A Model and Temporal Proof System for Networks of Processes

Van Nguyen[1]
David Gries[1]
Susan Owicki[2]

TR 84-651
November 1984

Department of Computer Science
Cornell University
Ithaca, New York 14853

[1] Computer Science, Cornell University, Ithaca, NY 14853.
[2] Computer Systems Lab., Stanford University, Stanford, CA 94305.

A Model and Temporal Proof System for Networks of Processes

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

Abstract

A model and a sound and complete proof system for networks of processes in which component processes communicate exclusively through messages is given. The model, an extension of the trace model, can describe both synchronous and asynchronous networks. The proof system uses temporal-logic assertions on sequences of observations – a generalization of traces. The use of observations (traces) makes the proof system simple, compositional and modular, since internal details can be hidden. The expressive power of temporal logic makes it possible to prove temporal properties (safety, liveness, precedence, etc.) in the system. The proof system is language-independent and works for both synchronous and asynchronous networks.

1. Introduction

A number of trace models exist for networks of processes [3, 4, 18, 22] (none of which handles both synchronous and asynchronous networks). The advantage of a trace model is that a network is specified solely by its input-output behavior. This makes it possible to hide irrelevant information, e.g. the internal structure of the network. Our model uses a generalization of trace, which allows the specification of more *liveness properties*, especially for synchronous networks.

Our model uses the notions of *observation* (the generalization of trace) and *behavior*. An *observation* records the data read and written on all ports of a network (or single 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 execution of the network.

Recently, temporal logic has been widely used for verifying programs, especially concurrent programs, due to its expressive power. Also, a number of proof systems for networks of processes that use assertions on traces, rather than on program codes, have been proposed [4, 5, 8, 18, 19]. The main advantages of such proof systems are modularity, simplicity and generality. Modularity comes from hiding of information. One reason for simplicity is that proofs of non-interference, as defined in [13], are not needed in these systems. By not dealing with program codes, these proof systems are language-independent.

Our proof system uses temporal-logic assertions on behaviors. The system is sound and complete. Unlike most other temporal proof systems, it is compositional, i.e. a specification of a network is formed from specifications of its component processes. Two other proof systems on traces [5, 18] are special cases of our system. That is, the set of specifications (assertions) allowed in their systems are proper subclasses of those allowed in ours. In fact, by using extended temporal logic, as defined by Wolper [23], instead of temporal logic, a more expressive proof system can be obtained.

A further interesting point is that the model and the proof system work for both synchronous and

---

[1] Computer Science, Cornell University, Ithaca, NY 14853.

[2] Computer Systems Lab., Stanford University, Stanford, CA 94305.

This research is supported by the NSF under grants MCS81-03605 and DCR-8320274 and by NASA under contract NAGW419.

asynchronous networks.

This paper is organized as follows. Section 2 discusses the model of networks, including definitions of observations and behaviors. Section 3 introduces temporal logic and defines what it means for a behavior to satisfy a temporal assertion. Section 4 defines a specification of a process or network.

Section 5 outlines the various parts of the proof system and discusses two of them in detail: axioms that define properties of behaviors (section 5.1) and the actual proof rules for deriving a specification of a network from specifications of its components (section 5.2). Section 5.3 gives some examples of deriving a specification of a network, including the Brock-Ackerman example [3].

Section 6 proves soundness and relative completeness and section 7 contains a concluding discussion.

2. A model of networks of processes

A process, as depicted in Fig. 1, has a finite number of distinctly named *input ports* and *output ports* associated with it.

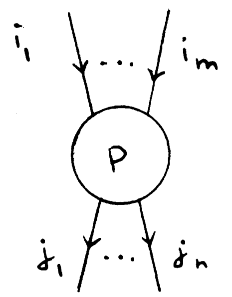

Figure 1. A (primitive) process

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. The following rule governs names of ports (see Fig. 2):

(2.1) The set of names of ports of a process or network are distinct, except that any pair of linked ports have the same name. A primitive process can not be linked to itself.

A network can also be thought of as a process whose input (output) ports are the unlinked input (output) ports of its component processes.

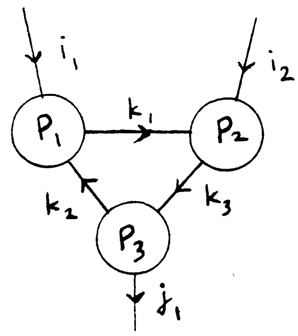

Figure 2. A network

We now give definitions of *event*, *trace*, *observation* and *behavior*.

(2.2) An *event* on port i is a pair (x, i) where x is a datum; (x, i) is said to occur on i. 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 that have been 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.

(2.3) An *observation* on a set S of ports (port names) is a tuple (t, In, Out), where t is a trace on S, function In maps the linked ports and the input ports of S to $\{T, F\}$, and function Out maps the linked ports and the output ports of S to $\{T, F\}$.

Intuitively, $In(k)$ ($Out(k)$) means "the process is ready to receive (produce) data on port k". For this reason, In (Out) is called an *input (output) communication function*.

(2.4) A *behavior* on a set of ports is an infinite sequence s_0, s_1 ... of observations on the ports satisfying the following properties:

• The trace of s_0 is empty.

• For $0 \le k$, the trace of s_{k+1} is the trace of s_k followed optionally by one event (e, h) (say). Suppose the extra event (e,h) is present. Then if h is an output or linked port, then $Out_k(h)$ must be T, where Out_k is the output communication function of s_k. If h is an input or linked port and the message transmission is synchronous then $In_k(h)$ must be T; if the message transmission is asynchronous, there is no condition on $In_k(h)$.

(2.5) A process is characterized by its set of behaviors. We require that if a behavior is in the set then any behavior obtained from it by repeating a (possibly infinite) number of observations, each some finite number of times, is also in the set, and vice versa.

We require the above repetition of observations because it allows our compositional system to have the important non-interference property (6.2) and facilitates information hiding. Lamport [12] also introduced the notion of repetition of state, which he called "stuttering", but for a different reason. He felt it should be impossible to express "how long" or "how many steps" an operation should take —this was a property of the 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" [12]. In contrast, the *next* operator plays an important part in our proof system; without it, we would not be able to characterize behaviors completely.

To summarize, a behavior of a process is the sequence of observations produced by some execution of the process, as time progresses. The trace in an observation records the events that have happened at the ports of the process up to some point; the communication functions indicate on which ports the process is ready to communicate at that point. Intuitively, a process is specified by the set of all observable behaviors under all environments, where an *environment* is a set of processes to which the process is connected.

(2.6) The *restriction of a trace* to a set of ports is the subsequence of the trace containing exactly the events occurring on ports in the set. The *restriction of a communication function In* (*Out*) to a set of ports S is the function obtained from *In* (*Out*) by restricting its domain to the linked ports and the input (output) ports of S. The *restrictions of observation* and *behavior* are defined similarly.

(2.7) The *set of behaviors of a network* is the set of behaviors on its ports whose restrictions to the ports of any component process are behaviors of that process. An *external behavior* of the network is the restriction of a network's behavior to the input and output ports of the network.

A network can be viewed as a process, in which case it is also characterized by its set of external behaviors. Such abstraction makes it possible to hide the internal structure of a network.

