GORMEL - Grammar ORiented ModEL checker

Rami Marelly Orna Grumberg *
Department of Computer Science
Technion — Israel
orna@cs.technion.ac.il

September 4, 1992

Abstract

GORMEL is a tool for automatic verification of distributed algorithms based on a method proposed in [SG89]. The algorithm is viewed as an infinite family of finite state programs and is described by means of a *network grammar*. Given a grammar and a specification, we first check that the grammar is *inductive* and then check the specification on one of the small programs. If a grammar is not inductive, the system provides us with a counter example that indicates how the grammar should be converted to an inductive one.

The system has been used to verify several algorithms presented in the literature. We examined topologies like trees, rings, buffers and combinations of the above.

1 Introduction

A typical distributed algorithm for communicating processes is designed to be applicable to a family of networks with similar topology. Such a topology has some finite number of process types and different number of processes. A few examples are: choosing a leader in a ring of any size [LeL77], mutual exclusion in a ring [Dij85], distributed termination detection in any connected network [Fra80] and deadlock detection [?]. An algorithm of this kind can be described by associating with each process type a program with a specified set of communication ports, together with rules to combine processes of these types (by pairing ports) to admissible networks.

In many cases, the programs associated with each process type is finite-state. Thus, combining processes into a network results in a finite-state program. In such cases an algorithm can be viewed as an infinite family of finite-state programs. If all programs in the family agree on a given specification (i.e., either all satisfy it or all falsify it), we say that the algorithm is *consistent* with respect to that specification. An algorithm satisfies a specification if each of its finite-state programs satisfies it.

Automatic verification of finite-state programs using temporal logic model checking has been widely investigated by [?], [?], [?], [BCM+90] and many others. Temporal logic model checking determines whether a distributed finite-state program satisfies a specification given as a temporal logic formula. The problem of automatic verification of infinite families of finite-state programs has been studied by several researchers: [CG87], [GS88], [BCG89], [KM89] and [WL89]. However, none of the suggested methods handles a wide class of networks in a fully automatic way.

*This research was partially supported by the U.S.- Isreali Binational Science Foundation.

In [SG89], Shtadler and Grumberg propose *network grammars* as a formalism for describing distributed algorithms. Network grammars are similar to graph grammars [?], [?] except that processes (programs) are assigned to nodes in the graph and edges represent channels in the network through which the processes communicate. The formalism is used to verify that an algorithm described by a grammar satisfies a given specification. To do that, a notion of *inductive grammar* is defined, and a method to check inductiveness is suggested. An algorithm is guaranteed to be consistent if it can be derived by an inductive grammar. Given a grammar, we can check that it is inductive and if so, check the specification on one of the networks it derives. The algorithm satisfies the specification if and only if the checked network satisfies it.

The proposed formalism handles topologies consisting of a fixed number of process types, each of which has a fixed number of communication ports. The specification language used is a two-leveled linear temporal logic called LTL^2. Formulas in the logic can relate the behaviors of processes to each other.

It turned out, that a non-computerized check of grammar inductiveness is very difficult even for small examples, and practically impossible for non trivial ones. Therefore, it was not clear how easy it is to describe algorithms by means of inductive network grammars. Consequently, it was hard to determine the usefulness of the method.

In this paper we present GORMEL, a tool for automatic verification of distributed algorithms based on the method proposed in [SG89]. Given a *network grammar* and a *formula*, the system checks whether the grammar is inductive and if so, verifies the specification on a relatively small derived network. Our tool also assists with the debugging of network grammars. If a grammar is found non-inductive, counter examples are provided, indicating the source of non-inductiveness. This knowledge is then exploited to modify the grammar.

In our work we extend the specification language presented in [SG89], by adding global counter variables (counters). Counters are assigned values from bounded domains of natural numbers. These counters hold *global information* about the network. A typical LTL^2 formula is of the form

$$AF \ \left(nmr\_reds[3] = 5 \ \wedge \ \bigvee_i AG \ black_i \right)$$

This formula is satisfied by a program if in each computation there exists a state (in the future) from which one of the processes is marked *black* forever in all possible computations, and in the same state, the value of the counter *nmr_reds* (which is represented by 3 bits) is equal to 5. Provided that each process assigns its counter to 1 if it is marked *red* and to 0 otherwise, the meaning of $nmr\_reds[3] = 5$ is that exactly 5 processes are marked *red*. As we show in this paper, the global counters enable to express, in a rather simple way, desired properties of programs that could not be expressed without them.

We apply our verifier to several examples. Through these examples we show some properties that can be expressed within our specification language and share our experience in constructing inductive grammars for given algorithms. We believe that the verifier we developed is a first step towards a widely applicable methodology for algorithm verification.

The rest of the paper is organized as follows. In Section 2 we describe our model of computation, and present the network grammars. In Section 3 we describe the logic LTL^2. Section 4 introduces the notion of communication behavior, later used in the verification procedure. This section also includes an algorithm to check equivalence of communication behaviors. Section 5 presents the verification procedure, and Section 6 includes some examples. The paper concludes in Section 7 with a discussion of our results.

2 Processes and Networks

The model of communication is described by means of synchronized communications through a channel that links two ports, one for each process. The model is similar both to CCS [Mil79] and to OCCAM's model of communications [INM84]. Each process is associated with a set of *communication action names* (referred to also as *ports*). ε denotes the internal (non communication) actions of a process. A process with no unpaired ports is called *a network*.

Each process contains some local information describing its current state. We add *global information* about the network by associating with each process a set of *counter variables* (referred to also as *global counters* or simply as *counters*). For each counter, a TOP value is defined. When processes are combined, the values of their counters are summed up. A global counter can be one of two types which differ by the way their summation is defined.

MOD-counter, in which the summation is defined modulo TOP.
TOP-counter, in which the summation is defined up to a ceiling of TOP, where TOP represents all values greater than or equal to TOP.

Definition 2.1: A **process** P is a 9-tuple,

$$P = \langle ACT_P, \mathrm{AP}_P, I_P, \mathrm{CP}_P, S_P, R_P, s_{0P}, L_P, LC_P \rangle$$

where

- ACT_P is a finite set of actions (ports) not containing ε.
- AP_P is a finite set of atomic propositions.
- I_P is a finite set of indices.
- CP_P is a finite set of counters. It consists of two disjoint sets, CM_P of MOD-counters and CT_P of TOP-counters where, $\mathrm{CP}_P = \mathrm{CM}_P \cup \mathrm{CT}_P$.
- S_P is a finite set of states.
- R_P is a labeled transition relation, $R_P \subseteq S_P \times (ACT_P \cup \{\varepsilon\}) \times S_P$. We will use $s \xrightarrow{\alpha} s'$ to indicate that $(s, \alpha, s') \in R_P$, and refer to α as the *transition label.*
- s_{0P} is the initial state.
- $L_P : S_P \to 2^{(\mathrm{AP}_P \times I_P)}$ is the function that labels each state with a set of indexed atomic propositions. We will write a_i instead of (a, i).
- $LC_P : S_P \times \mathrm{CP}_P \to \mathbb{N}$ is a function that assigns to each counter in every state a value over its domain. We will use $LC_P(s)$ to denote the subset of pairs contained in $\mathrm{CP}_P \times \mathbb{N}$ that associates with each counter its value in the state s.

Definition 2.2: A **path** Π in a process P is a finite or infinite sequence of transitions, not necessarily maximal, $\Pi = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \cdots$ such that for each $i \geq 0$, $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$.

Definition 2.3: The **empty process**

$$P_\emptyset = \langle ACT_\emptyset, \mathrm{AP}_\emptyset, I_\emptyset, \mathrm{CP}_\emptyset, S_\emptyset, R_\emptyset, s_{0\emptyset}, L_\emptyset, LC_\emptyset \rangle$$

is the process for which

$$ACT_\emptyset = \mathrm{AP}_\emptyset = I_\emptyset = \mathrm{CP}_\emptyset = R_\emptyset = \emptyset,\ S_\emptyset = \{s_{0\emptyset}\},\ L_\emptyset(s_{0\emptyset}) = LC_\emptyset(s_{0\emptyset}) = \emptyset$$

A **network** is a process in which the set of communication actions ACT is empty.

2.1 Combining processes

A basic process is a process with a single index. Each index in an index set refers to a different basic process. Hence, combining processes involves disjoint union of their index sets. When two processes P and Q are combined into a new process $P||Q$, zero or more channels are formed. Each channel pairs a port of P and a port of Q that have identical names. The channel is used merely for synchronization (i.e., no data is transfered). However, data can be encoded by sequences of synchronizations. The unpaired ports of P and Q become the ports of the combined process $P||Q$. When P and Q are combined together, each counter in the new process is assigned the sum of the counters (with the same name) in the original processes.

Definition 2.4: The **combination** of P and Q is a new process

$$P||Q = \langle ACT_{P||Q}, \mathrm{AP}_{P||Q}, I_{P||Q}, \mathrm{CP}_{P||Q}, S_{P||Q}, R_{P||Q}, s_{0P||Q}, L_{P||Q}, LC_{P||Q}\rangle$$

where

- $ACT_{P||Q} = (ACT_P \cup ACT_Q) - (ACT_P \cap ACT_Q)$
- $\mathrm{AP}_{P||Q} = \mathrm{AP}_P \cup \mathrm{AP}_Q$
- $I_{P||Q} = I_P \cup I_Q$
- $\mathrm{CP}_{P||Q} = \mathrm{CP}_P \cup \mathrm{CP}_Q$
- $S_{P||Q} = (S_P \times S_Q)$
- $R_{P||Q} =$

$$\begin{array}{rcl}
\left\{\left((s_P, s_Q), \varepsilon, (s'_P, s'_Q)\right)\right. & | & \left.\exists\alpha\ :\ s_P \xrightarrow{\alpha} s'_P\ \wedge\ s_Q \xrightarrow{\alpha} s'_Q\right\} \cup \\
\{((s_P, s_Q), \varepsilon, (s'_P, s_Q)) & | & \left. s_P \xrightarrow{\varepsilon} s'_P\right\} \cup \\
\left\{\left((s_P, s_Q), \varepsilon, (s_P, s'_Q)\right)\right. & | & \left. s_Q \xrightarrow{\varepsilon} s'_Q\right\} \cup \\
\{((s_P, s_Q), \alpha, (s'_P, s_Q)) & | & \left.\alpha \notin ACT_Q\ \wedge\ s_P \xrightarrow{\alpha} s'_P\right\} \cup \\
\left\{\left((s_P, s_Q), \alpha, (s_P, s'_Q)\right)\right. & | & \left.\alpha \notin ACT_P\ \wedge\ s_Q \xrightarrow{\alpha} s'_Q\right\}
\end{array}$$

- $s_{0P||Q} = (s_{0P}, s_{0Q})$
- $L_{P||Q}((s_P, s_Q)) = L_P(s_P) \cup L_Q(s_Q)$
- $LC_{P||Q}((s_P, s_Q)) =$

$$\begin{array}{rcl}
\{(mcnt, val) & | & mcnt \in \mathrm{CM}_{P||Q}, \\
& & (mcnt, val_1) \in LC_P(s_P), (mcnt, val_2) \in LC_Q(s_Q), \\
& & val = val_1 + val_2 \pmod{TOP_{mcnt}}\} \cup \\
\{(tcnt, val) & | & tcnt \in \mathrm{CT}_{P||Q}, \\
& & (tcnt, val_1) \in LC_P(s_P), (tcnt, val_2) \in LC_Q(s_Q), \\
& & val = \min\{val1 + val2, TOP_{tcnt}\}\} \cup \\
\{(cnt, val) & | & cnt \in \mathrm{CP}_{P||Q}, \\
& & cnt \notin \mathrm{CP}_P \wedge (cnt, val) \in LC_Q(s_Q) \vee \\
& & cnt \notin \mathrm{CP}_Q \wedge (cnt, val) \in LC_P(s_P)\}
\end{array}$$

When combining processes, we sometimes need to change their port names in order to form a network of a desirable structure. Let $N_P : ACT_P \rightarrow ACT$ be a 1-1 **renaming function** of the ports of P *onto* some set of actions ACT. $N_P(P)$ denotes a new process P', identical to P except that:

- $ACT_{P'} = N_P(ACT_P)$.
- $s \stackrel{N_P(\alpha)}{\rightarrow} t \in R_{P'}$ iff $s \stackrel{\alpha}{\rightarrow} t \in R_P$.

2.2 Network grammars

The formalism suggested in [SG89] for describing distributed finite-state algorithms is a *Network Grammar*, $G = \langle \Sigma, \Delta, \mathcal{P}, \mathcal{T} \rangle$. The grammar defines a subset of networks combined of basic processes in Σ.

- The terminals, Σ, are a set of basic processes.
- The nonterminals, Δ, are a set of process types, each of which represents an unspecified process with a specified set of ports. For a symbol A, this set is denoted by ACT_A.
- $\mathcal{P}$ is the set of production rules of the form

$$A \rightarrow N_B(B)||N_C(C)$$

  where $A \in \Delta$, $B, C \in \Sigma \cup \Delta$, and N_B and N_C are renaming functions. Every production rule must satisfy the condition:

$$ACT_A = N_B(ACT_B) \cup N_C(ACT_C) - N_B(ACT_B) \cap N_C(ACT_C)$$

- The start symbol $\mathcal{T} \in \Delta$ represents the network type, therefore, it has an empty set of action names.

