A Theory of Processes

Van Long Nguyen

(Ph.D. Thesis)
June 1985

TR 85-691

Department of Computer Science
Cornell University
Ithaca, NY 14853
A THEORY OF PROCESSES

A Thesis
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Van Long Nguyen

August 1985
Biographical Sketch

Van Long Nguyen was born in Vietnam in 1959. He received a B.S. degree from Monash University in 1982 and an M.S. degree from Cornell University in 1983.
Dedication

To my mother and Jean-Louis Lassez
Acknowledgements

I am deeply indebted to David Gries, my thesis advisor and friend, for his extraordinary support and guidance in the last three years. Without him I would have never been able to get a job! From him I learned to always strive for simplicity and to present things clearly. I thank him also for his financial support.

I am also indebted to Alan Demers and Susan Owicki for helping me with my thesis research. Alan’s versatility and curiosity made research fun and exciting while Susan’s depth and practical-mindedness provided the necessary discipline. David, Alan and Susan also taught me to rely more on intuition — and less on formal methods — in solving problems.

I would like to thank Zohar Manna for giving me help during my stay at Stanford. I learned a great deal from his papers and books. I thank Richard Waldinger for many valuable discussions on program synthesis.

Thanks also go to Charlie Van Loan for being nice and helpful. I thank Raimund Seidel, Ming Li, Rogerio Drummond, Anne Neiryck, Jean Zhou, Poh-Lean Tan, Devika Subramaniam, Lane Hemachandra, Lee Barford, Jim
Hook, David Rossiter and Haesun Park for being good friends. The Computer Science Department at Cornell has provided a very nice working environment. Thanks also go to Anil Nerode for serving on my Special Committee.

I wish to thank the Australian Computer Research Board for offering me their first Graduate Fellowship. This inspired my confidence and got me off to a good start. I also thank the Mathematics Department at Monash University for all the things they taught me.

Thanks go to Prabu N. for being such a wonderful friend. Finally, I would like to thank Jean-Louis Lassez, my first mentor and best friend, for all the things he has done for me. He is one of the kindest and most sincere persons I have ever met.
Table of Contents

Chapter 1. Introduction ................................................................. 1
Chapter 2. A model and temporal proof system for processes ............. 6
Chapter 3. A parallel functional language ...................................... 75
Chapter 4. A synthesis system for networks of processes ................. 87
Chapter 5. Discussion ................................................................. 124
References ................................................................. 127
CHAPTER 1

INTRODUCTION

1.1. General remarks

In recent years, there has been a lot of interest in concurrent programming and distributed computing. This gives rise to the notion of a process. A process is a computing agent that performs some task and communicates with other agents for inputs and outputs. Two models of process communication are: shared variable, where processes communicate through a set of shared variables, and message passing, where processes communicate by sending and receiving messages through interconnected ports (channels). In this thesis, we concentrate on the message passing model.

Concurrent processes are inherently more complex than sequential programs, due to the nondeterministic interleaving of events. This has led to several research efforts to gain a better understanding of processes. In one direction, tools to specify and formalize reasoning about processes have been developed. These tools enable us to abstract and study the essential properties of processes.
independent of their implementation. In other words, they provide non-operational semantics of processes. The main tools are algebraic models, e.g. [Ho83, Pr], and logical proof systems, e.g. [CH, LG, MP83]. Typically, in an algebraic model, processes are described by functions or sets, and process compositions are described by algebraic operators on these functions or sets. In a proof system, processes are described by logical formulas and their compositions are described by proof rules on these formulas.

In another direction, programming languages in which process communication and parallel composition are the basic primitives have been designed. Some are simple extensions of existing sequential languages, e.g. Concurrent C [GR] and Multilisp [Hal], while some are new languages that are radically different from conventional sequential languages, e.g. CSP [Ho78] and data flow languages [Ac]. These languages provide concise notations for describing processes algorithmically.

In yet another direction, some attempts have been made at synthesizing processes from given specifications, e.g. [Wo]. The synthesis problem is very difficult, however, so relatively little progress has been made. This problem is the opposite of the verification problem and is more difficult, since it requires more creativity to write a program than to verify that a program is correct.
1.2. Contributions

In this thesis, we attempt to give a comprehensive theory of processes. We give a general set-theoretic model of processes that forms the formal basis of our theory. The model is quite simple, and yet it is more expressive than any model of processes that we know of, e.g. [BA, Br, HB, Ho83, Ka, Mi, P]. A compositional temporal proof system is defined on the model. This provides a formal language and framework in which processes can be specified and reasoned about. The proof system is simpler than any temporal proof system for processes that we know of, e.g. [BKP, MP81a, MP81b, MP83], and yet it is just as expressive. Furthermore, it is as simple as any existing Hoare-like proof systems, e.g. [Br, CH, LG, MC, MCS], but is more expressive.

We also present a simple parallel functional language that incorporates most of the programming constructs in the model. The language enables us to describe processes algorithmically. It combines features of functional (applicative) and concurrent languages and has a general recursion scheme that makes it more expressive than Hoare's CSP [Ho78]. Another advantage is that it is easier to exploit concurrency in a language without side-effects, as argued in [Ac]. The language also demonstrates how communication and concurrency could be handled in a functional language.

Finally, we give a deductive system for synthesizing asynchronous networks of deterministic processes. Due to the difficulty of the problem, we are not able to
do this for more general types of network. The synthesis system uses the deductive tableau method that Manna and Waldinger developed for sequential programs [MW80, MW82]. In this approach, the synthesis of a program is regarded as a theorem-proving problem; the desired program is constructed as a by-product of the proof. This enables us to use such powerful tools like transformation rules and resolution rules in a single framework.

1.3. Organization

In chapter 2 of the thesis, we describe a model and temporal proof system for processes. In chapter 3, we present a parallel functional language. Chapter 4 gives a synthesis system for networks of processes. In Chapter 5, we discuss related work and directions for future research.

1.4. Notation

Throughout the thesis we use the following notation. \( T \) and \( F \) denote the Boolean constants \textit{true} and \textit{false}, respectively. For \( \sigma = s_1, s_2, \ldots \) a sequence, \( |\sigma| \) denotes its length \( (=\infty \text{ if } \sigma \text{ is infinite}) \), \( \sigma(i) \) denotes the \( i^{th} \) element of \( \sigma \), i.e. \( s_i \), and \( \sigma^{(k)} \) denotes the \( k \)-truncated suffix of \( \sigma \), i.e. \( s_k, s_{k+1}, \ldots \). \( \sigma \) can also be written as \([\sigma(1), \sigma(2), \ldots]\) and \textit{first}(\(\sigma\)) denotes the first element of \(\sigma\). If \(\sigma\) is infinite and has a constant suffix, \textit{last}(\(\sigma\)) denotes the repeated element of this
suffix.

Sequence catenation is denoted by "." and $\sigma \subseteq \tau$ means that sequence $\sigma$ is a prefix of sequence $\tau$. If $k > |\sigma|$ then $\sigma(k)$ appearing in another sequence is by convention empty. For example, if $|\sigma| = 0$ then $[a, \sigma(1), b] = [a, b]$. 
CHAPTER 2

A MODEL AND TEMPORAL PROOF SYSTEM FOR PROCESSES

2.1. Introduction

A number of models have been proposed for processes [BA, Br, HB, Ho83, Mi, Pr]. None of these models handles both synchronous and asynchronous communication in a single framework. In addition, their modeling of liveness properties is generally unsatisfactory. The models that seem most promising, due to their simplicity and ability to hide information, are those based on traces. A trace is a finite sequence of communication events, which can be thought of as an abstraction of a process state in which all irrelevant internal details are hidden. Roughly, 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. Liveness properties can deal with complete, possibly infinite execution sequences of a process, 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 \textit{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 may not represent a possible execution sequence, even though each trace in the chain does. We know of no simple model of processes with an appropriately defined partial ordering on traces that preserves continuity. Continuity is a desirable property, since it makes the analysis of the semantics more elegant. However, since there seems to be no natural way to achieve continuity, and we are able do without it, we see no reason to insist on having it.

To allow better specification of liveness properties, we introduce the notions of observation (a generalization of trace) and communication behavior. An \textit{observation} records the data read and written on all ports of a network 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 \textit{communication behavior} of a network is the sequence of observations recorded during one execution of the network. Thus
a communication behavior is an abstraction of an entire execution sequence in which irrelevant internal details are hidden. We also extend the notion of communication behavior to model termination, deadlock and recursive networks using either synchronous or asynchronous communication. The resulting model is simple and modular and facilitates information hiding.

Conventional sequential program constructs are modeled by a similar technique. We show how to combine the two by giving a behavior model for a version of CSP [Ho78].

Our models are well-suited for temporal reasoning about processes. Using them, we define temporal proof systems that are *compositional*: a specification of a process is formed naturally from specifications of its component processes. Hoare-like proof systems for concurrent processes [Br, CH, LG, MC, MCS] are compositional but lack expressive power and cannot deal with temporal properties; temporal proof systems are more complicated, and most of them [MP81a, MP81b, MP83] are not compositional. We believe that this is a problem with the underlying models that have been chosen, and not 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 postconditions in it correspond naturally to the initial and final states of the relation. For temporal proof systems, however, modeling processes by behaviors seems more
appropriate. Our temporal proof systems are compositional due to the modularity and information-hiding properties of the underlying models. They are also sound and relatively complete.

