Network Grammars, Communication Behaviors and Automatic Verification

Ze'ev Shtadler Orna Grumberg

Computer Science Department
The Technion
32000 Haifa, Israel.
steed@techunix.bitnet orna@techsel.bitnet

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) [L], mutual exclusion (in a ring) [D], distributed termination detection (in any connected network [F], or in a ring [FRS]). 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 program of each of the processes is finite-state. Thus, application of such an algorithm to a specific network is also a finite-state program. Hence, an algorithm of this kind can be viewed as an infinite family of finite-state programs.

Automatic verification of finite-state distributed programs by means of *temporal logic model checking* has been widely investigated [CES], [LP], [QS], [EL], [B], [SC] and successfully used [BCDM], [RRSV], [CBBG], [DC], [MC]. A model checking algorithm determines whether a finite-state distributed program satisfies a temporal logic formula by searching all possible paths in the global state graph induced by the program.

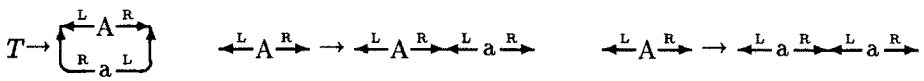
Recently, the problem of automatic verification of infinite families of finite-state programs was studied by several researchers ([CG], [SG]). However, these works dealt only with algorithms designed for a very restricted class of network topologies. More importantly, they didn't take advantage of the fact that every such infinite family of programs originates from one distributed algorithm. We exploit this observation to suggest a formalism for the description of algorithms that is, on the one hand, close to the designer's view of a distributed algorithm and, on the other hand, facilitates automatic reasoning about the described algorithm. We believe that our formalism will help to significantly narrow the gap between algorithm design and algorithm verification.

The main contributions of this paper are:

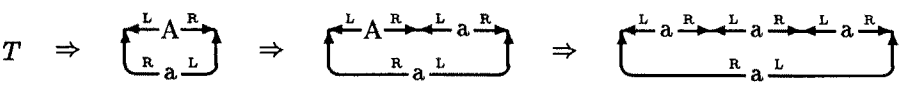
- We suggest a new formalism for the description of algorithms designed for networks with similar topology and different numbers of processes. Our formalism handles network topologies consisting of a fixed number of process types, each of which has a fixed number of communication ports.
- Based on this formalism, we develop a new method for the automatic verification of algorithms described by means of that formalism.

The formalism that we suggest is a context-free *network grammar* that derives networks of processes. With each derived network, a finite-state program is associated. In a network-grammar, the nonterminals represent *subnetwork types* (with possibly unpaired ports). The initial variable T represents the *network type* (with no unpaired ports), and terminals represent (instances of) process types.

Following is an example of a grammar that generates a family of ring networks with any number of processes. For simplicity, the processes are all of the same type a. Both the nonterminal A and the terminal a have ports L and R, representing the left port and the right port, respectively. The right hand side of the production rules specify how ports are paired.



The derivation of a ring of size 4 in the grammar above is:



Another important notion presented here is the *communication behavior* of a subnetwork P, given its environment Q (which is also a subnetwork). The communication behavior of P given Q, denoted by $P{:}Q$, is a new subnetwork, which records all the communications of P, which are in accord with Q, and represents all the others by some dummy communications.

The communication behavior is used as the main tool for reasoning about algorithms defined by means of network grammars. Its definition is motivated by the following observation: when replacing in a network $N = P||Q$ the subnetwork P with another subnetwork P', $P||Q$ and $P'||Q$ an equivalent (to be formally defined later) if P and P' have the same communication behavior given the same environment Q.

Following [CG] we would like to show that all programs derived from a given algorithm, satisfy the same specification, written in a specification language presented below. Showing that, we will be able to prove that an algorithm satisfies some specification in general by considering only one of its small derived programs. For any given algorithm, we employ induction on the derivation of its networks by a network-grammar to prove that all the derived programs are equivalent.

As the specification language we introduce LTL^2 , a two-leveled linear-time temporal logic. Its *basic* formulas are any Indexed Linear Temporal Logic (ILTL) formulas, similar to those defined in [CGB], except that linear-time operators are used instead of branching-time operators. The logic LTL^2 is defined by applying a second level of linear-time operators. A typical LTL^2 formula is:

$$\mathbf{AF}(\bigvee_i \mathbf{AGF}a_i \rightarrow \bigwedge_j \mathbf{AF}b_j)$$

where $\bigvee_i \mathbf{AGF}a_i$ and $\bigwedge_j \mathbf{AF}b_j$ are its basic ILTL components. This formula is satisfied by a program provided that along every computation of the program eventually the following will be true: if for some process i, a_i holds infinitely often along every computation, then for every process j, along every computation, b_j will eventually hold. A model for our logic is a labeled state transition graph or *Kripke structure* that represents the possible global state transitions of some network of finite-state processes.

Our paper is organized as follows: Section 2 describes processes, networks and network grammars. Section 3 presents the logic LTL^2 and its semantics with respect to Kripke structures. Two notions of equivalence are defined over Kripke structures, and are shown to imply each other. Section 4 introduces the communication behavior of a process given its environment, and shows how communication behaviors are used for algorithms verification. Section 5 presents a procedure to determine if an algorithm given by a network grammar satisfies an LTL^2 specification.

