\mathcal{AS}](A) = \mathcal{AS}(A)$.

—Assume that A is a nonterminal. A process $p \in \mathcal{U}[\mathcal{AS}](A)$ iff there exists a rule $A \rightarrow B \| C$ in G such that $p \in \mathcal{AS}(B) \| \mathcal{AS}(C)$ or $p \in \mathcal{AS}(A)$.

Now we define an iterative process to find representatives. First, we describe how to find the initial association. Given a symbol A of the grammar G let $D_k(A)$ be all the processes that can be derived from A (using the rules of G) in $\leq k$ steps. Let $\min_A$ be the minimum k such that $D_k(A)$ is not empty. We define the *initial association* as the association $\mathcal{AS}_0$ such that $\mathcal{AS}_0(A) = D_{\min_A}(A)$. A procedure to compute the initial association is described later. For a terminal A, $\mathcal{AS}_0(A) = \{A\}$. We repeat the following step until we have a nonempty association for each symbol A.

—Assume that $\mathcal{AS}_0(A)$ is empty. Let A_R be the set of rules $A \rightarrow B \| C$ such that $\mathcal{AS}_0(B)$ and $\mathcal{AS}_0(C)$ are nonempty:

$$\mathcal{AS}_0(A) \;=\; \bigcup_{\alpha \in A_R} \mathcal{AS}(\texttt{right}(\alpha))$$

If α is of the form $A \rightarrow B_1 \| \cdots \| B_k$, then

$$\mathcal{AS}(\texttt{right}(\alpha)) \text{ is equal to } \mathcal{AS}(B_1) \| \cdots \| \mathcal{AS}(B_k).$$

Once we find the initial association, we keep unfolding the current association until we find representatives satisfying the monotonicity property. The algorithm is given below.

(1) Set $\mathcal{AS} = \mathcal{AS}_0$, where $\mathcal{AS}_0$ is the initial association.
(2) If there exist representatives for each nonterminal A such that $rep(A) \in \mathcal{AS}(A)$ and the representatives satisfy the monotonicity property, we are done. Otherwise go to the next step.
(3) Create a new association $\mathcal{AS}$ by unfolding the current association, i.e., $\mathcal{AS} = \mathcal{U}[\mathcal{AS}]$. Go back to the previous step.

It is not hard to see that each iteration increases the set of processes associated with a nonterminal. Therefore, as we unfold, we have more choices to find representatives with the required property. Moreover, unfolding results in processes that are a combination of a larger number of basic processes. A (global) state of such a process consists of more local states and therefore may correspond to an abstract state that could not be produced before. For example, consider two abstract processes $I_1 = h(P \| P)$ and $I_2 = h(P \| P \| P)$ where P is an arbitrary process and h an abstraction function. Suppose we are trying to establish the inequality

$$I_1 \;\succeq\; h(I_1 \| h(P)).$$

The preceding inequality might fail because the process $h(I_1 \| h(P))$ might have more abstract states than I_1. This is exactly what happens in the example given in Section 7. On the other hand, I_2 may have more abstract states than I_1 and the equation with I_1 replaced by I_2 might be valid. In the examples we show how unfolding is used to find representatives. In light of the result given in Apt and Kozen [1986] the foregoing procedure might not terminate. Therefore, the user will have to put a limit on the number of iterations. In general, the user might replace the initial association with any arbitrary association (depending on the domain knowledge). In the algorithm given previously, users can also define their own unfolding heuristics.

Notice that if we find representatives with the monotonicity property such that $h(rep(\mathcal{S})) \not\models \phi$, we cannot conclude anything about the correctness of the network derived by the grammar G. In this case the counter-example might aid the user in finding a more refined representative or we may want to apply the unfolding technique again.

5. SYNCHRONOUS MODEL OF COMPUTATION

In this section we develop a synchronous framework that has the properties required by our verification method. We define a synchronous model of computation and a family of composition operators. We show that the composition operators are monotonic with respect to $\preceq$.

Our models are a form of LTSs, $M = (S, R, I, O, S_0)$, that represent *Moore machines*. Traditionally, in Moore machines the outputs are associated with the states. We assume that the outputs are moved from the states to the transitions emanating from it. Our models have an explicit notion of inputs I and outputs O that must be disjoint. In addition, they have a special internal action denoted by τ (called silent action in the terminology of CCS [Milner 1980]). The set of actions is $ACT = \{\tau\} \cup 2^{I \cup O}$, where each noninternal action is a set of inputs and outputs.

The composition of two LTSs M and M' is defined to reflect the synchronous behavior of our model. It corresponds to standard composition of Moore machines. To understand how this composition works, we can think of the inputs and outputs as "wires." If M has an output and M' has an input both named a, then in the composition the output wire a will be connected to the input a. Since an input can accept a signal only from one output, $M \| M'$ will not have a as input. On the other hand, an output can be sent to several inputs; thus $M \| M'$ still has a as output. Consequently, the set of outputs of $M \| M'$ is $O \cup O'$, while the set of inputs is $(I \cup I') \setminus (O \cup O')$.

A transition $s \xrightarrow{a} t$ from s in a machine M with $a = i \cup o$ such that $i \subseteq I$ and $o \subseteq O$ occurs only if the environment supplies inputs i and the machine M produces the outputs o. Assume transitions $s \xrightarrow{a} t$ in M and $s' \xrightarrow{a'} t'$ in M'. There is a transition from (s, s') to (t, t') iff the outputs provided by M agree with the inputs expected by M' and the outputs provided by M' agree with the inputs expected by M.