Our models and proof systems can handle either synchronous or asynchronous communication. Two proof systems on traces [CH, MC] are special cases of our basic proof system, in that the sets of specifications allowed in their systems are proper subclasses of those allowed in ours.

The remainder of the chapter is organized as follows. Section 2.2 describes the basic model and temporal proof system. In each of the next two sections we add a new property or program construct to the basic model and present a sound and relatively complete temporal proof system for it: Section 2.3 treats termination and deadlock, and Section 2.4 treats recursively-defined networks. The last section of the chapter deals with conventional sequential constructs and gives a behavior model for CSP.

2.2. The basic model and proof system

2.2.1. The basic model

Informally, a process has associated with it a finite number of distinctly-named input and output ports, as shown in Figure 2.1. Processes can be combined into networks, as shown in Figure 2.2, by linking some input ports of some
processes to some output ports of other processes in a one-to-one manner.

A network can itself be viewed as a process whose input and output ports are the unlinked ports of its component processes. Formally, a syntax to describe processes and networks is given by the following:

Definition 2.2.1.1: Assume for each \( n, m \geq 0 \) we are given a fixed set of zero or more primitive process names of arity \((m, n)\). Assume also an infinite set of distinct port names \( i_1, i_2, \ldots, j_1, j_2, \ldots \) etc. A primitive process description is

\[
P(i_1, \ldots, i_m ; j_1, \ldots, j_n)
\]

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.

![Figure 2.1 A (primitive) process](image1)

![Figure 2.2 A network](image2)
Ports $i_1, \ldots, i_m$ are the input ports of the process; ports $j_1, \ldots, j_n$ are its output ports.

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. However, we shall omit port names whenever they are clear from context.

Definition 2.2.1.2: A network description is either

- a primitive process description, or

- a parallel composition, of the form $|| (N_1, \ldots, N_k)$, where the components, $N_1, \ldots, N_k$, are network descriptions.

For primitive processes, input and output ports are defined above. For composite network $N = || (N_1, \ldots, N_k)$, ports are defined inductively: $k$ is an input (output) port of $N$ iff $k$ is an input (output) port of some component $N_i$. We also impose the following unique port naming requirement:

- The sets of input (output) port names of distinct components of $N$ must be disjoint.

A port of $N$ is

- an internal port if it occurs as both an input and output port of a component of $N$

- a linked port if it occurs as an input port of a component and as an output port of another component of $N$
• an external port otherwise.

Note that a primitive process is a (degenerate) network. We view a network as an active computing agent that receives and sends messages on its input and output ports. The semantics of a network is the set of all possible input/output behaviors that it can exhibit. This notion is developed formally in the next few definitions.

Definition 2.2.1.3: An event on port $k$ is a pair $(x, k)$ where $x$ is a datum; $(x, k)$ is said to occur on $k$.

Definition 2.2.1.4: A trace on set $S$ of ports is a finite sequence of events on ports in $S$.

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.
Definition 2.2.1.5: An observation on set $I$ of input ports and set $J$ of output ports is a quadruple $(t, \text{In}, \text{Out}, \text{Rd})$, where $t$ is a trace on $I \cup J$, $\text{In}$ is a function from $I$ to $\{T, F\}$, $\text{Out}$ is a function from $J$ to $\{T, F\}$ and $\text{Rd}$ is a function from $I$ to the natural numbers.

$\text{In} (\text{Out})$ is called an input (output) communication function. $\text{Rd}$ is called a length function and is such that, for all $i \in I$, $\text{Rd}(i) \leq |i|$, where $|i|$ denotes the length of the sequence of events in $t$ occurring on $i$.

Intuitively, $\text{Rd}(i)$ denotes the length of the sequence of events that have been read on input port $i$. Consequently, for synchronous communication, $\text{Rd}(i) = |i|$, for all $i \in I$, and so $\text{Rd}$ can be omitted from an observation. $\text{In}(k)$ ($\text{Out}(k)$) means “the process is ready to receive (produce) data on port $k$”.

Definition 2.2.1.6: A communication behavior on $I$ and $J$ is an infinite sequence

$$\sigma = (t_1, \text{In}_1, \text{Out}_1, \text{Rd}_1), (t_2, \text{In}_2, \text{Out}_2, \text{Rd}_2), (t_3, \text{In}_3, \text{Out}_3, \text{Rd}_3), \ldots$$

of observations on $I$ and $J$, such that

- $t_1$ is empty, and
- for all $n$, $t_{n+1}$ 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 passing is synchronous then $\text{In}_n(k) = T$, and if $k \in J$ then $\text{Out}_n(k) = T$, and
- for all positive integer $n$ and for all $i \in I$, $\text{Rd}_n(i) \leq \text{Rd}_{n+1}(i) \leq \text{Rd}_n(i) + 1$, and $\text{Rd}_n(i) < \text{Rd}_{n+1}(i)$ only if $\text{In}_n(i) = T$. 

Intuitively, a communication behavior is the sequence of observations produced by some execution of a network as time progresses. The trace in an observation records the events that have occurred at the ports up to some time; the communication functions indicate which ports are ready to communicate at that time; and the length function indicates the number of events read on each input port.

A network is characterized by its set of communication behaviors. We require the behaviors of a network to be closed under finite repetition, defined as follows:

Definition 2.2.1.7: A set $B$ of behaviors is closed under finite repetition iff for any two behaviors $\sigma$ and $\sigma'$ the following condition holds: if $\sigma'$ can be obtained from $\sigma$ 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.

We require the behaviors of a network to be closed under finite repetition because this allows us to model concurrent events by nondeterministic interleaving of sequential events without causing interference (see Lemma 2.2.2.7.1). This also has the consequence that the notion of "time" in our model is a qualitative notion, and not a quantitative one. Time has been abstracted to a total ordering, and we can talk about the relative order in which events occur, but not the exact time or step at which an event occurs. Invariance under repetition of states is also introduced by Lamport —for the latter reason— in [La83], where it is called
“stuttering”.

To give a formal semantics for networks, we assume the behaviors of primitive processes are given and define the behaviors of composite networks inductively from the behaviors of their components. Intuitively, a behavior of a network is a combination of behaviors of its components in which an event on an external port represents a message being passed between the network and its environment, while an event on a linked port represents a message being passed internally between two components. To formalize this, we define a restriction of a network behavior to one of its components as follows:

Definition 2.2.1.8: The restriction of trace $t$ to a set $S$ of ports is that subsequence of $t$ containing exactly those events occurring 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 $I \cup J$, $In'$, $Rd'$ are the functions obtained by restricting the domain of $In$, $Rd$ to $I$, and $Out'$ is the function obtained by restricting the domain of $Out$ to $J$. The restriction of a communication behavior $\sigma$ to $I$ and $J$, denoted $\sigma|_{I,J}$, is defined similarly. We let $\sigma|_N$ denote the restriction of behavior $\sigma$ to the input and output ports of network $N$.

We can now give a formal semantics for networks:

Definition 2.2.1.9: For each primitive process $P(i_1, \ldots, i_m ; j_1, \ldots, j_n)$ let $[P]$ be a given set of behaviors on the input and output ports of $P$. We require that $[P]$ is closed under finite repetition, as defined above, and that $[P]$ respects renaming
of ports:

\[ [P(h_1, \ldots, h_m; k_1, \ldots, k_n)] = [P(i_1, \ldots, i_m; j_1, \ldots, j_n)] \]

\[ [h_1, \ldots, k_n / i_1, \ldots, j_n] \]

where the notation \( B[\ldots p_r / \ldots q_r \ldots] \) represents the result of simultaneous substitution of port names \( p_r \) for \( q_r \) in every observation of set \( B \), provided no new link is formed as a result of the substitution. Finally, we require that an input port that is ready to communicate must be willing to accept any input value. That is, if \([P]\) contains behavior

\[ \sigma = (t_1, In_1, Out_1, Rd_1), \ldots, (t_n, In_n, Out_n, Rd_n), \ldots \]

(and in the case of synchronous communication, \( In_n(i) = T \) for some port \( i \)) then, for any data value \( x \), \([P]\) must contain a behavior

\[ \sigma' = (t_1, In_1, Out_1, Rd_1), \ldots, (t_n, In_n, Out_n, Rd_n), \]

\[ (t'_{n+1}, In'_{n+1}, Out'_{n+1}, Rd'_{n+1}), \ldots \]

such that \( t'_{n+1} = t_n(x, i) \).

For composite networks \( N = \parallel (N_1, \ldots, N_k) \), the meaning function \([\cdot]\) is defined inductively by

\[ \sigma \in [\parallel (N_1, \ldots, N_k)] \iff \sigma|_{N_i} \in [N_i], \ 1 \leq i \leq k \]

where \( \sigma \) ranges over communication behaviors on the ports of \( N \).

