A Model and Temporal Proof System for Networks of Processes*

Van Nguyen
Alan Demers
David Gries
Susan Owicki

June 1985

TR 85-690

Department of Computer Science
Cornell University
Ithaca, NY 14853

* To appear in the first issue of the new journal Distributed Computing in fall 1985.

A Model and Temporal Proof System for Networks of Processes*

Van Nguyen[1], Alan Demers[1], David Gries[1] and Susan Owicki[2]

* This work was supported by the NSF under grants MCS-81-03605, DCR-83-202-74, and DCR-83-123-19; by NASA under contract NAGW419; and by the third author's Guggenheim Fellowship.
[1] Computer Science Department, Cornell University, Ithaca, NY, 14853, USA.
This paper is based on part of the first author's Ph.D. thesis.
[2] Computer Systems Laboratory, Stanford University, Stanford, Ca. 94305, USA.

Summary.

An approach is presented for modeling networks of processes that communicate exclusively through message passing. A process (or a network) is defined by its set of possible behaviors, where each behavior is an abstraction of an infinite execution sequence of the process. The resulting model is simple and modular and facilitates information hiding. It can describe both synchronous and asynchronous networks. It supports recursively-defined networks and can characterize liveness properties such as progress of inputs and outputs, termination, and deadlock.

A sound and complete temporal proof system based on the model is presented. It is compositional —a specification of a network is formed naturally from specifications of its components.

Table of Contents

1. Introduction

A number of models exist for networks of processes [3], [4], [6], [7], [8], [15], [22]. None of these models handles both synchronous and asynchronous communication in a single framework. In addition, the modeling of liveness properties is generally unsatisfactory. The models that seem most promising, due to their simplicity and information-hiding ability, are those based on traces. A *trace*, an abstraction of a process state in which irrelevant internal details have been hidden, is a finite sequence of events that have occurred on the input-output ports of a process during some execution. A trace represents the state reached by the process after some computation in which the events of the trace occur.

Liveness properties, such as progress and termination, are difficult to specify in trace-based models. Some liveness properties deal with complete, possibly infinite, execution sequences, while traces specify only finite prefixes of execution sequences. For example, a property like "eventually a message is sent on port *k*" may fail to hold of a particular infinite computation even though every finite prefix (hence every trace) of the computation is also a prefix of some other computation for which the property does hold. It is difficult to see how a model based on finite traces could be used to specify such a property.

This problem is related to the question of *continuity* of processes. A process, defined as a set of traces, is *continuous* if the least upper bound (*lub*) of any ascending chain of traces in the set also belongs to the set. Using the partial order "is a prefix of" on traces, nondeterministic processes are not continuous in general —the *lub* of an infinite ascending chain of traces need not represent an execution sequence, even though each trace in the chain does. We know of no simple model of processes that preserves continuity. Continuity is desirable, since it makes analysis of semantics more elegant. However, since there seems to be no natural way to achieve continuity, and since we are able do without it, we see no reason to insist on it.

To allow better specification of liveness properties, our model of a network uses the notion of *observation* —a generalization of trace— and *behavior*. An *observation* records the data read and written on all ports of a network (or process) up to some point in an execution of the network and also records on which ports the network is ready to communicate at that point. A *behavior* of a network is the sequence of observations recorded during one of its executions. The resulting model is simple and facilitates information-hiding. Further, it supports both synchronous and asynchronous communication.

Our temporal proof system, which is based on the model, is *compositional*, i.e. a specification of a process is formed naturally from specifications of its component processes. Hoare-like proof systems for concurrent processes —e.g. [4], [5], [11], [16], [17]— are compositional but lack expressive power and cannot deal with temporal properties; temporal proof systems are more complicated, and most of them —e.g. [12], [13], [14]— are not compositional. We believe that this is a problem with the underlying models rather than with temporal logic itself. The models underlying most proof systems are *state-transition models*, in which a program is specified by a (binary) transition relation on the set of states. Such models are suitable for a Hoare-like proof system because the pre- and post-conditions in it correspond naturally to the initial and final states of the relation.

For temporal proof systems, modeling processes by behaviors seems more appropriate.

Our temporal proof system is compositional due to the modularity and information-hiding properties of the underlying model. It is also sound and relatively complete.

Two proof systems on traces, [5] and [16], are special cases of our system, in that the sets of specifications allowed in their systems are proper subclasses of those allowed in ours.

2. The basic system

Throughout, we will be referring to sequences of elements. Our terminology and notation for sequences is as follows. A (possibly infinite) sequence s of elements is written as $[s(0), s(1), s(2), \ldots]$ or $[s_0, s_1, s_2, \ldots]$. For example, $s(0)$ and s_0 refer to the first value of s. The notation $s(k..h)$ refers to the subsequence $[s(k), \ldots, s(h)]$, and $s(k..)$ refers to the subsequence beginning at $s(k)$, i.e. $s(k..) = [s(k), s(k+1), \ldots]$. The length of sequence s is denoted by $|s|$; $s \sqsubseteq t$ means that sequence s is a prefix of sequence t; and $\bullet$ is used as an infix catenation symbol. If $k \geq |s|$ then $s(k)$ appearing in another sequence is by convention empty. For example, if $|s| = 0$, then $[s(0), a, s(1), b] = [a, b]$.

Finally, T and F denote the Boolean constants "true" and "false", respectively.

2.1. The model

A process, as depicted in Fig. 1, has a finite number of distinctly named input and output ports. Networks of processes are formed by linking some input ports of some processes to some output ports of *other* processes in a one-to-one manner. This is done by making the names of the linked input and output ports identical. A network can itself be viewed as a process; its *external* ports are the unlinked ports of its component processes. Formally, a syntax to describe processes and networks is given by the following:

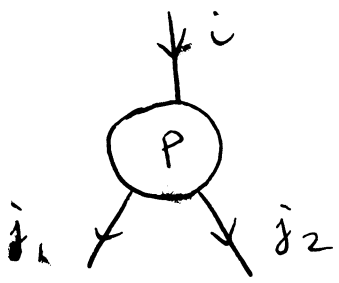


Figure 1. A primitive process

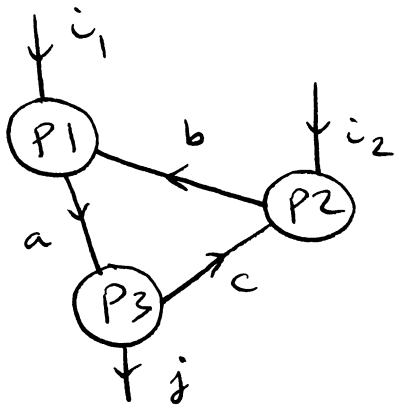


Figure 2. A network

(2.1.1) **Definition.** A *primitive process description*, with *input ports* $i_1, \ldots, i_m$ and *output ports* $j_1, \ldots, j_n$, has the form

$$P(i_1, \ldots, i_m; j_1, \ldots, j_n)$$

where P is a *primitive process name* of arity (m, n) and $i_1, \ldots, i_m, j_1, \ldots, j_n$ are distinct *port names*. □

The order of presentation of port names is significant. For example, processes $P(a, b; c, d)$ and $P(b, a; d, c)$ are in general different. We omit port names when they are clear from the context.

(2.1.2.) **Definition.** A *network description* is either a primitive process description or a *parallel composition* of the form

$$\| (N_1, \ldots, N_k)$$

where the *components* N_i are themselves network descriptions. The input (output) ports of a parallel composition are the input (output) ports of its components. The sets of input-port names of distinct components must be disjoint, and similarly for the output ports. This requirement allows a name to occur (once) as both an input and an output port; such a port is said to be *linked*. An *external* port of the network is a port that is not linked. □

Note that a primitive process is a (degenerate) network. On the other hand, we can view a network that is a parallel composition as a single process whose linked ports are hidden; we give it a description

$$P(\ldots, i_h, \ldots; \ldots, j_k, \ldots)$$

where P is a name that identifies the network, the i_h are its external input ports and the j_k its external output ports.

We view a network as an active computing agent that receives and sends messages on its ports. The semantics of a network is the set of all possible input/output behaviors that it can exhibit. This notion is now developed.

(2.1.3) **Definition.** An *event* is a pair (x, k) where x is a datum and k is a port name; (x, k) is said to *occur on* k. A *trace* on a set of ports is a finite sequence of events on those ports. □

There is a rather subtle point here concerning the input events:

- If the message transmission is *synchronous* —i.e. a process cannot send anything until the receiving process is ready to accept it as input— then the input events of a trace describe the data read by the process.
- If the message transmission is *asynchronous* —i.e. a process can send an output as soon as it is ready without having to wait for the receiving process— then the input events describe the data that have appeared at the input ports of the process.

(2.1.4) **Definition.** An *observation* on set I of input ports and set J of output ports is a quadruple (t, In, Out, Rd), where t is a trace on $I \cup J$, In is a function from I to $\{T, F\}$, Out is a function from J to $\{T, F\}$, and Rd is a function from I to the natural numbers. (Note that I and J need not be disjoint.)

In (*Out*) is called an *input (output) communication function.* The *length-of-sequence-read function Rd* satisfies $Rd(i) \leq |i|$ for all input ports i, where $|i|$ denotes the number of events in t that occur on i. For synchronous communication, $Rd(i) = |i|$, so Rd can be omitted from the observation. □

Intuitively, $In(k)$ ($Out(k)$) means "the process is ready to receive (produce) data on port k", while $Rd(i)$ is the number of events that have been read on input port i.

(2.1.5) **Definition.** The *restriction* of trace t to set S of ports is the subsequence of t that contains exactly those events that occur on ports in S. The restriction of observation (t, In, Out, Rd) to input ports I and output ports J is observation (t', In', Out', Rd') where t' is the restriction of t to $I \cup J$, function In' is obtained by restricting the domain of In to I, Out' by restricting the domain of Out to J, and Rd' by restricting the domain of Rd to I. □

We now define a *behavior*, which records the sequence of observations produced by some execution of a network as time progresses. The trace in an observation records the events that have happened at the ports up to some time; the communication functions indicate which ports are ready to communicate at that time; and the length-of-sequence-read function gives the number of values read on each input port.

(2.1.6) **Definition.** A *behavior* on I and J is an infinite sequence

$$\sigma = (t_0, In_0, Out_0, Rd_0), (t_1, In_1, Out_1, Rd_1), \ldots$$

of observations on I and J, such that

(a) t_0 is empty;

(b) for all $n \geq 0$, t_{n+1} equals t_n or is an extension of it, i.e. is t_n followed by some event (e, k). In the latter case, if $k \in I$ and the message transmission is synchronous then $In_n(k) = T$ (there is no condition on In_n if the message transmission is asynchronous), and if $k \in J$, then $Out_n(k) = T$.

(c) For all $n \geq 0$ and for all $i \in I$, $Rd_n(i) \leq Rd_{n+1}(i) \leq Rd_n(i) + 1$. Further, $Rd_n(i) < Rd_{n+1}(i)$ only if $In_n(i) = T$.