The above model can specify all liveness and safety properties expressible in temporal logic. However, in order to be able to *prove* liveness properties we need a *liveness assumption*:

(2.8) Associated with a synchronous network is a *liveness assumption* (e.g. justice, fairness). If Ψ is the liveness assumption then a process is specified by the set of Ψ*-behaviors* (e.g. fair behaviors), i.e. behaviors that satisfy Ψ. We require, of course, that Ψ be invariant under repetition of observations in a behavior, because (2.5) must hold —i.e. σ should satisfy Ψ iff any τ obtained from σ by repeating a (possibly infinite) number of observations, each a finite number of times, satisfies Ψ. All our definitions given above hold if behaviors are restricted to Ψ-behaviors.

3. Temporal logic and behaviors

We assume familiarity with temporal logic —see e.g. [15]— and make only the following comments. The temporal operators are: $\bigcirc$ (next), $\Box$ (always), $\diamond$ (eventually), $\mathcal{U}$ (until), $\mathcal{N}$ (unless), etc. Following [15], 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. The *local* symbols may assume different meanings and values in different states of the sequence. Quantification is not allowed over local symbols. Unlike [15], we allow local function and predicate symbols in the assertion language.

An example may help to indicate the difference between local and global symbols. Let i and j (port names) be local and n is global; n has one value throughout, while i (and j) has (possibly) different values from state to state. The example 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 a sequence of states σ. The interpretation I specifies a nonempty domain D and assigns concrete elements, functions and predicates to the global individual constants, function and predicate symbols. The assignment α assigns a value to each global free variable. The sequence $\sigma = s_0, s_1, \ldots$ is an infinite sequence of states. Each state is an assignment of values to the local free individual variables, and the function and predicate symbols. Let $\sigma^{(k)}$ denote $s_k, s_{k+1}, \ldots$, i.e. the k-truncated suffix of σ. The truth value of a temporal formula or term w (terms are defined just as in first order logic), denoted by $w|_\sigma^\alpha$, I being implicitly assumed, is defined as follows:

(1) If w is a term or a classical formula (containing no modal operator) then $w|_\sigma^\alpha$ is the value of w in s_0, under the assignment α.

(2) $(w_1 \vee w_2)|_\sigma^\alpha = \mathit{true}$ iff $w_1|_\sigma^\alpha = \mathit{true}$ or $w_2|_\sigma^\alpha = \mathit{true}$. Similarly for $\wedge$, $\neg$, etc...

(3) $\bigcirc w|_\sigma^\alpha = w|_{\sigma^{(1)}}^\alpha$. w can be a term or a formula.

(4) $\square w|_\sigma^\alpha = \mathit{true}$ iff for all $k \geq 0$, $w|_{\sigma^{(k)}}^\alpha = \mathit{true}$, i.e. $\square w$ means w is always true.

(5) $\diamond w|_\sigma^\alpha = \mathit{true}$ iff there exists $k \geq 0$ such that $w|_{\sigma^{(k)}}^\alpha = \mathit{true}$, i.e. $\diamond w$ means w will be true eventually.

(6) $(w_1 \mathcal{U} w_2)|_\sigma^\alpha = \mathit{true}$ iff there exists $k \geq 0$ such that $w_2|_{\sigma^{(k)}}^\alpha = \mathit{true}$ and for all i, $0 \leq i < k$, $w_1|_{\sigma^{(i)}}^\alpha = \mathit{true}$, i.e. $w_1 \mathcal{U} w_2$ means w_1 holds true continuously until w_2 becomes true, and w_2 does indeed become true.

(7) $(w_1 \mathcal{W} w_2)|_\sigma^\alpha = \mathit{true}$ iff $\square w_1|_\sigma^\alpha = \mathit{true}$ or $(w_1 \mathcal{U} w_2)|_\sigma^\alpha = \mathit{true}$.

(8) $\forall x.w|_\sigma^\alpha = \mathit{true}$ iff for all $d \in D$, $w|_\sigma^\beta = \mathit{true}$, where $\beta = \alpha \circ [x \leftarrow d]$ is the assignment obtained from α by assigning d to x. (x is a global variable.)

(9) $\exists x.w|_\sigma^\alpha = \mathit{true}$ iff for some $d \in D$, $w|_\sigma^\beta = \mathit{true}$, 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 [15, 17].

We now define what it means for a behavior −a sequence of observations− to satisfy a temporal assertion. This is done by showing how an observation is to be considered as a state:

(1) Assign to each local variable k the sequence $[a_0, \ldots, a_n]$, where $[(a_0, k), \ldots, (a_n, k)]$ is the restriction of the trace of the observation to port k.

(2) Assign to the local function symbols *In* and *Out* the corresponding communication functions of the observation. (Note that, to be rigorous, we should write *In*(''*k*'') instead of *In*(k), where "k" is some denotation of the port name k in the domain D. The reason is that *In* is a function on the link itself, not on its value. The same thing applies to *Out*.)

(3) Assign to the local predicate symbol $\ll$ the "precedes" relation on the trace of the observation: $(''h'', m) \ll (''k'', n)$ 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.

4. Specifications of processes

A specification of a process (network) P has the form

(4.1) $<P>\ R$

where R is a temporal assertion in which: the only local free variables are names of P's ports, the only local function symbols are *In* and *Out*, and the only local predicate symbol is $\ll$ ($\ll$ is needed to axiomatize behaviors completely). Furthermore, R contains no occurrence of *In*(k) (*Out*(k)) if k is an output (input) port of P.

(4.2) The interpretation of the specification $<P>\ R$ is:

Every behavior of P satisfies R.

A nice consequence of interpretation (4.2) is that if P is a network and the only free variables of R are the names of P's input and output ports (and not of linked ports), then interpretation (4.2) is equivalent to:

(4.3) Every external behavior of P satisfies R.

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

If Ψ is the liveness assumption, then interpretation (4.2) becomes:

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

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

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

4.1. 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 there is no particular liveness assumption Ψ. Throughout, $|x|$ denotes the length of x and $j \sqsubseteq i$ means j is a prefix of i. Also, 0^* is the set of all sequences consisting of a finite number of zeros and 0^*1 is similarly defined.

Example 1. Process *BUFF1* iteratively reads input on port i and reproduces it on port j.

The asynchronous specification of *BUFF1* is

$$<BUFF1>$$
$$\Box (j \sqsubseteq i \wedge (|j| = |i| \Rightarrow (In(i) \wedge \neg Out(j))))$$
$$\wedge\ \forall n (\Diamond\ |i| = n \Rightarrow \Diamond\ |j| = n)$$

The synchronous specification of *BUFF1* is

$$<BUFF1>$$
$$\Box (j \sqsubseteq i \wedge In(i) = \neg Out(j) = (|j| = |i|))$$

Example 2. Process *BUFF2* reads no input on port i and produces an arbitrary, finite number of 0's followed by a 1 on port j.

The asynchronous specification of *BUFF2* is

$$<BUFF2>\ \exists x(\Box (\neg In(i) \wedge j \sqsubseteq x \wedge x \in 0^*) \wedge \Diamond (j = x \wedge \neg Out(j)))$$

The synchronous specification of *BUFF2* is

$$<BUFF2>$$
$$\Box (\neg In(i) \wedge ((j \in 0^* \wedge Out(j)) \vee (j \in 0^*1 \wedge \neg Out(j))))$$

Note that the specification for *BUFF2* is invariant, but, in conjunction with appropriate specifications for a receiving process and the liveness axioms (5.4), it can be used to prove the liveness condition $\Diamond\ j \in 0^*1$.

5. The proof system

Our proof system consists of the following six parts:

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

(5.2) Axioms and inference rules for temporal logic.

(5.3) Axioms that define the properties of behaviors —see (2.4) and section 5.1.

