Verifying Parameterized Networks using Abstraction and Regular Languages *

E. M. Clarke[1] and O. Grumberg[2] and S. Jha[1]

[1] Carnegie Mellon University, Pittsburgh, PA 15213
[2] Computer Science Dept, The Technion, Haifa 32000, Israel

Abstract. This paper 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 an invariant which *simulates* all the state-transition systems in the family. A novel idea used in this paper is to use *regular languages* to express state properties. We have implemented our techniques and verified two non-trivial examples.

1 Introduction

Automatic verification of state-transition systems using temporal logic model checking has been investigated by numerous authors [3, 4, 5, 12, 16]. The basic model checking problem is easy to state

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

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 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 [1]. However, for *specific* families the problem may be solvable. This possibility has been investigated by [2]. 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 2-process token ring and an n-process token ring for any $n \geq 2$. It follows that the 2-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.

Induction at the process level has also been used to solve problems of this nature by two research groups [9, 10]. 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.

* This research was sponsored in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597 and in part by the National Science Foundation under Grant no. CCR-8722633 and in part by the Semiconductor Research Corporation under Contract 92-DJ-294. The second author was partially supported by grant no. 120-732 from The United States-Israel Binational Science Foundation (BSF), Jerusalem, Israel.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

In [20] an alternative method for checking properties of parametrized systems is proposed. 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* [11, 21]. 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 [11, 21] can only handle networks with one repetitive component. Also, the invariant I has to be explicitly provided by the user.

In [17, 13] 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 a 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", "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 of 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 given a $\forall CTL^\star$[6] formula, defined over this set of state properties, if it is true of the abstract system, then it is also true of the systems in the family.

Following [17] and [13] 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). While [17, 13] 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 use a simple mutual exclusion algorithm as the running example to demonstrate our ideas. Two realistic examples are given in a separate section.

Our paper is organized as follows. In Section 2 we define the basic notions, including network grammars and regular languages used 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. In Section 6 we apply our method to two non-trivial examples. Section 7 concludes with some directions for future research.

2 Definitions and Framework

Definition 1 (LTS). 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 $(s, a, s') \in R$.

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 the action set is the subset of ACT.

Definition 2 (Composition). 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 $(S_1 \times S_2, R', ACT, S_0^1 \times S_0^2)$. Notice that we write the composition function in infix notation.

Our verification method handles a set of LTSs referred to as a *network*. Intuitively, a network consists of LTSs is obtained by composing any number of LTSs from $L_{(S,ACT)}$. Thus, each LTS in a network is defined over the set of actions ACT, and over a set of states in S^i, for some i.

Definition 3 (Network). 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 [17], [13], 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 a LTS in $L_{(S,ACT)}$. These LTSs are sometimes referred to as *basic processes*.
- N is a set of non-terminals. Each non-terminal 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 1. We clarify the definitions on 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.

$$\begin{aligned} \mathcal{S} &\rightarrow Q \| A \\ A &\rightarrow P \| A \\ A &\rightarrow P \| P \end{aligned}$$

P and Q are LTSs defined over the set of states $\{nc, cs\}$ and the set of actions $ACT = \{\tau,$ `get-token`, `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 each process takes a step at any moment. We will not give a formal definition of the model here. In Section 5 we suggest a suitable definition for a synchronous model. 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. We refer to the rightmost process P as being to the left of process Q. We can apply the following derivation $\mathcal{S} \Rightarrow Q\|A \Rightarrow Q\|P\|P$ to obtain the LTS $Q\|P\|P$. The reachable states with their transitions are shown in Figure 2. Here, as well, we leave the precise definition of the composition operator unspecified.

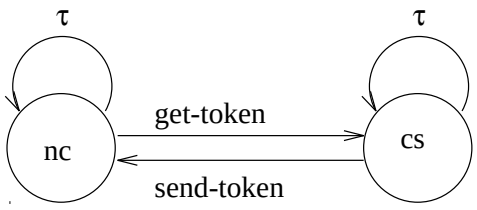

Fig. 1. Process Q, if $S_0 = \{cs\}$; process P if $S_0 = \{nc\}$

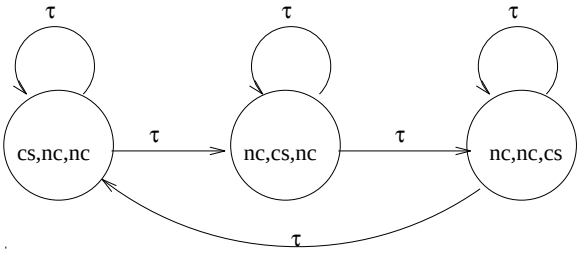

Fig. 2. Reachable states in LTS $Q\|P\|P$

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 below has S as its alphabet. Thus, it accepts words which are sequences of state names.