2 Processes and Networks

2.1 Combining Processes

Our model of communication is described by means of synchronization communications along channels. The model is similar both to CCS combination [M] and to OCCAM's model of communications [I]. Each process is associated with a set of *communication action names* (referred to also as *ports*). ϵ denotes the non-communication (internal) actions of a process. When two processes P and Q are combined to construct a new process $P||Q$, some of their ports are paired together. A pair of ports represents a channel of communication between P and Q. The channel is used merely for synchronization, i.e. no data is transferred. The unpaired ports of P and Q become the ports of $P||Q$. A process with no unpaired ports is called *a network.*

A process P is a 7-tuple, $P = \langle ACT_P, AP_P, I_P, S_P, R_P, s_{0\,P}, L_P \rangle$, where

- ACT_P is a finite set of actions (ports) that does not contain ϵ.
- AP_P is a finite set of atomic propositions.
- I_P is a finite set of indices.
- S_P is a finite set of states.
- $R_P \subseteq S_P \times (ACT_P \cup \{\epsilon\}) \times S_P$ is a labeled transition relation. We write $s_1 \xrightarrow{a} s_2$ to indicate that $(s_1, a, s_2) \in R_P$.

- $s_{0\ P}$ is the initial state.
- $L_P : S_P \rightarrow 2^{(AP_P \times I_P)}$ is the function that labels each state with a set of indexed atomic propositions. We will write a_i instead of (a, i).

Let P and Q be two processes, with disjoint sets of indices, I_P and I_Q. Let $N_P : ACT_P \rightarrow ACT$ and $N_Q : ACT_Q \rightarrow ACT$ be two 1-1 renaming functions, for some set of actions ACT. We say that two ports, a_P and a_Q, are paired together if $N_P(a_P) = N_Q(a_Q)$. The *combination* of P and Q using N_P and N_Q, is a new process, $P||Q = \langle ACT_{P||Q}, AP_{P||Q}, I_{P||Q}, S_{P||Q}, R_{P||Q}, s_{0\ P||Q}, L_{P||Q} \rangle$, where

$$ACT_{P||Q} = \big(N_P(ACT_P) \cup N_Q(ACT_Q)\big) - \big(N_P(ACT_P) \cap N_Q(ACT_Q)\big)$$
$$AP_{P||Q} = AP_P \cup AP_Q$$
$$I_{P||Q} = I_P \cup I_Q$$
$$S_{P||Q} = (S_P \times S_Q)$$
$$L_{P||Q}\big((s_P, s_Q)\big) = L_P(s_P) \cup L_Q(s_Q)$$
$$s_{0\ P||Q} = (s_{0\ P}, s_{0\ Q})$$

$$\begin{aligned}
R_{P||Q} = \\
&\Big\{\big((s_P, s_Q), \epsilon, (s'_P, s'_Q)\big) \mid \exists\, a_P, a_Q : (s_P, a_P, s'_P) \in R_P, (s_Q, a_Q, s'_Q) \in R_Q \\
&\qquad \text{and } N_P(a_P) = N_Q(a_Q)\Big\} \cup \\
&\Big\{\big((s_P, s_Q), \epsilon, (s'_P, s_Q)\big) \mid (s_P, \epsilon, s'_P) \in R_P\Big\} \cup \\
&\Big\{\big((s_P, s_Q), \epsilon, (s_P, s'_Q)\big) \mid (s_Q, \epsilon, s'_Q) \in R_Q\Big\} \cup \\
&\Big\{\big((s_P, s_Q), N_P(a_P), (s'_P, s_Q)\big) \mid N_P(a_P) \notin N_Q(ACT_Q) \text{ and } (s_P, a_P, s'_P) \in R_P\Big\} \cup \\
&\Big\{\big((s_P, s_Q), N_Q(a_Q), (s_P, s'_Q)\big) \mid N_Q(a_Q) \notin N_P(ACT_P) \text{ and } (s_Q, a_Q, s'_Q) \in R_Q\Big\}
\end{aligned}$$

Sometimes it will be useful to use a renaming function that leaves the names of the ports unchanged. We then use the identity renaming function, denoted by $\mathcal{I}$.

The empty process $\emptyset = \langle ACT_\emptyset, AP_\emptyset, I_\emptyset, S_\emptyset, R_\emptyset, s_{0\ \emptyset}, L_\emptyset \rangle$, is the process for which $ACT_\emptyset = AP_\emptyset = I_\emptyset = R_\emptyset = \emptyset$, $S_\emptyset = \{s_{0\ \emptyset}\}$ and $L_\emptyset(s_{0\ \emptyset}) = \emptyset$.

We say that two processes, P and Q are *isomorphic* if $ACT_P = ACT_Q$, $AP_P = AP_Q$, $I_P = I_Q$ and there is an isomorphism function $h : S_P \rightarrow S_Q$ such that $h(s_{0\ P}) = (s_{0\ Q})$, $(s_P, a_P, s'_P) \in R_P \iff (h(s_P), a_P, h(s'_P)) \in R_Q$ and for every state s_P, $L_P(s_P) = L_Q(h(s_P))$.

Lemma 1 *For every process P, $P||\emptyset$ with $N_P = \mathcal{I}$ is isomorphic to P.*

2.2 Associativity of Combination

Combination of processes is commutative. i.e., $P||Q$ is isomorphic to $Q||P$. However, combination is not associative in the usual way. When three processes P, Q and R are combined to a network $(P||Q)||R$, four renaming functions are needed. First N_P and N_Q are used to construct $P||Q$. Then $N_{P||Q}$ and N_R are used to combine $(P||Q)||R$. Similarly, to combine $P||(Q||R)$, the renaming functions $N'_Q, N'_R, N'_{Q||R}$ and N'_P are needed. The following theorem implies that combinations of a network may be defined such that associativity will hold up to isomorphism.