Σ^* is the set of processes combined of basic processes in Σ by applying all possible renaming functions. Each symbol A in the grammar derives a subset of Σ^*. Each occurrence of a basic process in a combined process is associated with a unique index.

Definition 2.5: The **set of processes derived from a symbol** A, denoted by $\mathcal{L}(A)$, is the smallest set that satisfies:

- If $A \in \Sigma$ then $\mathcal{L}(A) = \{A\}$.
- If $A \rightarrow N_B(B)||N_C(C)$ is a grammar rule, $b \in \mathcal{L}(B)$ and $c \in \mathcal{L}(C)$ then $N_B(b)||N_C(c) \in \mathcal{L}(A)$.

Definition 2.6: The **set of environments of a symbol** A, denoted by $\mathcal{E}(A)$, is the smallest set that satisfies:

- $P_\emptyset \in \mathcal{E}(\mathcal{T})$
- If $B \rightarrow N_A(A)||N_C(C)$ or $B \rightarrow N_C(C)||N_A(A)$ are grammar rules, $\bar{b} \in \mathcal{E}(B)$, $a \in \mathcal{L}(A)$ and $c \in \mathcal{L}(C)$ then $N_A^{-1}(\bar{b}||N_C(c)) \in \mathcal{E}(A)$.

For each symbol A such that $a \in \mathcal{L}(A)$, $\bar{a} \in \mathcal{E}(A)$ the following holds:

$$ACT_A = ACT_a = ACT_{\bar{a}}$$

In particular, every process derived from the start symbol $\mathcal{T}$ has an empty set of action names. That is, the symbol $\mathcal{T}$ derives a subset of *networks* in Σ^*. We consider only grammars in which every symbol is reachable, and derives at least one process.

3 The logic LTL^2

LTL^2 is a two-leveled linear time temporal logic. It is defined by means of three types of formulas: local, basic and global. *Local* formulas are LTL formulas describing the behavior of a single process. Quantification over local formulas results in *basic* formulas which describe the behavior of some process (all processes) in the network. Formulas that refer to counters' values are also considered as basic formulas. *Global* formulas relate the behaviors of processes in the network to each other by applying another level of LTL operators. Note that, global formulas can be regarded as LTL formulas where basic formulas serve as atomic propositions. In section 6 we present a variety of LTL^2 formulas and show their usefulness for specification.

LTL^2 is defined over a set of atomic propositions AP, indexed by a set of index variables IV; over a set of counter variables CP and a set of constants over $\mathbb{N}$. The set of the counter variables is the disjoint union of two sets, $\mathrm{CP} = \mathrm{CM} \cup \mathrm{CT}$. Each counter variable $cnt \in \mathrm{CP}$ is associated with a constant TOP_{cnt}. The presented logic extends the logic defined in [SG89] by providing global counters to express global information.

A local formula is either:

- a_i where $a \in \mathrm{AP}$ and $i \in IV$.
- $\neg f$, $f \vee g$ and fUg where f and g are local formulas.

A basic formula is either:

- $\bigvee_i Af(i)$ or $\bigvee_i Ef(i)$ where $f(i)$ is a local formula with exactly one index i.
- $cnt = val$ where $cnt \in \mathrm{CP}$ and $val \in \{0, \ldots, \mathrm{TOP}_{cnt} - 1\}$
- $tcnt < val$ where $tcnt \in \mathrm{CT}$ and $val = \mathrm{TOP}_{tcnt}$.
- $tcnt > val$ where $tcnt \in \mathrm{CT}$ and $val = \mathrm{TOP}_{cnt} - 1$.

A global formula is either:

- f where f is a basic formula.
- $\neg f$, $f \vee g$ or fUg where f and g are global formulas.

An LTL^2 formula is of the form Af or Ef where f is a global formula.

We define the semantics of LTL^2 with respect to a network $(ACT = \emptyset)$, $K = \langle \mathrm{AP}, I, \mathrm{CP}, S, R, s_0, L, LC \rangle$. Let $\Pi = s_0, s_1, \ldots$ be a path in K. Π^i will denote the suffix of Π starting in s_i. We only consider infinite paths in K.

We distinguish between two types of formulas. Formulas of the form $\neg f$, $f \vee g$ or fUg are path formulas. Formulas of the form a_i, Af, Ef, $\bigvee_i f$ or $(cnt\ op\ val)$ where $op \in \{=, <, >\}$ are state formulas. We use the notation $K, \Pi, J \models f$ $(K, s, J \models f)$ to denote that the formula f holds along a path Π (in a state s) for a subset $J \neq \emptyset$ of the index values in the structure K. A state formula f holds along path Π iff it holds in the first state of Π. The relation $\models$ is defined inductively as follows:

$$
\begin{array}{llcl}
K, s, \{i\} & \models a_i & \iff & a_i \in L(s). \\
K, \Pi, J & \models \neg f & \iff & K, \Pi, J \not\models f. \\
K, \Pi, J & \models f \vee g & \iff & K, \Pi, J \models f \text{ or } K, \Pi, J \models g. \\
K, \Pi, J & \models fUg & \iff & \text{there exists } l \geq 0 \text{ such that } K, \Pi^l, J \models g. \\
 & & & \text{and for all } 0 \leq j < l, \quad K, \Pi^j, J \models f. \\
K, s, J & \models \bigvee_i f & \iff & \text{there exists an } i_0 \in J \text{ such that } K, s, \{i_0\} \models f. \\
K, s, J & \models Af & \iff & \text{for every path } \Pi \text{ starting in } s, \quad K, \Pi, J \models f. \\
K, s, J & \models Ef & \iff & \text{there exists a path } \Pi \text{ starting in } s \\
 & & & \text{such that } K, \Pi, J \models f. \\
K, s, J & \models (cnt = val) & \iff & LC(s, cnt) = val. \\
K, s, J & \models (cnt < val) & \iff & LC(s, cnt) < val. \\
K, s, J & \models (cnt > val) & \iff & LC(s, cnt) > val.
\end{array}
$$

In the rest of the paper we use the following abbreviations:

$\bigwedge_i Af = \neg \bigvee_i E\neg f$, $\bigwedge_i Ef = \neg \bigvee_i A\neg f$, $Ff = true\ U\ f$ and $Gf = \neg F \neg f$.

For a formula $f \in \text{LTL}^2$, we define BF(f) as the set of basic formulas in f. Let $\mathcal{F}$ be some set of basic formulas over a set of atomic propositions $\text{AP}_{\mathcal{F}}$. Given two structures K, K', such that $\text{AP}_{\mathcal{F}} \subseteq \text{AP}_K = \text{AP}_{K'}$ and two index sets J_K and $J_{K'}$, such that $J_K \subseteq I_K$ and $J_{K'} \subseteq I_{K'}$, we define two notions of equivalence with respect to $\mathcal{F}$. The *specification equivalence* means that the two structures agree on every LTL^2 formula defined over basic formulas from $\mathcal{F}$. In other words, they agree on every LTL formula without the nexttime operator where $\mathcal{F}$ is the set of atomic propositions. The *computation equivalence* relates computations of the two structures in which states are labeled with elements of $\mathcal{F}$. Corresponding computations are identical up to stuttering. It is shown that the two equivalence notions coincide.

Specification equivalence: $(K, J_K) \equiv_{\mathcal{F}} (K', J_{K'})$ if and only if for every LTL^2 formula f such that $\text{BF}(f) \subseteq \mathcal{F}$,

$$K, s_{0K}, J_K \models f \iff K', s_{0K'}, J_{K'} \models f$$

Computation equivalence: $(K, J_K)\ C_{\mathcal{F}}\ (K, J_{K'})$ if and only if

1. For every path Π starting in s_{0K} there is a path Π' starting in $s_{0K'}$, and there exist partitions of both paths $B_1, B_2, \ldots, B'_1, B'_2, \ldots$ such that for every j, B_j and B'_j are both nonempty and finite. Moreover, for every state s in B_j, every state s' in B'_j, and for every $f \in \mathcal{F}$,
$$K, s, J_K \models f \iff K', s', J_{K'} \models f$$
2. For every path Π' starting in $s_{0K'}$ there is a path Π starting in s_{0K} such that Π and Π' satisfy the same conditions as above.

Theorem 3.1: $(K, J_K) \equiv_{\mathcal{F}} (K', J_{K'})$ iff $(K, J_K)\ C_{\mathcal{F}}\ (K', J_{K'})$.

The proof of this theorem appears in [**?**].

4 The communication behavior

The *communication behavior of a process P, given an environment Q*, is denoted by $P\!:\!Q$. P is the tested process and Q is the environment process. The communication behavior is defined only in case the combination $P||Q$ is a network. The main differences between $P||Q$ and $P\!:\!Q$ are:

- Transitions of $P:Q$ are labeled by actions in ACT_P, ε that denotes internal actions of the tested process P and $\bar{\varepsilon}$ that denotes internal actions of the environment process Q.
- A state s in $P:Q$ is labeled by *basic formulas* rather than atomic propositions.
- Counters in $P:Q$ are assigned their values in P rather than the sum of their values in P and Q.
- $P:Q$ has two special states *sink* and $\overline{sink}$. All communications on channels between P and Q in which P is capable of communicating while Q is not, are directed to *sink*. Similarly, communications in which Q is capable of communicating while P is not, are directed to $\overline{sink}$. This records the mutual influence of the tested process P and the environment process Q by means of communication ability.

In order to define $P:Q$ we first define the following notions. An ε**- path** is a path that contains only ε transitions. ε**- closure(s)** is the set of states reachable from s by ε - paths. Now, we can formally define the communication behavior.

Definition 4.1: Given P, Q, such that $ACT_P = ACT_Q$, the **communication behavior** is an 8-tuple

$$P:Q = \langle ACT_{P:Q}, BF_{P:Q}, \mathrm{CP}_{P:Q}, S_{P:Q}, R_{P:Q}, s_{0_{P:Q}}, \mathcal{M}_{P:Q}, LC_{P:Q}\rangle$$

where

- $ACT_{P:Q} = ACT_P$
- $\mathrm{BF}_{P:Q}$ is a set basic formulas defined over AP_P. It is used as the set of atomic labelings of states in $P:Q$.
- $\mathrm{CP}_{P:Q} = \mathrm{CP}_P$
- $S_{P:Q} = (S_P \times S_Q) \cup \{sink, \overline{sink}\}$ where *sink* and $\overline{sink}$ are special states.
- $s_{0_{P:Q}} = (s_{0P}, s_{0Q})$
- $\mathcal{M}_{P:Q}$ is the labeling of states with basic formulas.

$$\begin{array}{lcl} \mathcal{M}_{P:Q}((s_P, s_Q)) & = & \{f \mid f \in BF_{P:Q} \text{ and } P||Q, (s_P, s_Q), I_P \models f\} \\ \mathcal{M}_{P:Q}(sink) & = & issink \\ \mathcal{M}_{P:Q}(\overline{sink}) & = & \overline{issink} \end{array}$$

- $LC_{P:Q}((s_P, s_Q)) = LC_P(s_P)$. Since each counter can be regarded as a set of bits, we can consider $\mathrm{CP}_{P:Q}$ as an extension of $\mathrm{BF}_{P:Q}$ and $LC_{P:Q}$ as an extension of $\mathcal{M}_{P:Q}$.
- $R_{P:Q} =$

$$\begin{array}{lcl} \left\{\left((s_P, s_Q), \alpha, (s'_P, s'_Q)\right)\right. & | & \left. s_P \xrightarrow{\alpha} s'_P \wedge s_Q \xrightarrow{\alpha} s'_Q \right\} \cup \\ \{((s_P, s_Q), \varepsilon, (s'_P, s_Q)) & | & \left. s_P \xrightarrow{\varepsilon} s'_P \right\} \cup \\ \left\{\left((s_P, s_Q), \bar{\varepsilon}, (s_P, s'_Q)\right)\right. & | & \left. s_Q \xrightarrow{\varepsilon} s'_Q \right\} \cup \\ \{((s_P, s_Q), \alpha, sink) & | & \alpha \neq \varepsilon, \;\; \exists s_P \xrightarrow{\alpha} s'_P, \neg\exists s'_Q, s''_Q \;: \\ & & \left. s'_Q \in \varepsilon - closure(s_Q) \wedge s'_Q \xrightarrow{\alpha} s''_Q \right\} \cup \\ \left\{\left((s_P, s_Q), \alpha, \overline{sink}\right)\right. & | & \alpha \neq \varepsilon, \;\; \exists s_Q \xrightarrow{\alpha} s'_Q, \neg\exists s'_P, s''_P \;: \\ & & \left. s'_P \in \varepsilon - closure(s_P) \wedge s'_P \xrightarrow{\alpha} s''_P \right\} \cup \\ \{(sink, \varepsilon, sink)\} \cup & & \\ \left\{(\overline{sink}, \bar{\varepsilon}, \overline{sink})\right\} & & \end{array}$$

$R_{P:Q}$ is a union of 7 sets of transitions. The first one records the transitions in which P and Q synchronize on some channel. The second and the third contain internal transitions of P and Q, respectively. The fourth records the communications which are enabled by P but disabled by Q in the current state and all states reachable along ε-paths. The fifth set records the dual case where Q enables the communication but P disables it. The transition in both cases is labeled with the disabled action name. The sixth and seventh sets contain self loops for *sink* and $\overline{sink}$, and they are labeled ε and $\bar{\varepsilon}$ respectively.