A network can also be viewed as a process by "hiding" the internal structure represented by its internal and linked ports. The input and output ports of such a
process are just the external ports of the underlying network; its communication behaviors are the external communication behaviors of the network, defined as follows:

Definition 2.2.1.10: An external communication behavior of network $N$ is any behavior of the form $\sigma|_K$, where $\sigma \in \llbracket N \rrbracket$ and $K$ is the set of external ports of $N$.

In later sections we shall need the following notions.

Definition 2.2.1.11: Let $N$ be a network, and let $s = (t, In, Out, Rd)$ be an observation on $N$'s ports. Let $D$ be a nonempty set of components of $N$. In $s$, port $k$ of a member of $D$ is said to be disabled by $D$ if

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

Otherwise, $k$ is said to be enabled by $D$.

Definition 2.2.1.12: A communication behavior $\sigma$ is

- eventually constant if it has a constant suffix.
- eventually semi-constant if it has a suffix $\tau$ that is constant everywhere except on input ports, i.e. the only changes in $\tau$ are the addition of input
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 $\Psi$ is the liveness assumption then a process is specified by its $\Psi$-communication behaviors, i.e. communication behaviors that satisfy $\Psi$. To ensure that the set of $\Psi$-communication behaviors of a process is closed under finite repetition, we require that $\Psi$ itself be invariant under finite repetition, i.e. $\sigma$ satisfies $\Psi$ iff any $\tau$ obtained from $\sigma$ by finite repetition of observations satisfies $\Psi$. The results of this chapter all hold if communication behaviors are everywhere restricted to $\Psi$-communication behaviors.

2.2.2. The basic temporal proof system

2.2.2.1. Temporal logic and behaviors

We assume familiarity with temporal logic—see e.g. [MP81a]—and make only the following comments. The temporal operators include $\square$ (always), $\Diamond$ (eventually), $\mathcal{U}$ (until), $\mathcal{N}$ (unless) and $\mathcal{O}$ (next). Following [MP81a], 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 variables. But, unlike [MP81a], 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 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 example has the interpretation: if port $i$'s trace eventually has length $n$, then so does port $j$'s trace.

\[ \circ |i| = n \Rightarrow \circ |j| = n \]

A model $(I, \alpha, \sigma)$ for our language consists of a global interpretation $I$, a global assignment $\alpha$ and a sequence of states $\sigma$. 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 $\alpha$ assigns a value to each global free variable. The sequence $\sigma = s_1, s_2, \ldots$ 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. The truth value of a temporal formula or term $w$ (terms are defined just as in first order logic, except that they may contain the temporal operator $\bigcirc$), denoted by $w|_\sigma^I$, $I$ being implicitly assumed, is defined as follows:

- If $w$ is a classical term or formula (containing no temporal operator) then $w|_\sigma^I$ is the value of $w$ in $s_1$ under the assignment $\alpha$. 
\( (w_1 \lor w_2)_{[\alpha]} = T \) iff \( w_1_{[\alpha]} = T \) or \( w_2_{[\alpha]} = T \). Similarly for \( \land, \neg, \) etc...

\( \circlearrowleft w_{[\alpha]} = w_{[\alpha \circlearrowleft]} \). \( w \) can be a term or a formula.

\( \square w_{[\alpha]} = T \) iff for all \( k \geq 1 \), \( w_{[\alpha(k)]} = T \), i.e. \( \square w \) means \( w \) is always true.

\( \diamond w_{[\alpha]} = T \) iff there exists \( k \geq 1 \) such that \( w_{[\alpha(k)]} = T \), i.e. \( \diamond w \) means \( w \) will be true eventually.

\( (w_1 \lor w_2)_{[\alpha]} = T \) iff there exists \( k \geq 1 \) such that \( w_2_{[\alpha(k)]} = T \) and for all \( i, 1 \leq i < k \), \( w_1_{[\alpha(i)]} = T \), i.e. \( w_1 \lor w_2 \) means \( w_1 \) holds true continuously until \( w_2 \) becomes true, and \( w_2 \) does indeed become true.

\( (w_1 \land w_2)_{[\alpha]} = T \) iff \( \square w_1_{[\alpha]} = T \) or \( (w_1 \lor w_2)_{[\alpha]} = T \).

\( \forall x.w_{[\alpha]} = T \) iff for all \( d \in D \), \( w_{[\beta]} = T \), where \( \beta = \alpha[x \rightarrow d] \) is the assignment obtained from \( \alpha \) by assigning \( d \) to \( x \). (\( x \) is a global variable.)

\( \exists x.w_{[\alpha]} = T \) iff for some \( d \in D \), \( w_{[\beta]} = T \), where \( \beta \) 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 [MP81a, MP83].

We now define what it means for a communication 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:
• Assign to each local variable \( k \) the sequence \([a_1, \ldots, a_n]\), where
\([(a_1, k), \ldots, (a_n, k)]\) is the restriction of the trace of the observation to port \( k \).

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

• Assign to the local predicate symbol \(<\!\!\!\!\!\!\!\!\!\!\!<\) the "precedes" relation on the trace of the observation: \( (''h'', m) \!<\!\!\!\!\!\!\!\!\!\!\!<\!\!\!\!\!\!\!\!\!\!\<(''k'', n) \) iff the \( m \)th event on port \( h \) occurs before the \( n \)th event on port \( k \) in the trace. Thus \(<\!\!\!\!\!\!\!\!\!\!\!<\) is a total ordering.

2.2.2.2. Network specifications

We define a specification as follows:

Definition 2.2.2.2.1: A specification of a network \( N \) has the form

\[ <N> \ R \]

where \( R \) is a temporal assertion in which

• the only local free variables are names of \( N \)'s ports,
the only local function symbols are \textit{In}, \textit{Out} and \textit{Rd},

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

there is no occurrence of \(\text{In}(k)\) and \(\text{Rd}(k)\) (\(\text{Out}(k)\)) if \(k\) is an external output (input) port of \(N\). ■

The interpretation of specification \(<N>\ R\) is:

Every behavior of \(N\) satisfies \(R\).

A nice consequence of this interpretation is that, if the only free variables of \(R\) are the names of \(N\)'s external ports, the above interpretation is equivalent to:

\[\tag{2.2.2.2.1} \text{Every external behavior of } N \text{ satisfies } R.\]

This will be proved in later sections. In this case \(<N>\ R\) is called an \textit{external} specification.

Under liveness assumption \(\Psi\), the interpretation becomes:

Every \(\Psi\)-behavior of \(P\) satisfies \(R\).

In later sections, we will be dealing with precise specifications:

\textbf{Definition 2.2.2.2.2:} A specification \(<P>\ R\) is \textit{precise} if: every behavior on \(P\)'s port names is a behavior of \(P\) iff it satisfies \(R\). ■

\textbf{2.2.2.3. 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 \( \Psi \). We let \( 0^* \) \((0^*1)\) denote the set of all sequences consisting of a finite number of zeros (followed by a 1).

**Example 2.2.2.3.1:** Process \( \text{BUFF1} \) (one-slot buffer) iteratively reads input on port \( i \) and reproduces it on port \( j \).

The asynchronous specification of \( \text{BUFF1} \) is

\[
<\text{BUFF1}> \quad \square (j \subseteq i \land (\text{In}(i) = \lnot \text{Out}(j) = (|j| = \text{Rd}(i))))
\]
\[
\land \quad \forall m(\circ |i| = m \Rightarrow \circ \text{Rd}(i) = m)
\]
\[
\land \quad \forall n(\circ \text{Rd}(i) = n \Rightarrow \circ |j| = n)
\]

The synchronous specification of \( \text{BUFF1} \) is

\[
<\text{BUFF1}> \quad \square (j \subseteq i \land (\text{In}(i) = \lnot \text{Out}(j) = (|j| = |i|)))
\]

**Example 2.2.2.3.2:** Process \( \text{BUFF2} \) 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 \( \text{BUFF2} \) is

\[
<\text{BUFF2}> \quad \square (\neg \text{In}(i) \land \text{Out}(j) = j \in 0^*)
\]
\[
\land \quad \circ \square (j \in 0^{*}1)
\]

The synchronous specification of \( \text{BUFF2} \) is

\[
<\text{BUFF2}> \quad \square (\neg \text{In}(i) \land \text{Out}(j) = j \in 0^*)
\]
\[
\land \exists x \square (j \subseteq x \in 0^{*}1)
\]
Note that the synchronous specification for \textit{BUFF2} is invariant, but, in conjunction with appropriate specifications for a receiving process and the fairness assumption (2.2.2.6.1), it can be used to prove the liveness specification $\Diamond \Box (j \in 0^* 1)$.

2.2.2.4. Axioms for communication behaviors

Our basic proof system consists of the following six parts:

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

(2.2.2.4.2) Axioms and inference rules for temporal logic.

(2.2.2.4.3) Axioms that define the properties of communication behaviors (see below).

(2.2.2.4.4) 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.

(2.2.2.4.5) A set of primitive processes with precise specifications (see Definition 2.2.2.2.2).
(2.2.2.4.6) Proof rules to derive specifications of networks.