Theorem 1 (Associativity of Combination) *Given a network $(P||Q)||R$ combined using $N_P, N_Q, N_{P||Q}$, and N_R, there exist N'_P, N'_Q and N'_R such that the following networks are isomorphic:*

1. *The given network $(P||Q)||R$.*
2. *$(P||Q)||R$ combined using N'_P, N'_Q, N'_R, and $N'_{P||Q} = \mathcal{I}$.*
3. *$P||(Q||R)$ combined using N'_P, N'_Q, N'_R, and $N'_{Q||R} = \mathcal{I}$.*

Proof Let the new renaming functions be defined as follows: $N'_P = N_P$, $N'_Q = N_Q$, and $N'_R(a) = N^{-1}_{P||Q}(N_R(a))$ for every action $a \in ACT_R$.

First let us show that exactly the same ports are paired together by the new renaming functions as were paired by the given ones. Since $N'_{P||Q} = N'_{Q||R} = \mathcal{I}$, we need only to consider the following:

- $N_P(a_P) = N_Q(a_Q) \iff N'_P(a_P) = N'_Q(a_Q)$
- $N_{P||Q}(N_Q(a_Q)) = N_R(a_R) \iff N_Q(a_Q) = N^{-1}_{P||Q}(N_R(a_R)) \iff N'_Q(a_Q) = N'_R(a_R)$
- $N_{P||Q}(N_P(a_P)) = N_R(a_R) \iff N_P(a_P) = N^{-1}_{P||Q}(N_R(a_R)) \iff N'_P(a_P) = N'_R(a_R)$

This directly implies the isomorphism of the first and second networks. Moreover, the above implies that N'_P, N'_Q and N'_R are sufficient to determine the pairing of ports regardless of the order in which the processes P, Q and R are combined.

Now we define a mapping function from states of $(P||Q)||R$ to states of $P||(Q||R)$ and show that the two networks are isomorphic with respect to that function. Let the isomorphism function $h : S_{(P||Q)||R} \rightarrow S_{P||(Q||R)}$ be:

$$h\left(((s_P, s_Q), s_R)\right) = (s_P, (s_Q, s_R))$$

To show that $P||(Q||R)$ and $(P||Q)||R$ are isomorphic, we consider only the requirements on the transition relations, as all others are immediate. Since there are no unpaired ports in either network, all the transitions are labeled by ϵ, and $R_{(P||Q)||R}$ contains six kinds of transitions: Internal transitions of either P, Q or R, and communication transitions involving either P and Q, P and R, or Q and R. For similar reasons, $R_{P||(Q||R)}$ contains the same six kinds of transitions. By looking at each of the possible kinds of transitions, we can see that

$$\begin{aligned}
&\left(((s_P, s_Q), s_R), \epsilon, ((s'_P, s'_Q), s'_R)\right) \in R_{(P||Q)||R} \iff \\
&\left((s_P, (s_Q, s_R)), \epsilon, (s'_P, (s'_Q, s'_R))\right) \in R_{P||(Q||R)} \iff \\
&\left(h\left(((s_P, s_Q), s_R)\right), \epsilon, h\left(((s'_P, s'_Q), s'_R)\right)\right) \in R_{P||(Q||R)}
\end{aligned}$$

□

2.3 Network Grammars

The formalism we suggest for the description of distributed finite-state algorithms is a *Network Grammar*, $G = (\Sigma, \Delta, \mathcal{P}, T)$. $\Sigma^\star$ is the set of networks combined of instances of process in Σ. The grammar defines a subset of networks combined using the production rules.

- The terminals, Σ, are *process types*, which are processes with a singleton set of indices.
- The nonterminals, Δ, are *subnetwork types*, each of which represents an unspecified process with a specified set of action names. For a subnetwork type A, this set is denoted by ACT_A.
- Subnetwork types B and C may be combined together using renaming functions N_B and N_C, defined over ACT_B and ACT_C, respectively. Similar to process combination, the renaming functions specify those action names (ports) in B and C that are paired together. $ACT_{B||C}$, the set of actions names of $B||C$, contains all the unpaired action names of B and C *after renaming*.
- $\mathcal{P}$ is the set of production rules. A production rule in a network grammar specifies how subnetwork types and process types should be combined. For $A \in \Delta$, $B, C \in (\Sigma \cup \Delta)$, $(A, (B, N_B), (C, N_C))$, abbreviated to $A \to B||C$, describes a derivation in which A is replaces by $B||C$. It is required that $ACT_A = ACT_{B||C}$. For simplicity we assume that each production rule is a binary rule of the form $A \to B||C$. A unary production rule may be obtained by replacing C with the empty process. It is not allowed for both B and C to be $\emptyset$.
- The start symbol T represents a subnetwork type with an empty set of action names.

Each instance of a process type has a unique index value. Variables of G, other than T, can derive processes with unpaired ports.
We consider only grammars in which every symbol is reachable, and derives at least one subnetwork of terminals.

We used our formalism to define algorithms for a family of networks with nontrivial topologies.

3 The logic LTL^2

LTL^2 is a two-leveled linear time temporal logic. It is defined by means of two types of formulas: local and global. Let AP be a set of atomic propositions, which are indexed by a set of index variables IV. A local formula is either:

- a_i, if $a \in AP$ and $i \in IV$.
- $\neg f$, $f \vee g$ and $f\mathbf{U}g$, if f and g are local formulas.