(5.4) Axioms that describe the liveness assumptions. These axioms restrict the set of behaviors of a process to those satisfying the liveness assumptions; changing these axioms gives 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.

(5.5) A set of primitive processes with precise specifications (see (4.5)).

(5.6) Proof rules to derive specifications of networks.

Parts 5.1 and 5.2 are standard and need no further comment. Part 5.3, which captures the notion of a behavior (see (2.4)), is discussed in section 5.1. Part 5.4 describes the properties of Ψ-behaviors, thus capturing the liveness assumptions. We don't deal with any particular liveness assumptions here, but see (2.8). Part 5.5 defines the basic building blocks of networks of processes. Part 5.6 is given in section 5.2.

5.1. Axioms for behaviors

The properties that a behavior $\sigma = s_0, s_1, \ldots$ must satisfy are given in (2.4). Here we give a complete set of axioms for them. Let $k_1, k_2, \ldots$ be the list of local (port) variables.

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

(5.1.2) $\Box (|\bigcirc k_1| - |k_1| + \ldots + |\bigcirc k_n| - |k_n|) \le 1$, for $n = 1, 2, \ldots$, i.e. the next trace extends the current trace by at most one element.

(5.1.3) $\Box (k \sqsubseteq \bigcirc k \wedge$
$((k \ne \bigcirc k \wedge inp(k)) \Rightarrow In(k)) \wedge$
$((k \ne \bigcirc k \wedge outp(k)) \Rightarrow Out(k)) \wedge$
$((k \ne \bigcirc k \wedge lnkp(k)) \Rightarrow (In(k) \wedge Out(k))))$

where $inp(k)$, $outp(k)$ and $lnkp(k)$ mean k is an input, output and linked port, respectively. That is, an event can occur only on a port that is ready to communicate. (This is for synchronous message transmission; the axiom for the asynchronous case is similar.)

(5.1.4) $\forall m \forall n \, \Box ((m \le |k| \wedge n > |l|)$
$\Rightarrow \bigcirc (n \le |l|$
$\Rightarrow (\text{``}k\text{''}, m) \ll (\text{``}l\text{''}, n)))$
i.e. the event that extends a trace occurs after all the existing ones in that trace (see the end of section 3 for notation).

(5.1.5) $\forall m \forall n \, \Box ((\text{``}k\text{''}, m) \ll (\text{``}l\text{''}, n)$
$\Rightarrow \bigcirc ((\text{``}k\text{''}, m) \ll (\text{``}l\text{''}, n)))$,
i.e. the ordering among the elements of a trace is preserved as the trace is extended.

It is clear that any behavior satisfies these axioms. Now let $\sigma = s_0, s_1, \ldots$ be a 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, and the values of the port variables be the events of the trace. By induction on k, it is easy to show that each s_k is a legitimate observation and that σ satisfies the properties of behaviors. Axiom (5.1.1) implies that the trace of s_0 is empty. Axiom (5.1.2) states that a trace is extended by at most one event at a time. Axioms (5.1.4) and (5.1.5) ensure that $\ll$ is a total ordering and is the "precedes" relation. Axiom (5.1.3) implies that an event can occur only on a port that is ready to communicate.

5.2. Proof rules

There are 3 proof rules in the system:

(5.1) **Renaming rule:** $$\frac{<P>\ R}{<P'>\ R'}$$

where P' is obtained from P by changing some port names (without violating conventions (2.1) on port names) and R' is the result of replacing all free occurrences of the old port names in R by the new ones.

(5.2) **Network formation rule:**

$$\frac{<P_k>\ R_k,\ k = 1, \ldots, n}{<H>\ \wedge_k R_k}$$

where H is the network composed of the P_k, $k = 1, \ldots, n$ (assuming none of the conventions (2.1) on port names are violated).

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

where "$R \Rightarrow S$" can be proved using the first four components (5.1.1)-(5.1.4) of the proof system.

5.3. Examples

Example 1. Consider the network in Fig. 3. Process *P1* reads nothing on k_1 and produces a 1 on k_2. Process *P2* reads an input from k_2 and produces a 1 on k_1. This network behaves differently according to whether message transmission is asynchronous or synchronous: in the asynchronous case, a 1 is eventually produced on k_1; in the synchronous case, nothing is ever produced on k_1.

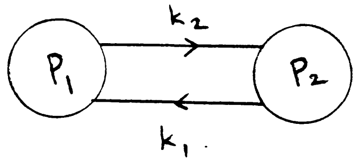

Figure 3. A network

Suppose that the network is asynchronous. Then we have

$<P1>\ \Box \neg In(k_1) \wedge \diamond\, k_2 = [1]$
$<P2>\ \Box ((|k_2| = 0 \Rightarrow In(k_2)) \wedge$
$(|k_2| > 0 \Rightarrow \diamond\, (\neg In(k_2) \wedge k_1 = [1])))$

where $[a_1, \ldots, a_n]$ denotes the sequence consisting of $a_1, \ldots, a_n$ in that order.

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

$$<NETWORK>\ \diamond\, (k_2 = [1] \wedge k_1 = [1])$$

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

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

We have

$<P1>\ (Out(k_2) \bowtie (k_2 = [1] \wedge \neg Out(k_2)))$
$\wedge \Box \neg In(k_1)$
$<P2>\ \neg Out(k_1)$
$\wedge (In(k_2) \bowtie (|k_2| > 0 \wedge Out(k_1)))$
$\wedge \Box (Out(k_1) \Rightarrow$

$$(Out(k_1) \mathcal{N} (k_1 = |1| \wedge \neg Out(k_1))))$$

By the fairness assumption and by the fact that $In(k_2)$ and $Out(k_2)$ are continuously enabled (i.e. = T) as long as $|k_2| = 0$, eventually $k_2 = [1]$ in the network. Since $In(k_1)$ is continuously disabled (i.e. = F), no output is ever produced on k_1. Therefore

$$<NETWORK> \ \diamond k_2 = [1] \wedge \Box k_1 = [\,]$$

Example 2. In [3], Brock and Ackerman give an example to show that specifying processes only by input-output relations gives rise to inconsistencies: two asynchronous networks whose component processes have the same input-output relations can have different input-output relations. We show how the processes can be specified in our system and formally derive the differences in the behaviors of the two networks.

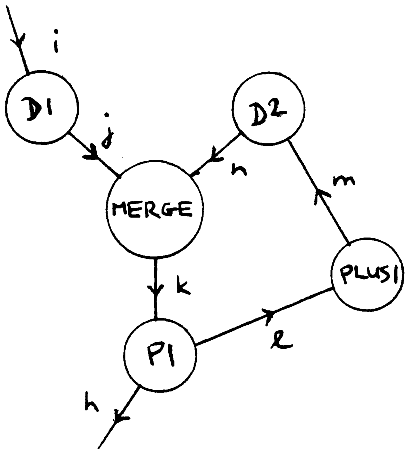

Figure 4. The Brock-Ackerman Example

We use the following notation. If $n > |s|$, where s is a sequence, then $s(n)$ appearing in a sequence is by convention empty, e.g. if $|s| = 0$, then $[a, s(1), b] = [a, b]$. Also, $l \oplus 1$ denotes the sequence calculated by adding 1 to each element of l. In the specifications, a proposition like $|j| = min(u,1)$, where j is a sequence, simply means that j always has length either 0 or 1, no matter how large u gets.

All the specifications contain a safety specification and a liveness specification.