Definition 4 Specification. $\mathcal{D} = (Q, q_0, \delta, F)$ is a *deterministic automaton* over S, where

1. Q is the set of 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.

$\mathcal{L}(\mathcal{D}) \subseteq S^*$ is the set of words accepted by $\mathcal{D}$.

Our goal is to specify a network of LTSs composed of any number of components (i.e., of basic processes). We will use finite automata over S in order to specify atomic state properties. Since a state of a LTS is a tuple from S^i, for some i, we can view such a state as a word in $S^\star$. Let $\mathcal{D}$ be an automaton over S. We say that $s \in S^\star$ 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., $\forall CTL$, $\forall CTL^*$ [6]) with finite automata over S as the atomic formulas. The relation $\models$ for other formulas of the logic is defined in the standard way (with respect to the temporal logic under consideration) and is omitted here.

Example 2. Consider again the network of Example 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 $\mathbf{AG\,AF}\,\mathcal{D}'$ specifies non-starvation for process Q. Note that, for our simple example non-starvation is guaranteed only if some kind of fairness is assumed.

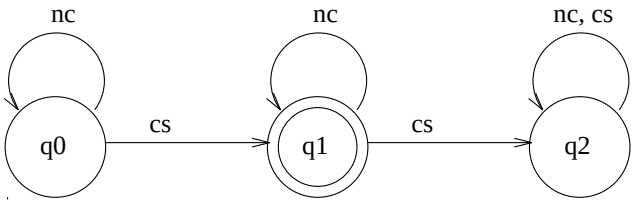

Fig. 3. Automaton $\mathcal{D}$ with $\mathcal{L}(\mathcal{D}) = \{nc\}^* cs \{nc\}^*$

3 Abstract *LTS*s

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 assume that the specification language contains a single atomic formula $\mathcal{D}$. In Appendix A we extend the framework to a set of atomic formulas $\mathcal{D}_i$.

3.1 State equivalence

We start by defining an equivalence relation over the state set of an LTS. The equivalence classes will then be used as the abstract states of the abstract LTS. Given a 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. This means that the two states are either both accepted or both rejected by the automaton $\mathcal{D}$. We also require that our equivalence relation is 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 will use $h(M)$ to denote the abstract LTS of M. The straightforward definition that defines s and s' to be equivalent iff they belong to the language $\mathcal{L}(\mathcal{D})$ has the first property, but does not have the second one. To see this, consider the following example.

Example 3. Consider LTSs defined by the grammar of Example 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 $w \in S^*$, $f_w : Q \mapsto Q$, the *function induced* by w on 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.

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. Then, two states s, s' in $S^\star$ are *equivalent*, denoted $s \equiv s'$, 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 $h(s') = f_1'$. Then, the abstraction of (s, s') is $h((s, s')) = f_1 \circ f_1'$ where $f_1 \circ f_1'$ denotes composition of functions.

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,

Lemma 5. 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 $h(s) \models \mathcal{D}$.

Example 4. 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 in $\mathcal{D}$. 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 will apply abstraction both to states $s \in S^*$ and to abstract states f_s for $s \in S^*$. 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 further 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 will consider LTSs in the network $\mathcal{N}$ on the tuple $(S \cup \mathcal{F}_\mathcal{D}, ACT)$.