Parts (2.2.2.4.1) and (2.2.2.4.2) are standard and need no further comment. Part (2.2.2.4.4) describes the properties of $\Psi$-behaviors, thus capturing the liveness assumptions. We don’t deal with any particular liveness assumptions here. Part (2.2.2.4.5) defines the basic building blocks of networks of processes. Part (2.2.2.4.6) is given in Section 2.2.2.5.

Part (2.2.2.4.3) captures the notion of a communication behavior (see Definition 2.2.1.6). Now we give a complete set of axioms that a communication behavior $\sigma = s_1, s_2, \ldots$ must satisfy. Let $k_1, k_2, \ldots$ be the list of (local) port variables.

\[(2.2.2.4.7)\quad k = [\ ] \land \Box (k \subseteq \bigcirc k),\text{ where } k \text{ is a port variable, i.e. the initial trace is empty and, at all times, the next trace extends the current one.}\]

\[(2.2.2.4.8)\quad \Box (0 \leq (|\bigcirc k_1| - |k_1| + \ldots + |\bigcirc k_n| - |k_n|) \leq 1), \text{ for } n = 1, 2, \ldots, \text{ i.e. the next trace extends the current trace by at most one element.}\]

\[(2.2.2.4.9)\quad \Box (0 \leq Rd(k) \leq |k| \land Rd(k) \leq \bigcirc Rd(k) \leq Rd(k)+1) \text{ for } k \text{ an input port, i.e. the number of input events read on an input port can never decrease and is always at most the number of input events occurring on that port. For synchronous communication, the axiom becomes } \Box Rd(k) = |k| \text{ – hence the length function } Rd \text{ is not}\]
needed.

(2.2.2.4.10) □ ((Rd(k) ≠ □ Rd(k)) ⇒ In(k)) for k an input port, i.e. an event can be read only on an input port that is ready to communicate.

(2.2.2.4.11) □ ((k ≠ □ k) ⇒ Out(k)), for k an output port, i.e. an event can be written only on an output port that is ready to communicate.

(2.2.2.4.12) ∀ m ∀ n □ ((|k| ≥ m ∧ |l| < n ∧ |O| = n) ⇒ □ ("k", m) ≪ ("l", n))

i.e. the event that extends a trace occurs after all the existing ones in that trace (see the end of Section 2.2.2.1 for notation).

(2.2.2.4.13) ∀ m ∀ n □ (((k", m) ≪ ("l", n) ⇒ □ ((("k", m) ≪ ("l", n))

i.e. the ordering among the elements of a trace is preserved as the trace is extended.

It is clear that any communication behavior satisfies these axioms. Now let σ = s₁, s₂, ... be a sequence of states that satisfies these axioms. Each state can be interpreted as an observation by letting ≪ be the ordering on the trace, ln and Out be the communication functions, Rd be the length function, and the values of the port variables be the events of the trace. By induction on k, it is straightforward to show that each sₖ is a legitimate observation and that σ satisfies the properties of behaviors.
2.2.2.5. Basic proof rules

There are three proof rules in the basic proof system:

Renaming rule

\[ \langle N \rangle \quad R \]

\[ \quad \frac{}{\langle N[h_1 \ldots h_m / k_1 \ldots k_m] \rangle \quad R[h_1 \ldots h_m / k_1 \ldots k_m]} \]

where the notation \([h_1 \ldots h_m / k_1 \ldots k_m]\) indicates simultaneous substitution of port names, with the restriction that no new link is formed as a result of the substitution.

Network formation rule

\[ \langle N_k \rangle \quad R_k, \ k = 1, \ldots, n \]

\[ \frac{}{\langle || (\ldots, N_k, \ldots) \rangle \quad \land_k R_k} \]

Note that the \(N_k\) must satisfy the unique port name requirement of Definition 2.2.1.9 in order for their parallel composition to be sensible.

Consequence rule

\[ \langle N \rangle \quad R, \ R \Rightarrow S \]

\[ \frac{}{\langle N \rangle \quad S} \]

where \(R \Rightarrow S\) can be proved from the axioms and inference rules for temporal
logic, the axioms and inference rules for the data domain and the axioms that characterize behaviors.

2.2.2.6. Examples

Example 2.2.2.6.1: Consider the network in Figure 2.3. Process \( P1 \) reads nothing on \( k_1 \) and produces a 1 on \( k_2 \). Process \( P2 \) reads an input from \( k_2 \) and then 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 \( k_1 \). Suppose that the network is asynchronous. Then we have

\[
\begin{align*}
<P1> & \quad \Box \neg \text{in}(k_1) \land \diamond k_2 = [1] \\
<P2> & \quad \diamond |k_2| > 0 \Rightarrow \diamond k_1 = [1]
\end{align*}
\]

Figure 2.3
By the network formation rule, the network satisfies the conjunction of the above assertions. By the consequence rule, it follows that

\[
<\text{NETWORK}> \circ (k_2 = [1] \land k_1 = [1])
\]

Now suppose the network is synchronous and assume the liveness assumption is that of fairness: if a linked port is enabled infinitely often then eventually communication must take place

\[(2.2.2.6.1) \quad \square ((|k| = n \land \square \circ (\text{In}(k) \land \text{Out}(k))) \Rightarrow \circ |k| > n)\]

We have

\[
<\text{P1}> \quad \square \neg \text{In}(k_1) \land (\text{Out}(k_2) \land k_2 = [1])
\]

\[
<\text{P2}> \quad (\text{In}(k_2) \land |k_2| > 0) \land (\circ |k_2| > 0 \Rightarrow (\circ k_1 = [1] \lor \square \text{Out}(k_1)))
\]

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

\[
<\text{NETWORK}> \circ k_2 = [1] \land \square k_1 = [ ]
\]

Example 2.2.2.6.2: In [BA], 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
communication behaviors of the two networks.

In the example, \( l \oplus 1 \) 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 Figure 2.4. The precise specifications for the component processes are:

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

\[
\langle D1 \rangle \quad \Box j \in [i(1), i(1)] \\
\land (\Diamond |i| = u \Rightarrow \Diamond |j| = 2 \times \min(u, 1))
\]

Figure 2.4 The Brock-Ackerman Example
D2 reads one value on m and writes it twice on n:

\[<D2> \quad \Box n \subseteq [m(1), m(1)] \]
\[\wedge (\varnothing |m| = u \Rightarrow \varnothing |n| = 2 \cdot \text{min}(u, 1))\]

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

\[<\text{MERGE}> \quad \Box \text{preshuffle}(j, n, k) \]
\[\wedge (\varnothing (|j| = u \wedge |n| = v) \Rightarrow \varnothing |k| = u + v)\]

where presuffle\((j, n, k)\) means that k is a prefix of an element of shuffle\((j, n)\).

shuffle is defined as

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

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

\[<\text{P1}> \quad \Box l \subseteq [k(1), k(2)] \]
\[\wedge (\varnothing |k| = u \Rightarrow \varnothing |l| = \text{min}(u, 2)) \]
\[\wedge \Box h \subseteq [k(1), k(2)] \]
\[\wedge (\varnothing |k| = u \Rightarrow \varnothing |h| = \text{min}(u, 2)) \]

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

\[<\text{PLUS1}> \quad \Box m \subseteq k \oplus 1 \]
\[\wedge (\varnothing |l| = u \Rightarrow \varnothing |m| = u) \]
Applying the network formation rule, we obtain

\[ <\text{NETWORKS}> R \]

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

\[ \square (j \subseteq [i(1), \text{i}(1)] \land n \subseteq [m(1), m(1)] \land m \subseteq l \oplus 1) \]

it follows that

\[ R \Rightarrow (\square l \subseteq [k(1), k(2)]) \]
\[ \land \square \text{preshuffle}([i(1), i(1)], [l(1) + 1, l(1) + 1], k)) \]

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

\[ R \Rightarrow \square (k \subseteq [i(1), i(1)] \lor k \subseteq [i(1), i(1) + 1]) \]
\[ R \Rightarrow \square (l \subseteq [i(1), i(1)] \lor l \subseteq [i(1), i(1) + 1]) \]

Similarly, we have

\[ R \Rightarrow \square (h \subseteq [i(1), i(1)] \lor h \subseteq [i(1), i(1) + 1]) \]

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

\[ |i| = u \]
\[ |j| = 2 \times \min(|i|, 1) \]
\[ |n| = 2 \times \min(|m|, 1) \]
\[ |k| = |j| + |n| \]
\[ |l| = \min(|k|, 2) \]
\[ |h| = \min(|k|, 2) \]
\[ |m| = |l| \]

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 \( \mathbb{N} \cup \{0\} \) with \(<\) as the partial order—by the usual least fixed point method (e.g. [Ka])—to yield the following least solution:

\[
\begin{align*}
|l| &= u \\
|j| &= 2 \times \min(1, u) \\
|n| &= 2 \times \min(1, u) \\
|k| &= 4 \times \min(1, u) \\
|l| &= 2 \times \min(1, u) \\
|h| &= 2 \times \min(1, u) \\
|m| &= 2 \times \min(1, u)
\end{align*}
\]