The *restriction of a behavior* σ to sets of input and output ports I and J, denoted by $\sigma|_{I,J}$, is the behavior obtained by restricting each observation of σ to I and J. Finally, $\sigma|_P$ denotes the restriction of behavior σ to the ports of network P. □

(2.1.7) **Definition.** A behavior is *eventually constant* if it has a constant suffix. It is *eventually semi-constant* if it has a suffix τ that is constant everywhere except on input ports, i.e. if the only changes in τ are the addition of events on its inport ports. □

A network is characterized by its set of behaviors. We require that the behaviors of a network be closed under finite repetition of observations:

(2.1.8) **Definition.** A set B of behaviors is *closed under finite repetition* iff for any two behaviors σ and σ' the following holds: if σ' can be obtained from σ by repeating a (possibly infinite) number of observations, each finitely many times, then $\sigma \in B$ iff $\sigma' \in B$. Any set of behaviors has a *closure* (i.e. smallest superset closed) under finite repetition. □

Closure under repetition allows us to model concurrent events by nondeterministic interleaving of sequential events without causing interference (see Lemma 2.5.3). This also has the consequence that the notion of "time" becomes a qualitative one. Time has been abstracted to a total ordering, and we can talk about the relative order in which events occur, but not about the exact time or step at which an event occurs. Lamport [10] also introduced invariance under repetition (of states), which he called "stuttering". He felt that it should be impossible to express "how long" or "how many steps" —this was a property of an implementation and not the operation— and stuttering was one way of preventing it. He also felt that "introducing the *next* operator would destroy the entire logical foundation for [the] use [of temporal logic] in hierarchical methods" [10]. We need the *next* operator in our system —not for specifying individual processes but for axiomatizing the notion of behavior.

(2.1.9) **Definition.** Let B be a set of behaviors. The notation $B[a,b,.../x,y...]$ denotes the result of simultaneously substituting port names $a,b,...$ for port names $x,y,...$ in every observation in every behavior of B. □

To give a formal semantics for networks, we assume the behaviors of primitive processes are given and define the behaviors of a composite network inductively from the behaviors of its components.

(2.1.10) **Definition.** For each primitive process $P(i_1,...,i_m; j_1,...,j_n)$ let $[\![P]\!]$ be a given set of behaviors on its input and output ports. $[\![P]\!]$ must satisfy the following three properties:

(a) $[\![P]\!]$ is closed under finite repetition.

(b) $[\![P]\!]$ respects renaming of ports: using i, j, h and k to denote vectors (of appropriate length) of port names, we have

$$[\![P(h;k)]\!] = [\![P(i;j)]\!][h,k/i,j]$$

provided that no unlinked port becomes linked as a result of the substitution and provided that the rules governing port names still hold (see Def. (2.1.2)); and

(c) a ready input port is willing to accept any input value: if $[\![P]\!]$ contains a behavior σ with $\sigma_n = (t, In, Out, Rd)$ and, for synchronous communication, $In(i) = T$ for some port i, then, for any data value x, $[\![P]\!]$ contains a

behavior

$$\sigma_0, \ldots, \sigma_n, \sigma'_{n+1}, \cdots$$

where the trace of σ'_{n+1} is $t \bullet (x,i)$. □

(2.1.11) **Definition.** For composite networks $P = \| (N_1, \ldots, N_k)$, the meaning function $[\![.]\!]$ is defined inductively by

$\sigma \in [\![\|(N_1, \ldots, N_k)]\!]$
iff for $1 \le i \le k$, $\sigma|_{N_i} \in [\![N_i]\!]$

where σ ranges over behaviors on P's ports. □

A network can be viewed as a process by "hiding" the internal structure represented by its linked ports. The input and output ports of such a process are just the external (i.e. unlinked) ports of the underlying network; its behaviors are the external behaviors of the network:

(2.1.12) **Definition.** An *external behavior* of network P is a behavior of the form $\sigma|_K$, where $\sigma \in [\![P]\!]$ and K is the set of external ports of P. □

Later, we will need the notion of a port being disabled or enabled by a process or network:

(2.1.13) **Definition.** Let $s = (t, In, Out, Rd)$ be an observation on the ports of network P. Let D be a set of components of P. In s, port k of a member of D is *disabled by D* if

- k is both an input and an output port of some members of D, the communication on k is synchronous (asynchronous), and $In(k) \wedge Out(k)$ ($Out(k)$) is F; or
- k is only an input (output) port of a member of D and $In(k)$ ($Out(k)$) is F.

Otherwise, k is *enabled*. □

To prove liveness properties of synchronous networks, we need to associate with each network a predicate on behaviors (e.g. justice, fairness), which we call a *liveness assumption*. If Ψ is a liveness assumption, then a process is specified by its Ψ*−behaviors*, i.e. its behaviors that satisfy Ψ. To ensure that the set of Ψ−behaviors of a process is closed under finite repetition, we require that Ψ itself be invariant under finite repetition, i.e. σ satisfies Ψ iff any τ obtained from σ by finite repetition of observations satisfies Ψ (see (2.1.8)). All results of this paper hold if behaviors are everywhere restricted to Ψ−behaviors.

2.2. Temporal Logic and Network Specifications

Temporal assertions on behaviors

We assume familiarity with temporal logic —see e.g. [12]— and make only the following comments. The temporal operators include $\Box$ (always), $\diamond$ (eventually), $\mathcal{U}$ (until), $\mathcal{N}$ (unless), and $\bigcirc$ (next). Following [12], we assume that the set of basic symbols in the language (individual constants and variables, proposition, predicate and function symbols) is partitioned into two subsets: global symbols and local symbols. The *global* symbols have a uniform interpretation and maintain their values or meanings from one state to another. Quantification is allowed over global variables only. The *local* symbols may assume different values in different states of the sequence. Unlike [12], we allow local function and predicate symbols in the assertion language.

An example will help to indicate the difference between local and global symbols. Let port names i and j be local and n be global; n has one value throughout, while i and j have (possibly) different values from state to state. The following temporal formula has the interpretation: if port i's trace eventually has length n, then so does port j's trace.

$$\diamond\, |i| = n \Rightarrow \diamond\, |j| = n$$

A *model* (I, α, σ) for our language consists of a global interpretation I, a global assignment α, and an infinite sequence of states $\sigma = \sigma_0, \sigma_1, \ldots$; interpretation I specifies a nonempty domain D and assigns concrete elements, functions and predicates to the global individual constants, function and predicate symbols. Assignment α assigns a value to each global free variable. Each state of σ is an assignment of values to the local free individual variables, functions and predicate symbols. The truth value of a temporal formula or term w (terms are defined as in first order logic, except that they may contain the temporal operator $\bigcirc$), denoted by $w|^{\alpha}_{\sigma}$, I being implicitly assumed, is defined as follows:

- If w is a classical term or $\not\exists$ formula (containing no modal operator) then $w|^{\alpha}_{\sigma}$ is the value of w in σ_0, under the assignment α.
- $(w_1 \vee w_2)|^{\alpha}_{\sigma} = T$ iff $w_1|^{\alpha}_{\sigma} = T$ or $w_2|^{\alpha}_{\sigma} = T$. Similarly for $\wedge$, $\neg$, etc.
- $\bigcirc w|^{\alpha}_{\sigma} = w|^{\alpha}_{\sigma(1..)}$. w is any term or formula.
- $\Box w|^{\alpha}_{\sigma} = T$ iff for all $k \geq 0$, $w|^{\alpha}_{\sigma(k..)} = T$, i.e. $\Box w$ means w is always true.
- $\diamond w|^{\alpha}_{\sigma} = T$ iff there exists $k \geq 0$ such that $w|^{\alpha}_{\sigma(k..)} = T$, i.e. $\diamond w$ means w will be true eventually.
- $(w_1 \,\mathcal{U}\, w_2)|^{\alpha}_{\sigma} = T$ iff there exists $k \geq 0$ such that $w_2|^{\alpha}_{\sigma(k..)} = T$ and for all i, $0 \leq i < k$, $w_1|^{\alpha}_{\sigma(i..)} = T$, i.e. $w_1 \,\mathcal{U}\, w_2$ means w_1 is true continuously until w_2 becomes true, and w_2 does indeed become true.
- $(w_1 \,\mathcal{N}\, w_2)|^{\alpha}_{\sigma} = T$ iff $\Box w_1|^{\alpha}_{\sigma} = T$ or $(w_1 \,\mathcal{U}\, w_2)|^{\alpha}_{\sigma} = T$.
- $\forall x.w|^{\alpha}_{\sigma} = T$ iff for all $d \in D$, $w|^{\beta}_{\sigma} = T$, where β is the assignment obtained from α by assigning d to x. (x is a global variable.)
- $\exists x.w|^{\alpha}_{\sigma} = T$ iff for some $d \in D$, $w|^{\beta}_{\sigma} = T$, where β is as above. (x is a global variable.)

Whenever w is true in a model, we say that the model *satisfies* w. For a set of axioms and theorems of temporal logic, see [12], [13], [14].

We now define what it means for a behavior to satisfy a temporal assertion. To do this, we show how an observation s is treated as a state:

- Assign to each (local) port variable k the sequence of values of events on k;
- Assign to the local function symbols *In*, *Out*, and *Rd* the corresponding communication and length-of-sequence-read functions of s. (To be rigorous, we should write $In(\text{"}k\text{"})$ instead of $In(k)$, where "k" is a denotation of the port name k in domain D, since *In* is a function of the port itself and not of its value. The same thing applies to *Out* and *Rd*.)
- Assign to the local predicate symbol $\ll$ the "precedes" relation on the events of the trace of the observation: $(\text{"}h\text{"}, m) \ll (\text{"}k\text{"}, n)$ iff the m^{th} event on port h occurs before the n^{th} event on port k in the trace. Thus $\ll$ is a total ordering.

Thus, temporal formulas can be interpreted over behaviors.

Network specifications

We define a specification as follows:

(2.2.1) **Definition.** A *specification* of a network P has the form

$$<P>\ R$$

where R is a temporal assertion in which

(a) the only local free variables are names of P's ports;

(b) the only local function symbols are *In*, *Out*, and *Rd*;

(c) the only local predicate symbol is $\ll$ ($\ll$ is needed only to axiomatize behaviors completely); and

(d) For external output ports k, $In(k)$ and $Rd(k)$ do not occur in R; for external input ports k, $Out(k)$ does not occur in R. □

(2.2.2) The interpretation of specification $<P>\ R$ is: Every behavior of P satisfies R.

Rather nicely, if the only free variables of R are the names of P's external ports, then (2.2.2) is equivalent to

(2.2.3) Every external behavior of P satisfies R.

This will be proved in later sections. In this case, $<P>\ R$ is called an *external* specification.