Next we define abstract LTSs over abstract states. The abstract transition relation is defined as usual for an abstraction that should be greater by the simulation preorder than the original structure [7]. If there is a transition between one state and another in the original structure, then there is a transition between the abstract state of the one to the abstract state of the other in the abstract structure. Formally,

Definition 6. Given a LTS $M = (S, R, ACT, S_0)$ in the network $\mathcal{N}$, the corresponding *abstract LTS* is defined by $h(M) = (S^h, R^h, ACT, S_0^h)$, where

- $S^h = \{h(s) \mid s \in S\}$ is the set of abstract states.
- $S_0^h = \{h(s) \mid s \in S_0\}$.
- 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' [14] (denoted $M \preceq M'$) if and only if there is a *simulation preorder* $\mathcal{E} \subseteq S \times S'$ that satisfies: 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. We have that $h(s) = h(s')$.
2. For every s_1 such that $s \xrightarrow{a} s_1$ there is s_1' such that $s' \xrightarrow{a} s_1'$ and $(s_1, s_1') \in \mathcal{E}$.

Lemma 7. $M \preceq h(M)$, i.e. M simulates $h(M)$.

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 [6].

Theorem 8. Let ϕ be a formula in $\forall CTL^*$ over an atomic formula $\mathcal{D}$. Then, $h(M) \models \phi$ implies $M \models \phi$.

Let M and M' be two LTSs in the network $\mathcal{N}$, and let $\|$ be a composition function. The abstract composition function corresponding to $\|$ (denoted by $\|_h$) is defined as follows:

$$M \|_h M' = h(M \| M')$$

Definition 9. A composition $\|$ is called *monotonic* with respect to a simulation preorder $\preceq$ iff given LTSs such that $M_1 \preceq M_2$ and $M_1' \preceq M_2'$ it should be true that $M_1 \| M_1' \preceq M_2 \| M_2'$.

4 Verification method

Given a grammar G, we associate with each terminal and nonterminal A of the grammar an abstract structure $rep(A)$ that *represents* all LTSs derived from A by the grammar. Thus for every LTS (a) derived from A we have that $rep(A) \succeq a$. This implies that for every network t, derived from the initial symbol $\mathcal{S}$, $rep(\mathcal{S}) \succeq t$ and therefore, any property of $rep(\mathcal{S})$, expressed in the logic $\forall CTL^*$ is also a property of t. We require that the composition functions used in the grammar G are monotonic with respect to $\preceq$ when applied to ordinary LTSs and to abstract LTSs.
Our verification method is as follows:

- 1. For every terminal A, choose $rep(A) = h(A)$ [3].
  2. Given a rule $A \rightarrow B \| C$ of the grammar, if $rep(A)$ is not defined yet, and if $rep(B)$ and $rep(C)$ are defined, then define $rep(A) = rep(B) \|_h rep(C)$.

  If every symbol of the grammar is reachable by some derivation from the initial symbol, and if each symbol derives at least one LTS then the algorithm will terminate and $rep(A)$ will be defined for every symbol A. In particular, when rules of the form $A \rightarrow A \| C$ are encountered, $rep(A)$ is already defined.
- For every rule $A \rightarrow B \| C$ in G show: $rep(A) \succeq rep(B) \|_h rep(C)$.

Theorem 10. Assume that the verification method has been successfully applied to the grammar G. Let A be a symbol in G and let a be a LTS derived from A in G, then $rep(A) \succeq a$.

5 Synchronous model of computation

In this section we develop a synchronous framework that will have 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*. They 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 [15]). The set of actions is $ACT = \{\tau\} \cup 2^{I \cup O}$, where each non-internal action is a set of inputs and outputs. In standard Moore machine the outputs are usually associated with the states while the inputs are associated with the transitions. Here, we associate both inputs and outputs with the transitions while maintaining the distinction between inputs and outputs. A transition $s \xrightarrow{\tau} s_1$ in a machine M can always be executed. It has not effect on other machines. It is used to hide wires once they are connected, in order to avoid the output signal from being connected to other input wires. Refer to the use of τ in the hiding function defined later.

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 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 will be a joint 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.

---

[3] Actually, for a terminal A it is sufficient to choose any abstract LTS (defined over $\mathcal{F}_D$) that satisfies $rep(A) \succeq A$

Formally, let $O \cap O' = \emptyset$. The *synchronous composition* of M and M', $M'' = M \parallel M'$ 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. $O'' = O \cup O'$.[4]
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 either $a = \tau$ and $a'' = a$ or $a' = \tau$ and $a'' = a$ or $a \cap (I' \cup O') = a' \cap (I \cup O)$ and $a'' = (a \cup a') \cap (I'' \cup O'')$.