Formally let $O \cap O' = \emptyset$. The *synchronous composition* of M and M', $M'' = M \| M'$ is defined by

(1) $S'' = S \times S'$,
(2) $S_0'' = S_0 \times S_0'$,
(3) $I'' = (I \cup I') \setminus (O \cup O')$,

(4) $O'' = O \cup O'$, and[1]

(5) $(s, s') \xrightarrow{a''} (s_1, s_1')$ is a transition in R'' iff the following holds: $s \xrightarrow{a} s_1$ is a transition in R and $s' \xrightarrow{a'} s_1'$ is a transition in R' for some a, a' such that $a \cap (I' \cup O') = a' \cap (I \cup O)$ and $a'' = a \cup a'$.

LEMMA 5.1. *The composition* $\|$ *is monotonic with respect to* $\preceq$.

PROOF. Let

$$\begin{aligned} M &= (S, R, I, O, S_0) \\ M' &= (S', R', I', O', S_0') \\ M_1 &= (S_1, R_1, I_1, O_1, S_{1,0}) \\ M_1' &= (S_1', R_1', I_1', O_1', S_{1,0}') \end{aligned}$$

be four Moore machines. Assume that $M \preceq M'$ and $M_1 \preceq M_1'$. Let $\mathcal{E} \subseteq S \times S'$ and $\mathcal{E}_1 \subseteq S_1 \times S_1'$ be the corresponding simulation relations. We say that $((s, s_1), (s', s_1')) \in \mathcal{E} \times \mathcal{E}_1$ iff $(s, s') \in \mathcal{E}$ and $(s_1, s_1') \in \mathcal{E}_1$. We show that $\mathcal{E} \times \mathcal{E}_1$ has the required properties. It is clear from the definition that given states $s_0 \in S_0$ and $s_{0,1} \in S_{1,0}$, there exists a $s_0' \in S_0'$ and $s_{1,0}' \in S_{1,0}'$ such that

$$((s_0, s_{1,0}), (s_0', s_{1,0}')) \in \mathcal{E} \times \mathcal{E}_1.$$

Assume that $((s, s_1), (s', s_1')) \in \mathcal{E} \times \mathcal{E}_1$.

(1) By assumption, we have that $h(s) = h(s')$ and $h(s_1) = h(s_1')$. Therefore, $h((s, s_1)) = h(s) \circ h(s_1)$ is equal to $h((s', s_1')) = h(s') \circ h(s_1')$.

(2) Let $(s, s_1) \xrightarrow{a''} (t, t_1)$ be a transition in $M \| M_1$. This means that there exists a transition $s \xrightarrow{a} t$ in M and a transition $s_1 \xrightarrow{a_1} t_1$ in M_1 such that $a \cap (I_1 \cup O_1) = a_1 \cap (I \cup O)$ and $a'' = a \cup a_1$. By definition there exists t' and t_1' such that $s' \xrightarrow{a} t'$ and $s_1' \xrightarrow{a_1} t_1'$, where $(t, t') \in \mathcal{E}$ and $(t_1, t_1') \in \mathcal{E}_1$. Therefore, $(s', s_1') \xrightarrow{a''} (t', t_1')$ and $((t, t_1), (t', t_1')) \in \mathcal{E} \times \mathcal{E}_1$.

The proof is thus complete. □

5.1 Network Grammars for Synchronous Models

Only a few additional definitions are required in order to adapt our general definition of network grammars to networks comprised of Moore machines. As before a network grammar is a tuple $G = \langle T, N, \mathcal{P}, \mathcal{S} \rangle$, but now every terminal and nonterminal A in $T \cup N$ is associated with a set of inputs I_A and a set of outputs O_A. In G we allow different composition operators $\|_i$ for the different production rules. In order to define the family of operators to be used in this framework we need the following definitions.

A *renaming function* $\mathcal{R}$ is an injection that acts on the inputs and outputs of the Moore machines. When applied to A, it maps inputs to inputs and outputs to outputs such that $\mathcal{R}(I_A) \cap \mathcal{R}(O_A) = \emptyset$. Applying $\mathcal{R}$ to a LTS M results in an

[1]Note that $ACT'' = 2^{I'' \cup O''} \cup \{\tau\}$ is not identical to ACT and ACT'. This is a technical issue that can be resolved by defining some superset of actions from which each LTS takes its actions.

LTS $M' = \mathcal{R}(M)$ with $S = S'$, $S_0 = S'_0$, $I' = \mathcal{R}(I) \setminus \{\tau\}$, $O' = \mathcal{R}(O) \setminus \{\tau\}$, and $(s, a, s') \in R$ iff $(s, \mathcal{R}(a), s') \in R'$.

A *hiding function* $\mathcal{R}_{act}$ for $act \subseteq I \cup O$, is a renaming function that maps each element in act to τ.

A typical composition operator in this family is associated with two renaming functions, $\mathcal{R}_{\texttt{left}}$, $\mathcal{R}_{\texttt{right}}$ and a hiding function $\mathcal{R}_{act}$, in the following way.

$$M \|_i M' = \mathcal{R}_{act}(\mathcal{R}_{\texttt{left}}(M) \| \mathcal{R}_{\texttt{right}}(M')),$$

where $\|$ is the synchronous composition previously defined. The renaming functions are applied to LTSs and not to nonterminals. Therefore, the derivations are composed bottom up. For example, consider the rule of the following form:

$$A \rightarrow \mathcal{R}_{act}(\mathcal{R}_{\texttt{left}}(B) \| \mathcal{R}_{\texttt{right}}(C))$$

In this case first we apply the derivations for B and C to derive two LTS b and c. Then the renaming and hiding functions are applied to complete the derivation.