If Ψ is the liveness assumption, then (2.2.2) becomes

(2.2.4) Every Ψ-behavior of P satisfies R.

Finally, we will be dealing with *precise* specifications of processes, where

(2.2.5) **Definition.** Specification $<P>\ R$ is *precise* if: every behavior on P's ports is a behavior of P iff it satisfies R. □

Examples

For each process below we give two specifications, one under the assumption that the communication is asynchronous, the other that it is synchronous. We assume that there is no particular liveness assumption Ψ. Below, 0^* is the set of all sequences consisting of a finite number of zeros and 0^*1 is similarly defined.

Example. Process *BUFF1* (one-slot buffer) iteratively reads input on port i and reproduces it on port j. Its asynchronous specification is

$$
\begin{aligned}
&<BUFF1> \\
&\Box\,(j \sqsubseteq i \ \wedge\ In(i) = \neg Out(j) = (|j| = Rd(i))) \\
&\wedge\ \forall n(\Diamond\ |i| = n \Rightarrow \Diamond\ Rd(i) = n) \\
&\wedge\ \forall n(\Diamond\ Rd(i) = n \Rightarrow \Diamond\ |j| = n)
\end{aligned}
$$

The synchronous specification of *BUFF1* is

$$
\begin{aligned}
&<BUFF1> \\
&\Box\,(j \sqsubseteq i \ \wedge\ In(i) = \neg Out(j) = (|j| = |i|))
\end{aligned}
$$

Example. Process *BUFF2* reads no input on port i and produces an arbitrary, finite number of 0's followed by a 1 on port j. Its asynchronous specification is

$$
\begin{aligned}
<BUFF2>\quad &\Box\,\neg In(i) \\
&\wedge\ \Box\,(Out(j) = j \in 0^*) \\
&\wedge\ \Diamond\,\Box\, j \in 0^*1
\end{aligned}
$$

The synchronous specification of *BUFF2* is

$$
\begin{aligned}
<BUFF2>\quad &\Box\,\neg In(i) \\
&\wedge\ \Box\,(Out(j) = j \in 0^*) \\
&\wedge\ \exists x(\,\Box\, j \sqsubseteq x \in 0^*1)
\end{aligned}
$$

The specifications for *BUFF2* illustrate some subtleties of such temporal-logic specifications. First, note that both specifications require $Out(j)$ to be continuously true until the final 1 is written on port j. Omitting the second conjunct of the asynchronous specification yields a specification that allows $Out(j)$ to be false from time to time, but the last conjunct would still specify that a member of 0^*1 is eventually on j.

Omitting the second conjunct from the synchronous specification, however, yields a specification that allows behaviors that write nothing on j. We can't place the conjunct $\Diamond\,\Box\, j \in 0^*1$ in the synchronous specification because whether anything is written on j depends on the whim of the process that will read from j. In conjunction with fairness assumption (2.4.1) and appropriate specifications for a receiving process, however, one *can* prove $\Diamond\,\Box\, j \in 0^*1$.

2.3. The proof system

Our basic proof system consists of the following six parts:

(2.3.1) Axioms and inference rules that describe the domain of values that can appear in events.

(2.3.2) Axioms and inference rules for temporal logic.

(2.3.3) Axioms that define the properties of behaviors.

(2.3.4) Axioms that describe the liveness assumptions. These axioms restrict the set of behaviors of a process to those satisfying the liveness assumptions; changing them yields a different model of computation. For example, if there are no such axioms, then all behaviors are considered; if the axioms describe fairness, then only fair behaviors are considered.

(2.3.5) A set of primitive processes with precise specifications (see Def. (2.2.5)).

(2.3.6) Proof rules to derive specifications of networks.

Parts (2.3.1) and (2.3.2) are standard and need no further comment. Part (2.3.3) is discussed below. Part (2.3.4) describes the liveness assumptions. We do not deal with any particular ones here, but see the comment following Def. (2.1.13). Part (2.3.5) defines the basic building blocks of networks of processes. Part (2.3.6) is given below, after the axioms for behaviors.

Axioms for behaviors

The properties of a behavior are discussed in (2.1.6). Here we give a complete set of axioms for them.

(2.3.7) $k = [\,]$, where k is a port variable,
i.e. the initial trace is empty.

(2.3.8) $\Box\,(0 \le |\bigcirc k_0| - |k_0| + \ldots + |\bigcirc k_n| - |k_n| \le 1)$, for $n = 0, 1, \ldots$, where k_0, k_1, ... is the list of (local) port variables,
i.e. the next trace extends the current trace by at most one element.

(2.3.9) $\Box\,(0 \le Rd(k) \le |k| \wedge Rd(k) \le \bigcirc Rd(k) \le Rd(k)+1)$ for k an input port,
i.e. the number of input events read on a port is always at most the number of events occurring on that port. For synchronous communication, the axiom becomes $\Box\, Rd(k) = |k|$, so function Rd is not needed.

(2.3.10) $\Box\,(Rd(k) \neq \bigcirc Rd(k) \Rightarrow In(k))$ for input port k, and $\Box\,(k \neq \bigcirc k \Rightarrow Out(k))$ for output port k,
i.e. an event can occur only on a port that is ready to communicate.

(2.3.11) $\forall m,n\ \Box\,(m \le |k| \wedge n > |l| \wedge n = |\bigcirc l| \Rightarrow \bigcirc((\text{“}k\text{”}, m) \ll (\text{“}l\text{”}, n)))$
i.e. the event that extends a trace occurs after all the existing ones in that trace.

(2.3.12) $\forall m,n\ \Box\,((\text{“}k\text{”}, m) \ll (\text{“}l\text{”}, n) \Rightarrow \bigcirc((\text{“}k\text{”}, m) \ll (\text{“}l\text{”}, n)))$,
i.e. the ordering among the elements of a trace is preserved as the trace is extended. (This axiom, together with (2.3.8) and (2.3.11), implies

that $\Box (k \sqsubseteq \bigcirc k)$ holds.)

It is clear that any behavior satisfies these axioms. Now let σ be an infinite sequence of states that satisfies these axioms. Each state can be interpreted as an observation by letting $\ll$ be the ordering on the trace, *In* and *Out* be the communication functions, *Rd* be the length-of-sequence-read function, and the values of the port variables be the events of the trace. By induction on k, it is easy to show that each σ_k is a legitimate observation and that σ satisfies the properties of behaviors.

Proof rules

There are 3 proof rules in the basic system:

(2.3.13) **Renaming rule:** $$\frac{<P>\ R}{<P[h/k]>\ R[h/k]}$$

where h and k are vectors of distinct port names. $P[h/k]$ is the network that results from simultaneous substitution of port names h for port names k in P, provided that no unlinked port becomes linked as a result of the substitution and provided that the rules governing port names still hold (see Def. (2.1.2)). $R[h/k]$ is conventional simultaneous substitution —of port names— in logical formulas.

(2.3.14) **Network formation rule:** $$\frac{<N_i>\ R_i, \quad i = 1, \ldots, n}{<\|(\ldots, N_i, \ldots)>\ \wedge_i R_i}$$

Note that the N_i must satisfy the unique port-name requirement of Def. (2.1.2) in order for their parallel composition to be sensible.

(2.3.15) **Consequence rule:** $$\frac{<N>\ R,\ R \Rightarrow S}{<N>\ S}$$

where $R \Rightarrow S$ can be proved using the first four components (2.3.1)-(2.3.4) of the proof system.

2.4. Examples

Example. Consider the network in Fig. 3. Process *P1* reads nothing on k and produces a 1 on h. Process *P2* reads an input from h and then produces a 1 on k. The network behaves differently according to whether message transmission is asynchronous or synchronous: in the asynchronous case, a 1 is eventually produced on k; in the synchronous case, nothing is produced on k.

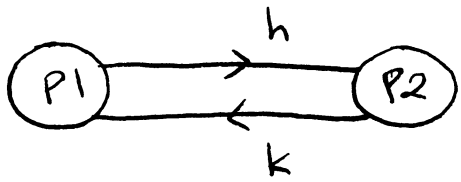


Figure 3. A network

Suppose the network is asynchronous. The process specifications are

$$\langle P1 \rangle \quad \Box \neg In(k) \ \wedge \ \Diamond \Box h = [1]$$

$$\langle P2 \rangle \quad \Box Rd(h) \leq 1 \wedge \Box (Rd(h) = 0 \Rightarrow (In(h) \wedge \neg Out(k))) \wedge (\Diamond |h| > 0 \Rightarrow \Diamond \Box k = [1])$$

By the network formation rule, the network satisfies the conjunction of the above assertions. By the consequence rule, it follows that

$$\langle NETWORK \rangle \quad \Diamond \Box (h = [1] \wedge k = [1])$$

Now suppose the network is synchronous and assume the liveness assumption is that of fairness:

(2.4.1) for any port i,
$\Box ((|i| = n \wedge \Box \Diamond (In(i) \wedge Out(i))) \Rightarrow \Diamond |i| > n)$

We have

$$\langle P1 \rangle \quad \Box \neg In(k) \wedge \Box ((Out(h) \wedge h = [\,]) \vee (\neg Out(h) \wedge h = [1]))$$

$$\langle P2 \rangle \quad \Box (k \sqsubseteq [1] \wedge |h| \leq 1) \wedge (In(h) \wedge \neg Out(k)) \,\mathcal{U}\, (|h| = 1 \wedge Out(k)) \wedge \Box (Out(k) \Rightarrow (Out(k) \,\mathcal{U}\, k = [1]))$$

By fairness assumption (2.4.1) and because $In(h)$ and $Out(h)$ are continuously T as long as $|h| = 0$, eventually $|h| = 1$ in the network, and $P1$'s specification yields $h = [1]$. Since $In(k)$ is continuously F, no output is ever produced on k. Therefore

$$\langle NETWORK \rangle \quad \Diamond \Box h = [1] \ \wedge \ \Box k = [\,]$$

Example. Brock and Ackerman [3] give an example to show that specifying processes only by "history relations" gives rise to inconsistencies: two asynchronous networks whose component processes have the same history relations have different history relations. We show here that this does not arise using our proof system; our proof system is expressive enough to express the difference between

these component processes using external specifications.