Following, we define an equivalence between communication behaviors. This notion captures two important properties. When two communication behaviors are equivalent, their corresponding networks agree on the specification. Moreover, replacing the tested component of one with the tested component of the other results in a communication behavior which is equivalent to the original ones. In order to define *communication behavior equivalence*, we first have to define *state equivalence*.

4.1 State Equivalence in Communication Behaviors

Definition 4.2: A path Π in $P\!:\!Q$ and a path Π' in $P'\!:\!Q'$ are **communication corresponding paths** (denoted $\Pi \approx_c \Pi'$) if their communication sequences are equal. Moreover, if their communication sequences are finite, then

- Π is finite iff Π' is finite.
- Π includes the state *sink* ($\overline{sink}$) iff Π' includes the state *sink* ($\overline{sink}$).
- Π includes infinite number of ε ($\bar{\varepsilon}$) transitions iff Π' includes infinite number of ε ($\bar{\varepsilon}$) transitions.

Definition 4.3: Let $P\!:\!Q$ and $P'\!:\!Q'$ be such that $\mathrm{BF}_{P:Q} = \mathrm{BF}_{P':Q'}$, $\mathrm{CP}_{P:Q} = \mathrm{CP}_{P':Q'}$ and $ACT_{P:Q} = ACT_{P':Q'}$. The states $s \in P\!:\!Q$ and $s' \in P'\!:\!Q'$ are **state equivalent** (denoted $s \equiv_{P:Q} s'$) if and only if

- For every path Π from s there exists a *communication corresponding path* Π' from s'.
- For every path Π' from s' there exists a *communication corresponding path* Π from s.
- $\mathcal{M}_{P:Q}(s) = \mathcal{M}_{P:Q}(s')$ and $LC_{P:Q}(s) = LC_{P:Q}(s')$.

State equivalence indicates that the states are labeled with the same subset of basic formulas. Moreover, they can not be distinguished by any communication sequence. The equivalence definition has been suggested in [SG89]. Below we present an algorithm to compute the equivalence relation and prove its correctness.

4.2 Communication Behavior Equivalence

Definition 4.4: Let Π be a path in $P\!:\!Q$, and let $B = B_0, B_1, \ldots$ be a partition of Π into blocks. B is a **maximal partition** of Π if for every two consecutive states, s_i and s_{i+1} are in the same block if and only if $s_i \equiv_{P:Q} s_{i+1}$ and either $s_i \xrightarrow{\varepsilon} s_{i+1}$ or $s_i \xrightarrow{\bar{\varepsilon}} s_{i+1}$.

Note that, if s_i and s_{i+1} are not in the same block, then either $s_i \not\equiv_{P:Q} s_{i+1}$ or $s_i \xrightarrow{\alpha} s_{i+1}$, where α is a communication action. Also, note that all blocks are finite except for the last block which may be infinite.

Definition 4.5: Two communication behaviors $P\!:\!Q$ and $P'\!:\!Q'$ are equivalent ($P\!:\!Q \equiv P'\!:\!Q'$) iff

- $\mathrm{ACT}_{P:Q} = \mathrm{ACT}_{P':Q'}$
- $\mathrm{BF}_{P:Q} = \mathrm{BF}_{P':Q'}$
- For every path Π starting in $s_{0_{P:Q}}$ there is a path Π' starting in $s_{0_{P':Q'}}$ such that the maximal partition of Π, $B_1, B_2, \ldots$, and the maximal partition of Π', $B'_1, B'_2, \ldots$ satisfy that for every j:
  1. For every state s in B_j and every state s' in B'_j, $s \equiv_{P:Q} s'$.
  2. $\mathrm{last}(B_j) \xrightarrow{\alpha_j} \mathrm{first}(B_{j+1})$ iff $\mathrm{last}(B'_j) \xrightarrow{\alpha_j} \mathrm{first}(B'_{j+1})$.
  3. The last block B_n is infinite and contains an infinite number of ε ($\bar{\varepsilon}$) transitions iff the last block B'_n is infinite and contains an infinite number of ε ($\bar{\varepsilon}$) transitions.
- For every path Π' starting in $s_{0_{P':Q'}}$ there is a path Π starting in $s_{0_{P:Q}}$ that satisfies the above conditions.

This equivalence definition has also been presented in [SG89]. An algorithm to compute it is presented below.

4.3 An Algorithm for Communication Behavior Equivalence

The algorithm for communication behavior equivalence has two main phases. First we partition the states of the given communication behaviors into *state equivalence* classes. Then, we check if the initial states of the two communication behaviors are equivalent according to the *communication behavior equivalence* defined above.

Our notion of state equivalence is very similar to the *Failure equivalence* of *Restricted NFSA's* described in [KS83] which is PSPACE-Complete and to the *testing equivalence* described in [NH84]. The algorithm is similar to the algorithms for determinization and minimization of regular automata, as described in [HU79]. We define a *deterministic process* for each communication behavior and then partition their states into equivalence classes.

4.3.1 Deterministic processes of type I

Definition 4.6: Given a communication behavior $P\!:\!Q$, a **deterministic process of type I**, denoted $\mathcal{D}_{P:Q}$, is a tuple $\mathcal{D} = \langle \mathcal{U}, \mathcal{V}, \mathcal{W} \rangle$ where:

- $\mathcal{U}$ is the set of states.

$$\mathcal{U} = 2^{S} \cup \{inf_{\varepsilon}, inf_{\bar{\varepsilon}}, inf_{\varepsilon\bar{\varepsilon}}\} \cup \{sink, \overline{sink}\} - \{\emptyset\}$$

- $\mathcal{V}$ is the set of transitions. In order to define $\mathcal{V}$, we first define the *($\varepsilon/\bar{\varepsilon}$)-closure* of a state s in $P\!:\!Q$.

$$\begin{aligned} ST_0(s) &= \{s\} \\ ST_{n+1}(s) &= \{s' \mid \exists s'' \in ST_n(s) \;:\; s'' \xrightarrow{\varepsilon} s' \;\vee\; s'' \xrightarrow{\bar{\varepsilon}} s'\} \\ ST(s) &= \bigcup_n ST_n(s) \end{aligned}$$

We also define the following three predicates:

$loop_{\varepsilon}(s) = true$	$\iff$	There exists a state $s' \in ST(s)$ from which there is a cycle containing only ε transitions.
$loop_{\bar{\varepsilon}}(s) = true$	$\iff$	There exists a state $s' \in ST(s)$ from which there is a cycle containing only $\bar{\varepsilon}$ transitions.
$loop_{\varepsilon\bar{\varepsilon}}(s) = true$	$\iff$	There exists a state $s' \in ST(s)$ from which there is a cycle containing only ε and $\bar{\varepsilon}$ transitions, and at least one of each.

The transition relation contains the following transitions:

$$\begin{aligned}
U_1 \xrightarrow{\alpha} U_2 &\iff U_2 = \{s' \mid \exists s \in U_1, \exists s'' \in ST(s) \; : \; s'' \xrightarrow{\alpha} s' \in R_{P:Q}\} \\
U_1 \xrightarrow{\varepsilon} inf_{\varepsilon} &\iff \exists s \in U_1 \; : \; loop_{\varepsilon}(s) = true \\
U_1 \xrightarrow{\bar{\varepsilon}} inf_{\bar{\varepsilon}} &\iff \exists s \in U_1 \; : \; loop_{\bar{\varepsilon}}(s) = true \\
U_1 \xrightarrow{\varepsilon} inf_{\varepsilon\bar{\varepsilon}} &\iff \exists s \in U_1 \; : \; loop_{\varepsilon\bar{\varepsilon}}(s) = true \\
U_1 \xrightarrow{\alpha} sink &\iff \exists s \in U_1, \exists s' \in ST(s) \; : \; s' \xrightarrow{\alpha} sink \in R_{P:Q} \\
U_1 \xrightarrow{\alpha} \overline{sink} &\iff \exists s \in U_1, \exists s' \in ST(s) \; : \; s' \xrightarrow{\alpha} \overline{sink} \in R_{P:Q}
\end{aligned}$$

- $\mathcal{W}$ is the labeling function that labels states in $\mathcal{D}$ with basic formulas. Each singleton is labeled as the state it contains, while states containing more then one original state are labeled by the empty set.

We define **type-I equivalence, E** over the states of a deterministic process, similarly to the *bisimulation equivalence* ([Mil79], [BCG88]), i.e., states are equivalent according to this notion if every α-son of one is type-I equivalent to some α-son of the other.

Definition 4.7:

$$\begin{aligned}
\mathrm{E}_0 &= \{(U, U') \mid \; U \in \mathcal{D} \wedge U' \in \mathcal{D}\} \\
\mathrm{E}_{n+1} &= \{(U, U') \mid \; U \mathrm{E}_n U' \; \wedge \\
&\qquad \forall U_1 (U \xrightarrow{\alpha} U_1) \implies (\exists U_1' \; : \; U' \xrightarrow{\alpha} U_1' \wedge U_1 \mathrm{E}_n U_1') \\
&\qquad \forall U_1' (U' \xrightarrow{\alpha} U_1') \implies (\exists U_1 \; : \; U \xrightarrow{\alpha} U_1 \wedge U_1 \mathrm{E}_n U_1')\} \\
\mathrm{E} &= \bigcap_n \mathrm{E}_n
\end{aligned}$$

Theorem 4.1: $s \equiv_{P:Q} s'$ if and only if $\{s\}$ and $\{s'\}$ are type-I equivalent, $\mathcal{M}_{P:Q}(s) = \mathcal{M}_{P:Q}(s')$ and $LC_{P:Q}(s) = LC_{P:Q}(s')$.

The proof of this theorem is left to the appendix.

4.4 An Algorithm for Communication Behavior Equivalence

The algorithm we present here is based on ideas similar to those employed in the algorithm for state equivalence. As before, we first construct deterministic processes and then partition their states into equivalence classes.

4.4.1 Deterministic processes of type II

Definition 4.8: Given a communication behavior $P:Q$, a **deterministic process of type II**, denoted $\mathcal{D}_{P:Q}$, is a tuple $\mathcal{D} = \langle \mathcal{U}, \mathcal{V}, \mathcal{W} \rangle$ where:

- $\mathcal{U}$ is the set of states.

$$\mathcal{U} = \{U \mid U \subseteq S, U \neq \emptyset, \forall s_1, s_2 \in U \ : \ s_1 \equiv_{P:Q} s_2\} \cup \{inf_{\varepsilon}, inf_{\bar{\varepsilon}}, inf_{\varepsilon\bar{\varepsilon}}\} \cup \{sink, \overline{sink}\}$$

- $\mathcal{V}$ is the set of transitions. As before, we define the *$(\varepsilon/\bar{\varepsilon})$-closure* of a state s in $P:Q$. Yet now we consider only $\varepsilon/\bar{\varepsilon}$ transitions to states which are state equivalent to s. We also define the predicates $loop_{\varepsilon}$, $loop_{\bar{\varepsilon}}$ and $loop_{\varepsilon\bar{\varepsilon}}$ with respect to the new definition of ST.
For each communication action α let R_α be a relation over the states of $\mathcal{D}$ such that

$$(U_1, U_2) \in R_\alpha \iff \forall s \in U_2\ \exists s' \in U_1\ \exists s'' \in ST(s')\ \ (s'' \xrightarrow{\alpha} s)$$

The transition relation contains the following transitions:

$$\begin{array}{rcl}
U_1 \xrightarrow{\alpha} U_2 & \iff & U_2 \text{ is a maximal set that satisfies } (U_1, U_2) \in R_\alpha \\
U_1 \xrightarrow{\varepsilon(\bar{\varepsilon})} U_2 & \iff & U_2 \text{ is a maximal set that satisfies} \\
 & & \forall s \in U_2\ \exists s' \in U_1\ \exists s'' \in ST(s')\ \ s'' \xrightarrow{\varepsilon(\bar{\varepsilon})} s \wedge s \notin ST(s') \\
U_1 \xrightarrow{\varepsilon} inf_{\varepsilon} & \iff & \exists s \in U_1\ :\ loop_{\varepsilon}(s) = true \\
U_1 \xrightarrow{\bar{\varepsilon}} inf_{\bar{\varepsilon}} & \iff & \exists s \in U_1\ :\ loop_{\bar{\varepsilon}}(s) = true \\
U_1 \xrightarrow{\varepsilon} inf_{\varepsilon\bar{\varepsilon}} & \iff & \exists s \in U_1\ :\ loop_{\varepsilon\bar{\varepsilon}}(s) = true \\
U_1 \xrightarrow{\alpha} sink & \iff & \exists s \in U_1, \exists s' \in ST(s)\ :\ s' \xrightarrow{\alpha} sink \in R_{P:Q} \\
U_1 \xrightarrow{\alpha} \overline{sink} & \iff & \exists s \in U_1, \exists s' \in ST(s)\ :\ s' \xrightarrow{\alpha} \overline{sink} \in R_{P:Q}
\end{array}$$

- $\mathcal{W}$ is the labeling function that labels each state in $\mathcal{D}$ with exactly those basic formulas that label its constituents.

We define a **type-II equivalence** over the states of a deterministic process. States are type-II equivalent if they are state equivalent and every α-son of one is type-II equivalent to some α-son of the other.

Theorem 4.2: $P:Q$ and $P':Q'$ are equivalent communication behaviors if and only if their initial states are type-II equivalent.

The proof of this theorem generally follows the same lines as the proof of Theorem 4.1. One main difference is that unlike deterministic processes of type I, where a state can have at most one outgoing transition with a given label, in a deterministic process of type II, a state can have more than one outgoing transition with a given label but these transitions can not lead to states that contain equivalent original states.