To use our framework, we need to show that every composition operator used in the grammar is monotonic, i.e., if $M_1 \preceq M_2$ and $M'_1 \preceq M'_2$ then $M_1 \|_i M'_1 \preceq M_2 \|_i M'_2$. The latter means that

$$\mathcal{R}_{act}(\mathcal{R}_{\texttt{left}}(M_1) \| \mathcal{R}_{\texttt{right}}(M'_1)) \preceq \mathcal{R}_{act}(\mathcal{R}_{\texttt{left}}(M_2) \| \mathcal{R}_{\texttt{right}}(M'_2)).$$

The following lemma, together with monotonicity of the synchronous composition $\|$ implies the required result.

LEMMA 5.1.1. *Let M, M' be Moore Machines, and let $\mathcal{R}$ be a renaming function. If $M \preceq M'$ then $\mathcal{R}(M) \preceq \mathcal{R}(M')$.*

PROOF. Let $\mathcal{E}$ be the simulation preorder between M and M'. It is easy to show that $\mathcal{E}$ is also a simulation preorder between $\mathcal{R}(M)$ and $\mathcal{R}(M')$. □

COROLLARY 5.1.2. *The composition operators $\|_i$, previously defined are monotonic.*

Example 5.1.3. We return to Example 2.1.1 and reformulate it within the synchronous framework. During the process we can describe more precisely the processes and the network grammar that constructs rings with any number of processes. The processes P and Q will be identical to those described in Figure 1 except that now we also specify for both processes $I = \{$ `get-token`$\}$ and $O = \{$`send-token`$\}$.

The derivation rules in the grammar apply two different composition operators:

$$\mathcal{S} \rightarrow Q \|_1 A$$

$$A \rightarrow P \|_2 A$$

$$A \rightarrow P \|_2 P$$

$\|_1$ is defined as follows:

—$\mathcal{R}^1_{\texttt{left}}$ maps `send-token` to some new action `cr` (stands for `connect right`) and `get-token` to `cl` (stands for `connect left`).

—$\mathcal{R}^1_{\texttt{right}}$ maps `send-token` to `cl` and `get-token` to `cr`.

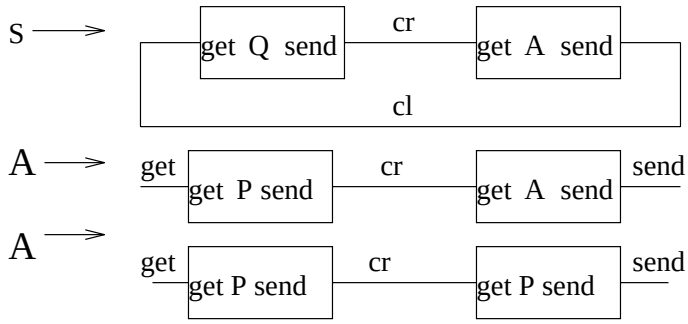

Fig. 4. Derivation rules with renaming.

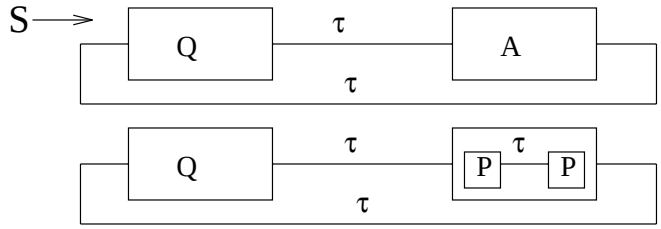

Fig. 5. Derivation of a ring of size 3.

—The hiding function $\mathcal{R}_{act}$ hides both `cr` and `cl` by mapping them to τ.

Thus, the application of this rule results in a network with one terminal Q and one nonterminal A, connected as a *ring*. $\|_2$ is defined by (see Figure 4):

—$\mathcal{R}^2_{\texttt{left}}$ maps `send-token` to `cr` and leaves `get-token` unchanged.
—$\mathcal{R}^2_{\texttt{right}}$ maps `get-token` to `cr` and leaves `send-token` unchanged.
—The hiding function $\mathcal{R}_{act}$ hides `cr`.

For instance, the application of the third rule results in a network in which the nonterminal A is replaced by a LTS consisting of two processes P, such that the `send-token` of the left one is connected to the `get-token` of the right one. The `get-token` of the left process and `send-token` of the right one will be connected according to the connections of A (see Figure 4 and Figure 5).

6. ASYNCHRONOUS MODEL OF COMPUTATION

In this section we describe a model for asynchronous communication. The model is similar to CCS [Milner 1980]. Each process is associated with a set of actions. The symbol τ denotes the internal (noncommunication) actions of a process.

Definition 6.1. A **process** M is a 4-tuple,

$$M = (ACT, S, R, S_0)$$

where

—ACT is a finite set of actions (ports or channels) not containing τ,

—S is a finite set of states,
—R is a labeled transition relation, $R \subseteq S \times (ACT \cup \{\tau\}) \times S$, and
—S_0 is the set of initial states.