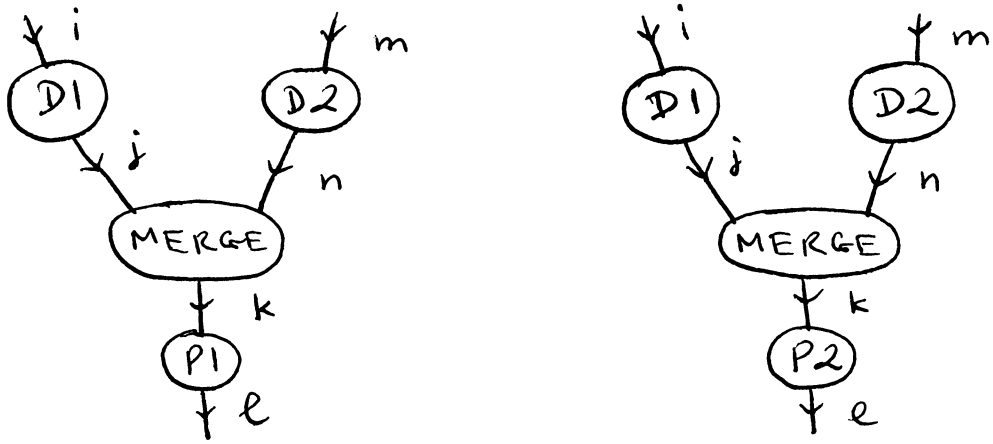


Figure 4. Asynchronous Networks *S1* and *S2*

Consider the network of Fig. 4. The component processes have the following precise specifications (in English and in our system). All the specifications contain a safety specification and a liveness specification.

D1 reads one value on i and writes it twice on j:

$$\begin{aligned}&\langle D1\rangle\ \ \Box\, j \sqsubseteq [i(0), i(0)]\\ &\qquad \wedge\ (\diamond\, |i| \geq 1 \Rightarrow \diamond\, \Box\, |j| = 2)\end{aligned}$$

D2 reads one value on m and writes it twice on n:

$$\begin{aligned}&\langle D2\rangle\ \ \Box\, n \sqsubseteq [m(0), m(0)]\\ &\qquad \wedge\ (\diamond\, |m| \geq 1 \Rightarrow \diamond\, \Box\, |n| = 2)\end{aligned}$$

MERGE nondeterministically merges the values from j and n onto k:

$$\begin{aligned}&\langle MERGE\rangle\\ &\ \ \Box\, preshuffle(j, n, k)\\ &\ \ \wedge\ (\diamond\, (|j| + |n| = v) \Rightarrow \diamond\, |k| = v)\end{aligned}$$

where $preshuffle(j, n, k)$ means that k is a prefix of an element of $shuffle(j, n)$:

$$\begin{aligned}&shuffle(a, [\,]) = shuffle([\,], a) = \{a\}\\ &shuffle(a \bullet j,\ b \bullet n)\\ &\qquad = \{a \bullet k \mid k \in shuffle(j,\ b \bullet n)\}\\ &\qquad\ \ \cup \{b \bullet k \mid k \in shuffle(a \bullet j,\ n)\}\end{aligned}$$

P1 reads a value on k and reproduces it on l, reads another value on k and reproduces it on l, and stops:

$$\begin{aligned}&\langle P1\rangle\ \ \Box\, l \sqsubseteq [k(0), k(1)]\\ &\qquad \wedge\ (\diamond\, |k| = 1 \Rightarrow \diamond\, |l| = 1)\\ &\qquad \wedge\ (\diamond\, |k| \geq 2 \Rightarrow \diamond\, \Box\, |l| = 2)\end{aligned}$$

P2 reads two values from k and *then* writes them on l ($\div$ is integer division):

$$\begin{aligned}&\langle P2\rangle\ \Box\, l \sqsubseteq [k(0), k(1)]\\ &\qquad \wedge\ \Box\ |l| \leq 2*(|k| \div 2)\\ &\qquad \wedge\ (\diamond\, |k| \geq 2 \Rightarrow \diamond\, \Box\, |l| = 2)\end{aligned}$$

P1 produces an output each time it reads an input, whereas *P2* produces no output until there are at least two inputs.

A history relation specification of a network gives for each possible set of input sequences on the input ports all possible sequences on the output ports. For example, the history relation specifications for *S1* and *S2* are the same and are given in the following table. The trouble with a history relation specification is that is does not describe the relative order in which events occur. Thus, the difference between *P1* and *P2* has been lost in embedding them in *S1* and *S2*.

i	m	ℓ
[]	[]	[]
[]	[b,...]	[b, b]
[a,...]	[]	[a, a]
[a,...]	[b,...]	[a, a] or [a, b] or [b, a] or [b, b]

We now show that *S1* and *S2* have different *external* specifications in our system. In (2.4.2) below we give an external specification for *S1*. The first conjunct of the specification restricts the values that can appear on output port *l*. The second conjunct indicates that if there is one input then two values will eventually be written on *l*. The third and fourth conjuncts capture the fact that if one input port remains empty until output appears on *l*, then the first value on *l* is the first value on the other input port. Such a statement cannot be made using history relations.

(2.4.2) $\langle S1 \rangle$
$\quad \Box\, \mathit{preshuffle}([i(0), i(0)], [m(0), m(0)], l)$
$\quad \wedge\ (\diamond\ (|i| + |m| \geq 1) \Rightarrow \diamond \Box\, |l| = 2)$
$\quad \wedge\ ((i = [\,] \,\mathcal{U}\, l \neq [\,]) \Rightarrow \diamond\, l[0] = m(0))$
$\quad \wedge\ ((m = [\,] \,\mathcal{U}\, l \neq [\,]) \Rightarrow \diamond\, l[0] = i(0))$

We now show that this specification holds. By the network formation rule, *S1* satisfies the conjunction of the specifications for *D1*, *D2*, *MERGE*, and *P1*. From the first conjuncts of these specifications, $\Box\, j \sqsubseteq [i(0), i(0)]$, $\Box\, n \sqsubseteq [m(0), m(0)]$, $\Box\, \mathit{preshuffle}(j, n, k)$, and $\Box\, l \sqsubseteq [k(0), k(1)]$, follows

(2.4.3) $\Box\, \mathit{preshuffle}([i(0), i(0)], [m(0), m(0)], k)$ and the first conjunct of (2.4.2).

From the last conjuncts of the specifications of the components, $\diamond\, |i| \geq 1 \Rightarrow \diamond \Box\, |j| = 2$, $\diamond\, |m| \geq 1 \Rightarrow \diamond \Box\, |n| = 2$, $\diamond\, |j| + |n| = v \Rightarrow \diamond\, |k| = v$, and $\diamond\, |k| \geq 2 \Rightarrow \diamond \Box\, |l| = 2$, follows the second conjunct of specification (2.4.2):

(2.4.4) $\diamond\ (|i| + |m| \geq 1) \Rightarrow \diamond \Box\, |l| = 2$

We now prove that *S1* satisfies the third conjunct of (2.4.2); by symmetry, *S1* also satisfies the fourth. Suppose that

$$i = [\,] \,\mathcal{U}\, l \neq [\,]$$

holds. This implies that $\diamond\, l \neq [\,]$. From the specification of *P1* it follows that

$l = [\,] \cup |k| = 1$, so that

(2.4.5) $i = [\,] \cup |k| = 1$

From (2.4.3) and (2.4.5), follows $preshuffle([\,],[m(0),m(0)],k) \cup |k| = 1$, i.e. $\diamond\, k(0) = m(0)$. But $(\diamond\, |k| = 1 \Rightarrow \diamond\, |l| = 1)$ and $\Box\, l \sqsubseteq [k(0),k(1)]$, so that $\diamond\, l(0) = m(0)$. Therefore, *S1* satisfies the third conjunct of specification (2.4.2).

Thus, specification (2.4.2) holds.

We now show that the following specification of *S2* holds:

(2.4.6) $<S2>$
$\Box\, preshuffle([i(0),i(0)],[m(0),m(0)],l)$
$\wedge\ (\diamond\, (|i| + |m| \geq 1) \Rightarrow \diamond\, \Box\, |l| = 2)$
$\wedge\ ((i = [\,] \cup l \neq [\,]) \Rightarrow \diamond\, l = [m(0),m(0)])$
$\wedge\ ((m = [\,] \cup l \neq [\,]) \Rightarrow \diamond\, l = [i(0),i(0)])$

The first two conjuncts are shown in the same way as for *S1*. We now show that the third holds, and the fourth holds by symmetry. Assume that

$$i = [\,] \cup l \neq [\,]$$

is true. This implies that $\diamond\, l \neq [\,]$. From this and the specification of *P2* it follows that $l = [\,] \cup |k| = 2$. Hence,

(2.4.7) $i = [\,] \cup |k| = 2$

From (2.4.3) (which holds for *S2* as well as *S1*) and (2.4.7) we have $\Box\, preshuffle([\,], [m(0),m(0)], k) \cup |k| = 2$, i.e. $\diamond\, k = [m(0), m(0)]$. But $\diamond\, |k| = 2 \Rightarrow \diamond\, |l| = 2$ and $\Box\, l \sqsubseteq [k(0), k(1)]$, so that $\diamond\, l = [m(0),m(0)]$. We have proved that the truth of the antecedent of the implication of the third conjunct of (2.4.6) implies the consequent, and the conjunct holds. Hence, specification (2.4.6) holds.

Now, it is straightforward to verify that there is a behavior of *S1* whose final trace is (0,*i*), (0,*j*), (0,*j*), (0,*k*), (0,*l*), (1,*m*), (1,*n*), (1,*n*), (1,*k*), (0,*k*), (1,*k*), (1,*l*). This behavior violates (2.4.6), so it is not a behavior of *S2*. Therefore, *S1* and *S2* indeed have different external specifications.

2.5. Soundness and Completeness

Soundness and completeness are defined as follows.

Let L be a temporal assertion language whose only local function symbols are *In*, *Out*, and *Rd* and whose only local predicate symbol is $\ll$. Let I be an interpretation whose domain D contains a set of elements (e.g. integers) and a set of sequences of these elements (e.g. sequences of integers). Global variables range over elements or sequences; local variables over sequences. Let $\{P_i\}$ be a set of *primitive* processes, from which networks of processes are to be formed.

(2.5.1) **Definition.** With L, I, $\{P_i\}$ as above, define L to be *expressive relative to* I and $\{P_i\}$ if for every primitive process P_i there exists an assertion R_i such that $<P_i> R_i$ is a precise specification. We denote this by $I \in E(L, \{P_i\})$. $\Box$

(2.5.2) **Definition.** A temporal proof system is *sound* if, for every $I \in E(L, \{P_i\})$, every specification $<P> R$ that is provable (with all the $<P_i> R_i$ as axioms and the basic proof rules as inference rules, together with a complete proof system for temporal logic and behaviors) is true (i.e. every behavior of P satisfies R in I). The proof system is *relatively complete* if, for every $I \in E(L, \{P_i\})$, every specification that is true is provable. □

This definition of soundness and relative completeness follows closely that for sequential programs (as in [1]).

We now establish a result that explains why proofs of non-interference —as defined in [11]— are not needed in our proof system. The proofs of soundness and completeness of the basic proof system depend on this "non-interference property".

(2.5.3) **Lemma.** Let I and J be sets of port names, and let R be an assertion in which

(a) the only free variables are local (port) variables in $I \cup J$,

(b) there is no occurrence of $In(j)$ and $Rd(j)$ for $j \in J - I$, and

(c) there is no occurrence of $Out(i)$ for $i \in I - J$.