Lemma 11. The composition $\parallel$ is monotonic with respect to $\preceq$.

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 of synchronous models. Like 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. 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 LTS $M' = \mathcal{R}(M)$ with $S = S'$, $S_0 = S'_0$, $I' = \mathcal{R}(I)$, $O' = \mathcal{R}(O)$, 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 function that maps each element in act to τ. Let $M' = \mathcal{R}_{act}(M)$ then $S' = S$, $S'_0 = S_0$, $I' = I \setminus act$, and $O' = O \setminus act$. Moreover, $s \xrightarrow{a} s'$ is a transition in M iff $a \setminus act \neq \emptyset$ and $s \xrightarrow{a \setminus act} s'$ is a transition in M' or $a \setminus act = \emptyset$ and $s \xrightarrow{\tau} s'$ is a transition in M'.

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 defined before.

To be used in our framework, we need to show that every such operator 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 $\|$ imply the required result.

Lemma 12. Let M, M' be synchronous LTSs and let $\mathcal{R}$ be a renaming function and let $\mathcal{R}_{act}$ be a hiding function. If $M \preceq M'$, then $\mathcal{R}(M) \preceq \mathcal{R}(M')$ and $\mathcal{R}_{act}(M) \preceq \mathcal{R}_{act}(M')$.

Corollary 13. The composition operators $\|_i$, defined as above are monotonic.

Example 5. We return to Example 1 and reformulate it within the synchronous framework. Doing so 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:

---

[4] Note that, $ACT'' = 2^{I'' \cup O''}$ 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.

$$\boxed{\begin{array}{lcl} \mathcal{S} & \rightarrow & Q\|_1 A \\ A & \rightarrow & P\|_2 A \\ A & \rightarrow & P\|_2 P \end{array}}$$

$\|_1$ is defined as follows (see also Figure 7 in appendix B):

- $\mathcal{R}^1_{\texttt{left}}$ 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`.
- The hiding function 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 7 in appendix B):

- $\mathcal{R}^2_{\texttt{left}}$ 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 hides `cr`

The application of the third rule, for instance, 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 7 and Figure 8 in Appendix B). Note that, in the derivation of an LTS by the grammar, the derivation is completed before the renaming and the hiding functions are applied. These functions are applied to LTSs and not to the non-terminals representing them.

6 Examples

We implemented the algorithm for network verification for the synchronous model and applied it to two examples of substantial complexity. These examples were verified with the aid of our verification tool.

6.1 Dijkstra's Token Ring

The first is the famous Dijkstra's token ring algorithm [8]. This algorithm is significantly more complicated than the one used as a running example along the paper. There is a token t which passes in the clockwise direction. To avoid the token from passing unnecessarily, there is a signal s which passes in the counter-clockwise direction. Whenever a process wishes to have the token, it sends the signal s to its left neighbor. The states of the processes have the following three properties:

- 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 b (black–an interest in the token exists to the right), w (white–no one is interested in the token)
- It is either t (with the token), or e (empty–without the token).

The name of a state is a combination of its properties. Thus wne is a neutral state with no request on the right and no token. Each process has `get-token` and `get-signal` as inputs and `send-token` and `send-signal` as outputs. The notation $x_1 \stackrel{\alpha/\beta}{\rightarrow} x_2$ means that on the input α a transition is made from the state x_1 to the state x_2 producing output β. If the input is missing, it means that on any input the transition is made. If the output is missing, it means that no output is produced on that transition. The list of transitions for a process that performs the token ring protocol is shown in the table

below. The symbol s stands for either `get-signal` or `send-signal` and t stands for either `get-token` or `send-token`. Each state has also a self loop on the internal transition τ. These transitions are omitted from the table. Note that, when a process makes a τ transition it does not communicate with its neighbors. Thus, its neighbors have to be involved in some other transition–either internal or a communication with another process.