Consider the network given in Fig. 4. The precise specifications for the component processes are:

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

$$<D1> \ \Box j \sqsubseteq [i(1), i(1)] \\ \wedge (\diamond |i| = u \Rightarrow \diamond |j| = 2 * min(u, 1))$$

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

$$<D2> \ \Box n \sqsubseteq [m(1), m(1)] \\ \wedge (\diamond |m| = u \Rightarrow \diamond |n| = 2 * min(u, 1))$$

MERGE reads values from j and n and nondeterministically merges them on k:

$$<MERGE> \\ \Box preshuffle(j, n, k) \\ \wedge (\diamond (|j| = u \wedge |n| = v) \Rightarrow \diamond |k| = u + v)$$

where $preshuffle(j, n, k)$ means that k is a prefix of an element of $shuffle(j, n)$. Using "." to denote catenation, *shuffle* is defined as

$$shuffle(j, [\,]) = shuffle([\,], j) = \{j\} \\ shuffle(a.j, b.n) = \{a.k \mid k \in shuffle(j, b.n)\} \\ \cup \{b.k \mid k \in shuffle(a.j, n)\}$$

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

$$<P1> \ \Box l \sqsubseteq [k(1), k(2)] \\ \wedge (\diamond |k| = u \Rightarrow \diamond |l| = min(u, 2)) \\ \wedge \Box h \sqsubseteq [k(1), k(2)] \\ \wedge (\diamond |k| = u \Rightarrow \diamond |h| = min(u, 2))$$

PLUS1 reads values on l, adds 1 to each of them and writes the resulting values on m:

$$<PLUS1> \ \Box m \sqsubseteq k \oplus 1 \\ \wedge (\diamond |l| = u \Rightarrow \diamond |m| = u)$$

Applying the network formation rule, we obtain

$$<NETWORK1> \ R$$

where R is the conjunction of assertions in the above five specifications. Since

$$\Box (j \sqsubseteq [i(1), i(1)] \wedge n \sqsubseteq [m(1), m(1)] \\ \wedge m \sqsubseteq l \oplus 1)$$

it follows that

$$R \Rightarrow \Box (l \sqsubseteq [k(1), k(2)]) \\ \wedge \Box (preshuffle([i(1), i(1)], \\ [l(1)+1, l(1)+1], k))$$

Hence, $k(1)$ can only be $i(1)$ or $l(1) + 1$. But it cannot be $l(1) + 1$ because $l(1)$ can only be $k(1)$! So $k(1)$ is $i(1)$. >From this, we have

$$R \Rightarrow \Box (k \sqsubseteq [i(1), i(1)] \vee k \sqsubseteq [i(1), i(1) + 1]) \\ R \Rightarrow \Box (l \sqsubseteq [i(1), i(1)] \vee l \sqsubseteq [i(1), i(1) + 1])$$

Similarly, we have

$$R \Rightarrow \Box (h \sqsubseteq [i(1), i(1)] \vee h \sqsubseteq [i(1), i(1) + 1])$$

Now consider the relationship between the lengths of the ports. To simplify it, one would naturally think of solving the set of recursive equations

$$|i| = u$$

$$
\begin{aligned}
|j| &= 2 * min(|i|, 1)\\
|n| &= 2 * min(|m|, 1)\\
|k| &= |j| + |n|\\
|l| &= min(|k|, 2)\\
|h| &= min(|k|, 2)\\
|m| &= |l|
\end{aligned}
$$

The first equation assigns a constant to the length of the input port of the network and the last six express the length relations in the five given process specifications. We can solve this set of recursive equations on the complete partially-ordered set of nonnegative integers $\cup \{\infty\}$ with $<$ as the partial order —by the usual least fixed point method (e.g. [10])— to yield the following least solution:

$$
\begin{aligned}
|i| &= u\\
|j| &= 2 * min(1, u)\\
|n| &= 2 * min(1, u)\\
|k| &= 4 * min(1, u)\\
|l| &= 2 * min(1, u)\\
|h| &= 2 * min(1, u)\\
|m| &= 2 * min(1, u)
\end{aligned}
$$

>From this, we get the following specification for *NETWORK1*:

$$
\begin{aligned}
&<NETWORK1>\\
&\quad \Box (h \sqsubseteq [i(1), i(1)] \vee h \sqsubseteq [i(1), i(1) + 1])\\
&\quad \wedge (\Diamond\, |i| = u \Rightarrow \Diamond\, |h| = 2 * min(1, u))
\end{aligned}
$$

Now consider the same network with *P1* replaced by *P2*, where *P2* has the following specification:

P2 reads 2 values from k and then writes them on h and l:

$$
\begin{aligned}
<P2>\ &\Box\, l \sqsubseteq [k(1), k(2)]\\
&\wedge \Box\, |l| \leq 2 * min(1, |k| \dot{-} 1)\\
&\wedge (\Diamond\, |k| = u \Rightarrow \Diamond\, |l| = 2 * min(1, u \dot{-} 1))\\
&\wedge \Box\, h \sqsubseteq [k(1), k(2)]\\
&\wedge \Box\, |h| \leq 2 * min(1, |k| \dot{-} 1)\\
&\wedge (\Diamond\, |k| = u \Rightarrow \Diamond\, |h| = 2 * min(1, u \dot{-} 1))
\end{aligned}
$$

where $a \dot{-} b$ is $a - b$ if $a > b$ and 0 otherwise.

P1 produces an output as soon as it reads the first input, whereas *P2* does not produce any output until it receives the second input.

Applying the network formation rule and arguing as before yields the specification

$$
\begin{aligned}
&<NETWORK2>\\
&\quad \Box\, h \sqsubseteq [i(1), i(1)]\\
&\quad \wedge (\Diamond\, |i| = u \Rightarrow \Diamond\, |h| = 2 * min(1, u))
\end{aligned}
$$

The behavior whose final trace is

$[(5,i), (5,j), (5,j), (5,k), (5,h), (5,l), (6,m), (6,n), (6,n), (6,k), (5,k), (6,k), (6,h), (6,l), (7,m)]$

satisfies R —which means that it is a behavior of the first network, by preciseness of the specifications— but does not satisfy the external specification for the second network. Thus the two networks have different behaviors.

6. Soundness and completeness

6.1. Preliminaries

Let L be a temporal assertion language whose only local function symbols are *In* and *Out* 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). The global variables range over elements or sequences, the local variables over sequences. Let $\{P_i\}$ be a set of *primitive* processes, from which networks of processes are to be formed.

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 (see (4.5)). We denote this by $I \in E(L, \{P_i\})$.

The proof system is defined to be *sound* if, for each $I \in E(L, \{P_i\})$, every specification $<P> R$ that is provable (with the $<P_i> R_i$ as axioms and proof rules (5.1), (5.2) and (5.3) as inference rules) is true —i.e. every behavior of P satisfies R under I.

The proof system is *relatively complete* if, for every $I \in E(L, \{P_i\})$, every specification that is true is provable. (Actually, we assume that parts (5.1.1), (5.1.2) and (5.1.4) of the proof system are given and prove the relative completeness of parts (5.1.3), (5.1.5) and (5.1.6) taken together.)

All the definitions and results still hold if "behavior" is replaced by "Ψ-behavior". This definition of soundness and relative completeness follows closely that for sequential programs (as in [1]).

6.2. Non-interference

We now establish a result that explains why proofs of non-interference are not needed in our proof system.