Then for any behaviors σ and τ,

$\sigma|_{I,J} = \tau|_{I,J}$ implies
σ satisfies R iff τ satisfies R,

that is, satisfaction of R depends only on the interpretations of port variables occurring (free) in R.

Proof. The proof is by induction on the structure of R. The induction hypothesis is:

$\sigma|_{I,J} = \tau|_{I,J}$ implies
$\sigma(k..)$ satisfies R iff $\tau(k..)$ satisfies R, for all k.

Note that the induction hypothesis implies the lemma.

Consider the structure of R. Suppose R is an atomic formula. Then $\sigma(k..)$ satisfies R iff R is true in σ_k. But σ_k and τ_k assign the same values to all the terms and predicate symbols in R. So $\sigma(k..)$ satisfies R iff $\tau(k..)$ does.

Suppose R is composed using classical logical operators, temporal operators, or quantification over global variables. It is easy to see from the definition of the truth values of the formulas that the induction hypothesis is preserved in each of these cases. □

Note that if we do not rule out quantification over port variables, then interference may occur. For example, if R is the assertion "for all ports k different from i and j, k is empty at all times", then clearly R does not satisfy the non-interference property. This in turns implies that the network formation rule is unsound. This condition is also needed —but is unmentioned— in the proof systems of [5], [16], [17].

Now, it is easy to see why the remarks surrounding (2.2.3) and concerning interpretations of $<P> R$ are true. An external behavior of a network is just the restriction of a behavior of the network to its external ports. So every external behavior of a network satisfies an assertion on its external ports iff every behavior of the network satisfies the assertion.

(2.5.4) **Theorem.** The basic proof system is sound and relatively complete.

Proof. *Soundness*: It is clear that the renaming rule and the consequence rule are sound. Consider the network formation rule. Let σ be a behavior of $\| (..., N_k, ...)$. By our model of behaviors, $\sigma|_{N_k}$ is a behavior of N_k, $k = 1, ..., n$. Hence $\sigma|_{N_k}$ satisfies R_k for $k = 1, ..., n$. By the non-interference property, σ satisfies R_k, for $k = 1, ..., n$. This is true for all k. Therefore σ satisfies $\wedge_k R_k$. So the network formation rule is sound. It follows that the proof system is sound.

Relative completeness: First of all, we prove that the network formation rule preserves preciseness. That is, if $<N_k> R_k$ is precise for all $k = 1, ..., n$ then $<\| (..., N_k, ...)> \wedge_k R_k$ is also precise. Let σ be a behavior on $\| (..., N_k, ...)$'s ports that satisfies $\wedge_k R_k$. For each k, σ satisfies R_k. So $\sigma|_{N_k}$ must satisfy R_k, $k = 1, ..., n$, by the non-interference property. By preciseness of $<N_k> R_k$, $\sigma|_{N_k}$ is a behavior of N_k. Hence σ must be a behavior of $\| (..., N_k, ...)$. Conversely, if σ is a behavior of $\| (..., N_k, ...)$, then σ must satisfy $\wedge_k R_k$, by the soundness of the network formation rule.

Now, let $<\| (..., N_k, ...)> R$ be a specification that is true and let $<N_k> R_k$ be precise specifications of primitive processes N_k, for $k = 1, ..., n$. Then, $<\| (..., N_k, ...)> \wedge_k R_k$ is a precise specification of $\| (..., N_k, ...)$. It follows that $\wedge_k R_k \Rightarrow R$ is satisfied by every behavior on the ports of $\| (..., N_k, ...)$. By the non-interference property, every behavior must satisfy $\wedge_k R_k \Rightarrow R$. By the consequence rule, we can infer $<\| (..., N_k, ...)> R$, i.e. $<\| (..., N_k, ...)> R$ is provable. By induction on the structure of a network, we can prove that every network specification that is true is provable.

Hence the proof system is relatively complete. □

3. Procedural and recursive networks

We now extend the model to include recursive networks, a useful abstraction that can model constructs of languages such as Concurrent Prolog [23] and the parallel language of [9]. This requires us first to define procedural networks, in which certain components do not begin execution until activated by neighboring components, so that we can restrict attention to a useful class of infinite networks in which only finitely many processes can be active at any time.

3.1. Procedural networks and subroutine components

A procedural network is one in which certain components are designated as subroutines, which may not execute until activated externally.

(3.1.1) **Definition.** A *procedural network* description is either an (ordinary) network description or a *procedural composition* of the form

$$\| (\ldots, M_k, \ldots;\ \ldots, Q_q, \ldots) \ ,$$

in which each of finitely many *main* components M_k and each of perhaps infinitely many *subroutine* components Q_q is a procedural network description.

We impose the same unique port-naming requirement as in Def. (2.1.2) for ordinary networks: the sets of input (output) port names of distinct components of a procedural composition are disjoint. The *input, output, linked* and *external* ports are also defined as for ordinary networks. Finally, we require that all ports of subroutine components be linked in P, i.e. no subroutine component is connected to an external port of the procedural composition. □

Graphically, we represent a subroutine component of a procedural network by a double circle.

A procedural network may have infinitely many ports, though only finitely many of them can be external. The definitions of event, observation and behavior are not affected by this.

In a procedural network, each subroutine is initially inactive and may not begin executing until a neighboring process attempts to communicate with it. To formalize this notion, we define activation and execution of network components in terms of behaviors.

(3.1.2) **Definition.** Let Q be a subroutine component of network P. Let I and J respectively be the sets of external input and output ports of Q, I' the set of *all* (external and linked) input ports of Q, and J' the set of *all* output ports of Q. Then predicates $act(Q)$ and $inert(Q)$ are defined as follows.

$$act(Q) = (\vee_{i \in I} Out(i)) \vee (\vee_{j \in J} In(j))$$
$$inert(Q) = (\wedge_{i \in I'} \neg In(i)) \wedge (\wedge_{j \in J'} \neg Out(j))$$

Formula $act(Q)$ is true of a behavior iff in its first observation some process is ready to send to or receive from Q; we say that the observation *activates* Q. Formula $inert(Q)$ is true of a behavior iff in its first observation Q is not ready to send or receive on any port. We say that Q *is inert in* any such observation. □

(3.1.3) **Definition.** For procedural network P, the set $[\![P]\!]$ of behaviors of P must satisfy

(a) each behavior is a behavior on the ports of P.

(b) if P is an ordinary network, then $[\![P]\!]$ is given by Defs. (2.1.10) and (2.1.11).

(c) if P is a composition $\|(\ldots, M_i, \ldots; \ldots, Q_j, \ldots)$, then a behavior σ is in $[\![P]\!]$ iff for every main component M_i of P

$$\sigma|_{M_i} \in [\![M_i]\!],$$

and for every subroutine component Q_j of P,

(i) Q_j is inert in every observation of σ up to (and including) the observation in which it is first activated (hence, if never activated, Q_j is always inert).

(ii) Suppose Q_j is activated in some observation of σ. Then for some q, Q_j is inert in the observations of $\sigma(1..q-1)$ and $\sigma(q..)|_{Q_j} \in [\![Q_j]\!]$. □

A subroutine process Q_j does not begin execution —its communication functions and those to which it is connected remain false, so its trace is empty— until a neighboring process activates it. Once activated, however, Q_j must eventually execute.

The point at which Q_j begins execution is subtle. If the empty observation (trace empty and communication functions false) is a valid initial observation of Q_j, then execution can be thought of as beginning at the observation in which Q_j is activated —or indeed anywhere before that. Nothing needs to be done to initiate its execution. However, if the empty observation is not valid, so that a communication function must be initially true on some port of Q_j, then something needs to be done to initiate execution of Q_j, and the observation at which its execution begins is the first one in which Q_j is not inert. Technically, this definition helps to preserve the finite repetition property (2.1.8).

The requirement in Def. (3.1.1) that all ports of subroutine processes be linked ensures that each behavior of a procedural network uniquely determines whether its subroutine components are activated. The requirement that there be only finitely many main components ensures that, even in an infinite procedural network, only finitely many subroutine processes have been activated at any time.

The proof system to cover procedural networks is the basic proof system with the following replacement (3.1.4) for the network formation rule (2.3.14). In the conclusion of the proof rule, the second part is a temporal formula expressing exactly (3.1.3)(c)(i) for all subroutine processes and the third part expresses (3.1.3)(c)(ii). Because a procedural network may contain infinitely many components, a complete proof rule requires the use of infinitary logical operators.

(3.1.4) **Procedural-network formation rule:**

$$\frac{\begin{array}{l}<M_i> R_i,\ 1 \le i \le m \\ <Q_j> S_j,\ 1 \le j\end{array}}{\begin{array}{l}< \|(\ldots, M_i, \ldots;\ \ldots, Q_j, \ldots) > \\ \wedge_i R_i \\ \wedge \wedge_j (inert(Q_j)\ \mathcal{U}\ (act(Q_j) \wedge inert(Q_j))) \\ \wedge \wedge_j (\diamond\, act(Q_j) \Rightarrow (inert(Q_j)\ \mathcal{U}\ S_j))\end{array}}$$

(3.1.5) **Theorem.** The proof system for procedural networks is sound and relatively complete.

Proof. *Soundness*: This follows straightforwardly from the non-interference Lemma (2.5.3), which still holds.

Completeness: Since lemma (2.5.3) still holds, to prove relative completeness, it is sufficient to prove that rule (3.1.4) preserve preciseness of specifications.

Let $<M_i> R_i$ and $<Q_j> S_j$ be precise specifications, and let σ be a behavior on the ports of $||(..., M_i, ...; ..., Q_j, ...)$ that satisfies the procedural network's specification in (3.1.4). We have to show that σ is a behavior of the procedural network, i.e. that it satisfies (3.1.3)(c). First, for every main component M_i, $\sigma|_{M_i}$ satisfies R_i, by non-interference. So $\sigma|_{M_i} \in [\![M_i]\!]$. Second, the behavior satisfies the second conjunct of the conclusion of the proof rule, which is a temporal-logic formulation of (3.1.3)(c)(i), so (3.1.3)(c)(i) is satisfied. Finally, suppose Q_j is activated in some observation. By the third conjunct of the conclusion of the proof rule, Q_j remains inert until some observation σ_q such that $\sigma(q..)$ satisfies S_j. By preciseness of $<Q_j> S_j$, $\sigma(q..)|_{Q_j} \in [\![Q_j]\!]$. Hence, (3.1.3)(c)(ii) is satisfied, and σ is a behavior of the procedural network.

Conversely, if σ is a behavior of the procedural network, then σ satisfies the network's specification in the proof rule, by the soundness of the rule. Hence the proof rule is precise. □

3.2. Recursive networks

Informally, a procedural network is *recursive* if some of its subroutine processes are designated as "recursive copies" of itself. For clarity, we restrict attention to recursive networks with a single recursive copy and no other subroutine process. Relaxing this restriction is tedious but straightforward.