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

Let Q be the process with wnt as the initial state and the transition relation shown above. Let P be the process with wne as the initial state and with the same transition relation as Q. The network grammar generating a token-ring of arbitrary size is similar to that of Example 5, where `get-signal` or `send-signal` are treated similarly to `get-token` or `send-token`. It turns out that LTSs consisting of less than three processes of type P have different behaviors than the LTSs composed of three or more P processes. We exclude such LTSs by replacing the last rule in the grammar by:

$$A \rightarrow P \| P \| P.$$

Let S be the set of states in a *basic* process of the token ring. Let `t` be the subset of states which has the token. Let `not-t` be the set $S \setminus t$. 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. We choose $rep(P) = h(P)$,and $rep(Q) = h(Q)$ and $rep(A) = h(P) \|_h h(P) \|_h h(P)$. Using the first rule of the grammar we have that $rep(\mathcal{S}) = h(Q) \|_h h(A)$. Using our verification tool we were automatically able to check that $rep(A) \succeq rep(A) \|_h rep(P)$. By Theorem 10 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 8 implies that every LTS generated by the grammar G satisfies $\mathbf{AG}\,\mathcal{D}$. Using our verification tool we established that $rep(\mathcal{S})$ is a model for $\mathbf{AG}\,\mathcal{D}$.

6.2 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 leaves values. The algorithm is taken from [19]. 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.

$\mathcal{S} \rightarrow$ `root`$\|$SUB$\|$SUB
SUB $\rightarrow$ `inter`$\|$SUB$\|$SUB
SUB $\rightarrow$ `inter`$\|$`leaf`$\|$`leaf`

The algorithm works as 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 $\oplus$ 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. This description is somewhat informal. Actually, at any step of the computation each process outputs its

relevant state variables to its neighbors. It also gets inputs from its neighbors and updates its state variables accordingly. Thus saying that a process gets a signal actually means that at the current step, the value of this signal (received by the process as input) is 1. The semantics of the composition used in the grammar G should be clear from Figure 4. 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` is the process

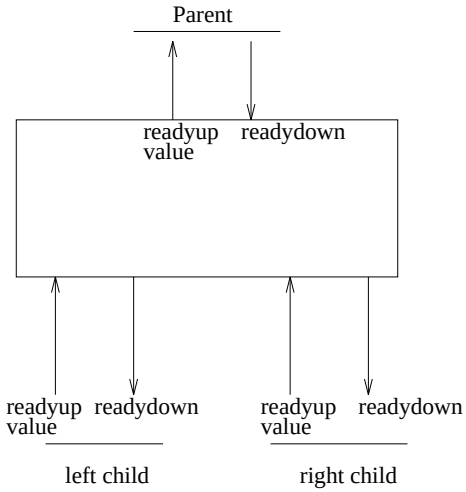

Fig. 4. Internal node of the tree

in the internal node of the tree. The various variables for the process are shown in the table:

state vars	output vars	input vars
$root\_or\_leaf$	$readydown$	$readydown$
$readydown$	$readyup$	$readyup\_l$
$readyup\_l$	$value$	$readyup\_r$
$readyup\_r$		$value\_l$
$value$		$value\_r$
$readyup$		

The following equations are invariants for the state variables:

$$root\_or\_leaf = 0$$
$$readyup = readyup\_l \wedge readyup\_r$$

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 equations given below show how the input variables affect the state variables. In the equations given below the primed variables on the left hand side refer to the next state variables and the right hand side refers to the input variables.

$$readydown' = readydown$$
$$readyup\_l' = readyup\_l$$
$$readyup\_r' = readyup\_r$$
$$value' = (readyup\_l \wedge value\_l) \oplus (readyup\_r \wedge value\_r)$$

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 `leaf` process. Since the `leaf` process does not have a child, the output variable *readydown* is absent. The `leaf` process 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 non-deterministically in the initial state and then kept the same throughout.

A state in the basic processes (`root`,`leaf`,`inter`) is a specific assignment to the state variables. We call this state set S. Notice that the state set $S \cong \{0,1\}^6$ because there are 6 state variables. The automata we describe accept 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 $value_i$ $(1 \leq i \leq n)$, 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:

$$p = \{s \in S | s \text{ satisfies } root\_or\_leaf \wedge value\}.$$

Let $not(p) = S - p$. The automaton $\mathcal{D}_{par}$ given in Figure 5 accepts the strings in $S^\star$ which satisfy the equation given above. Since $root\_or\_leaf = 0$ for internal nodes, the automaton essentially ignores the values at the internal nodes. We also want to assert that everybody is finished with their computation.