4.5 The Algorithm

Given two communication behaviors $P:Q$ and $P':Q'$:

1. Create two deterministic processes of type I.

2. Partition the states of the deterministic processes into *type I equivalence* classes.
3. Create two deterministic processes of type II.
4. Partition the states of the deterministic processes into *type II equivalence* classes.
5. The communication behaviors are equivalent iff their initial states are type II equivalent.

Following, communication behavior equivalence between communication behaviors is related to specification equivalence between the corresponding networks.

Theorem 4.3: If $P:Q \equiv P':Q'$ then $(P||Q, I_P) \equiv_{\mathrm{BF}_{P:Q}} (P'||Q', I_{P'})$.

A proof of this theorem appears in [SG89].

5 The Verification Method

We will show here a modified version of the verification procedure introduced in [SG89].

Definition 5.1: Let an **algorithm** be a set of networks. An algorithm is **consistent** with respect to a given specification if all the networks in the set agree on the specification. I.e., either all networks satisfy the specification or all falsify it.

Definition 5.2: A grammar G is **inductive** with respect to a given specification if for every symbol $A \in \Sigma \cup \Delta$, for every $a, a' \in \mathcal{L}(A)$ and $\bar{a}, \bar{a}' \in \mathcal{E}(A)$, $a : \bar{a} \equiv a' : \bar{a}'$.

Intuitively, this means that all subnetworks derived from A have the same behavior in all environments derived for A by the grammar. The specification determines the set of basic formulas that label the states of the communication behaviors.

Theorem 5.1: Every set of networks derived by an inductive grammar is consistent.

Corollary 5.2: An algorithm is consistent if there exists an inductive grammar that derives its set of networks.

Theorem 5.3: Let G be a network grammar. For every symbol $W \in G$ let w and $\bar{w}$ be a representative and an environment of W. If for every rule $A \to N_B(B)||N_C(C)$, $a : \bar{a} \equiv (N_B(b)||N_C(c)) : \bar{a}$, $\bar{b} : b \equiv N_B^{-1}(\bar{a}||N_C(c)) : b$ and $\bar{c} : c \equiv N_C^{-1}(\bar{a}||N_B(b)) : c$ then G is inductive.

The proofs of these theorems appear in [SG89].

The verification procedure accepts a network grammar and an LTL[2] formula, describing a distributed algorithm and a specification, respectively, and checks whether the grammar is *inductive*. If so, it concludes that all networks derived by the grammar agree on the specification (i.e., the algorithm is consistent). The specification then holds for the algorithm if and only if it holds for some of its representative networks.

5.1 When are Environments Needed ?

In the verification procedure presented in the next section, one has to construct processes representing environments of symbols in the grammar. Below we consider the cases in which environments should be constructed.

- When a symbol appears more than once as a head of a derivation rule, we test processes derived from that symbol and compare their behavior. These comparisons are done with respect to the symbol's environment.
- When a symbol appears more than once in a body of a derivation rule, there are several environments in which it can be located. We have to compare the behaviors of those environments and therefore, we build a representative environment to which all are compared.
- When a symbol's environment is required for the construction of another environment, it should be constructed.

According to these rules, we can inductively decide which environments should be constructed and which may be avoided. This improvement to the algorithm, though trivial, is significant since in general, environments whose construction is avoided are particularly large. A typical case are algorithms in which a unique process appears only once (e.g. the one with the token, the initiator etc.). The environment of such a process is the whole network excluding the process itself. With our modification this environment is not built, thus time and space are saved.

5.2 The Verification procedure

We assume that the network grammar is given as a sequence of derivation rules satisfying the following condition

> If B appears in a body of a derivation rule then B is either a terminal or appears as a head of a rule located **after** this rule in the sequence.

It is clear that any grammar, in which all symbols are reachable and each derives at least one word, can be ordered to satisfy the above condition. Given a *Network grammar* describing a distributed algorithm $G = \langle \Sigma, \Delta, \mathcal{P}, \mathcal{T} \rangle$ and a specification *SPEC*, the algorithm consists of the following steps:

1. Find $BF(SPEC)$, the set of basic formulas in *SPEC*.
2. Find which symbols require environments according to the rules described above.
3. For each symbol $W \in \Sigma \cup \Delta$, construct a representative as follows:
   (a) The representative of each terminal is the basic process associated with it.
   (b) Proceed through the grammar from the last rule to the first. For every rule $A \rightarrow N_B(B)||N_C(C)$, if no representative is associated with A then define $a = N_B(b)||N_C(c)$ to be the representative of A. Since the grammar is ordered as required, it is guaranteed that when this rule is reached, the representatives b and c of B and C are already defined.
4. For each symbol $W \in \Sigma \cup \Delta$, construct an environment representative as follows:
   (a) The empty process $P_\emptyset$ is the environment of the start symbol $\mathcal{T}$.
   (b) Proceed through the grammar rules from the first to the last. For every rule $A \rightarrow N_B(B)||N_C(C)$, if the environment of B is not defined and if B requires an environment, define $\bar{b} = N_B^{-1}(\bar{a}||N_C(c))$ to be the environment of B. The environment of C is defined similarly, if required.
5. For each derivation rule $A \rightarrow N_B(B)||N_C(C)$,

- If the representative of A is not defined with respect to this rule, build the communication behaviors $a : \bar{a}$ and $(N_B(b)||N_C(c)) : \bar{a}$, label them with basic formulas from $BF(SPEC)$ using a model checker for LTL and check if they are equivalent.
- If the environment of B is not defined with respect to this rule, build the communication behaviors $\bar{b} : b$ and $N_B^{-1}(\bar{a}||N_C(c)) : b$, label them with basic formulas and check if they are equivalent.
- If the environment of C is not defined with respect to this rule, build the communication behaviors $\bar{c} : c$ and $N_C^{-1}(\bar{a}||N_B(b)) : c$, label them with basic formulas and check if they are equivalent.

If any of these equivalence checks does not hold then the grammar is not inductive. A counter example is provided to indicate how G should be modified for the checks to hold.

6. If all checks succeed then G is inductive and we check *SPEC* by applying an LTL Model Checker to $t : P_\emptyset$. The algorithm satisfies *SPEC* if and only if $t : P_\emptyset$ satisfies it.

6 Examples

In this section, we present some examples of distributed algorithms verified by GORMEL. We show how these algorithms can be described by means of inductive grammars. If a grammar is not inductive, we exemplify the use of counter examples to find the source of its non-inductiveness. This knowledge is then exploited to modify the grammar. Each network grammar is described by a list of its rules together with a description of the programs running in each node. The rules are binary and the obvious way to write them is in the form $A \to B||C$. However, writing the rules in such form lacks the information of how the processes' ports are combined together. This information is given by a pictorial description of the rules. The programs are written in pseudo-code that enables non-deterministic statements and communication between processes. Statements that are carried out simultaneously will be written between $\langle \ldots \rangle$. When the programs are simple and a literal description suffices, the code is omitted.

We run the examples on a SUN4/490. The LTL model checker we use was developed by Nir Friedman in the Weizmann Institute [SP89]. We provide with each example the CPU-time and space it required.

6.1 Mutual Exclusion in a Ring

In this section we verify an algorithm for mutual exclusion in a ring of arbitrary size. This example has already been presented but not fully verified in [SG89]. The algorithm is based on a single token passed between the processes according to their order on the ring. There are two basic process types in this algorithm, f and e. f starts with a token and e starts without a token. The processes participating in the protocol are described in Figure 1. Every ring contains one process of type f and an arbitrary number (≥ 3) of processes of type e. The network grammar is illustrated in Figure 2.

The specification we want to check consists of the following statements:

Mutual exclusion: in every moment, along every possible computation, exactly one of the processes owns a token. This statement can easily be expressed using global counters. Provided that each process assigns its counter to 1 if it has a token and to 0 if it doesn't, the required property holds if and only if the sum of the counters over all processes is always 1. This in

```
process f;
AP:     token = true;
CP:     nmr_tokens[TOP 2] = 1;
begin
    do forever
        ⟨ send  token; {  Three operations are done simultaneously  }
            token = false;
            nmr_tokens = 0; ⟩
        ⟨ recv  token; {  Three operations are done simultaneously  }
            token = true;
            nmr_tokens = 1; ⟩
end .

process e;
AP:     token = false;
CP:     nmr_tokens[TOP 2] = 0;
begin
    do forever
        ⟨ recv  token; {  Three operations are done simultaneously  }
            token = true;
            nmr_tokens = 1; ⟩
        ⟨ send  token; {  Three operations are done simultaneously  }
            token = false;
            nmr_tokens = 0; ⟩
end .
```

Figure 1: processes of the ring

$T \longrightarrow F \,\|\, E$

$F \longrightarrow f \,\|\, e$

$E \longrightarrow E \,\|\, e$

$E \longrightarrow e \,\|\, e$

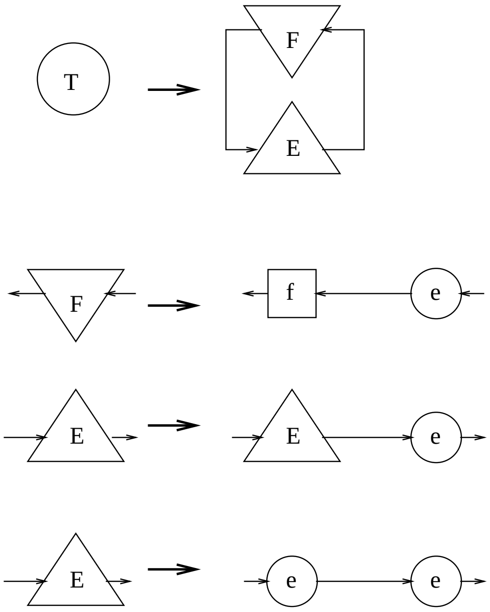

Figure 2: network grammar for a ring

turn, is equivalent to the fact that the value of the global counter in the combined network is always 1.

Starvation freedom: Each process in enabled infinitely often along every possible computation. This statement can be formulated in a rather straightforward way.

Hence, our specification can be captured in the LTL^2 formula:

$$A\ G\ \left(nmr\_tokens[2] = 1 \text{ AND } \bigwedge_i A\ F\ token_i \right)$$

The given grammar has been verified with respect to this specification. It took 9.0 seconds to check that the grammar is inductive and that it satisfies the specification. Memory used in this example was 32 KB.

6.2 XOR in a Binary tree

Here we verify an algorithm for calculating the parity (XOR) of the values in the leaves of a binary tree. The algorithm is based on an upgoing wave in which, each leaf sends its value to its father (deciding upon the value is done non-deterministically while sending it). Every internal node receives values from its sons, computes their XOR and sends the result to its father. The root node receives values from its sons and computes their XOR.

A preliminary network grammar for this algorithm is given in Figure 3. The statement we

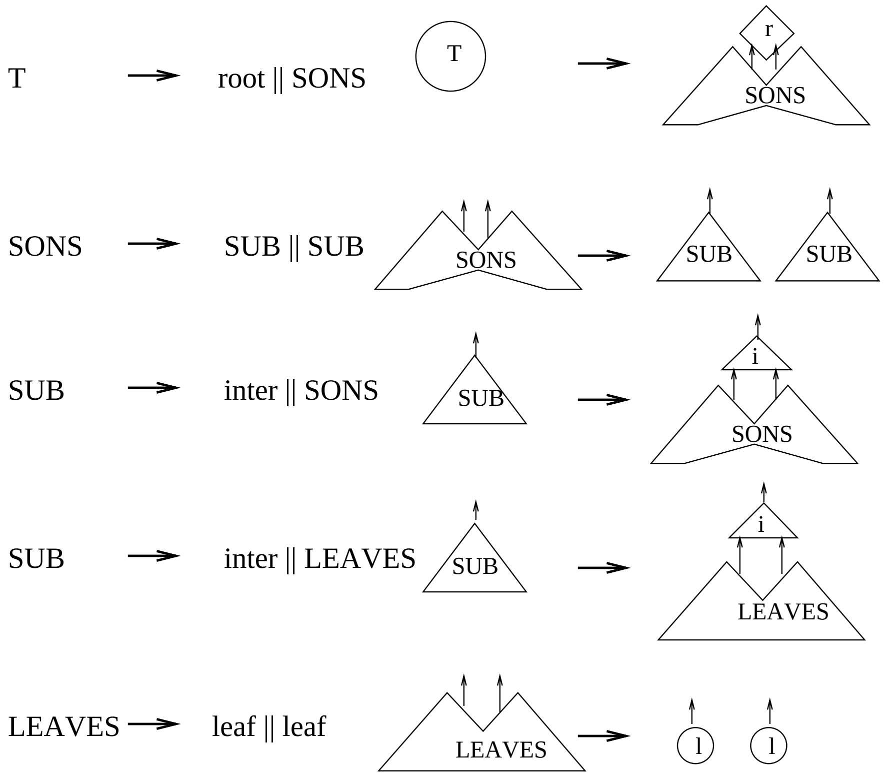

Figure 3: Preliminary network grammar for a binary tree

would like to verify is:

The value computed in the root is indeed the XOR of the values decided by the leaves.

In order to express such an assertion in our specification language, we add a global counter **xor** *modulo 2*. Initially, the value of this counter is 0 in all nodes. Each leaf, after deciding its value, assigns its counter to that value. Thus, the value of the counter **xor** in the combined network will be the sum of the leaves values modulo 2, or in other words, the XOR of their values. Now, if we also make the root assign its counter to the value computed, we get that the value of the counter in the global network is 0 if and only if the value computed by the root is the XOR of the leaves' values.