(3.2.1) **Definition.** A *recursive network description* is

$$X(i_1, ..., i_m ; j_1, ..., j_n) = ||(..., N_i, ...; X(h_1, ..., h_m ; k_1, ..., k_n))$$

where, except for the fact that X is not a primitive process name, each $X(...; ...)$ is a primitive process description, the righthand side is a procedural network description, and the two sides have the same external input and output ports. □

See Fig. 5 for an example.

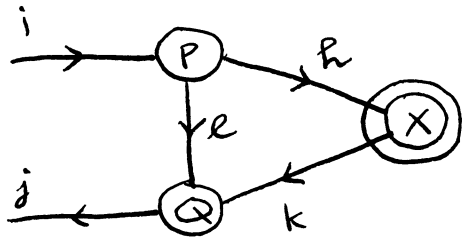


Figure 5. A recursive network $X(i,j) = ||(P,Q;X(h,k))$

We will define the behaviors of a recursive network to be the behaviors of the infinite procedural network obtained by "unrolling" the recursive definition. Defining unrolling requires a uniform method for obtaining new port names. For any port k, we define

$k^0 = k$; and

k^{r+1} is a new port name, distinct from h^s if $h \neq k$ or $s \neq r+1$.

We extend this notation to networks in the obvious way: P^r is the network obtained from P by replacing every (linked or external) port k of P by k^r.

Given recursive network description (in terms of vectors of port names of suitable lengths)

$$X(i;j) = ||(...,N_t,... ; X(h;k))$$

we define a sequence $G_r(X)$, $r = 1, 2, ...$ by:

$$G_r(X) = (|| (...,N_t,... ;))^r [h^{r-1}, k^{r-1}/i^r, j^r]$$

Intuitively, $G_{r+1}(X)$ is a uniquely renamed copy of the "body" of X, with its external ports renamed so that they link to $G_r(X)$ instead of the recursive instance of X. An example appears in Fig. 6.

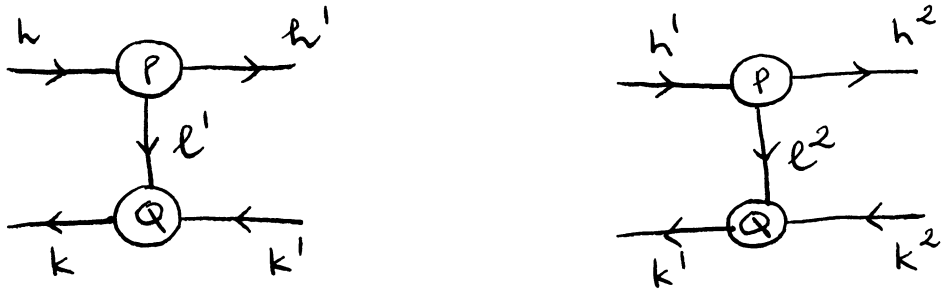


Figure 6. $G_1(X)$, $G_2(X)$

The "completely unrolled" infinite procedural network for X is

$$F(X) = || (...,N_t,... ; G_1(X), G_2(X), ...)$$

Such a network is depicted in Figure 7.

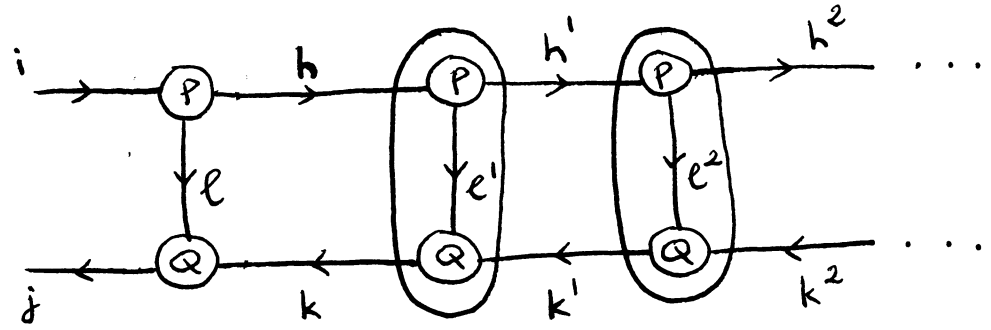


Figure 7. F(X)

The behaviors of X are defined in terms of $F(X)$:

(3.2.2) **Definition.** Let X and $F(X)$ be as above. We define $[\![X]\!] = [\![F(X)]\!]$. □

The reader may find the unrolling process slightly unconventional. The "obvious" way to unroll a recursive definition would lead to an infinite procedural network in which the nesting of subroutine subnetworks was infinitely deep, and this would be inconvenient for technical reasons. Its equivalence to our "flattened" network can be seen from the following lemma.

(3.2.3) **Lemma.** For procedural networks N_i, M_j and Q,

$$[\![\ ||\ (\ldots, M_m, \ldots;\ ||\ (\ldots, N_n, \ldots; Q))\ ]\!]$$
$$= [\![\ ||\ (\ldots, M_m, \ldots;\ ||\ (\ldots, N_n, \ldots;\ ), Q)\ ]\!]$$

provided these compositions satisfy the requirements for unique port names (note that both sides of the equality are legal if either is).

Proof. The proof is a direct application of Defs. (3.2.2) and (3.1.3). □

This lemma justifies our definition of $F(X)$ and allows us to devise a proof rule for recursive networks directly from the rule for procedural networks. The proof rule for recursive networks is:

(3.2.4) **Recursive network formation rule:**

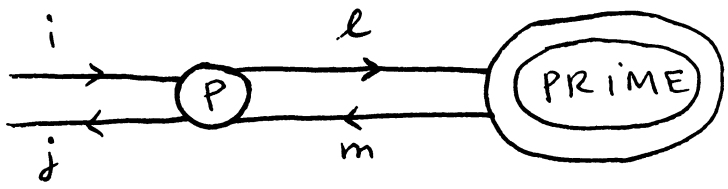
$$\frac{<||(\ldots, N_t, \ldots;\ )> R}{\begin{array}{l} <X(i; j)> \\ R \\ \wedge\ \wedge_n (inert(G_n)\ \mathcal{N}\ (act(G_n) \wedge inert(G_n))) \\ \wedge\ \wedge_n (\diamond\, act(G_n) \Rightarrow (inert(G_n)\ \mathcal{U}\ R_n)) \end{array}}$$

where R_n is obtained from R using the same substitution of names used in generating G_n from $||(\ldots, N_t, \ldots;\ )$.

The soundness and completeness of this rule follow directly from the soundness and completeness of the rule for procedural networks, from which this rule was derived.

3.3. Example

Consider synchronous recursive network *PRIME* shown in Fig. 8.



$$PRIME(i; j) = ||(P(i,m; j,l);\ PRIME(l; m))$$

Figure 8

Process P produces on j the first value from i followed by all the values from m. At the same time, P produces on l those values from i that are not divisible by the first value from i.

A formal specification for P is

$$<P>\ \Box S \wedge T,$$

where

$$S = j \sqsubseteq i(0) \bullet m \wedge l \sqsubseteq indiv(i(1..),\ i(0))$$

$$T = \Box\,(In(i) \wedge In(m))$$
$$(\wedge \diamond |i(0) \bullet m| = n \Rightarrow (\diamond |j| = n \vee \Box \diamond Out(j)))$$
$$\wedge\ (\diamond |indiv(i(1..),\ i(0))| = n$$
$$\Rightarrow (\diamond |l| = n \vee \Box \diamond Out(l)))$$

and $indiv(s, a)$ is the subsequence of s containing the elements that are not divisible by a.

We want to prove

$$<PRIME>$$
$$\Box\, i \sqsubseteq ODDNUM \Rightarrow$$
$$(\Box\, j \sqsubseteq ODDPRIME \wedge (\diamond |prime(i)| = n \Rightarrow$$
$$(\diamond |j| = n \vee \Box \diamond Out(j))))$$

where *ODDNUM* and *ODDPRIME* are the infinite ascending sequences of odd numbers and odd primes greater than 1, respectively, and $prime(i)$ is the sequence of primes in i.

By the renaming rule, we obtain

$$<G_n(PRIME)>\ \Box S_n \wedge T_n$$

where

$$S_n = S[l^{n-1}, m^n, m^{n-1}, l^n\ /\ i, m, j, l]$$

$$= m^{n-1} \sqsubseteq l^{n-1}(0) \bullet m^n \wedge l^n \sqsubseteq indiv(l^{n-1}(1..),\ l^{n-1}(0))$$

$$\begin{aligned} T_n &= T[l^{n-1}, m^n, m^{n-1}, l^n \,/\, i, m, j, l] \\ &= \Box\,(In(l^{n-1}) \wedge In(m^n)) \\ &\quad \wedge (\diamond\, |l^{n-1}(0) \bullet m^n| = v \Rightarrow (\diamond\, |m^{n-1}| = v \vee \Box \diamond\, Out(m^{n-1}))) \\ &\quad \wedge (\diamond\, |indiv(l^{n-1}(1..),\ l^{n-1}(0))| = v \\ &\qquad \Rightarrow (\diamond\, |l^n| = v \vee \Box \diamond\, Out(l^n))) \end{aligned}$$

By the proof rule for recursive networks, we have

(3.3.1) $<PRIME>$
$\Box S \wedge T$
$\wedge \wedge_n (inert(G_n)\ \mathcal{W}\ (act(G_n) \wedge inert(G_n)))$
$\wedge \wedge_n (\diamond\, act(G_n) \Rightarrow (inert(G_n)\ \mathcal{U}\ (\Box S_n \wedge T_n)))$

See Fig. 9.

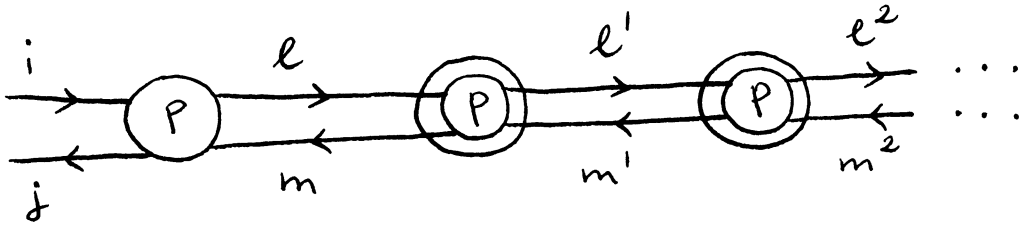


Figure 9

Safety

We first prove the safety specification

$$<PRIME>\ \Box i \sqsubseteq ODDNUM \Rightarrow \Box j \sqsubseteq ODDPRIME$$

From $<P>\ \Box S$, by applying the proof rule for recursive networks and using the fact that S_n is satisfied by the empty observation on $G_n(PRIME)$, we obtain

$$<PRIME>\ \Box S \wedge \wedge_n \Box S_n$$

Since

$$\begin{aligned} S &\Rightarrow j \sqsubseteq i(0) \bullet m \wedge l \sqsubseteq indiv(i(1..),\ i(0)) \\ S_1 &\Rightarrow m \sqsubseteq l(0) \bullet m^1 \wedge l^1 \sqsubseteq indiv(l(1..),\ l(0)) \\ S_2 &\Rightarrow m^1 \sqsubseteq l^1(0) \bullet m^2 \wedge l^2 \sqsubseteq indiv(l^1(1..),\ l^1(0)) \\ &\ldots \end{aligned}$$

it follows that *PRIME* satisfies at all times

$$\begin{aligned} j &\sqsubseteq i(0) \bullet m \\ &\sqsubseteq i(0) \bullet l(0) \bullet m^1 \\ &\sqsubseteq i(0) \bullet l(0) \bullet l^1(0) \bullet m^2 \end{aligned}$$

...