A global formula is either:

- $\bigvee_i \mathbf{A}\, f(i)$ and $\bigvee_i \mathbf{E}\, f(i)$, if $f(i)$ is a local formula that has exactly one free index variable i. We will refer to this type of global formulas as *basic formulas*. Basic formulas have no free index variables.
- $\neg f$, $f \vee g$ and $f\mathbf{U}g$, if f and g are global formulas.

An LTL^2 formula is either $\mathbf{A}\, f$ or $\mathbf{E}\, f$, where f is a global formula.

We define the semantics of LTL^2 with respect to a Kripke structure

$$K = \langle AP, I, S, R, s_0, L \rangle$$

where

- AP is the set of atomic propositions.
- I is the set of index values.
- S is the set of states.
- $R \subseteq S \times S$ is the transition relation, which must be total in both of its arguments. We write $s_1 \rightarrow s_2$ to indicate that $(s_1, s_2) \in R$.
- s_0 is the initial state.
- $L : S \rightarrow 2^{(AP \times I)}$ is the indexed proposition labeling.

We define a *path in* K to be a sequence of states, $\pi = s_1, s_2, \ldots$ such that for every $i \geq 1, s_i \rightarrow s_{i+1}$. π^i will denote the suffix of π starting in s_i, and $first(\pi)$ will denote the first state in π. We only consider infinite paths in K.

We distinguish between two types of formulas. Formulas of the form $\neg f$, $f \vee g$ or $f\mathbf{U}g$ are path formulas. Formulas of the form a_i, $\mathbf{A}f$, $\mathbf{E}f$ or $\bigvee_i f$ 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 path π (in a state s) for a subset $J \neq \emptyset$ of the index values in the structure K. The relation $\models$ is defined inductively as follows:

$$
\begin{aligned}
K, s, \{i\} &\models a_i && \Leftrightarrow && a_i \in L(s).\\
K, \pi, \{i\} &\models a_i && \Leftrightarrow && K, \mathit{first}(\pi), \{i\} \models a_i\\
K, \pi, J &\models \neg f && \Leftrightarrow && K, \pi, J \not\models f.\\
K, \pi, J &\models f \vee g && \Leftrightarrow && K, \pi, J \models f \text{ or } K, \pi, J \models g.\\
K, \pi, J &\models f\mathbf{U}g && \Leftrightarrow && \text{there exists a } l \geq 1 \text{ such that } K, \pi^l, J \models g\\
& && && \text{and for all } 1 \leq j < l \;\; K, \pi^j, J \models f.\\
K, \pi, J &\models \bigvee_i f && \Leftrightarrow && K, \mathit{first}(\pi), J \models \bigvee_i f.\\
K, s, J &\models \bigvee_i f && \Leftrightarrow && \text{there exists an } i_0 \in J \text{ such that } K, s, \{i_0\} \models f.\\
K, s, J &\models \mathbf{A}f && \Leftrightarrow && \text{for every path } \pi \text{ starting in } s \;\; K, \pi, J \models f.\\
K, s, J &\models \mathbf{E}f && \Leftrightarrow && \text{there exists a path } \pi \text{ starting in } s \text{ such that } \;\; K, \pi, J \models f.
\end{aligned}
$$

We will use the following abbreviations:

$\bigwedge_i \mathbf{A}f = \neg \bigvee_i \mathbf{E}\neg f$, $\bigwedge_i \mathbf{E}f = \neg \bigvee_i \mathbf{A}\neg f$, $\mathbf{F}f = \mathit{true}\,\mathbf{U}f$ and $\mathbf{G}f = \neg\mathbf{F}\neg f$

For a formula $f \in \mathrm{LTL}^2$, we define $BF(f)$ as the set of basic formulas in f . Let $\mathcal{F}$ be some set of basic formulas over a set $AP_{\mathcal{F}}$ of atomic propositions. Given two structures K, K', such that $AP_K = AP_{K'} \supseteq AP_{\mathcal{F}}$ 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 $\mathcal{F}$.

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

$$\forall f\,[BF(f) \subseteq \mathcal{F}] : K, s_{0\,K}, J_K \models f \iff K', s_{0\,K'}, J_{K'} \models f$$

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

1. For every path π starting in $s_{0\,K}$ there is a path π' starting in $s_{0\,K'}$, 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, and 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 $s_{0\,K'}$ there is a path π starting in $s_{0\,K}$ that satisfies the same conditions as above.

Theorem 2 $(K, J_K) \equiv_{\mathcal{F}} (K', J_{K'})$ *if and only if* $(K, J_K)\ \mathbf{C}_{\mathcal{F}}\ (K', J_{K'})$.

Proof First we note that once we refer to the basic formulas as atomic propositions, LTL^2 is reduced to a variant of LTL without the next-time operator. This variant also includes a *single* top level path quantifier which is either **A** ("for all paths") or **E** ("there exists a path").

We show that both the specification equivalence and the computation equivalence correspond to LTL-equivalence of structures labeled by the basic formulas in $\mathcal{F}$.