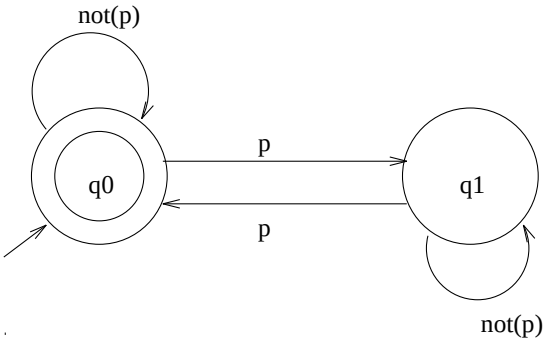

Fig. 5. Automaton (D_{par}) for parity

This is signaled by the fact that $readyup = 1$ for each process. The automaton $\mathcal{D}_{ter}$ given in Figure 6 accepts strings in $S^\star$ iff $readyup = 1$ in each state. i.e. all processes have finished their computation. The property $D_{ter} \Rightarrow D_{par}$ says that if the computation is finished in a state, then the parity is correct at the `root`. We want to check that every reachable state of an LTS in the network has the desired property, i.e. $\mathbf{AG}(D_{ter} \Rightarrow D_{par})$ is true. We use as our atomic formula the union of $\mathcal{D}_{par}$ and $\mathcal{D}_{\neg ter}$ (the complement of $\mathcal{D}_{ter}$). Let h be the abstraction function induced by this automaton (see Section 3 for the definition of h). Let $\|_h$ be the abstract composition operator and $\preceq$ the simulates relation. Let I_1, I_2 be abstract processes defined as follows:

$$I_1 = h(inter) \|_h h(leaf) \|_h h(leaf)$$
$$I_2 = h(inter) \|_h I_1 \|_h I_1$$

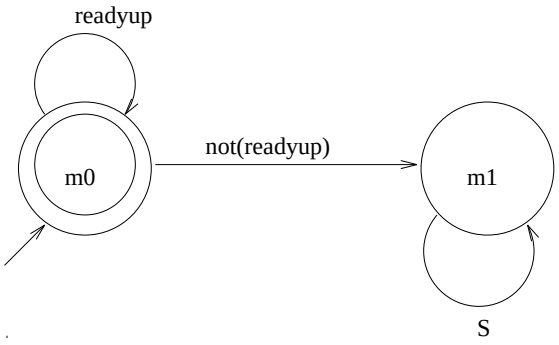

Fig. 6. Automaton (D_{ter}) for ready

The following equations were verified automatically by our verification tool:

$$\begin{aligned} h(inter)\|_h I_1\|_h I_1 &\not\preceq I_1 \\ I_1 &\preceq I_2 \\ h(inter)\|_h I_2\|_h I_2 &\preceq I_2 \end{aligned}$$

From the first equation given above it is clear the I_1 cannot be used as a representative for the non-terminal SUB, i.e if we set $rep(\text{SUB}) = I_1$, the induction corresponding to the second rule of the grammar does not hold. Notice that I_2 was derived from the second rule of the grammar by substituting I_1 for SUB. Suppose we use $rep(\text{SUB}) = I_2$ and $rep(\mathcal{S}) = h(\text{root})\|_h I_2\|_h I_2$ as the representatives for the non terminals. From the equations given above the following inequalities can be derived:

$$\begin{aligned} rep(\text{SUB}) &\succeq h(\text{inter})\|_h rep(\text{SUB})\|_h rep(\text{SUB}) \\ rep(\text{SUB}) &\succeq h(\text{inter})\|_h h(\text{leaf})\|_h h(\text{leaf}) \end{aligned}$$

Now using Theorem 10 we can conclude that $H = h(root)\|_h I_2\|_h I_2$ simulates all the networks generated by the context free grammar G. After we constructed H, we verified automatically, that all reachable states in H have the desired property. Now by Theorem 8 we have the result that every LTS derived by G has the desired property, i.e. when the computation is finished the root process has the correct parity.

7 Direction for future research

In this paper 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 non-trivial 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 like square grids and complete binary tree which cannot be generated by the context-free grammars. The specification language can be strengthened by replacing regular languages by more expressive formalisms. We might also consider adding fairness to our models (LTSs). Finally, we intend to extend the techniques described in this paper to asynchronous models of computation.