(6.2) **Non-interference property**: Let R be an assertion whose only free variables are local (port) variables and among $k_1, \ldots, k_n$ and that has no occurrence of $In(k)$ ($Out(k)$) for k an output (input) port. A behavior σ on $k_1, \ldots, k_n$ satisfies R iff any behavior τ whose restriction to $k_1, \ldots, k_n$ is σ satisfies R.

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

> Let R be an assertion whose free variables are either global variables or local variables from among $k_1, \ldots, k_n$ and that has no occurrence of $In(k)$ ($Out(k)$) where k is an output (input) port. Then $\sigma^{(k)}$ satisfies R iff $\tau^{(k)}$ satisfies R, for all k.

Note that the induction hypothesis implies the theorem.

Consider the structure of R.

(1). R is an atomic formula. Let s_k and t_k be the k^{th} elements of σ and τ. Then $\sigma^{(k)}$ satisfies R iff R is true in s_k. But s_k and t_k assign the same values to all the terms and predicate symbols in R. So $\sigma^{(k)}$ satisfies R iff $\tau^{(k)}$ does.

(2) 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. Q.E.D.

Note that if we do not have the condition that quantification over port variables is not allowed, 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, 8, 18, 19].

Now, it is easy to see why the remark concerning interpretations (4.2) and (4.3) of $<P> R$ is true. An external behavior of a network is just the restriction of a behavior of the network to its input and output ports. So every external behavior of a network satisfies an assertion on its input and output ports iff every behavior of the network satisfies the assertion.

6.3. Soundness

It is clear that the renaming rule and the consequence rule are sound. Consider the network formation rule. Let $\sigma = s_0, s_1, \ldots$ be a behavior of H. By our model of behaviors, $\sigma(P_k)$, the sequence with element $\sigma(P_k)_m$ equal to the restriction of s_m to the ports of P_k, for all m, is a behavior of P_k, $k = 1, \ldots, n$. Hence $\sigma(P_k)$ satisfies R_k for $k = 1, \ldots, n$. By the non-interference property, σ satisfies R_k, for $k = 1, \ldots, n$. 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.

6.4. Relative completeness

First of all, we prove that the network formation rule preserves preciseness. That is, if $<P_k> R_k$ is precise for all $k = 1, \ldots, n$ then $<H> \wedge_k R_k$ is also precise. Let $\sigma = s_0, s_1, \ldots$ be a behavior on H's ports that satisfies $\wedge_k R_k$. For each k, σ satisfies R_k. So $\sigma(P_k)$, as defined above, must satisfy R_k, $k = 1, \ldots, n$, by the non-interference property. By preciseness of $<P_k> R_k$, $\sigma(P_k)$ is a behavior of P_k. Hence σ must be a behavior of H. Conversely, if σ is a behavior of H, then σ must satisfy $\wedge_k R_k$, by the soundness of the network formation rule.

Now, let $<H> R$ be a specification that is true, and let H be formed from primitive processes P_k, where $<P_k> R_k$ is precise, for $k = 1, \ldots, n$. Then, $<H> \wedge_k R_k$ is a precise specification of H. It follows that $\wedge_k R_k \Rightarrow R$ is satisfied by every behavior on the ports of H. By the non-interference property, every behavior must satisfy $\wedge_k R_k \Rightarrow R$. By the consequence rule, we can infer $<H> R$, i.e. $<H> R$ is provable.

Hence, the proof system is relatively complete.

7. Discussion

7.1. Expressiveness

The proof system we just described is quite general and expressive. As an illustration, we look at 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, at all times, the trace produced by P satisfies R. This is equivalent to stating $<P> \Box R$ in our system.

In Misra and Chandy's system [18], a specification of a process H has the form $R \mid H \mid S$, where R and S are first-order logic assertions. The interpretation is as follows:

> S holds for the empty trace.
>
> If R holds up to point k in any trace of H, 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.)

This is equivalent to stating $<H>\ S \wedge \neg(R\ \mathcal{U}\ \neg S)$ in our system. According to the interpretation of temporal formulas, $R\ \mathcal{U}\ \neg S$ is true iff $\exists k \geq 0$ such that $\neg S$ is true in s_k and for all i, $0 \leq i < k$, R is true in s_i (for R and S are classical formulas). So $S \wedge \neg(R\ \mathcal{U}\ \neg S)$ is true iff S is true in s_0 and for all $k \geq 0$, if R is true in s_i for all $i < k$ then S is true in s_k. This is again equivalent to: S is true in s_0 and for all $k \geq 0$, if R is true in s_i for all $i < k$, then S is true in s_j for all $j \leq k$. This is not difficult to see, since if R is true in s_i then S is true in s_{i+1} (let k be $i + 1$). But this is exactly the interpretation of $R \mid H \mid S$ in Misra and Chandy's system.

However, temporal logic is by no means the most expressive language there is. Certain properties cannot be expressed in temporal logic, e.g. "formula p is true in every even state". As shown in [23], temporal logic can be extended by right-linear grammars. That is, for every right-linear grammar, an appropriately defined temporal operator can be added to the language. The resulting logic is called extended temporal logic.

We can enhance the expressive power of our proof system in the same way by using *extended temporal logic*, instead of temporal logic, as the assertion language. The proof rules remain the same, and the resulting proof system is still sound and complete. In fact, any language in which the assertions satisfy the non-interference property would serve our purpose.

At first glance, it looks as if extended temporal logic is of no use for our proof system because restriction (2.5), which destroys regularity, is required. However, we can save the situation by introducing the notion of normal form. A behavior is in *normal form* if no state – except the last one, if there is one – is repeated. A process is completely specified by its set of normal-form behaviors. Non normal-form behaviors are needed to make process linking easier to discuss. So we can have specifications of the form

$$<P>\ \mathit{normalform} \Rightarrow R,$$

where R is a formula in extended temporal logic and *normalform* means "behavior is in normal form", which can be easily expressed in temporal logic. To obtain more complete specifications, we can introduce a new temporal operator $\mathcal{R}$ (repeat). A sequence σ satisfies $\mathcal{R}(p)$ iff σ is obtained from some sequence τ by repeating some states of τ a finite number of times, where τ satisfies p. Then we can have specifications of the form

$$<P>\ (\mathit{normalform} \Rightarrow R) \wedge (\neg \mathit{normalform} \Rightarrow \mathcal{R}(\mathit{normalform} \wedge R))$$

7.2. Extension of model and proof system

The model we described here can specify liveness properties that involve progress of inputs and outputs but not liveness properties that involve internal states, e.g. *deadlock* and *termination*. *Recursive networks* and *sequential program constructs* such as assignment, if-then-else, while, etc. are not defined, either. Fortunately, the model and proof system can be extended in a simple way to deal with these matters. These issues will be addressed in a forthcoming paper by the first author and Alan Demers.

Acknowledgement The first author wishes to thank Zohar Manna for giving him a chance to read a draft of his book on temporal logic. We are grateful to Alan Demers for many valuable discussions and to Fred Schneider for useful remarks on earlier drafts of this paper.

References

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

(2) Ben-Ari, M., Manna, Z., and Pnueli, A. The logic of next time. 8*th* Annual ACM Symp. Principles of Programming Languages, Jan 1981, 164–176.

(3) Brock, J.D., and Ackerman, W.B. 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 Hoare, C.A.R. Partial correctness of communicating processes and protocols. Technical monograph PRG–20, Programming Research Group, Oxford University Computing Laboratory, May 1981.

(6) Emerson, E.A., and Halpern, J.Y. "Sometimes" and "not never" revisited; on branching versus linear time. 10*th* Annual ACM Symp. on Principles of Programming Languages, Jan 1983, 127–140.