From this, we get the following external specification for \textsc{Network1}:

\[
<\textsc{Network1}> \quad \square (h \sqsubseteq [i(1), i(1)] \lor h \sqsubseteq [i(1), i(1) + 1]) \\
\land (\circ |i| = u \Rightarrow \circ |h| = 2 \times \min(1, u))
\]

Now consider the same network with \textit{P1} replaced by \textit{P2}, where \textit{P2} has the following specification:

\textit{P2} reads 2 values from \( k \) and then writes them on \( h \) and \( l \):

\[
<\textit{P2}> \quad \square l \sqsubseteq [k(1), k(2)] \land \square |l| \leq 2 \times \min(1, |k| - 1) \\
\land (\circ |k| = u \Rightarrow \circ |l| = 2\times\min(1, u - 1)) \\
\land \square h \sqsubseteq [k(1), k(2)] \land \square |h| \leq 2 \times \min(1, |k| - 1)
\]
\( \land (\diamond |k| = u \Rightarrow \diamond |h| = 2*\min(1,u - 1)) \)

where \( a - b \) 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 external specification

\[
\langle \text{NETWORK2} \rangle \quad \square h \subseteq [i(1), i(1)] \\
\land (\diamond |i| = u \Rightarrow \diamond |h| = 2 \cdot \min(1, u))
\]

A communication 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 communication 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 communication behaviors.

2.2.2.7. Soundness and completeness

Soundness and completeness are defined as follows.

Let \( L \) be a temporal assertion language whose only local function symbols are \( \text{In}, \text{Out} \) and \( \text{Rd} \), 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 \textit{primitive} processes, from which networks of processes are to be formed.

**Definition 2.2.2.7.1:** With \( L, I, \{P_i\} \) as above, define \( L \) to be \textit{expressive relative to} \( I \) and \( \{P_i\} \) if for every primitive process \( P_i \) there exists an assertion \( R_i \) such that \( \langle P_i \rangle R_i \) is a precise specification. We denote this by \( I \in E(L, \{P_i\}) \).

**Definition 2.2.2.7.2:** A temporal proof system is \textit{sound} if, for every \( I \in E(L, \{P_i\}) \), every specification \( \langle P \rangle R \) that is provable (with all the \( \langle P_i \rangle R_i \) 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 \( P \) in \( I \) satisfies \( R \)). The proof system is \textit{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 [Ap]).

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

**Lemma 2.2.2.7.1:** Let \( I \) and \( J \) be sets of port names and \( R \) an assertion in which

- the only free variables are (local) port variables in \( I \cup J \),
• there is no occurrence of \(\text{In}(j)\) and \(\text{Rd}(j)\) for \(j \in J - I\), and

• there is no occurrence of \(\text{Out}(i)\) for \(i \in I - J\).

Then for any communication behaviors \(\sigma\) and \(\tau\),

\[
\sigma|_{I,J} = \tau|_{I,J} \quad \text{implies} \quad \sigma \text{ satisfies } R \iff \tau \text{ satisfies } R,
\]

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:

\[
\sigma|_{I,J} = \tau|_{I,J} \quad \text{implies} \quad \sigma^{(k)} \text{ satisfies } R \iff \tau^{(k)} \text{ satisfies } R, \text{ for all } k.
\]

Note that the induction hypothesis implies the lemma.

Consider the structure of \(R\).

• \(R\) is an atomic formula. Let \(s_k\) and \(t_k\) be the \(k^{th}\) elements of \(\sigma\) and \(\tau\).

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.

• \(R\) 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 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 [CH, Ho81, MC, MCS].

Now, it is easy to see why the remark surrounding (2.2.2.2.1) concerning interpretations of \( \langle P \rangle \ R \) is 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.

**Theorem 2.2.2.7.1:** 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 \( \sigma = s_1, s_2, \ldots \) be a behavior of \( \| (\ldots, N_k, \ldots) \). By our model of behaviors, \( \sigma_{\mid N_k} \) is a behavior of \( N_k \), \( k = 1, \ldots, n \). Hence \( \sigma_{\mid N_k} \) satisfies \( R_k \) for \( k = 1, \ldots, n \). By the non-interference property, \( \sigma \) satisfies \( R_k \), for \( k = 1, \ldots, n \). This is true for all \( k \). Therefore \( \sigma \) satisfies \( \land_k R_k \). 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 \(k = 1, \ldots, n\) then \(<\| (\ldots, N_k, \ldots)> \land_k R_k\) is also precise. Let \(\sigma = s_1, s_2, \ldots\) be a behavior on \(\| (\ldots, N_k, \ldots)\)'s ports that satisfies \(\land_k R_k\). For each \(k\), \(\sigma\) satisfies \(R_k\). So \(\sigma|_{N_k}\) must satisfy \(R_k\), \(k = 1, \ldots, n\), by the non-interference property. By preciseness of \(<N_k> R_k\), \(\sigma|_{N_k}\) is a behavior of \(N_k\). Hence \(\sigma\) must be a behavior of \(\| (\ldots, N_k, \ldots)\). Conversely, if \(\sigma\) is a behavior of \(\| (\ldots, N_k, \ldots)\), then \(\sigma\) must satisfy \(\land_k R_k\), by the soundness of the network formation rule.

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

2.2.2.8. 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 [CH], a specification of process $P$ has the form $P \text{ sat } R$, 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 [MC], 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 $<H> \ S \land \lnot(R \cup \lnot S)$ in our system. According to the interpretation of temporal formulas, $R \cup \lnot S$ is true iff $\exists k \geq 1$ such that $\lnot S$ is true in $s_k$ and for all $i$, $1 \leq i < k$, $R$ is true in $s_i$ (for $R$ and $S$ are classical formulas). So $S \land \lnot(R \cup \lnot S)$ is true iff $S$ is true in $s_1$ and for all $k \geq 1$, 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 $s_1$ and for all $k \geq 1$, 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 $s_1, \ldots, s_i$ 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.

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

2.3. Termination and deadlock

The difficulty in modeling termination and deadlock is that 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.

2.3.1. 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 communication behavior "appears" terminated. Formally,

**Definition 2.3.1.1:** A $T$-behavior is a pair $(t, \sigma)$, where $t \in \{T, F\}$ and $\sigma$ is a communication behavior, such that if $t = T$ and the communication is synchronous (asynchronous) then $\sigma$ is eventually constant (eventually semi-constant), converging on an observation in which every port of the network is disabled by
every network component it belongs to. ■

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

**Definition 2.3.1.2:** As in definition 2.2.1.9, assume $[P]_T$ is given for each primitive process $P$. For composite network $N = \| (N_1, \ldots, N_m)$, we define $[N]_T$ as follows:

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

- $t = \land_i t_i$, and
- $\sigma|_{N_i} = \sigma_i$.

Thus a network communication behavior terminates iff each of its component communication behaviors terminates. ■

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:

**Network formation rule (termination)**

\[
<\langle N_i \rangle > R_i, \quad i = 1, \ldots, n
\]

\[
\begin{array}{c}
<\|_{i} N_i > \exists t_{N_1} \ldots \exists t_{N_n} ((\land_i R_i) \land t|_{N_i} = \land_i t_{N_i})
\end{array}
\]
Theorem 2.3.1.1: The proof system is sound and relatively complete.

Proof:

Soundness:

Let \((t, \sigma) \in \llbracket \llbracket \llbracket N_i \rrbracket \rrbracket_T\). Then, \((t, \sigma)\) results from combining T-behaviors \((t_1, \sigma_1), \ldots, (t_n, \sigma_n)\), where \((t_i, \sigma_i) \in \llbracket N_i \rrbracket_T, i = 1, \ldots, n\). Hence, \((t_i, \sigma_i)\) satisfies \(R_i\). Equivalently, \(\sigma_i\) satisfies \(R_i[t_i / t_{N_i}]\) \((R_i\) with all free occurrences of \(t_{N_i}\) replaced by \(t_i\)). By the non-interference property (Lemma 2.2.2.7.1), it follows that \(\sigma\) satisfies \(\land_i R_i[t_i / t_{N_i}]\). Thus, \((t, \sigma)\) satisfies the specification of \(\llbracket \llbracket N_i \rrbracket\) given in the proof rule, and the proof rule is sound.

Completeness:

We first prove that the proof rule preserves preciseness of specifications. Assume \(<N_i> R_i\) are precise. Let \((t, \sigma)\) satisfy the specification of \(\llbracket \llbracket N_i \rrbracket\) given in the proof rule. Then \(\sigma\) satisfies \(\exists t_{N_1} \ldots \exists t_{N_i} (\land_i R_i \land t = \land_i t_{N_i})\). So \(\sigma\) satisfies \(\land_i R_i[t_i / t_{N_i}] \land t = \land_i t_i\) 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, \((t_i, \sigma|_{N_i}) \in \llbracket \llbracket N_i \rrbracket_T\). Therefore, \((t, \sigma) \in \llbracket \llbracket \llbracket N_i \rrbracket_T\). Conversely, if \((t, \sigma) \in \llbracket \llbracket \llbracket N_i \rrbracket_T\), then it satisfies the specification of \(\llbracket \llbracket N_i \rrbracket\) by soundness of the rule. Thus, the proof rule preserves preciseness.
Now, let \(<N> R\) be a specification that is true and let \(N\) be built from primitive components \(P_1, \ldots, P_n\) by network formation. From the network formation rule and precise specifications of \(P_i\)'s we obtain a precise specification \(<N> S\).