When two processes M and M' are combined into a new process $M \| M'$, zero or more channels are formed. Each channel pairs a port of M and a port of M' that have identical names. The channel is used merely for synchronization, i.e., no data are transferred. However, data can be encoded by sequences of synchronizations. The unpaired ports of M and M' become the ports of the combined process $M \| M'$.

Definition 6.2. The **composition** of $M = (S, R, ACT, S_0)$ and $M' = (S', R', ACT', S_0')$ is a new process

$$M'' = M \| M' = (S'', R'', ACT'', S_0'')$$

where

—$ACT'' = (ACT \cup ACT') \backslash (ACT \cap ACT')$,
—$S'' = (S \times S')$,
—$S_0'' = S_0 \times S_0'$, and
—$R'' =$

$$\begin{aligned}
&\{((s,s'),\tau,(s_1,s_1')) \mid \exists\alpha : s \xrightarrow{\alpha} s_1 \wedge s' \xrightarrow{\alpha} s_1' \wedge \alpha \in ACT \cap ACT'\} \cup \\
&\{((s,s'),\alpha,(s_1,s')) \mid s \xrightarrow{\alpha} s_1 \wedge \alpha \notin ACT'\} \cup \\
&\{((s,s'),\alpha,(s,s_1')) \mid s' \xrightarrow{\alpha} s_1' \wedge \alpha \notin ACT\}.
\end{aligned}$$

When combining processes, we sometimes need to change their action names in order to form a network of a desirable structure. The definitions of renaming and hiding functions are similar to the ones introduced for the synchronous model. Let $\mathcal{R}$ be an 1-1 *renaming function* of the ports of the process $M = (ACT, S, R, S_0)$. $\mathcal{R}(M) = (ACT, S, R', S_0)$ denotes a new process M' identical to M except that

—$s \xrightarrow{\mathcal{R}(\alpha)} t \in R'$ iff $s \xrightarrow{\alpha} t \in R$.

A *hiding function* $\mathcal{R}_{act}$ (where $act \subseteq ACT$) disallows synchronization on the actions in the set act by renaming them to the silent action τ. Formally, $\mathcal{R}_{act}(M)$ is a new process M', which is identical to M except that its transition relation R' is

—$s \xrightarrow{\alpha} t \in R'$ iff $s \xrightarrow{\alpha} t \in R$ and $\alpha \notin act$.
—$s \xrightarrow{\tau} t \in R'$ iff $s \xrightarrow{\alpha} t \in R$ and $\alpha \in act$.

The network grammar $G = \langle T, N, \mathcal{P}, \mathcal{S} \rangle$ has the production rules of the following form:

—$\mathcal{P}$ is the set of production rules of the form:

$$A \rightarrow \mathcal{R}_{act}(\mathcal{R}_{\texttt{left}}(B) \| \mathcal{R}_{\texttt{right}}(C))$$

where $A \in N$, $B, C \in T \cup N$, $\mathcal{R}_{\texttt{left}}$ and $\mathcal{R}_{\texttt{right}}$ are renaming functions, and $\mathcal{R}_{act}$ is the hiding function for that rule.

The following theorem proves the monotonicity of the asynchronous composition.

THEOREM 6.3 (MONOTONICITY THEOREM). *Let*

$$\begin{aligned} M &= (S, R, ACT, S_0) \\ M_1 &= (S_1, R_1, ACT_1, S_{0,1}) \\ M' &= (S', R', ACT, S'_0) \\ M'_1 &= (S'_1, R'_1, ACT_1, S'_{0,1}) \end{aligned}$$

be processes such that $M \preceq M'$ *and* $M_1 \preceq M'_1$. *In this case* $M \| M_1 \preceq M' \| M'_1$.

PROOF. Let $\mathcal{E} \subseteq S \times S'$ and $\mathcal{E}_1 \subseteq S_1 \times S'_1$ be the simulation relations. Consider the following relation $\mathcal{E} \times \mathcal{E}_1 \subseteq (S \times S_1) \times (S' \times S'_1)$:

—$((s, s_1), (s', s'_1)) \in \mathcal{E} \times \mathcal{E}_1$ iff $(s, s') \in \mathcal{E}$ and $(s_1, s'_1) \in \mathcal{E}_1$

We prove that $\mathcal{E} \times \mathcal{E}_1$ is the simulation relation between $M \| M_1$ and $M' \| M'_1$ and, hence, $M \| M_1 \preceq M' \| M'_1$. Note that set of actions of $M \| M_1$ and $M' \| M'_1$ are both identical to $(ACT \cup ACT_1) \backslash (ACT \cap ACT_1)$.

—For all $(s_0, s_{0,1}) \in S_0 \times S_{0,1}$ there exists $(s'_0, s'_{0,1}) \in S'_0 \times S'_{0,1}$ such that

$$((s_0, s_{0,1}), (s'_0, s'_{0,1})) \in \mathcal{E} \times \mathcal{E}_1.$$

This follows because we can find $s'_0 \in S'_0$ and $s'_{0,1} \in S'_{0,1}$ such that $(s_0, s'_0) \in \mathcal{E}$ and $(s_{0,1}, s'_{0,1}) \in \mathcal{E}_1$.

—Let $((s, s_1), (t, t_1)) \in \mathcal{E} \times \mathcal{E}_1$. By assumption, we have that $h(s) = h(t)$ and $h(s_1) = h(t_1)$. Therefore, $h((s, s_1)) = h(s) \circ h(s_1)$ is equal to $h((t, t_1)) = h(t) \circ h(t_1)$. Assume that $(s, s_1) \xrightarrow{\alpha} (s', s'_1)$. There are three cases to consider.