Therefore, the LTL^2 formula to be checked is:

$$A\ F\ G\ xor[1] = 0$$

While running GORMEL on this example, when the rule

$$SUB \quad \rightarrow \quad inter \,||\, LEAVES$$

was checked for inductiveness, the representative process of the internal node (*inter*) was tested under two environments as shown in Figure 4 (actually, *inter* plays the role of the environment in this check). Testing these two communication behaviors for equivalence has yielded the message that

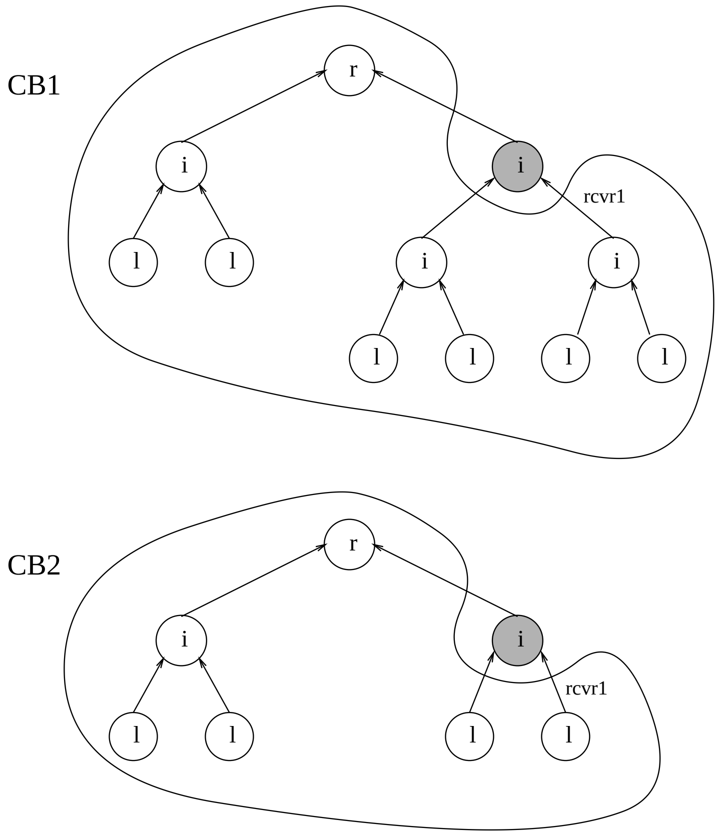

Figure 4: inter in two environments

the path containing the action *rcvr1* and ending with the state $\overline{sink}$, in CB1, has no corresponding path in CB2.

The communication action *rcvr1* denotes receiving the value 1 from the right son. Hence, the meaning of this message is that in CB1 there exists a path starting with internal moves of the environment, reaching a state from which the environment can not send the value 1 to *inter* from its right son. The internal node is capable of receiving this value from its son, and therefore, a transition to $\overline{sink}$ is created. In CB2 there is no such path since as long as the internal node is capable of receiving the value 1, the environment is capable of sending it.

The difference in the communication behaviors originates from the fact that a leaf determines its value only upon sending it, therefore it is capable of any communication with the internal node. In CB1 however, before any communication between *inter* and its right son occurs, the XOR of the leaves' values in the right subtree is computed. Thus, it is possible that the value computed by the right son is 0, and then it can send only 0 and is incapable of sending 1. A computation like that will end with a *rcvr1* transition to $\overline{sink}$.

This difference in the behaviors leads us to the conclusion that even though an internal node runs the same algorithm wherever it is located, some internals differ from others depending on the environments they operate in. If *inter* is connected directly to the leaves then its environment is capable of all communications, and if it is connected to other internal nodes, there are some computations in which some communications are disabled.

Reaching these conclusions, converting the grammar to be inductive is quite simple. We define two distinct terminals for the two kinds of internal nodes. There is a way to tell the verifier that two terminals are associated with the same basic process. This is quite convenient when we want two symbols to use the same algorithm but do not want them to be tested for equivalence, such as in this case. The new grammar is described in Figure 5.

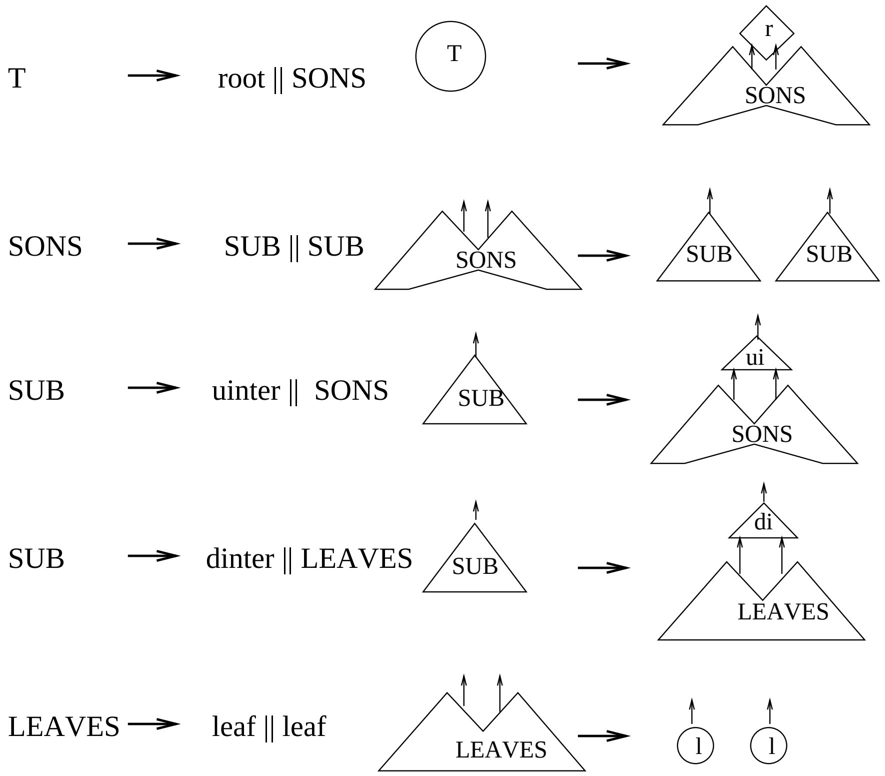

Figure 5: Network grammar for a binary tree

The new grammar was verified to be inductive and to satisfy the given specification in 1:55

minutes. Memory used in this example was 3328 KB.

6.3 Token Waves in a Tree with a Ring

In this example we verify an algorithm in which token waves are passed down a tree (this example has been suggested to us by Nissim Francez). We will use this example to illustrate some of the problems that arise in creating *inductive* network grammars. The networks on which the algorithm runs are a combination of a tree and a ring. The tree is a simple binary tree (except for the root which has only one son). The ring is a unidirectional ring from the root to the leftmost leaf, through all the leaves and back to the root. Initially, all the nodes are identically colored. The root sends a token of the other color to its son, and a signal to its neighbor on the ring (the leftmost leaf). It changes its color and waits for the signal to get back from its other neighbor on the ring (the rightmost leaf). The procedure then starts all over again. An internal node receives a token from its father, changes its color to be the same as that of the token and sends tokens to its sons. A leaf receives a token from its father and a signal from its left neighbor on the ring, changes its color and sends a signal to its right neighbor.

The algorithm, though simple, can be viewed as an abstraction of many algorithms that involve information propagation with feedback. Algorithms for value passing in a tree and algorithms for distributed termination detection in a tree can be regarded as special cases of this abstraction. A preliminary grammar network for a tree with a ring, describing the above algorithm, is given in Figure 6.

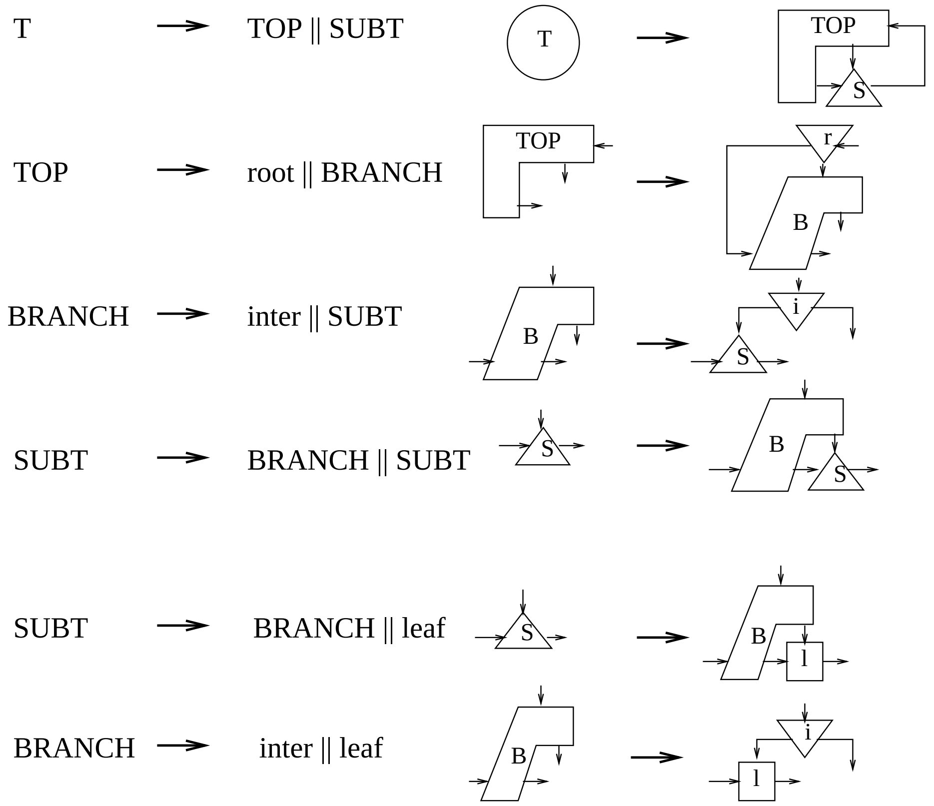

Figure 6: Preliminary network grammar for a tree with a ring

The specification we want to check for the algorithm is :

> Along every computation, infinitely often all the nodes are simultaneously colored by one of the colors, and infinitely often all are colored by the other.

The LTL^2 formula that captures this statement is the following:

$$A\ G\ ((F\ (\bigwedge_i A\ odd_i))\ \mathrm{AND}\ (F\ (\bigwedge_i A\ \neg odd_i)))$$

While running GORMEL on this grammar, several of the *state equivalence* checks have failed. We consider one of the checks and see how the counter examples provided by the verifier, can be used to understand the problem and to fix it. During the check of the rule

$$BRANCH \;\; \rightarrow \;\; inter \,||\, SUBT$$

two processes that could be derived from the symbol *BRANCH* were tested for equivalence. The two communication behaviors are given in Figure 7.

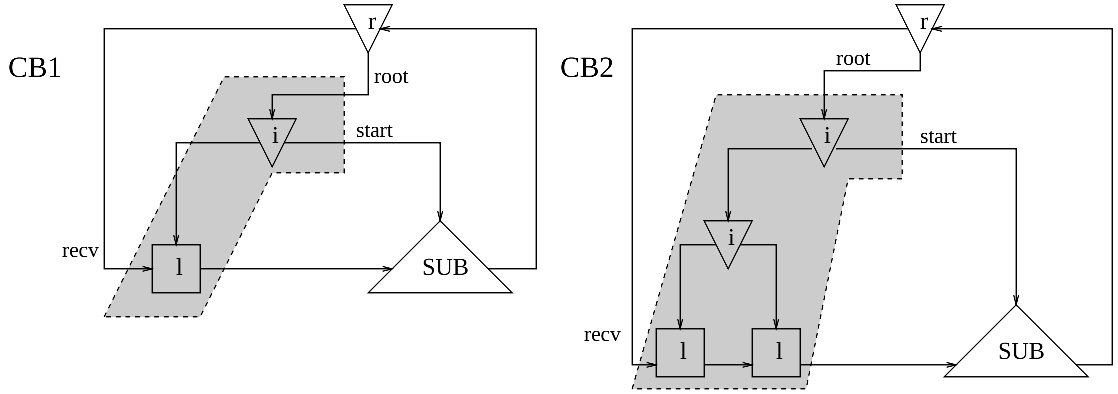

Figure 7: Two BRANCH processes

The message produced by GORMEL indicated that the sequence of communications

$$\text{root, start, recv, recv} \;\; \rightarrow \;\; sink$$

in CB2 has no corresponding sequence in CB1. This message means that the sequence of communications in which the root sends a token to its son (*root*), that son sends a token to its right son (*start*) and the root sends a signal to the leftmost leaf (*recv*) is possible in both communication behaviors, but in CB2 there is another transition to *sink* labeled with *recv* which does not have a match in CB1. That is, the *BRANCH* in CB2 can reach a state in which it is ready to get another *recv* from the root, while the *BRANCH* in CB1 can not reach such a state. This difference originates from the fact that the *BRANCH* in CB2 has more than one leaf and therefore, by an internal action, can pass the signal from the left leaf to its right neighbor, thus enabling the left leaf to receive another signal from the root. The root is not capable of communicating with the left leaf since it did not get a signal from the right leaf yet, and thus we have a *recv* transition to *sink*. The *BRANCH* in CB1, containing only one leaf, can not reach such a state along internal actions.