So \(S \Rightarrow R\) is satisfied by every pair \((t, \sigma)\), where \(\sigma\) is a behavior on \(N\)'s ports. Consider any T-behavior \((t', \sigma')\). \((t', \sigma'|_N)\) satisfies \(S \Rightarrow R\). Hence, \(\sigma'|_N\) satisfies \((S \Rightarrow R)[t' / t_N]\). By the non-interference property, \(\sigma'\) satisfies it, too. So \((t', \sigma')\) satisfies \(S \Rightarrow R\). \(S \Rightarrow R\) is satisfied by every T-behavior. Hence \(<N> R\) is provable from \(<N> S\) 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.

Therefore, the proof system is relatively complete.  ■

2.3.2. Deadlock

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 now 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 ports of members of $D$ that are linked or external ports of the network 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:

Definition 2.3.2.1: A $D$-behavior is a triple $(d, w, \sigma)$, where $d, w \in \{T, F\}$ and $\sigma$ is a communication behavior, such that if $w = T$ and the communication is synchronous (asynchronous) then $\sigma$ is eventually constant (eventually semi-constant), converging on an observation in which all internal and linked ports are disabled by the network.

We remark that the condition “all internal and linked ports 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 cannot be in a wait state, since it can clearly change state (by sending a message on the enabled internal or linked port) without an external communication event taking place.

The meaning of network $N$ is now a set $\llbracket N \rrbracket_D$ of $D$-behaviors:

Definition 2.3.2.2: As in definition 2.2.1.9, assume $\llbracket P \rrbracket_D$ is given for each primitive process $P$. For composite network $N = \| (N_1, \ldots, N_m)$, we define $\llbracket N \rrbracket_D$ as follows:
A D-behavior \((d, w, \sigma)\) on \(N\)'s ports is in \([N]_D\) iff there exist 
\((d_i, w_i, \sigma_i) \in [N_i]_D\) such that 

- \(w = T\) iff \(w_i = T\) for all \(i\), and eventually all linked ports in \(N\) are disabled by \(N\) forever; and 
- \(d = T\) iff either \(d_i = T\) for some \(i\), or there exists a nonempty subset \(D\) of components of \(N\) such that 
  
  (a) \(w_i = T\) for each \(i\) such that \(N_i \in D\); and 

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

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

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 of its component processes get into deadlock —i.e., the set $D$ in Definition 2.3.2.2 above consists of all the components of $N$. This is the notion of deadlock treated in [Br]. Using a wait bit and deadlock bit as above, the rule for forming the deadlock bit in Definition 2.3.2.2 above now becomes

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

- $(a') w_i = T$ for all $i$'s, and

- $(b')$ eventually all linked and external ports of the network are disabled by the network forever.

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

- $d = T$ iff $(a')$ and $(b')$ hold.

In fact, we do not even need a deadlock bit, since $(a')$ and $(b')$ do not mention $d$. Now it is straightforward to prove that $\| (P_1, \ldots, P_n)$ is totally deadlocked iff $\| (\| (P_1, P_2), \ldots, P_n)$ 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
Network formation rule (deadlock)

\[ <N_i> R_i, \ i = 1, \ldots, n \]

\[ <||_i N_i > \ \exists d_{N_1} \ldots \exists d_{N_n} \ \exists w_{N_1} \ldots \exists w_{N_n} \]

\[ (((\land_i R_i) \land w_{||_i N_i}) = ((\land_i w_{N_i}) \land \diamond \ \square \ \text{disabled}(||_i N_i)) \]

\[ \land d_{||_i N_i} = ((\lor_i d_{N_i}) \lor (\lor_{D \in A} dlck(D))) \]

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

\[ dlck(D) = ((\land_{N_i \in D} w_{N_i}) \land \square \ \text{inactive}(D)) \]

where \( \text{disabled}(||_i N_i) \) means that all linked ports of \( ||_i N_i \) are disabled by \( ||_i N_i \), and \( \text{inactive}(D) \) means that all ports of members of \( D \) that are linked or external ports of \( ||_i N_i \) are disabled by \( D \). It is clear how these formulas are expressed in temporal logic.

**Theorem 2.3.2.1:** The proof system is sound and relatively complete.

**Proof:** The proof of soundness and completeness of this rule is similar to that of Theorem 2.3.1.1.  

2.4. Procedural and recursive networks

We now define procedural networks, in which certain components do not begin execution until they are activated by neighboring components. A restricted but useful class of infinite networks, in which only finitely many processes can be
active at any time, can be defined using procedural networks. This leads naturally to the notion of recursive network, a useful abstraction that can model constructs of languages such as Concurrent Prolog [ST] and the parallel language of [KM].

2.4.1. Procedural networks and subroutine components

Informally, a procedural network is one in which certain components, designated as subroutines, may not execute until activated externally. This restriction on the behaviors of subroutine components makes possible a simple definition of infinite procedural network, in which all but finitely many components are subroutines.

Definition 2.4.1.1: A procedural network description is either

- an (ordinary) network description, or
- a procedural composition of the form $\| (M_1, \ldots, M_m ; Q_1, Q_2, \ldots)$, in which each of finitely many main components $M_1, \ldots, M_m$ and each of perhaps infinitely many subroutine components $Q_1, Q_2, \ldots$ is a procedural network description.

For composite procedural network $N$, we impose the same unique port naming requirement as in Definition 2.2.1.9 for ordinary networks, i.e.

- The sets of input (output) port names of distinct components of $N$ must be disjoint.
The *input*, *output*, *internal*, *linked* and *external* ports of $N$ are also defined as for ordinary networks. Finally, we require that

- All ports of subroutine components of $N$ must be internal or linked in $N$.

That is, no subroutine component of $N$ can be connected to an external port of $N$.

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 without having been activated by some neighboring process attempting to communicate with it. To formalize this notion we define activation and execution of network components in terms of communication behaviors.

**Definition 2.4.1.2**: Let $N$ be a procedural network and $Q$ be a subroutine component of $N$. Let $I$ and $J$ be all the input and output ports of $Q$, and $I'$ and $J'$ be the external input and output ports of $Q$. Then predicates $\text{act}(Q)$ and $\text{inert}(Q)$ are defined as follows.
\[
act(Q) = (\bigvee_{i \in \mathcal{I}} \text{Out}(i)) \lor (\bigvee_{j \in \mathcal{I}} \text{In}(j))
\]

\[
inert(Q) = (\bigwedge_{i \in \mathcal{I}} \text{In}(i)) \land (\bigwedge_{j \in \mathcal{I}} \text{Out}(j))
\]

Finally, let \(s\) be an observation of \(N\). \(Q\) is said to be activated (inert) in \(s\) if \(s\) satisfies \(\act(Q)\) (\(\inert(Q)\)).

Intuitively, \(Q\) is activated in \(s\) means that some neighbor of \(Q\) in \(N\) is ready to communicate with \(Q\). \(Q\) is inert in \(s\) means that \(Q\) is not ready to communicate on any of its ports—including its linked and internal ports.

**Definition 2.4.1.3:** Let \(N\) be a procedural network. Then \([N]\) is a set of behaviors on the ports of \(N\) defined as follows:

- if \(N\) is an ordinary network, then \([N]\) is given by Definition 2.2.1.9.

- if \(N\) is a composition \(\| (M_1, \ldots, M_m ; Q_1, Q_2, \ldots)\), then a behavior \(\sigma = s_1, s_2, \ldots\) on \(N\)'s ports is in \([N]\) iff for every main component \(M_i\) of \(N\)

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

and for every subroutine component \(Q_i\) of \(N\), either

(a) \(Q_i\) is never activated and is always inert (i.e. never begins execution), or

(b) there exist positive integers \(p \leq q\) such that

(i) \(Q_i\) is activated in \(s_p\),
(ii) $Q_i$ is inert in $s_j$ for all $j \leq p$ and $j < q$, and

(iii) $s^{(q)}|_{Q_i} \in \text{[Q]}$;

that is, $Q_i$ is eventually activated and begins execution.  ■

Thus, a subroutine component $Q_i$ does not begin execution until a neighboring component activates it. Once activated, however, $Q_i$ must execute. Note that, before $Q_i$ begins execution, its observations are empty (trace empty and communication functions everywhere false). The point at which $Q_j$ begins execution is subtle. If the empty observation is a valid initial observation of $Q_j$, then execution can be thought of as beginning at the point where $Q_j$ is activated — since nothing needs to be done to initiate the execution. Otherwise, execution of $Q_j$ begins sometime after the point of activation. Technically, this assumption helps to preserve the finite repetition property.