We define a function $h_{\mathcal{F}}$ that transforms a pair (K, J_K) to a structure (without an index set) labeled by $\mathcal{F}$.

$$h_{\mathcal{F}}(K, J_K) = \langle \mathcal{F}, S, R, s_0, L_{\mathcal{F}} \rangle$$

where $L_{\mathcal{F}}(s) = \{f | f \in \mathcal{F} \text{ and } K, s, J_K \models f\}$.
According to the semantics of LTL^2 and LTL:

$$\forall f\,[BF(f) \subseteq \mathcal{F}] : K, s, J_K \models_{\text{LTL}^2} f \iff h_{\mathcal{F}}(K, J_K), s \models_{\text{LTL}} f$$

$$(K, J_K) \equiv_{\mathcal{F}} (K', J_{K'}) \text{ if and only if } h_{\mathcal{F}}(K, J_K) \equiv_{\text{LTL}} h_{\mathcal{F}}(K', J'_K)$$

On the other hand, using a stuttering equivalence relation for LTL derived from the equivalence for PTL suggested in [SG], $h_{\mathcal{F}}(K, J_K) \equiv_{\text{LTL}} h_{\mathcal{F}}(K', J'_K)$ if and only if

1. For every path π starting in $s_{0\ K}$ there is a path π' starting in $s_{0\ K'}$, 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, and for every state s in B_j, every state s' in B'_j, $L_{\mathcal{F}}(s) = L'_{\mathcal{F}}(s')$.

2. For every path π' starting in $s_{0\ K'}$ there is a path π starting in $s_{0\ K}$ that satisfies the same conditions as above.

Therefore $(K, J_K) \equiv_{\mathcal{F}} (K', J_{K'})$ if and only if $(K, J_K)\ \mathbf{C}_{\mathcal{F}}\ (K', J_{K'})$. □

It will sometime be convenient to refer to a network in a context which requires a structure instead. When this happens, the required structure is the one obtained from the network by omitting the transition labels (all of which are ϵ). Since Kripke structures have total transition relations, so must the networks.

In the remainder of the paper we refer only to *finite* Kripke structures.

4 The Communication Behavior

The *communication behavior of a process P given an environment Q*, is denoted by $P{:}Q$. Similarly to $P||Q$, it is defined by means of P, Q, N_P and N_Q, but only in case the combination of $P||Q$, using N_P and N_Q, results in a network (with no unpaired ports). The main differences between $P||Q$ and $P{:}Q$ are:

- Transitions of $P{:}Q$ are labeled by $\epsilon, \overline{\epsilon}$ or actions in ACT_P.
- A state s in $P{:}Q$ is labeled by *basic formulas*, rather than atomic propositions.
- $P{:}Q$ has two special states *sink* and $\overline{sink}$. All communications on channels between P and Q in which P is willing to participate while Q is not, are directed to *sink*. Similarly, communications in which Q is willing to participate but P is not, are directed to $\overline{sink}$. This records the mutual influence of the environment Q and the process P by means of communication enableness.

Given P, Q, N_P and N_Q such that $N_P(ACT_P) = N_Q(ACT_Q)$, the communication behavior $P{:}Q$ is formally defined as follows:

$$P{:}Q = \left\langle ACT_{P:Q}, BF_{P:Q}, S_{P:Q}, R_{P:Q}, s_{0\ P:Q}, M_{P:Q} \right\rangle$$

where

- $ACT_{P:Q} = ACT_P$
- $BF_{P:Q}$ is a set of basic formulas, defined over AP_P, the set of atomic propositions of P.
- $S_{P:Q} = \{sink, \overline{sink}\} \cup (S_P \times S_Q)$ where $sink$ and $\overline{sink}$ are special states.
- $M_{P:Q}\left((s_P, s_Q)\right) = \{f | f \in BF_{P:Q} \text{ and } P||Q, (s_P, s_Q), I_P \models f\}$
- $M_{P:Q}(sink) = issink$
- $M_{P:Q}(\overline{sink}) = \overline{issink}$
- $s_{0\ P:Q} = (s_{0\ P}, s_{0\ Q})$

In order to define $R_{P:Q}$ we need the following definitions:

- A path containing only ϵ transitions of Q is called an *ϵ-path* in Q.
- *ϵ-closure*(s) is the set of states that are reachable from s by an *ϵ-path*.

$$
\begin{aligned}
&R_{P:Q} = \\
&\Big\{\big((s_P, s_Q), a_P, (s'_P, s'_Q)\big) \quad | \quad (s_P, a_P, s'_P) \in R_P \text{ and } (s_Q, a_Q, s'_Q) \in R_Q, \\
&\qquad\qquad\qquad\qquad \text{where } N_P(a_P) = N_Q(a_Q)\Big\} \cup \\
&\Big\{\big((s_P, s_Q), \epsilon, (s'_P, s_Q)\big) \quad | \quad (s_P, \epsilon, s'_P) \in R_P\Big\} \cup \\
&\Big\{\big((s_P, s_Q), \overline{\epsilon}, (s_P, s'_Q)\big) \quad | \quad (s_Q, \epsilon, s'_Q) \in R_Q\Big\} \cup \\
&\Big\{\big((s_P, s_Q), a_P, sink\big) \quad | \quad \exists (s_P, a_P, s'_P) \in R_P : a_P \neq \epsilon \text{ and} \\
&\qquad\qquad\qquad\qquad \forall s'_Q \in \epsilon\text{-}closure(s_Q)\, \forall (s'_Q, a_Q, s''_Q) \in R_Q : \\
&\qquad\qquad\qquad\qquad N_P(a_P) \neq N_Q(a_Q)\Big\} \cup \\
&\Big\{\big((s_P, s_Q), a_P, \overline{sink}\big) \quad | \quad \exists (s_Q, a_Q, s'_Q) \in R_Q : N_P(a_P) = N_Q(a_Q) \text{ and} \\
&\qquad\qquad\qquad\qquad \forall (s'_P, a_P, s''_P) \in R_P : s'_P \notin \epsilon\text{-}closure(s_P)\Big\} \cup \\
&\Big\{\big(sink, \epsilon, sink\big)\Big\} \cup \\
&\Big\{\big(\overline{sink}, \overline{\epsilon}, \overline{sink}\big)\Big\}
\end{aligned}
$$

Explanation: $R_{P:Q}$ is a union of 7 sets of transitions. The first one records the transitions in which P and Q cooperate. 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 the states in its *ϵ-closure*. The fifth set records the dual case, in which Q enables the communication while P disables it. The transition in both cases is labeled by an action in ACT_P. The sixth and seventh sets contain self loops for $sink$ and $\overline{sink}$.