To overcome this problem, we introduce a modified grammar containing two nonterminals to derive branches. The nonterminal *BRANCH* derives the set of branches containing a single leaf while the nonterminal *BIG-BR* derives the set of branches containing more than one leaf. An important feature of the modified grammar is that it derives exactly the same set of networks as the original one.

T → TOP || SUBT

TOP → root || MAIN-BR

MAIN-BR → minter || SUB

SUB → BIG-BR || SUB

BIG-BR → uinter || SUB

SUB → BRANCH || leaf2

BRANCH → dinter || leaf1

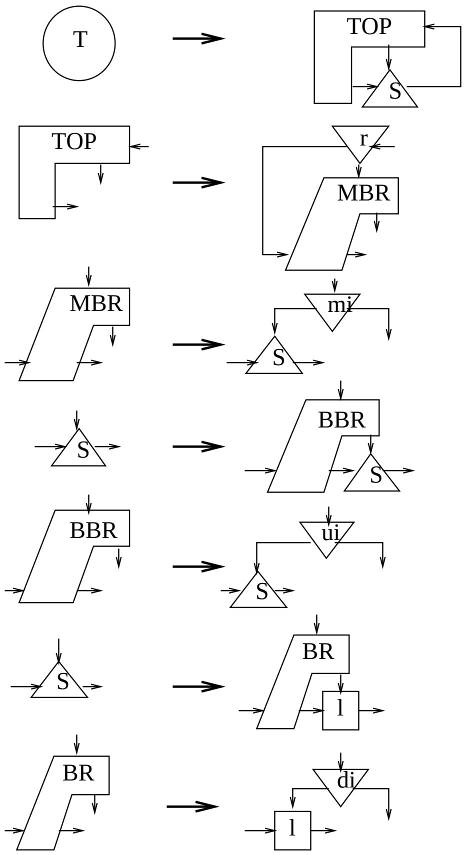

Figure 8: Network grammar for a tree with a ring

Similar problems arose in other equivalence checks and were solved similarly. The modified grammar is given in Figure 8.

The modified grammar was checked by GORMEL. This time, some *communication behavior equivalence* checks have failed. For example, one of the problems revealed by the verifier was that the behavior of a *SUBTREE* depends on its location in the network. That is, the rightmost *SUBTREE* differs from the leftmost *SUBTREE* in the network, and they both differ from all other subtrees. Let us compare the behaviors of the rightmost *SUBTREE* with the behavior of some *SUBTREE* located in another place as shown in Figure 9. In this case, one can see how

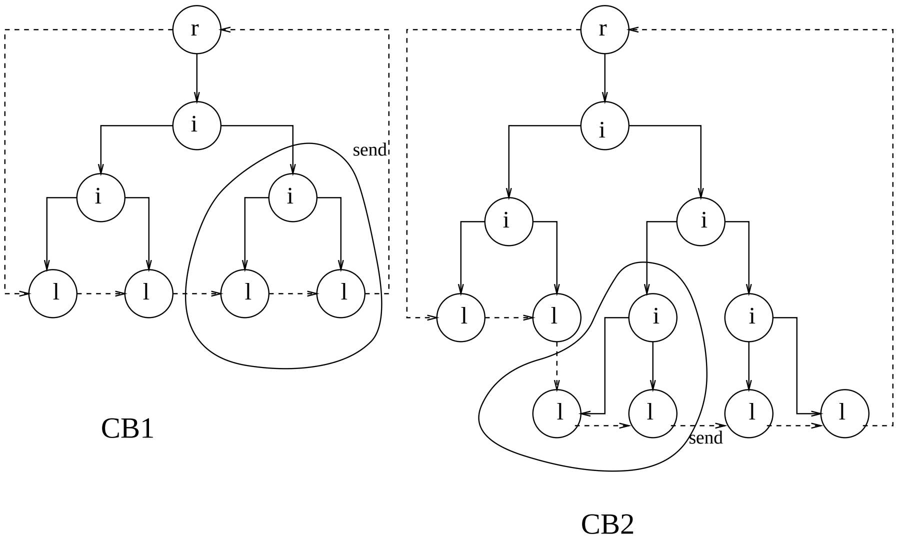

Figure 9: Two SUBTREES in different locations

the communication behavior equivalence depends upon the given specification. The specification consists of basic formulas stating that all processes have the same color simultaneously.

While running the program, the message received was that CB1 and CB2 are not equivalent since the transition *send* to *sink* in CB2 has no corresponding transition in CB1. The initial states of the two communication behaviors are equivalent according to the *state equivalence* and therefore such a transition must exist.

Let us examine the behaviors of CB1 and CB2 starting at their initial states. In both networks, initially, all processes are identically colored, and therefore the initial states satisfy one of the basic formulas (the same one).

In CB2, the leaf to the right of *SUBTREE* is ready to receive the message *send* from *SUBTREE* while the latter is not capable of sending it. Therefore, a *send* transition to *sink* exists from the initial state.

On the other hand, in CB1, the node to the right of *SUBTREE* is the root which is not ready to accept a *send* message in the initial state. Only after sending a token down the tree, a message along the ring and changing its color, the root is in a state in which it expects a *send* message. As before, this results in a *send* transition to *sink*. However, this time, the transition to *sink* originates from a state which is not equivalent to the initial one, since it satisfies none of the basic formulas.

This problem can be solved by handling the rightmost *SUBTREE* by a separate nonterminal. A similar problem arose for the leftmost *SUBTREE* and was solved similarly. The new grammar rules are therefore:

$$
\begin{array}{lcl}
T & \rightarrow & TOP||RSUBT \\
TOP & \rightarrow & root||MBRANCH \\
MBRANCH & \rightarrow & minter||LSUBT \\
LSUBT & \rightarrow & LBRANCH||SUBT \\
SUBT & \rightarrow & BIGBRANCH||SUBT \\
RSUBT & \rightarrow & BIGBRANCH||RSUBT \\
BIGBRANCH & \rightarrow & uinter||SUBT \\
SUBT & \rightarrow & BRANCH||leaf2 \\
RSUBT & \rightarrow & RBRANCH||rleaf \\
BRANCH & \rightarrow & dinter||leaf1 \\
 & & \\
FILES & & \\
minter & \rightarrow & inter.proc \\
dinter & \rightarrow & inter.proc \\
uinter & \rightarrow & inter.proc \\
leaf1 & \rightarrow & leaf.proc \\
leaf2 & \rightarrow & leaf.proc \\
rleaf & \rightarrow & leaf.proc \\
LBRANCH & \rightarrow & BRANCH.proc \\
RBRANCH & \rightarrow & BRANCH.proc
\end{array}
$$

Here we use an aliasing option provided by GORMEL. This option allows for two symbols to have the same representative. Yet, it prevents the comparison of this representative in different environments.

This network grammar was proved to be inductive and to satisfy the above specification. The verification took 1:43:06 hours and used 6752 KB.

6.4 Finding a Leader in an Acyclic Graph

In this example we verify an algorithm for finding a leader in an acyclic graph of degree 3. This algorithm can also be regarded as an algorithm for directing a tree, since it is based on directing the graph from the leaves towards some root and declaring the root as the leader. The networks, on which this algorithm runs, contain two process types - internal nodes of degree 3, and leaves of degree 1. Each leaf can, non-deterministically, receive a message from its neighbor and declare itself as leader, or send a message to its neighbor. Each internal node waits for messages from its neighbors. After receiving two messages, it can either receive one more message and declare itself as leader, or send a message to the neighbor from which no message has arrived. Sending a message corresponds to directing the edge in the graph from the sending node to the receiving one. The network grammar for acyclic graphs of degree 3 is given in Figure 10. The symbol *PAIR1* is a variable that derives the same process as *PAIR*. We omit the actual derivation rule from the grammar and instead, define the representative of *PAIR1* to be that of *PAIR*.

The specification we want to check contains two parts:

- Eventually, a *single* leader will be declared.
  Here also, we establish an assertion referring to the satisfiability of a certain property in *one* process and *only one*. We chose to do that using a global counter that counts up to 2 (TOP 2). Every process that declares itself as leader will assign its counter to 1, and all others will leave their counters set to 0. Thus, we get that in the global network, this counter will contain

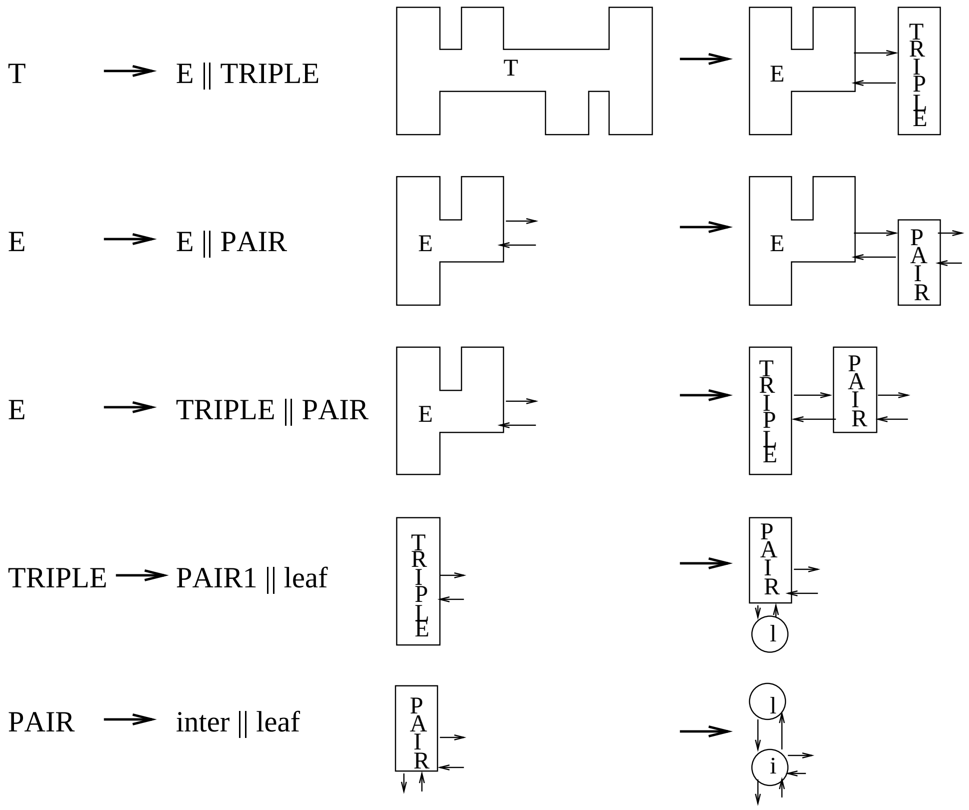

Figure 10: Network grammar for tree of degree 3

0 if no leader was elected, 1 if a single leader was elected, and 2 if ,more than 1 leaders were elected. All we have to do is to compare the value of this counter in the global network to 1.

- For every process, there exists a computation in which this process will be elected as leader. This statement can be expressed in a rather straightforward way in our language.

The LTL[2] formula to be checked is therefore:

$$A\ ((F\ G\ leaders[2] = 1)\ AND\ (\bigwedge_i E\ F\ root_i))$$

This specification was verified for the given grammar. The grammar was found to be inductive and to satisfy the specification. It took 36.8 seconds and 192 KB to verify this example.

6.5 Shared Memory

In this example we show how to simulate algorithms in a model where all the processes can access a shared memory. The legal operations on the memory are *test&set*. That is, each process can read the contents of the memory, and write a new value in a single operation. every *test&set* operation is *atomic*. The algorithm we show, is not complicated, and its only purpose is to show how to verify algorithms in the shared memory model. The model scheme is given in Figure 11. The shared memory in our example contains a single bit which can be assigned 0 or 1 (initialized to 0). The number of participating processes is arbitrarily large.

Each process starts in a *sleep* state, and can wake up at any time. The goal of the algorithm is to enable every process, that wakes up, to know whether there are any other processes in a *wake*

Figure 11: shared memory model

state. In order to do that, each process, after waking up, checks the value of the memory. If it is 0, it changes it to 1. If it is 1, it leaves it unchanged, and marks that there are more *wake* processes ($found\_t = true$).

The way in which we transform an algorithm from the shared memory model to the communication model is by combining the processes into a *ring*. The atomicity of the actions is achieved by a token passed between the processes in the ring. Each process can perform meaningful actions (actions related to the algorithm) only if it owns the token. In order to keep all the possible sequences of actions we allow each process to receive the token and immidietly send it, without performing any meaningful actions. Thus, we enable action sequences that are not necessarily induced by a *round robin* on the ring. In addition, we have to keep the memory updated in all the processes. Actually, it suffices to keep the memory of a process updated only when it owns the token. To do that, we will use the token itself. Instead of one token, we will have a token for every possible value of the memory (2 tokens in our case). Every process will have its own copy of the memory, and the token will save the status of the memory at every moment. When receiving a token, the process will update its copy of the memory, act according to the updated values, and finally, send a token representing the new status of the memory. The distributed model, substituting the shared memory model, is shown in Figure 12.

Figure 12: The communication model

In the algorithm presented, there is only one process type. Due to the transformation of the algorithm into the communication model, we have to distinguish between two process types. One process will start with the token while others will start without it. The two process types participating in the algorithm are given in Figure 13. The grammar for the networks on which the algorithm runs is basically a grammar for rings and is similar to the one shown in Figure 2, where p_0 and p_1 substitute e and f respectively. There are also two kinds of messages instead of one in the original grammar.