By induction, we obtain

move this to the left so that "□" is on top of "I"

- $\Box (j \sqsubseteq i(0) \bullet l(0) \bullet l^1(0) \bullet l^2(0) \ldots)$
- If $\Box i \sqsubseteq ODDNUM$ then $i(0)$ and $l(0)$ are the first two odd primes and each $l^{n+1}(0)$ is the prime that follows prime $l^n(0)$.

Hence

$$\Box i \sqsubseteq ODDNUM \Rightarrow \Box j \sqsubseteq ODDPRIME$$

Liveness

We take the liveness asumption to be that of *fairness*: if a linked port is enabled infinitely often then eventually communication must take place.

(3.3.2) $\Box ((|k| = n \wedge \Box \diamond (In(k) \wedge Out(k))) \Rightarrow \diamond |k| > n)$

We now prove the liveness specification

$$\begin{array}{l} <PRIME> \\ \Box i \sqsubseteq ODDNUM \Rightarrow \\ (\diamond |prime(i)| = n \Rightarrow (\diamond |j| = n \vee \Box \diamond Out(j))) \end{array}$$

Since $<P>$ $\Box In(m)$, $G_1(PRIME)$ is activated as soon as P starts executing. Similarly, all the $G_n(PRIME)$ are activated eventually. Hence the specification (3.3.1) of *PRIME* can be simplified to

$$\begin{array}{ll} <PRIME> & \Box S \wedge T \wedge \\ & \wedge_n \diamond (\Box S_n \wedge T_n) \end{array}$$

Assume $\Box i \sqsubseteq ODDNUM$. By fairness assumption (3.3.2), from specifications T and T_1 we obtain

$$\diamond |indiv(i(1..),\ i(0))| = n \Rightarrow \diamond |l| = n$$

Since $\Box l \sqsubseteq indiv(i(1..),\ i(0))$, it follows that

$$\diamond |prime(i(1..))| = n \Rightarrow \diamond |prime(l)| = n$$

By a similar argument, from T_1 and T_2 we obtain

$$\diamond |prime(l(1..))| = n \Rightarrow \diamond |prime(l^1)| = n$$

...

Hence

$$\begin{array}{l} \diamond |prime(i(1..))| = n \Rightarrow \diamond |l| = n \Rightarrow \diamond |l^1| = n-1 \\ \Rightarrow \cdots \Rightarrow \diamond |l^{n-1}| = 1 \\ \Rightarrow \diamond |l(0) \bullet l^1(0) \bullet \ldots \bullet l^{n-1}(0) \bullet m^n| = n \end{array}$$

The fairness assumption and specifications T_i imply

$$\begin{array}{l} \diamond |l(0) \bullet l^1(0) \bullet \ldots \bullet l^{n-1}(0) \bullet m^n| = n \\ \Rightarrow \diamond |l(0) \bullet l^1(0) \bullet \ldots \bullet l^{n-2}(0) \bullet m^{n-1}| = n \\ \Rightarrow \cdots \Rightarrow \diamond |l(0) \bullet m^1| = n \Rightarrow \diamond |m| = n \end{array}$$

← indent this line

Hence

$$\diamond\, |prime(i(1..))| = n \Rightarrow \diamond\, |m| = n$$

By this and specification T, we have for $n > 0$

$$\begin{aligned} &\diamond\, |prime(i)| = n \Rightarrow \diamond\, |prime(i(1..))| = n - 1 \\ &\Rightarrow \diamond\, |m| = n - 1 \Rightarrow \diamond\, |i(0) \bullet m| = n \\ &\Rightarrow (\diamond\, |j| = n \vee \Box \diamond Out(j)) \end{aligned}$$

The case $n = 0$ is trivial. Hence *PRIME* satisfies the required liveness specification.

4. Extensions to handle termination and deadlock

The basic model and proof system allow us to deal with any specification concerning communication between processes that can be written in temporal logic. Thus, one can specify such properties as progress of inputs and outputs.

However, the basic model cannot describe termination and deadlock because these properties involve internal states of a process. For example, a network should not be considered terminated unless each of its component processes is terminated. To model termination and deadlock in the presence of information hiding, we add global bits of information to each communication behavior. These bits contain the essential abstraction of the information that would otherwise be lost when information is hidden.

4.1. Termination

To characterize *termination* of a network, we add to each behavior a termination bit, $t \in \{T, F\}$. Intuitively, $t = T$ means the network terminates; thus, if $t = T$ we require that the behavior "appears" terminated. Formally,

(4.1.1) **Definition.** A *T−behavior* is a pair (t, σ), where $t \in \{T, F\}$ and σ is a behavior, such that if $t = T$ and the communication is synchronous (asynchronous) then σ is eventually constant (eventually semi-constant), converging on an observation in which every port of the network is disabled by all the network components it belongs to. □

The meaning of a network N is now a set $[\![N]\!]_T$ of T-behaviors:

(4.1.2) **Definition.** As in Def. (2.1.9), assume that $[\![P]\!]_T$ is given for each primitive process P. For composite network $N = ||(\ldots, N_i, \ldots)$, we define $[\![N]\!]_T$ as follows:

A T-behavior (t, σ) on N's ports is in $[\![N]\!]_T$ iff there exist $(t_i, \sigma_i) \in [\![N_i]\!]_T$ such that

(a) $t = \wedge_i t_i$ and

(b) $\sigma|_{N_i} = \sigma_i$.

Thus, a network behavior terminates iff each of its component behaviors terminates. □

To be able to prove termination, we associate with each network component N a global variable $t_N \in \{T, F\}$. An assertion on N can have as free variables t_N and N's port names. The new proof system consists of the renaming and consequence rules, obtained from the basic rules in the obvious way, together with the following new network formation rule:

(4.1.3) **Network formation rule (termination):**

$$\frac{<N_i> R_i, \quad i = 1, \ldots, n}{<||_i N_i> \exists \ldots t_{N_i} \ldots ((\wedge_i R_i) \wedge t_{||_i N_i} = \wedge_i t_{N_i})}$$

(4.1.4) **Theorem.** The proof system is sound and relatively complete.

Proof. Soundness: Let $(t, \sigma) \in [\![||_i N_i]\!]_T$. Then, (t, σ) results from combining T-behaviors (t_1, σ_1), ..., (t_n, σ_n), where $(t_i, \sigma_i) \in [\![N_i]\!]_T$, $i = 1, \ldots, n$. Hence, (t_i, σ_i) satisfies R_i. Equivalently, σ_i satisfies $R_i[t_i/t_{N_i}]$. By the non-interference property, it follows that σ satisfies $\wedge_i R_i[t_i/t_{N_i}]$. Thus, (t, σ) satisfies the specification of $||_i N_i$ given in the proof rule, and the proof rule is sound.

Completeness: We first prove that the proof rule preserves preciseness of specifications. Assume the specifications $<N_i> R_i$ are precise. Let (t, σ) satisfy the specification of $||_i N_i$ given in the proof rule. Then σ satisfies $\exists \ldots t_{N_i} \ldots (\wedge_i R_i \wedge t = \wedge_i t_{N_i})$. So σ satisfies $\wedge_i R_i[t_i/t_{N_i}] \wedge t = \wedge_i t_i$ for some t_i, $i = 1, \ldots, n$. By the non-interference property, it follows that $\sigma|_{N_i}$ satisfies $R_i[t_i/t_{N_i}]$. Equivalently, $(t_i, \sigma|_{N_i})$ satisfies R_i. By preciseness of the specifications, $(t_i, \sigma_i) \in [\![N_i]\!]_T$. Therefore, $(t, \sigma) \in [\![||_i N_i]\!]_T$. Conversely, if $(t, \sigma) \in [\![||_i N_i]\!]_T$, then it satisfies the specification of $||_i N_i$ by soundness of the rule. Thus, the proof rule preserves preciseness.

Now, let $<P> R$ be a specification that is true and let P be built from primitive components ..., N_i, ... by network formation. From the network formation rule and the precise specifications of the N_i, we obtain a precise specification $<P> S$. So $S \Rightarrow R$ is satisfied by every pair (t, σ), where σ is a behavior on P's ports. Consider any (t', σ'). $(t', \sigma'|_P)$ satisfies $S \Rightarrow R$. Hence, $\sigma'|_P$ satisfies $(S \Rightarrow R)[t/t_P]$. By the non-interference property, σ' satisfies it, too. So (t', σ') satisfies $S \Rightarrow R$. $S \Rightarrow R$ is satisfied by every T-behavior. Hence $<P> R$ is provable from $<P> S$ by the consequence rule, i.e. it is provable in the system. By induction on the structure of a network, we can prove that every network specification that is true is provable.

Hence the proof system is relatively complete. □

4.2. Deadlock

We introduce some terminology concerning the ports of a network N. The *external ports* of N are, as before, the unlinked ports of N. The *hidden ports* of N are the linked ports of the individual components N_i. The *exposed* ports of N are

the external ports of the individual components N_i that are linked in N. Note that the *linked ports* of the network are the hidden and exposed ports.

For example, consider $N = ||(N1, N2)$ as shown in Fig. 10. The external ports of N are a, c, and f; the hidden ports b and e; and the exposed port d. The reason for this distinction is as follows. When composing networks hierarchically from smaller ones, sometimes one wants to describe components only in terms of their external behavior, so the linked ports of the individual components are essentially hidden from view.

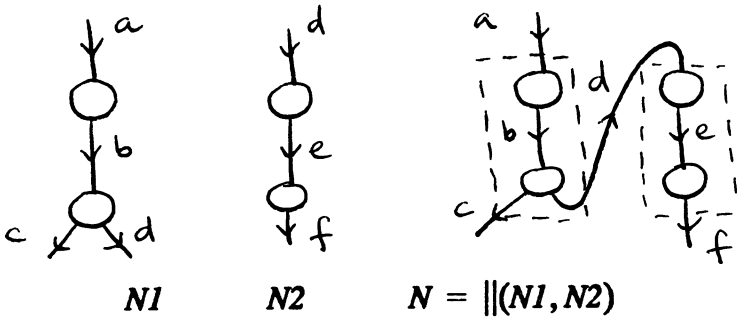


Figure 10

To characterize *deadlock*, we introduce the notion of waiting: a network is in a *wait state* if it cannot change state without a communication event taking place on one of its external ports. We add two bits of information to each behavior. The wait bit w means that eventually the network reaches a wait state and remains in that state forever. The deadlock bit d means that eventually the network becomes deadlocked, i.e. there exists a nonempty set D of components of the network such that

> Every member of D is in a wait state, and
>
> All exposed and external ports of the network that are ports of members of D are disabled by D (i.e. members of D cannot communicate with one another and refuse to communicate with outsiders).