—$(s, s_1) \xrightarrow{\tau} (s', s'_1)$ and $s \xrightarrow{\alpha} s'$, $s_1 \xrightarrow{\alpha} s'_1$ for $\alpha \in ACT \cap ACT_1$. In this case, there exists t' and t'_1 such that $t \xrightarrow{\alpha} t'$ and $t_1 \xrightarrow{\alpha} t'_1$ such that $(s', t') \in \mathcal{E}$ and $(s'_1, t'_1) \in \mathcal{E}_1$. It is clear that $(t, t_1) \xrightarrow{\tau} (t', t'_1)$ and $((s', s'_1), (t', t'_1))$ is in $\mathcal{E} \times \mathcal{E}_1$.

—$(s, s_1) \xrightarrow{\alpha} (s', s'_1)$ and $s \xrightarrow{\alpha} s'$, $s_1 = s'_1$ and $\alpha \notin ACT_1$. In this case, there exists t' such $t \xrightarrow{\alpha} t'$ and $(s', t') \in \mathcal{E}$. It is clear that $(t, t_1) \xrightarrow{\alpha} (t', t_1)$ and $((s', s_1), (t', t_1))$ is in $\mathcal{E} \times \mathcal{E}_1$. Notice that by assumption $(s_1, t_1) \in \mathcal{E}_1$.

—$(s, s_1) \xrightarrow{\alpha} (s', s'_1)$ and $s = s'$, $s_1 \xrightarrow{\alpha} s'_1$, and $\alpha \notin ACT$. This case is similar to the one given previously.

The proof is thus complete. □

The next theorem states that renaming and hiding preserve monotonicity.

THEOREM 6.4. *Let* M *and* M' *be two processes such that* $M \preceq M'$. *Let* $\mathcal{R}$ *and* $\mathcal{R}_{act}$ *be renaming and hiding functions respectively. In this case* $\mathcal{R}(M) \preceq \mathcal{R}(M')$ *and* $\mathcal{R}_{act}(M) \preceq \mathcal{R}_{act}(M')$.

PROOF. The proof of this theorem is straightforward. □

Using the preceding two theorems we can prove that the composition operators used in the grammar are monotonic.

7. EXAMPLES

We implemented the algorithm for network verification and applied it to two examples of substantial complexity. The first example uses the asynchronous composition. The second example uses the synchronous model of composition.

$wne \xrightarrow{r/r} bne$	$wne \xrightarrow{/r} wde$	$bne \rightarrow bde$
$bne \xrightarrow{t/t} wne$	$wde \xrightarrow{r/} bde$	$wde \xrightarrow{t/} wct$
$bde \xrightarrow{t/} bct$	$wnt \xrightarrow{r/t} wde$	$wnt \rightarrow wct$
$wct \xrightarrow{r/} bct$	$wct \rightarrow wnt$	$bct \xrightarrow{/t} wne$

Fig. 6. Transitions for a process that performs the token ring protocol.

7.1 Dijkstra's Token Ring

Our first example is the famous Dijkstra's token ring algorithm [Dijkstra 1985]. This algorithm is significantly more complicated than the one used as a running example throughout the article. The example uses the asynchronous model of computation. There is a token t which passes in the clockwise direction. To avoid the token from passing unnecessarily, there is a signal r (r stands for request) which passes in the counter-clockwise direction. Whenever a process wishes to have the token, it sends the signal r to its left neighbor, i.e., the process on the counterclockwise side. The states of the processes have the following four components:

—It is either b (black;an interest in the token exists to the right), w (white;no one is interested in the token).

—It is either n (in the neutral state), d (the process is delayed waiting for the token), or c (the process is in the critical section).

—It is either t (with the token), or e (empty;without the token).

—A set of outputs o enabled from the state. The possible values of o are $\emptyset$, $\{r\}$, $\{t\}$, and $\{r, t\}$.

Each process has $r?$ and $t?$ as input actions and $r!$ and $t!$ as output actions. This notation is borrowed from CSP. While synchronizing, $r!$ is matched with $r?$. Similarly, $t!$ is matched with $t?$. Each state has an output associated with it (kept in the component o). The name of the state is a combination of its properties. Thus $\langle wne, \{r\}\rangle$ is a neutral state with no request on the right and no token and has output r. We describe a state with a 2-tuple, i.e., $\langle x, o\rangle$ where x describes the first three components and o is the set of outputs enabled from that state. If the output from a state s' is nonempty ($o \neq \emptyset$), then there is a transition to s' with every action in o labeling the transition. Given a state $\langle x, o\rangle$, the following transitions are present in the system:

$$\forall(\alpha \in o)\left[\langle x, o\rangle \xrightarrow{\alpha!} \langle x, o \backslash \{\alpha\}\rangle\right]$$

For conciseness, the list of transitions for a process that performs the token ring protocol is given in Figure 6.