Lemma 2 $\big((s_P, s_Q), \alpha, (s'_P, s'_Q)\big) \in R_{P:Q}$ *if and only if* $\big((s_P, s_Q), \epsilon, (s'_P, s'_Q)\big) \in R_{P||Q}$. *i.e., every transition in* $P{:}Q$ *that does not result in either* sink *or* $\overline{\text{sink}}$ *has a corresponding transition in* $P||Q$, *and vice versa.*

Proof

- By the definition of $R_{P:Q}$, a transition, $\big((s_P, s_Q), \alpha, (s'_P, s'_Q)\big) \in R_{P:Q}$ if and only if one of the following conditions holds:
  1. $(s_P, a_P, s'_P) \in R_P$ and $(s_Q, a_Q, s'_Q) \in R_Q$, where $N_P(a_P) = N_Q(a_Q)$
  2. $(s_P, \epsilon, s'_P) \in R_P$ and $s_Q = s'_Q$
  3. $(s_Q, \epsilon, s'_Q) \in R_Q$ and $s_P = s'_P$
- Because $P||Q$ is a network, and by the definition of $R_{P||Q}$, $\big((s_P, s_Q), \epsilon, (s'_P, s'_Q)\big) \in R_{P||Q}$ if and only if one of the above conditions holds.
- Therefore, $\big((s_P, s_Q), \alpha, (s'_P, s'_Q)\big) \in R_{P:Q}$ if and only if $\big((s_P, s_Q), \epsilon, (s'_P, s'_Q)\big) \in R_{P||Q}$.

□

We define $\pi = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \rightarrow \cdots$ to be a *path in* $P{:}Q$ if $\forall i \geq 0\, (s_i, \alpha_{i+1}, s_{i+1}) \in R_{P:Q}$

Corollary 1 *For every path in* $P{:}Q$ *from the initial state,* $s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \rightarrow \cdots$, *with* $\alpha_i \in ACT_P \cup \{\epsilon, \overline{\epsilon}\}$, *that does not contain* sink *or* $\overline{\text{sink}}$, $s_0, s_1, s_2, \cdots$ *is a path in* $P||Q$.

Corollary 2 *For every path in* $P||Q$ *from the initial state,* $s_0, s_1, s_2, \cdots$ *there exists a sequence of actions* $\alpha_1, \alpha_2, \alpha_3, \ldots$ *where* $\alpha_i \in ACT_P \cup \{\epsilon, \overline{\epsilon}\}$ *such that* $s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \rightarrow \cdots$ *is a path in* $P{:}Q$.

4.1 Equivalence of Communication Behaviors

Two communication behaviors $P{:}Q$ and $P'{:}Q'$ are equivalent ($P{:}Q \equiv P'{:}Q'$) if

1. $ACT_{P:Q} = ACT_{P':Q'}$
2. $BF_{P:Q} = BF_{P':Q'}$
3. For every path π starting in $s_{0\ P:Q}$ there is a path π' starting in $s_{0\ P':Q'}$ and partition of both paths $B_1, B_2, \ldots,\ B'_1, B'_2, \ldots$ such that for every j:
   (a) B_j and B'_j are both finite, non-empty and contain only ϵ and $\overline{\epsilon}$ transitions.
   (b) For every state s in B_j and every state s' in B'_j $\quad M_{P:Q}(s) = M_{P':Q'}(s')$.
   (c) If $\text{last}(B_j) \xrightarrow{\alpha_j} \text{first}(B_{j+1})$ then $\text{last}(B'_j) \xrightarrow{\alpha_j} \text{first}(B'_{j+1})$.
4. For every path π' starting in $s_{0\ P':Q'}$ there is a path π starting in $s_{0\ P:Q}$ that satisfies the same conditions as in 3.

Lemma 3 *If* $P{:}Q \equiv P'{:}Q'$ *then* $(P||Q, I_P) \equiv_{BF_{P:Q}} (P'||Q', I_{P'})$

Proof First we show that if $P{:}Q \equiv P'{:}Q'$ then $(P||Q, I_P)\ \mathbf{C}_{BF_{P:Q}}\ (P'||Q', I_{P'})$.

- $P{:}Q \equiv P'{:}Q'$ therefore $BF_{P:Q} = BF_{P':Q'}$.
- The initial states of $P{:}Q$ and $P||Q$ are both $(s_{0\ P}, s_{0\ Q})$, and thus are equal. Similarly, the initial states of $P'{:}Q'$ and $P'||Q'$ are equal, too.
- Let $\pi = s_0, s_1, s_2, \cdots$ be a path in $P||Q$ starting in the initial state $s_{0\ P||Q}$.