The specification we want to check is

```
process p0;
AP: reg = false, found_t = false;
CP: wake[TOP 2] = 0;
begin
     while not UP do
          [              {  i ∈ {0,1}  }
          □ recv_i →
                    send_i;
          □ recv_0 →
                    UP = true; wake = 1; send_1;
          □ recv_1 →
                    UP = true; found_t = true; wake = 1; send_1;
          ]
     do forever
          recv_i;
          send_i;
end .

process p1;
AP: reg = false, found_t = false;
CP: wake[TOP 2] = 0;
begin
     choose if UP or not non deterministically.
     [
          □ UP →
                    send_1; wake = 1;
          □ ¬ UP →
                    send_0;
     ]
     while not UP do
          [              {  i ∈ {0,1}  }
          □ recv_i →
                    send_i;
          □ recv_0 →
                    UP = true; wake = 1; send_1;
          □ recv_1 →
                    UP = true; found_t = true; wake = 1; send_1;
          ]
     do forever
          recv_i;
          send_i;
end .
```

Figure 13: processes in the ring

If a certain process decides that there are *wake* processes besides itself, then there are at least two *wake* processes in the network.

In order to check this specification, we use a global counter (**wake**) which counts until 2 (TOP 2). Every process raises his counter to 1 when waking up, and thus, the value of this counter in the global network will be the minimum between the number of *wake* processes and 2.

The LTL^2 formula is therefore:

$$A\ G\ ((\bigvee_i E\ found\_t_i)\ \ ->\ \ \ wake[2] > 1)$$

This specification was checked for the given grammar and was found to be true. It took 48 seconds to verify that the grammar is consistent and satisfies the specification.

6.6 The Alternating Bit Protocol

In this example we will demonstrate some additional aspects of the expressive power of our language. To do this, we consider a variant of the Alternating Bit Protocol, first presented in [**?**]. The purpose of the protocol is to enable passing messages from a source to a destination when messages can be lost (we do not distinguish between lost messages and erroneous ones). The principle is simple, the source associates a control bit with each message and keeps the control bit unchanged until it gets an acknowledgment with the same control value. When the destination receives a message, it acknowledges by sending the same bit as associated with the received message. Both the source and the destination use timeout mechanisms. If the destination does not receive a message within a given time period it repeats its last acknowledgment. Similarly, if the source does not receive an acknowledgment within a given time period, it repeats its last message with the same control bit. The destination only considers the content of the first message in any sequence that have the same control bit.

In our implementation we assume, for simplicity, that the line from the destination to the source never fails, while the line from the source to the destination can fail arbitrarily. We ignore the contents of the messages and refer only to the control bits associated with them. We regard the communication line as going through transmission stations where each of the stations can fail arbitrarily.

Since we can not model timeout, we will use a slightly different model. When a station fails (thus causing the lost of a message), it sends a message of type *lost*, independent of the message received. Receiving a *lost* message by the destination means that one of the stations has failed and will cause a request for another transmission of the same message. *lost* messages can be considered as simulating timeout. The network consists of one source, one destination and an arbitrary number of transmitters between them. The process types are described in Figure 14.

Let us now explain the role of the atomic propositions. *msg* indicates the value of the control bit in each process. It is meaningful only in the source and destination. *ok* indicates whether the station passed the message correctly. The source and destination are always *ok*. *src* and *dst* are true in the source and destination respectively and false elsewhere. *send* is used to indicate the moments in which the source send messages. Before it sends a message it raises this flag, and lowers it immediately after it is sent. *recv* is the symmetric flag for the destination. Whenever a message is received, *recv* is raised, and is lowered when the acknowledgment is sent. We will specify the use of these atomic propositions later.

The network grammar for this algorithm is given in Figure 15. The reason for the distinc-

```
process src;
AP: msg = 0, ok = true, src = true, dst = false, send = true, recv;
begin
    do forever
        ⟨ send msg; send = false; ⟩
        ⟨ recv ack; send = true; ⟩
        if (msg = ack) then
            msg = ¬ msg;
end .

process dst;
AP: msg = 1, ok = true, src = false, dst = true, send, recv = false;
begin
    do forever
        ⟨ recv newmsg; recv = true; ⟩
        if (newmsg = msg) ∨ (newmsg = lost) then
            ⟨ recv = false; send msg; ⟩ {  As an acknowledge  }
        else
            msg = ¬ msg;
            ⟨ recv = false; send msg; ⟩ {  As an acknowledge  }
end .

process trns;
AP: msg = 0, ok, src = false, dst = false, send, recv;
begin
    do forever
        recv msg;
        [
            □ true→
                ⟨ ok = true; send msg; ⟩
            □ true→
                ⟨ ok = false; send lost; ⟩
        ]
        recv ack;
        send ack;
end .
```

Figure 14: Process in the AB - protocol

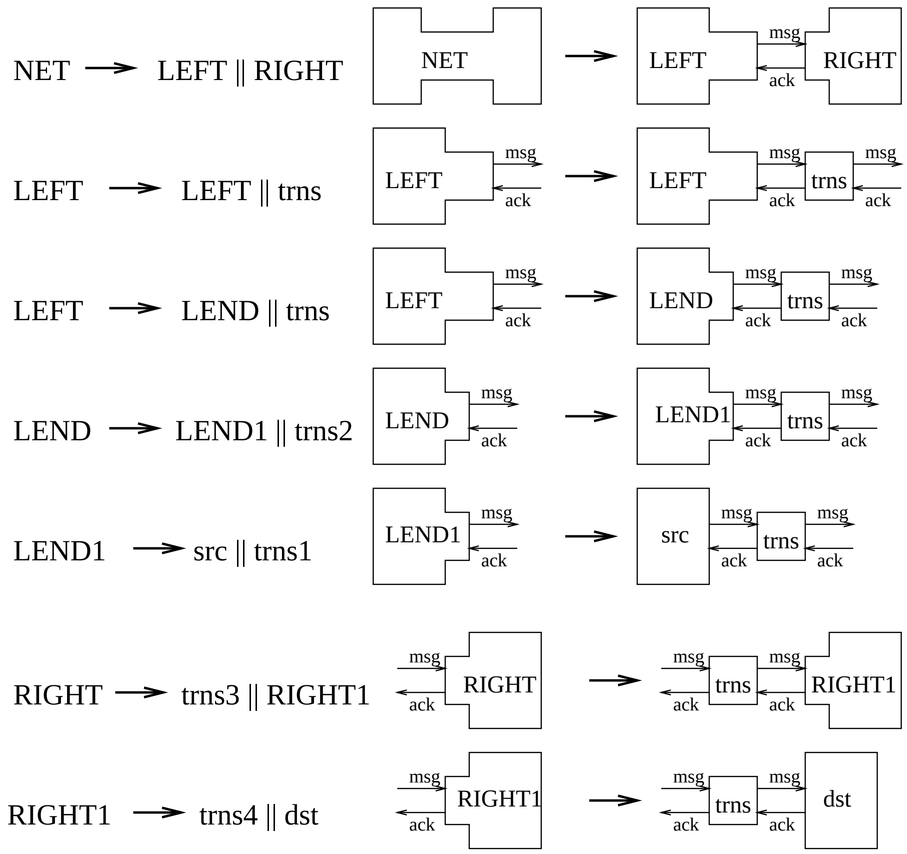

Figure 15: Network grammar for a src - dst line

tion between the two right stations, the two left stations and the other stations originates from the differences in their environments. For example, the left station can never get a *lost* message while other stations can. The right station can never pass correctly the value 1 and receive an acknowledgment 0, while other stations can (provided that some station located to their right will fail).

The protocol as defined, does not guarantee the arrival of any message from the source to the destination, since stations can fail continuously. Let us define *fair* computations to be all computations that satisfy the following:

> Infinitely many times, all stations work correctly from the moment of sending the message in the source until the moment it is received in the destination.

Once we restrict our attention to fair computations we can verify the property:

> For all *fair* computations, it is guaranteed that the destination will receive an infinite number of messages labeled with 1 and an infinite number of messages labeled with 0.

It should be noted that for expressing such a property, we must be able to refer to a single process (sending by the source). Moreover, we must have the ability to refer to fair computations as defined. In general, it is undesirable to be able to refer to a *specific* process in the network, since this enables to distinguish between networks of different sizes. However, there are several cases (quite a lot) where the network contains some unique processes and many other identical processes. In such cases we would like to specify properties of those unique processes. LTL^2 does not allow this directly, since all atomic propositions are quantified with process quantifiers. We overcome this problem by adding a special atomic proposition which is true only in the unique process. Then, we require that *for all processes*, if the new special proposition is true, then so is the property we would like to verify.

The ability to express *fairness constraints* is not exclusive to LTL^2, and is possible due to the LTL structure of the top level of the LTL^2 formulas. In order to formulate the fairness constraint defined above, we use the flags *send* and *recv*, and require that from the moment that *send* is true in the source until *recv* is true in the destination, all the stations work correctly. The LTL^2 formula is therefore:

$$
\begin{array}{l}
A\ (\\
\quad (GF\ ((\bigwedge_i A\ (src -> \ send))\ \wedge\ ((\bigwedge_i A\ ok)\ U\ (\bigwedge_i A\ (dst -> \ recv)))))\ \ -> \\
\quad ((GF\ \ \bigwedge_i A\ (dst -> \ msg))\ \wedge\ ((GF\ \ \bigwedge_i A\ (dst -> \ \neg msg))))\\
)
\end{array}
$$

This example was given as input to GORMEL. The grammar was proved to be inductive and the specification to be true. It took 22.6 seconds and 192 KB to verify this example.

7 Conclusions

In this paper we presented a system for automatic verification of distributed algorithms based on a method suggested in [SG89]. The system has been used to verify several algorithms presented in the literature. We tried several network topologies like trees, rings, buffers and combinations of the above. We have developed algorithms for finding *state equivalence* and *communication behavior equivalence*. These algorithms involve the construction of structures that are theoretically exponential in the size of the graphs they originate from. Yet, our experience shows that in practice, they grow linearly with the size of the original graphs.

As mentioned before, our method is applicable only to *inductive* network grammars. One of the main questions we have tried to resolve is how easily can algorithms be described as inductive network grammars. We have found that the naive way to come up with such a description sometimes does not work. However, our verifier has been proved quite efficient in the process of debugging, yielding inductive network grammars.

Once an inductive grammar is given, our method is fully automatic. One may argue that finding an inductive grammar needs as much human intuition as required in finding an invariant. Human intuition is indeed required, but we believe that the former task is easier than the latter. More important, here the intuition is aided by an automatic system.

One may also wonder why our method is so complicated comparing to other inductive methods that use invariants [WL89], [KM89]. The answer lies in the fact that network grammars are more expressive than other methods that use ordinary induction. These methods start with some basis and extend it in a single manner (e.g., adding a process to a ring or to a buffer). On the other hand, network grammar enables a combined extension of the basis in several manners. Thus, we can regard the methods that use plain induction as a special case of a network grammar in which only one symbol can derive more than one subnetwork. In these special cases, our method is not more complicated than others.

It should be noted that this is only an experimental implementation, and can handle relatively small examples due to space limitations. Recently, Binary Decision Diagrams (BDD) [Bry86] were investigated in the context of automatic model checking [BCM+90]. BDDs have proved to be quite efficient in representing finite state systems. We believe that a BDD-based implementation of our system may result in an automatic verifier applicable to a large family of distributed algorithms.

A Proof of Theorem 4.1

Lemma A.1: Let s be a state in $P:Q$, then the following conditions are equivalent:

1. There exist an infinite path from s containing only ε transitions and an infinite path from s containing only $\bar{\varepsilon}$ transitions.
2. There exists a path from s containing an infinite number of ε transitions, an infinite number of $\bar{\varepsilon}$ transitions and those only.

Proof outline:

Note that, a communication behavior is a combination of two processes and therefore, each of its states is a combination of two original states, one from P and one from Q. We will denote s as a pair (s_0, s_0'). We also recall that ε denotes an internal transition of P (which is independent of Q) and that $\bar{\varepsilon}$ denotes an internal transition of Q (which is independent of P).

In order to prove that the first condition implies the second, we construct the desired path by interleaving states from the two given paths, and show that all transitions are legal, using the above observation. In order to prove the other direction, we eliminate from the given path all the $\bar{\varepsilon}$- transitions while leaving the second component permanently s_0'. We show that this yields a legal infinite path containing only ε transitions. An infinite path containing only $\bar{\varepsilon}$ transitions is obtained similarly. □

Lemma A.2: [The embedding lemma]

1. For every path $\Pi = s_0, s_1, \ldots$ in $P\!:\!Q$ there exists a path $\Pi_D = U_0, U_1, \ldots$ in $\mathcal{D}_{P:Q}$ such that:
   (a) $U_0 = \{s_0\}$
   (b) If $s_{i-1} \stackrel{\alpha_k}{\rightarrow} s_i$ is the kth communication transition then $s_i \in U_k$ and $U_{k-1} \stackrel{\alpha_k}{\rightarrow} U_k$.
   (c) If Π ends with an infinite number of ε ($\bar{\varepsilon}$) transitions (with no communications) then Π_D is finite and ends with the state inf_{ε} ($inf_{\bar{\varepsilon}}$). If Π ends with an infinite number of ε transitions and an infinite number of $\bar{\varepsilon}$ transitions, with no communications, then Π_D is finite and ends with the state $inf_{\varepsilon\bar{\varepsilon}}$.
2. For every path $\Pi_D = U_0, U_1, \ldots$ in $\mathcal{D}_{P:Q}$ there exists a path $\Pi = s_0, s_1, \ldots$ in $P\!:\!Q$ such that conditions (b) and (c) hold, and in addition, $s_0 \in U_0$.