The requirement in Definition 2.4.1.1 that all ports of subroutine components must be internal or linked ensures that each communication behavior of a procedural network uniquely determines whether its subroutine components are activated. The requirement that there are only finitely many main components ensures that, even in an infinite procedural network, only finitely many components have been activated at any time.

Because a procedural network may contain infinitely many components, a complete proof rule requires the use of infinitary logical operators. For convenience, we introduce the temporal operator $\mathcal{U}$ (strict until), where $\mathcal{U}$ is defined in
terms of other operators as follows

\[ p \sqcup q =_{\text{def}} p \land (p \lor q) \]

Note that we do not increase the complexity of our temporal logic here, since the new temporal operator is expressible in terms of the existing ones. The network formation rule for procedural networks is as follows:

Procedural network formation rule

\[
\begin{align*}
\langle M_i \rangle & R_i, \ 1 \leq i \leq m \\
\langle Q_j \rangle & S_j, \ 1 \leq j
\end{align*}
\]

\[
\langle \parallel (M_1, \ldots, M_m ; Q_1, Q_2, \ldots) \rangle > \land_i R_i \\
\land \land_j ((\neg \text{act}(Q_j) \land \text{inert}(Q_j)) \\
\land (\text{act}(Q_j) \land (\text{inert}(Q_j) \lor S_j)))
\]

Theorem 2.4.1.1: The proof system for procedural networks is sound and relatively complete.

Proof:

Soundness:

It is easy to see that the non-interference property (Lemma 2.2.2.7.1) still holds. The proof of soundness follows from this straightforwardly.

Completeness:

To prove relative completeness, it is sufficient to prove that the procedural network formation rule preserves preciseness of specifications.
Let \(<M_i> R_i\) and \(<Q_j> S_j\) be precise specifications, and let \(\sigma = s_1, s_2, \ldots\) be a communication behavior on the ports of \(\| (M_1, \ldots, M_m ; Q_1, Q_2, \ldots)\) that satisfies the procedural network's specification in the proof rule. For every main component \(M_i\), \(\sigma|_{M_i}\) satisfies \(R_i\), by non-interference. So \(\sigma|_{M_i} \in [M_i]\). By definitions of \(\mathcal{N}\) and \(\mathcal{U}\), for every subroutine component \(Q_j\), either

- \(Q_j\) is never activated and always inert in \(\sigma\), or

- \(Q_j\) is activated in \(s_p\), for some \(p\), and \(Q_j\) is inert in \(s_k\) for all \(k \leq p\) and \(k < q\), for some \(q > p\), and \(\sigma(q)|_{Q_j}\) satisfies \(S_j\). By preciseness of \(<Q_j> S_j\), \(\sigma(q)|_{Q_j} \in [Q_j]\).

From Definition 2.4.1.3, it follows that \(\sigma\) is a communication behavior of the procedural network. Conversely, if \(\sigma\) is a behavior of the procedural network, then \(\sigma\) satisfies the network's specification in the proof rule, by the soundness of the rule. Hence the proof rule is precise. \(\square\)

2.4.2. Recursive networks

Informally, a procedural network is recursive if some of its subroutine components are designated as "recursive copies" of itself. To make the exposition clearer, we restrict ourselves to the case in which the recursive network has a single recursive copy and no other subroutine component. Relaxing this restriction is a tedious but straightforward exercise.
Definition 2.4.2.1: A recursive network description is

\[ X(i_1, \ldots, i_m \ ; j_1, \ldots, j_n) = \parallel (N_1, \ldots, N_i \ ; X(h_1, \ldots, h_m \ ; k_1, \ldots, k_n)) \]

where, except for the fact that \( X \) is not a primitive process name,

- the left-hand side and \( X(\ldots h_r \ldots \ ; \ldots k_s \ldots) \) are primitive process descriptions,

- the right-hand side is a procedural network description, and

- the two sides have the same external input and output ports. □

See Figure 2.5 for an example.

We shall define the behaviors of a recursive network to be the behaviors of the infinite procedural network obtained by “unrolling” the recursive definition in a natural way. To define the unrolling operation requires a uniform method of obtaining new port names. For any port \( k \), we define

![Figure 2.5](image-url)  

Figure 2.5 A recursive network
- $k^0 = k$, and

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

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

Given recursive network description

$$X(i_1, \ldots, i_m; j_1, \ldots, j_n) = || (N_1, \ldots, N_i; X(h_1, \ldots, h_m; k_1, \ldots, k_n))$$

as above, we define a sequence $G_r(X), \ r = 1, 2, \ldots$ by:

$$G_r(X) = (|| (N_1, \ldots, N_i; ))^r [h_1^{r-1} \ldots h_m^{r-1} k_1^{r-1} \ldots k_n^{r-1} / i_1^r \ldots i_m^r j_1^r \ldots j_n^r]$$

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 Figure 2.6.

![Diagram](image)

**Figure 2.6** $G_1(X), \ G_2(X)$
The "completely unrolled" infinite procedural network for $X$ is:

$$F(X) = \| (N_1, \ldots, N_i ; G_1(X), G_2(X), \ldots)$$

Such a network is depicted in Figure 2.7. The behaviors of $X$ are defined in terms of $F(X)$:

**Definition 2.4.2.2:** Let $X$ and $F(X)$ be as above; then we define $[X] = [F(X)]$.

The reader may find our unrolling process slightly unconventional. However, 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. Such a network would be inconvenient for technical reasons. Its equivalence to our "flattened" network can be seen from the following:

![Diagram](image.png)

*Figure 2.7 X*
Lemma 2.4.2.1: For procedural networks $N_i, M_j$ and $Q$,

\[
[\parallel (M_1, \ldots, M_m ; \parallel (N_1, \ldots, N_n ; Q)) ]
= [\parallel (M_1, \ldots, M_m ; \parallel (N_1, \ldots, N_n ; ), Q) ]
\]

provided these compositions satisfy the requirements for unique port names.

Proof: A direct application of Definition 2.4.1.3. ■

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

Recursive network formation rule

\[
<\parallel (N_1, \ldots, N_t ; )> R
\]

\[
<\text{X}> R \land
\land_n ((\neg \text{act}(G_n(X)) \land \text{inert}(G_n(X)))
\land (\text{act}(G_n(X)) \land (\text{inert}(G_n(X)) \lor R_n)))
\]

where $<G_n(X)> R_n$ is obtained from $<\parallel (N_1, \ldots, N_t ; )> R$ by a suitable application of the renaming rule.

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.
2.4.3. Examples

Example 2.4.3.1: Consider synchronous recursive network \textit{PRIME} shown in Figure 2.8. Process \( R \) produces the first input from \( i \) on \( j \) and then all the inputs from \( m \) on \( j \). At the same time, \( R \) produces on \( l \) those inputs from \( i \) that are not divisible by the first input from \( i \).

A formal specification for \( R \) is

\[
\langle R \rangle \quad \Box S \land T
\]

where

\[
S = j \subseteq \text{first}(i).m \land l \subseteq \text{indiv}(i^{(2)}, \text{first}(i))
\]

\[
T = \Box (\text{In}(i) \land \text{In}(m))
\]

\[
\land \Box |\text{first}(i).m| = n \Rightarrow (\Box |j| = n \lor \Box \Box \text{Out}(j))
\]

\[
\land \Box |\text{indiv}(i^{(2)}, \text{first}(i))| = n \Rightarrow (\Box |l| = n \lor \Box \Box \text{Out}(l))
\]
and \( \text{indiv}(s, a) \) is the subsequence of \( s \) containing those elements that are not divisible by \( a \).

We want to prove

\[
<\text{PRIME}> \quad \lozenge i \subseteq \text{ODDNUM} \Rightarrow
\]

\[
(\lozenge j \subseteq \text{ODDPRIME} \land (\diamond \mid \text{prime}(i) \mid = n
\Rightarrow (\lozenge \mid j \mid = n \lor \lozenge \diamond \text{Out}(j))))
\]

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

By the renaming rule, we obtain

\[
<\text{G}_n(\text{PRIME})> \quad \lozenge S_n \land T_n
\]

where

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

\[
T_n = T[l^{n-1}, m^n, m^{n-1}, l^n / i, m, j, l]
\]

By the proof rule for recursive networks, we have

\[
<\text{PRIME}> \quad \lozenge S \land T \land
\]

\[
\land_n ((\neg \text{act}(G_n(\text{PRIME})) \land \text{inert}(G_n(\text{PRIME})))
\land (\text{act}(G_n(\text{PRIME})) \land (\text{inert}(G_n(\text{PRIME})) \lor (\lozenge S_n \land T_n))))
\]

See Figure 2.9.
Safety

We first prove the safety specification

\[ <PRIME> \; \Box i \subseteq ODDNUM \Rightarrow \Box j \subseteq ODDPRIME \]

From \( <R> \; \Box S \), 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

\[ <PRIME> \; \Box S \land \land_n \Box S_n \]

Since

\[
S \Rightarrow j \subseteq first(i).m \land l \subseteq indiv(i^{(2)}, first(i)) \\
S_1 \Rightarrow m \subseteq first(l).m^1 \land l^1 \subseteq indiv(l^{(2)}, first(l)) \\
S_2 \Rightarrow m^1 \subseteq first(l^1).m^2 \land l^2 \subseteq indiv(l^{(2)}, first(l^1)) \\
... 
\]

It follows that \( PRIME \) satisfies at all times
\[ j \subseteq \text{first}(i).m \]
\[ \subseteq \text{first}(i).\text{first}(l).m^1 \]
\[ \subseteq \text{first}(i).\text{first}(l).\text{first}(l^1).m^2 \]

By induction, we obtain

- \( \Box (j \subseteq \text{first}(i).\text{first}(l).\text{first}(l^1).\text{first}(l^2)...) \)

- If \( \Box i \subseteq \text{ODDNUM} \) then \( \text{first}(i), \text{first}(l) \) are the first two primes and \( \text{first}(l^n) \) is the \( (n+2) \)th prime in \( i \).