  By corollary 1 there is a path $s_{0\ P:Q} \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \rightarrow \cdots$ in $P{:}Q$.

  By the equivalence of $P{:}Q$ and $P'{:}Q'$ there is a path $s_{0\ P':Q'} \xrightarrow{\beta_1} s'_1 \xrightarrow{\beta_2} s'_2 \rightarrow \cdots$ in $P'{:}Q'$, and a partitioning of both paths that satisfy conditions 3 of the equivalence definition above.

  By corollary 2, $\pi' = s_{0\ P'||Q'}, s'_1, s'_2, \cdots$ is a path in $P'||Q'$. The partition $B_1, B_2, \ldots$ and $B'_1, B'_2, \ldots$, applied to π and π' respectively, satisfy the conditions of computation equivalence of Kripke structures with respect to $BF_{P:Q}$.
- Similarly, for every a path in $P'||Q'$, starting in the initial state, there exists a path in $P||Q$ that satisfies the conditions of computation equivalence of Kripke structures with respect to $BF_{P:Q}$.

Thus, $(P||Q, I_P)\ \mathbf{C}_{BF_{P:Q}}\ (P'||Q', I_{P'})$, and by Theorem 2, $(P||Q, I_P) \equiv_{BF_{P:Q}} (P'||Q', I_{P'})$. □

Theorem 3 (3-Way Orthogonality) *Let all the communication behaviors be defined over a fixed set of basic formulas. If* $P'{:}(Q||R) \equiv P{:}(Q||R)$, $Q'{:}(P||R) \equiv Q{:}(P||R)$ *and* $R'{:}(P||Q) \equiv R{:}(P||Q)$ *then* $P'{:}(Q'||R') \equiv P{:}(Q||R)$.

Essentially, the theorem says that it is sufficient to check the replacement of each component of a communication behavior separately to conclude that replacing all components at the same time results in an equivalent communication behavior . Due to lack of space, the proof is omitted.

Corollary 3 *If* $P'{:}Q \equiv P{:}Q$ *and* $Q'{:}P \equiv Q{:}P$ *then* $P'{:}Q' \equiv P{:}Q$.

Corollary 4 *If* $P'{:}Q \equiv P{:}Q$ *then* $Q{:}P' \equiv Q{:}P$.

4.2 Verification of Algorithms

Verification of an algorithm means to verify all the programs it describes. The purpose of this section is to establish a theoretical basis for automatic verification of algorithms defined by network grammars. Let $G = (\Sigma, \Delta, \mathcal{P}, T)$ be a description of an algorithm, and let SPEC $\in$ LTL^2 be a specification to be verified for that algorithm. With every terminal and nonterminal symbol $W \in \Sigma \cup \Delta$ we associate two *representative* processes $w, \bar{w} \in \Sigma^\star$. w is the representative of the subnetworks derived from W by the grammar, while $\bar{w}$ is the representative of the possible environments for these subnetworks. For terminal symbols $W \in \Sigma$, w is (an instance of) process type W. The environment representative, $\bar{t}$, of the start symbol T is the empty process $\emptyset$.

From now on, we only consider communication behaviors labeled by BF(SPEC).

Lemma 4 *If for every production rule* $A \rightarrow B||C$,

$$\forall b', c' \in \Sigma^{\star} \left[\textit{if } b':\bar{b} \equiv b:\bar{b} \wedge c':\bar{c} \equiv c:\bar{c} \textit{ then } a:\bar{a} \equiv (b'||c'):\bar{a} \right]$$

then for every $W \in \Sigma \cup \Delta$, *and every* $w' \in \Sigma^{\star}$ *such that* $W \Rightarrow^{\star} w'$, $w':\bar{w} \equiv w:\bar{w}$.

The proof is by induction on the number of productions in the derivation of w'.

Lemma 5 *For every production rule* $A \rightarrow B||C$, *if* $a:\bar{a} \equiv (b||c):\bar{a}$, $\bar{b}:b \equiv (\bar{a}||c):b$ *and* $\bar{c}:c \equiv (\bar{a}||b):c$, *then* $\forall b', c' \in \Sigma^{\star} \left[\textit{if } b':\bar{b} \equiv b:\bar{b} \wedge c':\bar{c} \equiv c:\bar{c} \textit{ then } a:\bar{a} \equiv (b'||c'):\bar{a} \right]$

The proof is based on theorem 3.

Theorem 4 (Consistent Grammars) *If for every production rule* $A \rightarrow B||C$, $a:\bar{a} \equiv (b||c):\bar{a}$, $\bar{b}:b \equiv (\bar{a}||c):b$ *and* $\bar{c}:c \equiv (\bar{a}||b):c$, *then*

$$(t, s_{0\ t}, I_t \models \mathit{SPEC}) \Leftrightarrow \forall t' \in \Sigma^{\star} [T \Rightarrow^{\star} t'] \ (t', s_{0\ t'}, I_{t'} \models \mathit{SPEC})$$

The proof is based on lemma 4, lemma 5 and the definition of specification equivalence.

A network grammar is *consistent* if is satisfies the conditions of theorem 4. Once the consistency of a grammar is established, the representative t, of the start symbol T, satisfies SPEC if and only if the algorithm does.