References

1. K. Apt and D. Kozen. Limits for automatic verification of finite-state systems. *IPL*, 15:307–309, 1986.
2. M. Browne, E. Clarke, and O. Grumberg. Reasoning about networks with many identical finite-state processes. *Inf. and Computation*, 81(1):13–31, Apr. 1989.
3. J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. and Computation*, 98(2):142–170, June 1992.
4. E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*, volume 131 of *LNCS*. Springer-Verlag, 1981.
5. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Prog. Lang. Syst.*, 8(2):244–263, 1986.
6. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proc. 19th Ann. ACM Symp. on Principles of Prog. Lang.*, Jan. 1992.
7. D. Dams, O. Grumberg, and R. Gerth. 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, June 1994.
8. E. Dijkstra. Invariance and non-determinacy. In C. Hoare and J. Sheperdson, editors, *Mathematical Logic and Programming Languages*. 1985.
9. E. Emerson and K. S. Namjoshi. Reasoning about rings. In *Proc. 22nd Ann. ACM Symp. on Principles of Prog. Lang.*, Jan. 1995.
10. S. German and A. Sistla. Reasoning about systems with many processes. *J. ACM*, 39:675–735, 1992.
11. R. P. Kurshan and K. L. McMillan. A structural induction theorem for processes. In *Proc. 8th Ann. ACM Symp. on Principles of Distributed Computing*. ACM Press, Aug. 1989.
12. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th Ann. ACM Symp. on Principles of Prog. Lang.*, Jan. 1985.
13. R. Marelly and O. Grumberg. GORMEL—Grammar ORiented ModEL checker. Technical Report 697, The Technion, Oct. 1991.
14. R. Milner. An algebraic definition of simulation between programs. In *In proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pages 481–489, 1971.
15. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer-Verlag, 1980.
16. J. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. Fifth Int. Symp. in Programming*, 1981.
17. Z. Shtadler and O. Grumberg. Network grammars, communication behaviors and automatic verification. In Sifakis [18].
18. J. Sifakis, editor. *Proc. 1989 Int. Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*. Springer-Verlag, June 1989.
19. J. D. Ullman. *Computational Aspects of VLSI*. Computer Science Press, 1984.
20. I. Vernier. Parameterized evaluation of CTL-X formulae. In *Workshop accompanying the Internation Conference on Temporal Logic (ICTL'94)*, 1994.
21. P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In Sifakis [18].

A Appendix: Extension to multiple atomic formulas

Our framework can easily be extended to any set of atomic formulas. The restriction to one atomic formula was done in order to simplify presentation. However, in practice we may want to have several such formulas, related by boolean and temporal operators.

The notion of equivalence can be extended to any set of atomic formulas. 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, s' are *equivalent* if and only if for every 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, $s \in \mathcal{L}(\mathcal{D}_i) \Leftrightarrow s' \in \mathcal{L}(\mathcal{D}_i)$. Abstract *LTS*s are defined as before.

The relation $\models$ is extended for 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 $h(s) \models \mathcal{D}_i$.

B Appendix: Figures explaining derivation rules

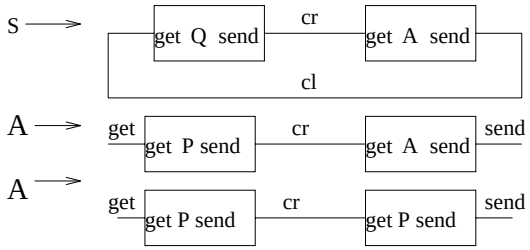

Fig. 7. Derivation rules with renaming

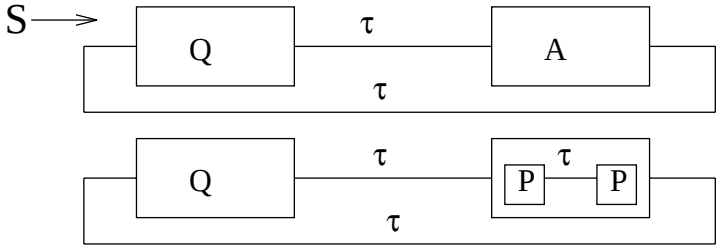

Fig. 8. Derivation of a ring of size 3

This article was processed using the LaTeX macro package with LLNCS style