We explain how the transitions given in Figure 6 can be translated into our notation. The first component on the transition label is the input, and the second is the output. We consider the four kinds of transitions:

—$x \rightarrow y$ corresponds to the transitions $\langle x, o\rangle \xrightarrow{\tau} \langle y, o\rangle$,

—$x \xrightarrow{/v} y$ corresponds to transitions $\langle x, o\rangle \xrightarrow{\tau} \langle y, o \cup \{v\}\rangle$,

—$x \xrightarrow{u/} y$ corresponds to transitions $\langle x, o\rangle \xrightarrow{u?} \langle y, o\rangle$, and

—$x \xrightarrow{u/v} y$ corresponds to transitions $\langle x, o\rangle \xrightarrow{u?} \langle y, o \cup \{v\}\rangle$.

Notice that each entry in the table actually corresponds to a group of transitions that only differ from each other by the value of the second component.

Let Q be the process with $\langle wnt, \emptyset\rangle$ as the initial state and the preceding transitions. Let P be the process with $\langle wne, \emptyset\rangle$ as the initial state and the same transitions as Q. The network grammar generating a token-ring of arbitrary size is similar to the one given for Example 5.1.3.

Let S be the set of states in a *basic* process of the token ring. Let `t` be the subset of states that have the token. Let `not-t` denote $S \setminus$ `t`. The automaton $\mathcal{D}$ is the same as the automaton in Figure 3 with `t` substituted for cs and `not-t` substituted for nc. The automaton accepts strings $S^\star$ such that the number of processes with the tokens is exactly one. Let h be the abstraction function induced by the automaton. The initial association is

$$\begin{aligned}\mathcal{AS}_0(Q) &= \{Q\}\\ \mathcal{AS}_0(P) &= \{P\}\\ \mathcal{AS}_0(A) &= \{P \| P\}\\ \mathcal{AS}_0(\mathcal{S}) &= \{Q \| P \| P\}.\end{aligned}$$

Unfortunately, the corresponding representatives did not satisfy the monotonicity condition. Therefore, we unfolded the initial association to get the following association:

$$\begin{aligned}\mathcal{AS}(A) &= \{P \| P \| P, P \| P\}\\ \mathcal{AS}(\mathcal{S}) &= \{Q \| P \| P \| P, Q \| P \| P\}\end{aligned}$$

Notice that unfolding does not change the associations for the terminals. If we choose the representatives as

$$\begin{aligned}rep(P) &= P\\ rep(Q) &= Q\\ rep(A) &= P \| P \| P\\ rep(\mathcal{S}) &= Q \| P \| P \| P\end{aligned}$$

then the monotonicity condition holds, i.e.,

$$\begin{aligned}h(rep(A)) &\succeq h(h(rep(P)) \| h(rep(P)))\\ h(rep(A)) &\succeq h(h(rep(A)) \| h(rep(P)))\\ h(rep(\mathcal{S})) &\succeq h(h(rep(Q)) \| h(rep(A))).\end{aligned}$$

By Theorem 4.1 we conclude that $rep(\mathcal{S})$ simulates all the LTSs generated by the grammar G. Notice that if $rep(\mathcal{S})$ satisfies the property $\mathbf{AG}\,\mathcal{D}$, then Theorem 3.2.4 implies that every LTS generated by the grammar G satisfies $\mathbf{AG}\,\mathcal{D}$. Using our system we established that $rep(\mathcal{S})$ is a model for $\mathbf{AG}\,\mathcal{D}$.

7.2 Parity Tree

We consider a network of binary trees, in which each leaf has a bit value. We describe an algorithm that computes the parity of the values at the leaves. The algorithm is described in Ullman [1984]. A context-free grammar G generating a binary tree is given below, where `root, inter,` and `leaf` are terminals (basic processes), and $\mathcal{S}$ and SUB are nonterminals. The precise definition of the composition operator is given later in this section.

$$\begin{aligned}\mathcal{S} &\rightarrow \texttt{root}\|\text{SUB}\|\text{SUB}\\ \text{SUB} &\rightarrow \texttt{inter}\|\text{SUB}\|\text{SUB}\\ \text{SUB} &\rightarrow \texttt{inter}\|\texttt{leaf}\|\texttt{leaf}\end{aligned}$$

An intuitive description of the algorithm follows. The `root` process initiates a wave by sending the *readydown* signal to its children. Every internal node that gets the signal sends it further to its children. When the signal *readydown* reaches a leaf process, the leaf sends the *readyup* signal and its *value* to its parent. An internal node that receives the *readyup* and *value* from both its children, sends the *readyup* signal and the `xor`($\oplus$) of the values received from the children to its parent. When the *readyup* signal reaches the root, one wave of the computation is terminated and the root can initiate another wave. The semantics of the composition used in the grammar G should be clear from Figure 7. For example, the inputs *readyup_l* and *value_l* of an internal node are identified with the outputs *readyup* and *value* of its left child.

Next, we describe the various signals in detail. First we describe the process `inter`. The process `inter` corresponds to the internal node of the tree. The various variables for the process are shown in Figure 8.