Definition A.1: Two paths Π and Π_D that satisfy the above conditions are **corresponding w.r.t. the embedding lemma** (denoted $\Pi \approx_e \Pi_D$).

Proof outline:

1. The proof consists of three stages:
   (a) We prove that for every finite path Π there exists a corresponding path Π_D, by induction on the length of the path.
   (b) We use könig's lemma to conclude that the lemma also holds for paths with an infinite number of communications.
   (c) We describe this case in detail:
   If Π is infinite but contains only a finite sequence of communications and $s_n \stackrel{\alpha_k}{\rightarrow} s_{n+1}$ is the last communication action, then, according to 1a, there exists a path $\Pi'_D = U'_0, U'_1, \ldots U_k$' corresponding ($\approx_e$) to the path $\Pi' = s_0, \ldots, s_{n+1}$ such that $s_{n+1} \in U_k$. Since there exists an infinite path containing only ε, $\bar{\varepsilon}$ transitions or both, from s_{n+1} in $P\!:\!Q$, and since the number of states in $P\!:\!Q$ is finite, it follows that $loop_{\varepsilon}$, $loop_{\bar{\varepsilon}}$ or both are true, and therefore there exists a transition from U_k to inf_{ε}, $inf_{\bar{\varepsilon}}$ or $inf_{\varepsilon\bar{\varepsilon}}$ respectively. Hence, by definition, $\Pi_D = \Pi'_D \stackrel{\varepsilon/\bar{\varepsilon}}{\rightarrow}$ INF corresponds to Π where INF is inf_{ε}, $inf_{\bar{\varepsilon}}$ or $inf_{\varepsilon\bar{\varepsilon}}$ respectively.
2. Here also, the proof consists of three stages:
   (a) We prove that for every finite path Π_D that does not end with one of the INF states there exists a corresponding path Π. The proof is done by induction on the length of the path.
   (b) We use könig's lemma to conclude that the lemma holds for infinite paths as well.
   (c) If $\Pi_D = \Pi'_D \stackrel{\varepsilon}{\rightarrow} inf_{\varepsilon}$, then by the definition of deterministic processes, there exsists a state s_n in the last state of Π'_D such that $loop_{\varepsilon}(s_n)$ is true. It follows from 2a that there exsists a path $\Pi = s_0, \ldots, s_n$ corresponding to Π'_D. We define $ST^{\omega}_{\varepsilon}(s)$ to be some infinite path starting with s and containing ε transitions only. Such a path exists iff $loop_{\varepsilon}(s) = true$. It is clear now that, $\Pi = \Pi' \circ ST^{\omega}_{\varepsilon}(s_n)$ corresponds to Π_D (where $\circ$ is path concatenation). □

Definition A.2: Two paths Π_D in $\mathcal{D}$ and Π'_D in $\mathcal{D}'$ are **corresponding** ($\approx$) if

1. For every k, $U_{k-1} \stackrel{\alpha_k}{\rightarrow} U_k$ iff $U'_{k-1} \stackrel{\alpha_k}{\rightarrow} U'_k$.

2. The last state in Π_D is one of *sink*, $\overline{sink}$, $inf_{\mathcal{E}}$, $inf_{\overline{\mathcal{E}}}$, $inf_{\mathcal{E}\overline{\mathcal{E}}}$ iff the last state in Π'_D is *sink*, $\overline{sink}$, $inf_{\mathcal{E}}$, $inf_{\overline{\mathcal{E}}}$, $inf_{\mathcal{E}\overline{\mathcal{E}}}$ respectively.

Corollary A.3: If $\Pi_D \approx \Pi'_D$ then for every i, $\Pi^i_D \approx \Pi'^i_D$.

Theorem A.4: Let Π_D be a path from $\{s\}$ in $\mathcal{D}$.
If there exist paths Π in $P:Q$, Π' in $P':Q'$ and Π'_D in $\mathcal{D}'$ such that:

- $\Pi \approx_e \Pi_D$.
- $\Pi' \approx_c \Pi$.
- $\Pi'_D \approx_e \Pi'$.

then, $\Pi_D \approx \Pi'_D$.

The correctness of this theorem follows directly from the various correspondence definitions.

Definition A.3: Let $U \in \mathcal{D}$ and $U' \in \mathcal{D}'$ be states in deterministic processes of type I. We say that $U \approx U'$ if:

1. For every path Π from U, there exists a path Π' from U' such that $\Pi \approx \Pi'$.
2. For every path Π' from U', there exists a path Π from U such that $\Pi \approx \Pi'$.

Lemma A.5: If $U \approx U'$, U_1 is an α-son of U in $\mathcal{D}$, and U'_1 is an α-son of U' in $\mathcal{D}'$, then $U_1 \approx U'_1$.

This lemma follows from the definition of the relation ($\approx$) and from the fact that in a deterministic process of type I, a state can have at most one outgoing transition labeled with α.

Lemma A.6: If $U \approx U'$ then $U \mathrm{E}_n U'$ for every n.

This lemma is proved by induction on n, using Lemma A.5.

Theorem A.7: Let $s \in P:Q$ and $s' \in P':Q'$ then:
$s \equiv_{P:Q} s' \implies \{s\}\mathrm{E}\{s'\}, \mathcal{M}(s) = \mathcal{M}(s')$ and $LC(s) = LC(s')$.

Proof :

$s \equiv_{P:Q} s'$, therefore $\mathcal{M}(s) = \mathcal{M}(s')$ and $LC(s) = LC(s')$ by definition. Let Π_D be a path from $\{s\}$. by Lemma A.2 (The embedding lemma)(2), there exists a path Π in $P:Q$ from s such that $\Pi \approx_e \Pi_D$. Since $s \equiv_{P:Q} s'$ there exists a path Π' from s' such that $\Pi' \approx_c \Pi$. by Lemma A.2 (1) there exists a path Π'_D from $\{s'\}$ such that $\Pi'_D \approx_e \Pi'$. Now, by Theorem A.4 we get that $\Pi_D \approx \Pi'_D$.

The proof of the other direction is similar. That is, for every path Π'_D from $\{s'\}$ there exists a path Π_D from $\{s\}$ such that $\Pi_D \approx \Pi'_D$. Putting it all together, we get that $\{s\} \approx \{s'\}$. According to Lemma A.6 we get that $\{s\}\mathrm{E}_n\{s'\}$ for every n, and by definition of E we get that $\{s\}\mathrm{E}\{s'\}$. □

Lemma A.8: There exists a number n such that $\mathrm{E}_n = \mathrm{E}_{n+1} = \cdots = \mathrm{E}$.

This lemma trivially follows from the facts that $\mathrm{E}_0 \supseteq \mathrm{E}_1 \supseteq \mathrm{E}_2 \supseteq \cdots$ and that the number of states in $\mathcal{D}$ and $\mathcal{D}'$ is finite.

Corollary A.9: The equivalence of type I can be defined as the coarsest equivalence relation that satisfies:

$$
\begin{array}{lll}
U\mathrm{E}U' & \Longleftrightarrow & \\
& & \forall U_1 \ \ (U \xrightarrow{\alpha} U_1) \ \Rightarrow \ \exists U_1' \ \ U' \xrightarrow{\alpha} U_1' \wedge U_1\mathrm{E}U_1' \\
& & \forall U_1' \ \ (U' \xrightarrow{\alpha} U_1') \ \Rightarrow \ \exists U_1 \ \ U \xrightarrow{\alpha} U_1 \wedge U_1\mathrm{E}U_1'
\end{array}
$$

Lemma A.10: If $U_0\mathrm{E}U_0'$ and $\Pi_D = U_0, U_1, \ldots$ and $\Pi_D' = U_0', U_1', \ldots$ are paths such that $\Pi_D \approx \Pi_D'$ then $U_i\mathrm{E}U_i'$ for every i.

The proof is done by induction on i, using the fact that a deterministic state can have, at most, one outgoing transition labeled with a given label.

Lemma A.11: If $U_0\mathrm{E}U_0'$ then $U_0 \approx U_0'$.

Proof outline:

We show that for every finite path Π_D from U_0 there exists a path Π_D' from U_0' such that $\Pi_D \approx \Pi_D'$. This is done by iduction on the length of the path, using Lemma A.10. Then, we use könig's lemma to conclude that infinite paths have corresponding paths as well. □

Lemma A.12: Let Π be a path from s in $P:Q$.
If there exist paths Π_D in $\mathcal{D}$, Π_D' in $\mathcal{D}'$, and Π' in $P':Q'$ such that:

- $\Pi_D \approx_e \Pi$.
- $\Pi_D' \approx \Pi_D$.
- $\Pi' \approx_e \Pi_D'$.

then $\Pi \approx_c \Pi'$.

The correctness of this theorem follows directly from the various correspondence definitions.

Theorem A.13: If $\{s\}\mathrm{E}\{s'\}, \mathcal{M}(s) = \mathcal{M}(s')$ and $LC(s) = LC(s')$ then $s \equiv_{P:Q} s'$.

Proof :

We will prove that $s \equiv_{P:Q} s'$ by proving the three necessary conditions.

- $\mathcal{M}(s) = \mathcal{M}(s')$ and $LC(s) = LC(s')$ - given.
- Let Π be a path from s. By Lemma A.2(1) there exists a path Π_D from $\{s\}$ such that $\Pi_D \approx_e \Pi$. $\{s\}\mathrm{E}\{s'\}$ and therefore, according to Lemma A.11, $\{s\} \approx \{s'\}$. According to the definition of the relation $\approx$, there exists a path Π_D' from $\{s'\}$ such that $\Pi_D' \approx \Pi_D$. Now, according to Lemma A.2(2), there exists a path Π' from s in $P:Q$ such that $\Pi' \approx_e \Pi_D'$. From Theorem A.12 we get that $\Pi \approx_c \Pi'$.
- It can be similarly shown that for every path Π' from s' there exists a path Π from s such that $\Pi \approx_c \Pi'$. □

The correctness of Theorem 4.1 is a direct consequence of Theorems A.7 and A.13.

References

[BCG88] M.C. Browne, E.M. Clarke, and O. Grumberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59:115–131, 1988.

[BCG89] M.C. Browne, E.M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite-state processes. *Information and Computations*, 81(1):13 – 31, 1989.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *LICS*, 1990.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677– 691, August 1986.

[BSW69] K.A. Bartlet, R.A. Scantlebury, and P.T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *CACM*, 12(5):260 – 261, May 1969.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. on Programming Languages and Systems*, 8(2):244 – 263, 1986.

[CG87] E.M. Clarke and O. Grumberg. Avoiding the state explosion in temporal logic model checking algorithms. In *Proc. of the 6th Annual Symp. on Principles Of Distributed Computing*, pages 294 – 303. ACM, August 1987.

[CM82] K.M. Chandy and J. Misra. A distributed algorithm for detecting resource deadlocks in distributed systems. In *Symp. on Principles of Distributed Computing*, pages 157 – 164. ACM, August 1982.

[Dij85] E. Dijkstra. Invariance and non-determinancy. In C.A.R. Hoare and J.C. Shepherson, editors, *Mathematical Logic and Programming Languages*, pages 157 – 163. Prentice-Hall, 1985.

[Fra80] N. Francez. Distributed termination. *ACM-TOPLAS 2,1*, pages 42 – 55, 1980.

[GS88] S.M. German and A.P. Sistla. Reasoning about systems with many processes. Technical Report TR 88-378.02, GTE Laboratories Inc., Waltham, MA, March 1988.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* ADDISON-WESLEY PUBLISHING COMPANY, 1979.

[INM84] INMOS Limited. *OCCAM programming manual*, 1984. Prentice-Hall.

[KM89] R. P. Kurshan and K. McMillan. A structural induction theorem for processes. In *8th PODC ACM*, pages 239–247, 1989.

[KS83] P.C. Kanellakis and S.A. Smolka. Ccs expressions, finite-state processes and three problems of equivalence. In *Proceedings of the Second ACM Symposium on Principles of Distributed Computing*, pages 228 – 240, Montreal, Quebec, August 1983.

[LeL77] G. LeLann. Distributed systems - towards a formal approach. In *Information Processing*, pages 155 – 160. North Holland, Amsterdam, 1977.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite states concurrent programs satisfy their linear specification. In *Conf. Record of the 20th Annual ACM Symp. on Principles of Programming Languages*, New Orleans, LA., January 1985.

[Mil79] R. Milner. A calculus of communicating systems. In *Lecture Notes in Computer Science. Vol. 92.* Springer-Verlag, 1979.

[NH84] R. De Nicola and M. Hennesey. Testing equivalences for processes. *Theoretical Computer Science*, 34:83 – 133, 1984.

[QS82] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proc. of the 5th Inter. Symp. on Programming*, 1982. LNCS 137.

[SG89] Z. Shtadler and O. Grumberg. Network grammars, communication behaviors and automatic verification. In *Automatic Verification Methods for Finite State Systems*, pages 151 – 165. Springer-Verlag, 1989.

[SP89] R. Sherman and A. Pnueli. Model checking for linear temporal logic: An efficient implementation. Technical report, USC/ISI, 1989.

[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Workshop on Automatic Verification Methods for Finite State Systems, LNCS 407*, pages 68 – 80. Springer-Verlag, 1989.