5 The Verification Procedure

Given a description of an algorithm by means of a network grammer $G = (\Sigma, \Delta, \mathcal{P}, T)$, and a specification SPEC, the verification procedure consists of the following steps:

1. Find BF(SPEC), the set of basic formulas in SPEC.
2. For every $W \in \Sigma \cup \Delta$ choose a representative $w \in \Sigma^{\star}$:
   (a) For every terminal, its representative is the process type associated with it.
   (b) For any symbol A that does not have a representative, if there is a production rule $A \rightarrow B||C$, for which both B and C have representatives, then $a = b||c$ is the representative of A.
   (c) Repeat step 2b until every symbol has a representative.
3. For $W \in \Sigma \cup \Delta$ choose an environment representative $\bar{w} \in \Sigma^{\star}$:
   (a) For the start symbol T, the representative is the empty process $\emptyset$.
   (b) For any symbol B that does not have an environment representative, if there is a production rule $A \rightarrow B||C$ (or $A \rightarrow C||B$), for which A has an environment representative $\bar{a}$, then $\bar{b} = \bar{a}||c$ is the environment representative of B.
   (c) Repeat step 3b until every symbol has an environment representative.

4. For every production rule $A \rightarrow B||C$ construct the communication behavior of $a{:}\bar{a}, \bar{b}{:}b, \bar{c}{:}c, (b||c){:}\bar{a}, (\bar{a}||c){:}b, (\bar{a}||b){:}c$ labeled by basic formulas in BF(SPEC). The labeling is determined using an LTL model checker.

5. If G is consistent then apply an LTL model checker to $t{:}\emptyset$. The algorithm satisfies SPEC if and only if $t{:}\emptyset$ satisfies SPEC.

6. If G is not consistent, out method is incompetent with the verification of the grammar.

Acknowledgements

We thank Nissim Francez and Amir Pnueli for helpful discussions and hints.

The second author, O. Grumberg, was partly supported by the V.P.R. fund – Loewengart Research Fund.

References

[B] M.C. Browne, "An Improved Algorithm for the Automatic Verification of Finite State Systems using Temporal Logic", *Proc. of the 1986 Conference on Logic in Computer Science,* Cambridge, MA, pp. 260-267, June 1986.

[BCDM] M.C. Browne, E.M. Clarke, D. Dill, and B. Mishra, "Automatic Verification of Sequential Circuits using Temporal Logic", *IEEE Trans. on Computers,* C-35(12), December 1986.

[CBBG] E.M. Clarke, S. Bose, M.C. Browne, and O. Grumberg, "The design and verification of finite-state hardware controllers", conference on VLSI, Taiwan, 1987.

[CG] E.M. Clarke, and O. Grumberg, "Avoiding the State Explosion in Temporal Logic Model Checking Algorithms", *Proc. of the 6th Annual Symposium on Principles of Distributed Computing, ACM,* pp. 294-303, August 1987.

[CGB] E.M. Clarke, O. Grumberg, and M.C. Browne, "Reasoning about Networks with Many Identical Finite-State Processes", *Information and Computations,* 81(1), pp. 13-31, 1989.

[CES] 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), pp. 244-263, 1986.

[D] E. Dijkstra, *Invariance and Non-Determinacy,* (C.A.R. Hoare, and J.C. Shepherson, eds.), *Mathematical Logic and Programming Languages,* pp. 157-163, Prentice-Hall, 1985.

[DC] D. Dill, and E.M. Clarke, "Automatic Verification of Asynchronous Circuits using Temporal Logic", *IEEE Proc.,* 133, pt. E(5), September 1986.

[EL] E.A. Emerson, and C.L. Lei, "Modalities for Model Checking: Branching Time Strikes Back," *Conf. Record of the 20th Annual ACM Symp. on Principles of Programming Languages,* New Orleans, La., January 1985.

[F] N. Francez, "Distributed Termination", *ACM-TOPLAS* 2, 1, pp. 42-55, 1980.

[FRS] N. Francez, M. Rodeh, and M. Sintzoff, "Distributed Termination with Interval Assertions", *Proc. of the Inf. Col. on Formalization of Programming Concepts,* Peninscula, Spain, LNCS 107, Springer-Verlag 1981.

[I] INMOS Limited, *OCCAM Programming Manual,* Prentice-Hall, 1984.

[L] G. LeLann, "Distributed Systems - Towards a Formal Approach", *Information Processing,* 77, pp. 155-160, North-Holland, Amsterdam.

[LP] O. Lichtenstein, and A. Pnueli, "Checking that Finite State Concurrent Programs Satisfy their Linear Specification", *Conf. Record of the 20th Annual ACM Symp. on Principles of Programming Languages,* New Orleans, La., January 1985.

[M] R. Milner, *A Calculus of Communicating Systems*, LNCS 92, Springer-Verlag 1980.

[MC] B. Mishra, and E.M. Clarke, "Hierarchical Verification of Asynchronous Circuits using Temporal Logic", *Theoretical Computer Science,* 38, pp. 269-291, 1985.

[QS] J.P. Quielle, and J. Sifakis, "Specification and Verification of Concurrent Systmes in CESAR", *Proc. of the 5th Inter. Symp. on Programming,* 1981.

[SC] A.P. Sistla, and E.M. Clarke, "Complexity of Propositional Temporal Logics", *J. of the Association for Computing Machinery,* 32(3), pp. 733-749, July 1986.

[SG] A.P. Sistla, and S. German, "Reasoning with Many Processes", *Proc. of the Symp. on Logic in Computer Science,* Ithaca, N.Y., June 1987.

[RRSV] J.L. Richier, C. Rodriguez, J. Sifakis, and J. Voiron, "XESAR - User's Guide", Technical Report, IMAG, Institut National Polytechnique de Grenoble, September 1987.