(7) Hoare, C.A.R. Communicating sequential processes. *Comm. ACM 21*, 8 (Aug 1978), 666–677.

(8) ___. A calculus of total correctness for communicating processes. Technical Monograph PRG-23, Programming Research Group, Oxford University Computing Laboratory, May 1981.

(9) Hughes, G.E., and Cresswell, M.J. *An introduction to modal logic*. Menthuen & Co., London, 1968.

(10) Kahn, G. The semantics of a simple language for parallel programming. *Inf. Process. Letters 74* (1974), 471–475.

(11) Lamport, L. "Sometimes" is sometimes "not never". 7*th* Annual ACM Symp. Principles of Programming Languages, Jan 1980, 174–185.

(12) ___. What good is temporal logic? Proceedings IFIP 1983, 657–668.

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

(14) Lipton, R.J. A necessary and sufficient condition for the existence of Hoare logics. 18*th* Annual Symp. on Foundations of Computer Science, 1977, 1–6.

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

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

(17) ___ and ___. How to cook a temporal proof system for your pet language. 10*th* Annual ACM Symp. Principles of Programming Languages, Jan 1983, 141–154.

(18) Misra, J., and Chandy, K.M. Proofs of networks of processes. *IEEE Trans. Software Eng. SE–7*, 4 (July 1981).

(19) Misra, J., Chandy, K.M., and Smith, T. Proving safety and liveness of communicating processes with examples. SIGACT-SIGOPS Symp. Principles of Distributed Computing, Aug 1982, 201–208.

(20) Owicki, S., and Lamport, L. Proving liveness properties of concurrent programs. *ACM TOPLAS 4*, 3 (July 1982), 455–495.

(21) Pnueli, A. The temporal logic of programs,. 18*th* Annual Symp. Foundations of Computer Science, 1977, 46–57.

(22) Pratt, V. On the composition of processes. 9*th* Annual ACM Symp. Principles of Programming Languages, Jan 1982, 213–223.

(23) Wolper, P.L. Synthesis of communicating processes from temporal logic specifications. Ph.D. thesis, Stanford University, Aug 1982.

A Model and Temporal Proof System for Networks of Processes

Van Nguyen[1]
David Gries[1]
Susan Owicki[2]

TR 84-651
November 1984

Department of Computer Science
Cornell University
Ithaca, New York 14853

[1] Computer Science, Cornell University, Ithaca, NY 14853.
[2] Computer Systems Lab., Stanford University, Stanford, CA 94305.

A Model and Temporal Proof System for Networks of Processes

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

Abstract

A model and a sound and complete proof system for networks of processes in which component processes communicate exclusively through messages is given. The model, an extension of the trace model, can describe both synchronous and asynchronous networks. The proof system uses temporal-logic assertions on sequences of observations – a generalization of traces. The use of observations (traces) makes the proof system simple, compositional and modular, since internal details can be hidden. The expressive power of temporal logic makes it possible to prove temporal properties (safety, liveness, precedence, etc.) in the system. The proof system is language-independent and works for both synchronous and asynchronous networks.

1. Introduction

A number of trace models exist for networks of processes [3, 4, 18, 22] (none of which handles both synchronous and asynchronous networks). The advantage of a trace model is that a network is specified solely by its input-output behavior. This makes it possible to hide irrelevant information, e.g. the internal structure of the network. Our model uses a generalization of trace, which allows the specification of more *liveness properties*, especially for synchronous networks.

Our model uses the notions of *observation* (the generalization of trace) and *behavior*. An *observation* records the data read and written on all ports of a network (or single 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 execution of the network.

Recently, temporal logic has been widely used for verifying programs, especially concurrent programs, due to its expressive power. Also, a number of proof systems for networks of processes that use assertions on traces, rather than on program codes, have been proposed [4, 5, 8, 18, 19]. The main advantages of such proof systems are modularity, simplicity and generality. Modularity comes from hiding of information. One reason for simplicity is that proofs of non-interference, as defined in [13], are not needed in these systems. By not dealing with program codes, these proof systems are language-independent.

Our proof system uses temporal-logic assertions on behaviors. The system is sound and complete. Unlike most other temporal proof systems, it is compositional, i.e. a specification of a network is formed from specifications of its component processes. Two other proof systems on traces [5, 18] are special cases of our system. That is, the set of specifications (assertions) allowed in their systems are proper subclasses of those allowed in ours. In fact, by using extended temporal logic, as defined by Wolper [23], instead of temporal logic, a more expressive proof system can be obtained.

A further interesting point is that the model and the proof system work for both synchronous and

[1] Computer Science, Cornell University, Ithaca, NY 14853.

[2] Computer Systems Lab., Stanford University, Stanford, CA 94305.

This research is supported by the NSF under grants MCS81-03605 and DCR-8320274 and by NASA under contract NAGW419.

communication function of s_k. If h is an input or linked port and the message transmission is synchronous then $In_k(h)$ must be T; if the message transmission is asynchronous, there is no condition on $In_k(h)$.

(2.5) A process is characterized by its set of behaviors. We require that if a behavior is in the set then any behavior obtained from it by repeating a (possibly infinite) number of observations, each some finite number of times, is also in the set, and vice versa.

We require the above repetition of observations because it allows our compositional system to have the important non-interference property (6.2) and facilitates information hiding. Lamport [12] also introduced the notion of repetition of state, which he called "stuttering", but for a different reason. He felt it should be impossible to express "how long" or "how many steps" an operation should take —this was a property of the 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" [12]. In contrast, the *next* operator plays an important part in our proof system; without it, we would not be able to characterize behaviors completely.

To summarize, a behavior of a process is the sequence of observations produced by some execution of the process, as time progresses. The trace in an observation records the events that have happened at the ports of the process up to some point; the communication functions indicate on which ports the process is ready to communicate at that point. Intuitively, a process is specified by the set of all observable behaviors under all environments, where an *environment* is a set of processes to which the process is connected.

(2.6) The *restriction of a trace* to a set of ports is the subsequence of the trace containing exactly the events occurring on ports in the set. The *restriction of a communication function In* (*Out*) to a set of ports S is the function obtained from *In* (*Out*) by restricting its domain to the linked ports and the input (output) ports of S. The *restrictions of observation* and *behavior* are defined similarly.

(2.7) The *set of behaviors of a network* is the set of behaviors on its ports whose restrictions to the ports of any component process are behaviors of that process. An *external behavior* of the network is the restriction of a network's behavior to the input and output ports of the network.

A network can be viewed as a process, in which case it is also characterized by its set of external behaviors. Such abstraction makes it possible to hide the internal structure of a network.

The above model can specify all liveness and safety properties expressible in temporal logic. However, in order to be able to *prove* liveness properties we need a *liveness assumption*:

(2.8) Associated with a synchronous network is a *liveness assumption* (e.g. justice, fairness). If Ψ is the liveness assumption then a process is specified by the set of Ψ*-behaviors* (e.g. fair behaviors), i.e. behaviors that satisfy Ψ. We require, of course, that Ψ be invariant under repetition of observations in a behavior, because (2.5) must hold —i.e. σ should satisfy Ψ iff any τ obtained from σ by repeating a (possibly infinite) number of observations, each a finite number of times, satisfies Ψ. All our definitions given above hold if behaviors are restricted to Ψ-behaviors.