These conditions agree with the usual intuitive requirements for deadlock. Formally:

(4.2.1) **Definition.** A *D-behavior* is a triple (d, w, σ), where $d, w \in \{T, F\}$ and σ is a behavior, such that if $w = T$ and the communication is synchronous (asynchronous) then σ is eventually constant (eventually semi-constant), converging on an observation in which all linked ports are disabled by the network. □

We remark that the condition "all linked ports of the network are disabled by the network" in the above definition is essential to our intuitive notion of a wait state —a network in which some linked port is enabled, even if hidden, cannot be in a wait state, since it can change state (by sending a message on the enabled linked port without an external communication event taking place).

The meaning of network N is now a set $[\![N]\!]_D$ of D-behaviors:

(4.2.2) **Definition.** As in Def. (2.1.9), assume $[\![P]\!]_D$ is given for each primitive process P. For composite network $N = ||(N_1, \ldots, N_m)$, we define $[\![N]\!]_D$ as follows:

A D-behavior (d, w, σ) on N's ports is in $[\![N]\!]_D$ iff there exist $(d_i, w_i, \sigma_i) \in [\![N_i]\!]_D$ such that

(a) $w = T$ iff $w_i = T$ for all i and eventually all exposed ports of N are disabled by N forever; and

(b) $d = T$ iff either $d_i = T$ for some i or there exists a nonempty subset D of components of N such that

(i) $w_i = T$ for each i such that $N_i \in D$; and

(ii) eventually all ports of members of D that are exposed or external ports of N become disabled by D, and remain so forever; and

(c) $\sigma|_{N_i} = \sigma_i$. □

An interesting fact about this characterization of deadlock is that network formation is no longer associative. It is easy to construct four processes a, b, c and d such that

$||(a, b, c, d)$ and $||(||(a, b), ||(c, d))$ are deadlocked in our model because a and b get into deadlock.

$||(||(a, c), ||(b, d))$ is not deadlocked in our model —due to information-hiding, the separate identity of a and b is lost.

While surprising, this property is not technically a problem, and it is reasonable to take the view that the way processes are composed should affect our view of whether a system is deadlocked.

There is, however, a different notion of deadlock for which associativity is preserved: a network is *totally deadlocked* if *all* its component processes get into deadlock —i.e. the set D in Def. (4.2.2) above consists of all the components of N. This notion of deadlock is treated in [4]. Using a wait bit and deadlock bit as above, rule (b) for forming the deadlock bit in Def. (4.2.2) above now becomes

$d = T$ iff all $d_i = T$ or

(i') $w_i = T$ for all i; and

(ii') Eventually all exposed and external ports of the network are disabled by the network forever.

It is not difficult to see that "all $d_i = T$" implies (i') and (ii'). Hence, the definition can be simplified to

$d = T$ iff (i') and (ii') hold.

In fact, we do not even need a deadlock bit, since (i') and (ii') do not mention d. Now it is straightforward to prove that $||(N_1, \ldots, N_n)$ is totally deadlocked iff $||(||(N_1, N_2), \ldots, N_n)$ is. By induction, associativity follows.

To prove deadlock (freedom), we associate with each network N global variables $d_N, w_N \in \{T, F\}$. It is clear how the renaming and consequence rules should be modified. The new network formation rule is

(4.2.3) **Network formation rule (deadlock):**

$$\frac{<N_i> R_i, \quad i = 1, \ldots, n}{<||_i N_i> \ \exists \ldots, d_{N_i}, \ldots, \exists \ldots, w_{N_i}, \ldots \ ((\wedge_i R_i) \wedge w_{||_i N_i} = ((\wedge_i w_{N_i}) \wedge \diamond \Box \mathit{disabled}(||_i N_i)) \wedge d_{||_i N_i} = ((\vee_i d_{N_i}) \vee (\vee_{D \in \mathbb{A}} \mathit{dlck}_D)))}$$

where $\mathbb{A}$ is the collection of all nonempty subsets of $\{N_1, \ldots, N_n\}$, and

$$\mathit{dlck}_D = ((\wedge_{N_i \in D} w_{N_i}) \wedge \diamond \Box \mathit{inactive}(D)))$$

where $\mathit{disabled}(||_i N_i)$ means that all exposed ports of $||_i N_i$ are disabled by $||_i N_i$ and $\mathit{inactive}(D)$ means that all ports of members of D that are exposed or external ports of $||_i N_i$ are disabled by D. Expressing these formulas in temporal logic is straightforward.

(4.2.4) **Theorem.** The proof system is sound and relatively complete.

Proof. The proof of soundness and completeness of this rule is similar to Theorem 4.1.4. □

5. Discussion

We have presented a new technique for process modeling that uses the notion of behavior. This technique gives rise to a model of processes that is as simple as those based on traces (e.g. [3], [4], [7], [22]) but that is more general and expressive. Our model is more suitable for temporal reasoning than state-transition models: a sound and complete temporal proof system based on the model is simpler than comparable proof systems based on state-transition models, e.g. [2], [13], [14].

As an illustration, we compare our basic proof system to two other proof systems.

In Chen and Hoare's system [5], a specification of process P has the form P *sat* R, where R is a first-order logic assertion. The interpretation is that every trace produced by P satisfies R. This is equivalent to stating $<P> \Box R$ in our system.

In Misra and Chandy's system [16], a specification of a process P has the form $R \mid P \mid S$, where R and S are first-order logic assertions. The interpretation is as follows:

(a) S holds for the empty trace.

(b) If R holds up to point k in any trace of P, then S holds up to point $(k+1)$ in that trace, for all $k \geq 0$. (An assertion R holds up to point k in a trace t means that R holds for all prefixes of t of length at most k.)

Part (b) can be restated as the following restriction on traces:

$$\forall k \geq 1\ (\neg\ (R \text{ holds up to point } k-1) \vee S \text{ holds up to point } k)$$

Together with Part(a) —S holds for the empty trace— we have

$$\begin{aligned} &\forall k \geq 0\ (\neg\ (R \text{ holds up to point } k-1) \vee S \text{ holds up to point } k) \\ &= \neg\ (\exists k \geq 0\ (R \text{ holds up to point } k-1 \wedge \neg\ (S \text{ holds up to point } k)) \\ &= \neg\ (\exists k \geq 0\ (R \text{ holds up to point } k-1 \wedge \exists j \leq k\ (\neg S \text{ holds at point } j))) \\ &= \neg\ (R\ \mathcal{U}\ \neg S) \end{aligned}$$

Hence, the specification $R \mid P \mid S$ can be written in our system as $<P>\ \neg\ (R\ \mathcal{U}\ \neg S)$.

Misra and Chandy's proof system is also shown to be incomplete in [19].

We have extended the technique to deal with sequential processes [18], and we are applying them to the shared-memory model of concurrency. These issues will be reported in a forthcoming paper.

Acknowledgements The first author would like to thank Zohar Manna and Amir Pnueli for the chance to read a draft of their book on temporal logic.

References

[1] Apt, K.R. Ten years of Hoare's logic: a survey —Part 1. *ACM TOPLAS 3*, 4 (Oct 1981), 431-483.

[2] Barringer, H., R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. *Proc. 16th ACM Symp. Theory of Comp.*, May 1984.

[3] Brock, J.D., and W.B Ackerman. Scenarios: a model of non-determinate computation. International Colloquium on Formalization of Programming Concepts, April 1981.

[4] Brookes, S.D. A semantics and proof system for communicating processes. *Lecture Notes in Computer Science 164*, 1984, 68-85.

[5] Chen, Z.C., and C.A.R. Hoare. Partial correctness of communicating processes and protocols. Technical monograph PRG-20, Programming Research Group, Oxford University Computing Laboratory, May 1981.

[6] Hewitt, C. and H.G. Baker. Laws for communicating parallel processes. *Proc. IFIP 77*, 1977, 987-992.

[7] Hoare, C.A.R. Notes on communicating sequential processes. Technical Monograph PRG-33, Programming Research Group, Oxford University Computing Laboratory, Aug 1983.

[8] Kahn, G. The semantics of a simple language for parallel programming. *Information Processing 74*, 471-475.

[9] ___ and D.B. MacQueen. Coroutines and networks of parallel processes. *Proc. IFIP 77*, 1977, 993-998.

[10] Lamport, L. What good is temporal logic? *Proc. IFIP 83*, 1983, 657-668.

[11] Levin, G.M., and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica 15* (1981), 281-302.

[12] Manna, Z., and A. Pnueli. Verification of concurrent programs, Part 1: The temporal framework. Tech. rep. STAN-CS-81-836, Stanford University, June 1981.

[13] ___ and ___. Verification of concurrent programs, Part 2: Temporal proof principles. Tech. rep. STAN-CS-81-843, Stanford University, Sept 1981.

[14] ___ and ___. How to cook a temporal proof system for your pet language. *Proc. 10th ACM Symp. Princ. of Prog. Lang.*, Jan 1983, 141-154.

[15] Milner, R. A calculus of communicating systems. *Lecture Notes in Computer Science 92*, 1980.

[16] Misra, J., and K.M. Chandy. Proofs of networks of processes. *IEEE Trans. Soft. Eng. SE-7*, 4 (July 1981).

[17] ___, ___, and T. Smith. Proving safety and liveness of communicating processes with examples. *Proc. SIGACT-SIGOPS Symp. Princ. of Distributed Computing*, Aug 1982, 201-208.

[18] Nguyen, V. A theory of processes. Ph.D. Thesis, Department of Computer Science, Cornell University, 1985.

[19] ___. The incompleteness of Misra and Chandy's proof systems. To appear in *Inf. Proc. Lett.*

[20] ___, A. Demers, D. Gries, and S. Owicki. Behavior: a temporal approach to process modeling. IBM Workshop on Logics of Programs, June 1985.

[21] ___, D. Gries, and S. Owicki. A model and temporal proof system for networks of processes, *Proc. 12th ACM Symp. Princ. of Prog. Lang.*, Jan 1985, 121-131.

[22] Pratt, V. On the composition of processes. Proc. 9th ACM Symp. Princ. of Prog. Lang., Jan 1982, 213-223.

[23] Shapiro, E. and A. Takeuchi. Object-oriented programming in Concurrent Prolog. *Journal of New Generation Computing 1*, 1, (1983), 25-48.