The following equations are invariants for the state variables:

$$\begin{aligned}root\_or\_leaf &= 0\\ readyup &= readyup\_l \wedge readyup\_r\end{aligned}$$

The output variables have the same value in each state as the corresponding state variable, e.g., the output variable *readydown* has the same value as the state variable *readydown*. The following equations show how the input variables affect the state variables. The primed variables on the left-hand side refer to the next state variables, and the right-hand side refers to the input variables.

$$\begin{aligned}readydown' &= readydown\\ readyup\_l' &= readyup\_l\\ readyup\_r' &= readyup\_r\\ value' &= (readyup\_l \wedge value\_l) \oplus (readyup\_r \wedge value\_r)\end{aligned}$$

Since the `root` process does not have a parent, it does not have the input variable *readydown*. The invariant $root\_or\_leaf = 1$ is maintained for the root and the

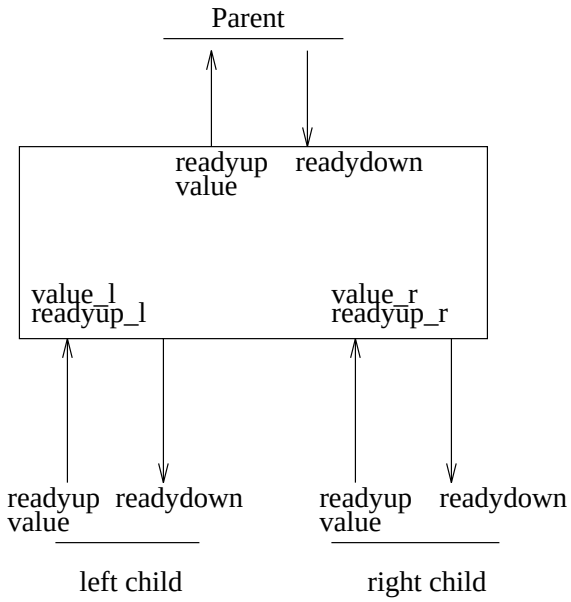

Fig. 7. Internal node of the tree.

state vars	output vars	input vars
root_or_leaf	*readydown*	*readydown*
readydown	*readyup*	*readyup_l*
readyup_l	*value*	*readyup_r*
readyup_r		*value_l*
value		*value_r*
readyup		

Fig. 8. Variables used in processes.

leaf process. Since the `leaf` process does not have a child, the output variable *readydown* is absent. The leaf variable has only one input variable *readydown* and the following equation between the next state variables and input variables is maintained:

$$readyup' = readydown$$

For each `leaf` process the assignment for the state variable *value* is decided nondeterministically in the initial state and then kept the same throughout the computation.

A state in the basic processes (`root,leaf,inter`) is a specific assignment to the state variables. We call this state set S. Since there are 6 state variables, the state set $S \cong \{0,1\}^6$.

The automata we describe accepts strings from $S^\star$. Let $value_1, \cdots, value_n$ be the values in the n leaves. Let *value* be the value calculated at the `root`. Since at the end of the computation the `root` process should have the parity of the bits $value_i$

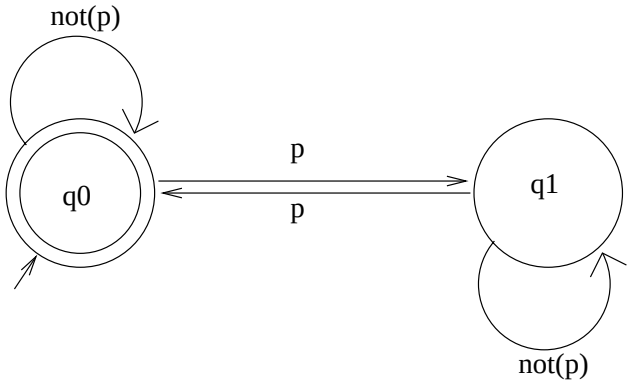

Fig. 9. Automaton for parity.

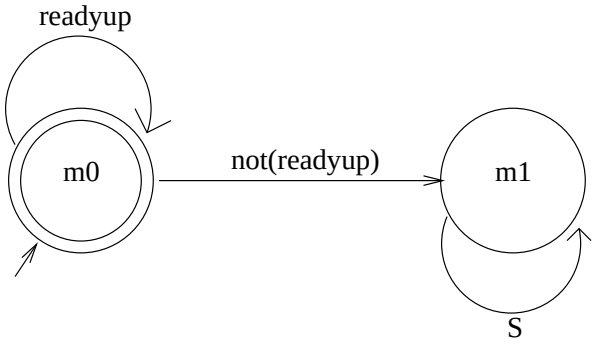

Fig. 10. Automaton for ready.

$(1 \leq i \leq n)$, the following equation should hold at the end of the computation:

$$value \oplus \bigoplus_{i=1}^{n} value_i \;=\; 0.$$

Let p be defined by the following equation:

$$\mathtt{p} \;=\; \{s \in S \mid s \,\text{satisfies}\, root\_or\_leaf \wedge value\}$$

Let not(p) $= S - \mathtt{p}$. The automaton given in Figure 9 accepts the strings in $S^\star$ that satisfy the preceding equation. Since $root\_or\_leaf = 0$ for internal nodes, the automaton essentially ignores the values at the internal nodes. We call this automaton **parity**. We also want to assert that everybody is finished with his or her computation. This is signaled by the fact that $readyup = 1$ for each process. The automaton given in Figure 10 accepts strings in $S^\star$ iff $readyup = 1$ in each state, i.e., all processes have finished their computation. We call this automaton **finished**. The automata **parity** and **finished** are our atomic formulas. We want to verify that if the computation is finished then the parity at the root is correct. Hence, we want to check that the initial state satisfies the property **AG**(**finished** $\rightarrow$ **parity**). Let h be the abstraction function induced by the atomic formulas (see Section 3 for

the definition of h). The initial association is

$$\begin{aligned} \mathcal{AS}_0(\text{SUB}) &= \{\texttt{inter}\|\texttt{leaf}\|\texttt{leaf}\} \\ \mathcal{AS}_0(\mathcal{S}) &= \mathit{root}\|\mathcal{AS}_0(\text{SUB})\|\mathcal{AS}_0(\text{SUB}). \end{aligned}$$