3. Temporal logic and behaviors

We assume familiarity with temporal logic —see e.g. [15]— and make only the foll[illegible] [illegible]ts. The temporal operators are: $\bigcirc$ (next), $\square$ (always), $\diamond$ (eventually), $\mathcal{U}$ (until), $\mathcal{N}$ (unless), etc. Following [15], we assume that the set of basic symbols in the [illegible]e (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 [illegible] from one state to another. The *local* symbols may assume different meanings and values in different states of the sequence. Quantification is not allowed over local symbols. Unlike [15], we allow local function and predicate symbols in the assertion language.

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

$$(\diamond\ |i| = n \Rightarrow \diamond\ |j| = n)$$

(4.3) Every external behavior of P satisfies R.

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

If Ψ is the liveness assumption, then interpretation (4.2) becomes:

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

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

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

4.1. 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 there is no particular liveness assumption Ψ. Throughout, $|x|$ denotes the length of x and $j \sqsubseteq i$ means j is a prefix of i. Also, 0^* is the set of all sequences consisting of a finite number of zeros and 0^*1 is similarly defined.

Example 1. Process *BUFF1* iteratively reads input on port i and reproduces it on port j.

The asynchronous specification of *BUFF1* is

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

The synchronous specification of *BUFF1* is

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

Example 2. Process *BUFF2* reads no input on port i and produces an arbitrary, finite number of 0's followed by a 1 on port j.

The asynchronous specification of *BUFF2* is

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

The synchronous specification of *BUFF2* is

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

Note that the specification for *BUFF2* is invariant, but, in conjunction with appropriate specifications for a receiving process and the liveness axioms (5.4), it can be used to prove the liveness condition $\Diamond\, j \in 0^*1$.

5. The proof system

Our proof system consists of the following six parts:

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

(5.2) Axioms and inference rules for temporal logic.

(5.3) Axioms that define the properties of behaviors —see (2.4) and section 5.1.

(5.4) Axioms that describe the liveness assumptions. These axioms restrict the set of behaviors of a process to those satisfying the liveness assumptions; changing these axioms gives 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.

(5.5) A set of primitive processes with precise specifications (see (4.5)).

(5.6) Proof rules to derive specifications of networks.

Parts 5.1 and 5.2 are standard and need no further [illegible]t. Part 5.3, which captures the notion of a behavior (see (2.4)), is discussed in section 5.1. Part 5.4 describes the properties of Ψ-behaviors, thus capturing the liveness assumptions. We don't deal with any particular liveness assumptions here, but see (2.8). Part 5.5 defines the basic building blocks of networks of processes. Part 5.6 is given in section 5.2.

5.1. Axioms for behaviors

The properties that a behavior $\sigma = s_0, s_1, \ldots$ must satisfy are given in (2.4). Here we give a complete set of axioms for them. Let $k_1, k_2, \ldots$ be the list of local (port) variables.

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

(5.1.2) $\Box\,(|\bigcirc k_1| - |k_1| + \ldots + |\bigcirc k_n| - |k_n|) \le 1$, for $n = 1, 2, \ldots$, i.e. the next trace extends the current trace by at most one element.

(5.1.3)
$$\begin{aligned}\Box\,(k \sqsubseteq \bigcirc k \wedge\ &\\ &((k \ne \bigcirc k \wedge inp(k)) \Rightarrow In(k)) \wedge\\ &((k \ne \bigcirc k \wedge outp(k)) \Rightarrow Out(k)) \wedge\\ &((k \ne \bigcirc k \wedge lnkp(k)) \Rightarrow (In(k) \wedge Out(k))))\end{aligned}$$

where $inp(k)$, $outp(k)$ and $lnkp(k)$ mean k is an input, output and linked port, respectively. That is, an event can occur only on a

$(Out(k_2) \bowtie (k_1 = |1| \wedge \neg Out(k_1))))$

By the fairness assumption and by the fact that $In(k_2)$ and $Out(k_2)$ are continuously enabled (i.e. = T) as long as $|k_2| = 0$, eventually $k_2 = [1]$ in the network. Since $In(k_1)$ is continuously disabled (i.e. = F), no output is ever produced on k_1. Therefore

$$\langle NETWORK \rangle \quad \Diamond\, k_2 = [1] \wedge \Box\, k_1 = [\,]$$

Example 2. In [3], Brock and Ackerman give an example to show that specifying processes only by input-output relations gives rise to inconsistencies: two asynchronous networks whose component processes have the same input-output relations can have different input-output relations. We show how the processes can be specified in our system and formally derive the differences in the behaviors of the two networks.

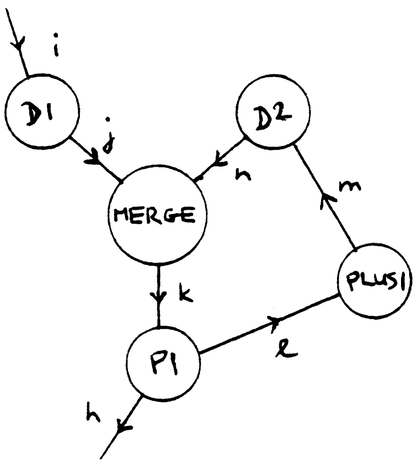

Figure 4. The Brock-Ackerman Example

We use the following notation. If $n > |s|$, where s is a sequence, then $s(n)$ appearing in a sequence is by convention empty, e.g. if $|s| = 0$, then $[a, s(1), b] = [a, b]$. Also, $l \oplus 1$ denotes the sequence calculated by adding 1 to each element of l. In the specifications, a proposition like $|j| = min(u,1)$, where j is a sequence, simply means that j always has length either 0 or 1, no matter how large u gets.

All the specifications contain a safety specification and a liveness specification.

Consider the network given in Fig. 4. The precise specifications for the component processes are:

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

$$\langle D1 \rangle \quad \Box\, j \sqsubseteq [i(1), i(1)] \\ \wedge (\Diamond\, |i| = u \Rightarrow \Diamond\, |j| = 2 * min(u, 1))$$

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

$$\langle D2 \rangle \quad \Box\, n \sqsubseteq [m(1), m(1)] \\ \wedge (\Diamond\, |m| = u \Rightarrow \Diamond\, |n| = 2 * min(u, 1))$$

MERGE reads values from j and n and nondeterministically merges them on k:

$$\langle MERGE \rangle \\ \Box\, preshuffle(j, n, k) \\ \wedge (\Diamond\, (|j| = u \wedge |n| = v) \Rightarrow \Diamond\, |k| = u + v)$$

where $preshuffle(j, n, k)$ means that k is a prefix of an element of $shuffle(j, n)$. Using "." to denote catenation, $shuffle$ is defined as

$$shuffle(j, [\,]) = shuffle([\,], j) = \{j\} \\ shuffle(a.j, b.n) = \{a.k \mid k \in shuffle(j, b.n)\} \\ \cup \{b.k \mid k \in shuffle(a.j, n)\}$$

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

$$\langle P1 \rangle \quad \Box\, l \sqsubseteq [k(1), k(2)] \\ \wedge (\Diamond\, |k| = u \Rightarrow \Diamond\, |l| = min(u, 2)) \\ \wedge \Box\, h \sqsubseteq [k(1), k(2)] \\ \wedge (\Diamond\, |k| = u \Rightarrow \Diamond\, |h| = min(u, 2))$$

PLUS1 reads values on l, adds 1 to each of them and writes the resulting values on m:

$$\langle PLUS1 \rangle \quad \Box\, m \sqsubseteq k \oplus 1 \\ \wedge (\Diamond\, |l| = u \Rightarrow \Diamond\, |m| = u)$$

Applying the network formation rule, we obtain

$$\langle NETWORK1 \rangle \quad R$$

where R is the conjunction of assertions in the above five specifications. Since

$$\Box\, (j \sqsubseteq [i(1), i(1)] \wedge n \sqsubseteq [m(1), m(1)] \\ \wedge m \sqsubseteq l \oplus 1)$$

it follows that

$$R \Rightarrow \Box\, (l \sqsubseteq [k(1), k(2)]) \\ \wedge \Box\, (preshuffle([i(1), i(1)], \\ [l(1)+1, l(1)+1], k))$$

Hence, $k(1)$ can only be $i(1)$ or $l(1) + 1$. But it cannot be $l(1) + 1$ because $l(1)$ can only be $k(1)$! So $k(1)$ is $i(1)$. >From this, we have

$$R \Rightarrow \Box\, (k \sqsubseteq [i(1), i(1)] \vee k \sqsubseteq [i(1), i(1) + 1]) \\ R \Rightarrow \Box\, (l \sqsubseteq [i(1), i(1)] \vee l \sqsubseteq [i(1), i(1) + 1])$$

Similarly, we have

$$R \Rightarrow \Box\, (h \sqsubseteq [i(1), i(1)] \vee h \sqsubseteq [i(1), i(1) + 1])$$

Now consider the relationship between the lengths of the ports. To simplify it, one would naturally think of solving the set of recursive equations

$$|i| = u$$

(input) port. A behavior σ on $k_1, \ldots, k_n$ satisfies R iff any behavior τ whose restriction to $k_1, \ldots, k_n$ is σ satisfies R.

Proof. The proof is by induction on the s[illegible]e of R. The induction hypothesis is:

> Let R be an assertion whose free variables are either global variables or local variables from among $k_1, \ldots, k_n$ and that has no occurrence of $In(k)$ $(Out(k))$, where k is an output (input) port. Then $\sigma^{(k)}$ satisfies R iff $\tau^{(k)}$ satisfies R, for all k.

Note that the induction hypothesis implies the theorem.

Consider the structure of R.

(1) R is an atomic formula. Let s_k and t_k be the k^{th} elements of σ and τ. Then $\sigma^{(k)}$ satisfies R iff R is true in s_k. But s_k and t_k assign the same values to all the terms and predicate symbols in R. So $\sigma^{(k)}$ satisfies R iff $\tau^{(k)}$ does.

(2) 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. Q.E.D.

Note that if we do not have the condition that quantification over port variables is not allowed, 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, 8, 18, 19].

Now, it is easy to see why the remark concerning interpretations (4.2) and (4.3) of $<P> R$ is true. An external behavior of a network is just the restriction of a behavior of the network to its input and output ports. So every external behavior of a network satisfies an assertion on its input and output ports iff every behavior of the network satisfies the assertion.

6.3. Soundness

It is clear that the renaming rule and the consequence rule are sound. Consider the network formation rule. Let $\sigma = s_0, s_1, \ldots$ be a behavior of H. By our model of behaviors, $\sigma(P_k)$, the sequence with element $\sigma(P_k)_m$ equal to the restriction of s_m to the ports of P_k, for all m, is a behavior of P_k, $k = 1, \ldots, n$. Hence $\sigma(P_k)$ satisfies R_k for $k = 1, \ldots, n$. By the non-interference property, σ satisfies R_k, for $k = 1, \ldots, n$. 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.

6.4. Relative completeness

First of all, we prove that the network formation rule preserves preciseness. That is, if $<P_k> R_k$ is precise for all $k = 1, \ldots, n$ then $<H> \wedge_k R_k$ is also precise. Let $\sigma = s_0, s_1, \ldots$ be a behavior on H's ports that satisfies $\wedge_k R_k$. For each k, σ satisfies R_k. So $\sigma(P_k)$, as defined above, must satisfy R_k, $k = 1, \ldots, n$, by the non-interference property. By preciseness of $<P_k> R_k$, $\sigma(P_k)$ is a behavior of P_k. Hence σ must be a behavior of H. Conversely, if σ is a behavior of H, then σ must satisfy $\wedge_k R_k$, by the soundness of the network formation rule.

Now, let $<H> R$ be a specification that is true, and let H be formed from primitive processes P_k, where $<P_k> R_k$ is precise, for $k = 1, \ldots, n$. Then, $<H> \wedge_k R_k$ is a precise specification of H. It follows that $\wedge_k R_k \Rightarrow R$ is satisfied by every behavior on the ports of H. By the non-interference property, every behavior must satisfy $\wedge_k R_k \Rightarrow R$. By the consequence rule, we can infer $<H> R$, i.e. $<H> R$ is provable.

Hence, the proof system is relatively complete.

7. Discussion

7.1. Expressiveness

The proof system we just described is quite general and expressive. As an illustration, we look at 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, at all times, the trace produced by P satisfies R. This is equivalent to stating $<P> \Box R$ in our system.

In Misra and Chandy's system [18], a specification of a process H has the form $R \mid H \mid S$, where R and S are first-order logic assertions. The interpretation is as follows:

> S holds for the empty trace.
>
> If R holds up to point k in any trace of H, 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.)

(8) ___. A calculus of total correctness for communicating processes. Technical Monograph PRG−23, Programming Research Group, Oxford University Computing Laboratory, May 1981.

(9) Hughes, G.E., and Cresswell, M.J. *An introduction to modal logic*. Menthuen & Co., London, 1968.

(10) Kahn, G. The semantics of a simple language for parallel programming. *Inf. Process. Letters 74* (1974), 471−475.

(11) Lamport, L. "Sometimes" is sometimes "not never". 7*th* Annual ACM Symp. Principles of Programming Languages, Jan 1980, 174−185.

(12) ___. What good is temporal logic? Proceedings IFIP 1983, 657−668.

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

(14) Lipton, R.J. A necessary and sufficient condition for the existence of Hoare logics. 18*th* Annual Symp. on Foundations of Computer Science, 1977, 1−6.

(15) Manna, Z., and Pnueli, A. Verification of concurrent programs, Part 1: The temporal framework. Tech. rep. STAN−CS−81−836, Stanford University, June 1981.

(16) ___ and ___. Verification of concurrent programs, Part 2: Temporal proof principles. Tech. rep. STAN−CS−81−843, Stanford University, Sept 1981.

(17) ___ and ___. How to cook a temporal proof system for your pet language. 10*th* Annual ACM Symp. Principles of Programming Languages, Jan 1983, 141−154.

(18) Misra, J., and Chandy, K.M. Proofs of networks of processes. *IEEE Trans. Software Eng. SE−7*, 4 (July 1981).

(19) Misra, J., Chandy, K.M., and Smith, T. Proving safety and liveness of communicating processes with examples. SIGACT−SIGOPS Symp. Principles of Distributed Computing, Aug 1982, 201−208.

(20) Owicki, S., and Lamport, L. Proving liveness properties of concurrent programs. *ACM TOPLAS 4*, 3 (July 1982), 455−495.

(21) Pnueli, A. The temporal logic of programs,. 18*th* Annual Symp. Foundations of Computer Science, 1977, 46−57.

(22) Pratt, V. On the composition of processes. 9*th* Annual ACM Symp. Principles of Programming Languages, Jan 1982, 213−223.

(23) Wolper, P.L. Synthesis of communicating processes from temporal logic specifications. Ph.D. thesis, Stanford University, Aug 1982.