The association corresponding to the terminals is just the process associated with the terminal. The association for $\mathcal{S}$ can be derived from $\mathcal{AS}(\text{SUB})$. The representatives corresponding to the initial association did not satisfy the monotonicity condition. Unfolding the initial association we get

$$\begin{aligned} I &= \texttt{inter}\|\texttt{leaf}\|\texttt{leaf} \\ I_1 &= \texttt{inter}\|I\|I \\ \mathcal{AS}(\text{SUB}) &= \{I_1\} \cup \mathcal{AS}_0(\text{SUB}) \\ \mathcal{AS}(\mathcal{S}) &= \{\texttt{root}\|I_1\|I_1\} \cup \mathcal{AS}(\mathcal{S}). \end{aligned}$$

Now we could find representatives that did satisfy the monotonicity condition. Using Theorem 4.1 we can conclude that $H = h(\mathit{rep}(\mathcal{S}))$ simulates all the networks generated by the context free grammar G. We checked that $H, s_0 \models \mathbf{AG}(\mathbf{finished} \rightarrow \mathbf{parity})$ (s_0 is the initial state of the process H). Theorem 3.2.4 implies that every network derived by G has the desired property, i.e., when the computation is finished the root process has the correct property.

8. DIRECTION FOR FUTURE RESEARCH

In this article we have described a new technique for reasoning about families of finite-state systems. This work combines network grammars and abstraction with a new way of specifying state properties using regular languages. We have implemented our verification method and used it to check two nontrivial examples. In the future, we intend to apply the method to even more complex families of state-transition systems.

There are several directions for future research. The context-free network grammars can be replaced by context-sensitive grammars. Context-sensitive grammars can generate networks such as square grids and complete binary trees that cannot be generated by the context-free grammars. The specification language can be strengthened by replacing regular languages by more expressive formalisms. We are also considering adding fairness to our models.

ACKNOWLEDGMENT

We thank the anonymous referees for their useful comments.

REFERENCES

Apt, K. and Kozen, D. 1986. Limits for automatic verification of finite-state systems. *Inf. Process Lett. 15*, 307–309.

Browne, M., Clarke, E., and Grumberg, O. 1989. Reasoning about networks with many identical finite-state processes. *Inf. Comput. 81,* 1 (Apr.), 13–31.

Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., and Hwang, L. J. 1992. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput. 98,* 2 (June), 142–170.

Clarke, E. M. and Emerson, E. A. 1981. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop* (*Yorktown Heights*), D. Kozen, Ed. Lecture Notes in Computer Science, vol. 131. Springer-Verlag, Berlin.

Clarke, E. M., Emerson, E. A., and Sistla, A. P. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst. 8,* 2 (Apr.), 244–263.

Clarke, E. M., Grumberg, O., and Long, D. E. 1992. Model checking and abstraction. In *Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York.

Dams, D., Grumberg, O., and Gerth, R. 1994. Abstract interpretation of reactive systems: Abstractions preserving ACTL*,ECTL*, and CTL*. In *IFIP Working Conference and Programming Concepts, Methods and Calculi* (*PROCOMET'94*), (San Miniato, Italy).

Dijkstra, E. 1985. Invariance and non-determinacy. In *Mathematical Logic and Programming Languages*, C. Hoare and J. Sheperdson, Eds. Prentice-Hall, Englewood Cliffs, N.J.

Eilenberg, S. 1974. *Automata, Languages, and Machines*. Academic Press, New York.

Emerson, E. and Namjoshi, K. S. 1995. Reasoning about rings. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Programming Languages*. ACM, New York.

German, S. and Sistla, A. 1992. Reasoning about systems with many processes. *J. ACM 39*, 675–735.

Grumberg, O. and Long, D. 1994. Model checking and modular verification. *ACM Trans. Program. Lang. Syst. 16,* 3 (May), 843–871.

Kurshan, R. P. and McMillan, K. L. 1989. A structural induction theorem for processes. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*. ACM, New York.

Lichtenstein, O. and Pnueli, A. 1985. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, 97–107.

Marelly, R. and Grumberg, O. 1991. GORMEL—Grammar ORiented ModEL checker. Tech. Rep. 697, The Technion, Haifa, Israel. Oct.

Milner, R. 1971. An algebraic definition of simulation between programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*.

Milner, R. 1980. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science, vol. 92. Springer-Verlag, Berlin.

Quielle, J. and Sifakis, J. 1982. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th International Symposium in Programming*.

Shtadler, Z. and Grumberg, O. 1989. Network grammars, communication behaviors and automatic verification. In *Proceedings of the 1989 International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France*, J. Sifakis, Ed. Lecture Notes in Computer Science, vol. 407. Springer-Verlag, Berlin.

Ullman, J. D. 1984. *Computational Aspects of VLSI*. Computer Science Press, New York.

Vernier, I. 1994. Parameterized evaluation of CTL-X formulae. In *Workshop Accompanying the International Conference on Temporal Logic* (*ICTL'94*).

Wolper, P. and Lovinfosse, V. 1989. Verifying properties of large sets of processes with network invariants. In *Proceedings of the 1989 International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France*, J. Sifakis, Ed. Lecture Notes in Computer Science, vol. 407. Springer-Verlag, Berlin.

Received August 1996; revised April 1997; accepted June 1997