Hence \text{PRIME} satisfies the safety specification

\[ <\text{PRIME}> \Box i \subseteq \text{ODDNUM} \Rightarrow \Box j \subseteq \text{ODDPRIME} \]

Liveness

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

\[ \Box ((|k| = n \land \Box \Diamond (\text{In}(k) \land \text{Out}(k))) \Rightarrow \Diamond |k| > n) \]

We now prove the liveness specification

\[ <\text{PRIME}> \Box i \subseteq \text{ODDNUM} \Rightarrow \\
(\Diamond |\text{prime}(i)| = n \Rightarrow (\Diamond |j| = n \lor \Box \Diamond \text{Out}(j))) \]

Since \( <R> \Box \text{In}(m) \), \( G_1(\text{PRIME}) \) is activated as soon as \( R \) starts executing.

Similarly, all the \( G_n(\text{PRIME}) \) are activated eventually. Hence the original specification of \text{PRIME} can be simplified to
\[ <\text{PRIME}> \quad \square S \land T \land \]
\[ \land_n \diamond (\square S_n \land T_n) \]

Assume \( \square i \subseteq \text{ODDNUM} \). By the fairness assumption, from specification \( T \) and \( T_1 \) we obtain

\[ \diamond |\text{indiv}(i^{(2)}, \text{first}(i))| = n \Rightarrow \diamond |l| = n \]

Since \( \square l \subseteq \text{indiv}(i^{(2)}, \text{first}(i)) \), it follows that

\[ \diamond |\text{prime}(i^{(2)})| = n \Rightarrow \diamond |\text{prime}(l)| = n \]

By a similar argument, from \( T_1 \) and \( T_2 \) we obtain

\[ \diamond |\text{prime}(l^{(2)})| = n \Rightarrow \diamond |\text{prime}(l^1)| = n \]

\[ \ldots \]

Hence

\[ \diamond |\text{prime}(i^{(2)})| = n \Rightarrow \diamond |l| = n \Rightarrow \diamond |l^1| = n - 1 \]

\[ \Rightarrow \ldots \Rightarrow \diamond |l^{n-1}| = 1 \]

\[ \Rightarrow \diamond |\text{first}(l) \cdot \text{first}(l^1) \ldots \text{first}(l^{n-1}) \cdot m^n| = n \]

The fairness assumption and specifications \( T_i \) also imply

\[ \diamond |\text{first}(l) \cdot \text{first}(l^1) \ldots \text{first}(l^{n-1}) \cdot m^n| = n \]

\[ \Rightarrow \diamond |\text{first}(l) \cdot \text{first}(l^1) \ldots \text{first}(l^{n-2}) \cdot m^{n-1}| = n \]

\[ \Rightarrow \ldots \Rightarrow \diamond |\text{first}(l) \cdot m^1| = n \Rightarrow \diamond |m| = n \]

Therefore
\[ \circ |\text{prime}(i^{(2)})| = n \Rightarrow \circ |m| = n \]

By this and specification \( T \), we have for \( n > 0 \)

\[ \circ |\text{prime}(i)| = n \Rightarrow \circ |\text{prime}(i^{(2)})| = n - 1 \]

\[ \Rightarrow \circ |m| = n - 1 \Rightarrow \circ |\text{first}(i).m| = n \]

\[ \Rightarrow (\circ |j| = n \lor \Box \circ \text{Out}(j)) \]

The case \( n = 0 \) is trivial. Hence \( \text{PRIME} \) satisfies the required liveness specification.

2.5. Sequential processes

A similar technique is used for modeling sequential processes—i.e. processes that are built from conventional sequential constructs. For this purpose, we introduce the notions of state and state behavior that are analogous to the notions of observation and communication behavior for concurrent processes.

A temporal proof system is also defined on this model. To allow simple and complete proof rules, new temporal operators that are similar to those of [BKP, HKP] are introduced.

2.5.1. A behavior model for sequential processes

The syntax of sequential processes is as follows.

Definition 2.5.1.1: A sequential process description is
• *SKIP*, or

• $x := e$ (assignment), where $x$ is a variable and $e$ is an expression, or

• $S_1 ; S_2$ (sequential composition), where $S_1$ and $S_2$ are sequential process descriptions, or

• if... $p_k \rightarrow S_k$... fi (nondeterministic choice), where $p_k$ are Boolean expressions and $S_k$ are sequential process descriptions, or

• while $p$ do $S$ (while loop), where $p$ is a Boolean expression and $S$ is a sequential process description.

To describe the semantics of sequential processes, we introduce the following notions.

**Definition 2.5.1.2:** A *state* on a set $V$ of variables is a total function $V \rightarrow D$, where $D$ is some domain of values.

**Definition 2.5.1.3:** A *state behavior* on a set $V$ of variables is a pair $(t, \sigma)$, where $t \in \{T, F\}$ and $\sigma$ is an infinite sequence of states on $V$ such that any two consecutive states in the sequence assign different values to at most one variable. Further, if $t$ then $\sigma$ is an eventually constant sequence.

**Definition 2.5.1.4:** The *restriction of a state* $s$ to a set $V$ of variables is the function obtained from $s$ by restricting the domain of $s$ to $V$. The *restriction of a state behavior* $(t, \sigma)$ to $V$ is defined similarly and is denoted by $(t, \sigma|_V)$. 


Intuitively, a state behavior is the sequence of states produced by some execution of a sequential process as time progresses. \( t = T \) means that the process eventually terminates under \( \sigma \).

For \( \sigma = s_1, \ldots, s_n, s_n, \ldots \) an eventually constant sequence, \( \tau = t_1, t_2, \ldots \) and \( s_n = t_1 \), we let \( \sigma \circ \tau \) (strict catenation) denote the sequence \( s_1, \ldots, s_n, t_2, \ldots \). A sequential process is characterized by its set of state behaviors. Formally,

**Definition 2.5.1.5:** For sequential process \( S \) and set \( V \) of variables, \( \llbracket S, V \rrbracket \) denotes the set of state behaviors of \( S \) on \( V \). We require that \( \llbracket S, V \rrbracket \) is closed under finite repetition and that, for every state on \( V \), there is a state behavior in \( \llbracket S, V \rrbracket \) that starts with this state.

If \( V \) contains all free variables of \( S \), \( \llbracket S, V \rrbracket \) is defined inductively as follows. For convenience of notation, sometimes the defining set we give is not closed under finite repetition (of states). But it should be clear that we mean to take the closure of the set under repetition.

- \( \llbracket \text{SKIP}, V \rrbracket = \{ (T, \sigma) \mid \sigma \text{ is constant} \} \).

- \( \llbracket x := e, V \rrbracket = \{ (T, \sigma) \mid \sigma = s_1, s_2, s_2, \ldots \land s_2 = s_1[x = s_1(e)] \} \),

where \( s(e) \) denotes the value of \( e \) in state \( s \), \( s[x = d] \) denotes the state obtained from \( s \) by giving \( x \) the value \( d \).

- \( \llbracket S_1 ; S_2, V \rrbracket = \{ (F, \sigma) \mid (F, \sigma) \in \llbracket S_1, V \rrbracket \} \)
  \( \cup \{ (t, \sigma_1 \circ \sigma_2) \mid (T, \sigma_1) \in \llbracket S_1, V \rrbracket \land (t, \sigma_2) \in \llbracket S_2, V \rrbracket \land \text{last}(\sigma_1) = \text{first}(\sigma_2) \} \)
The first set corresponds to the case where $S_1$ does not terminate and the second set to the case where it terminates.

- $[[if \ldots p_k \rightarrow S_k \ldots fi, V]] = \bigcup_k \{ (t, \sigma) \in [[S_k, V]] \mid first(\sigma)(p_k) \}$

- $[[while p do S, V]] = \{ (T, \sigma) \mid \sigma \text{ is constant} \land \neg first(\sigma)(p) \}
  \bigcup \{ (t, \sigma_1 \ldots \sigma_{n-1} \sigma_n) \mid (T, \sigma_i) \in [[S, V]]
  \land last(\sigma_i) = first(\sigma_{i+1})
  \land first(\sigma_i)(p), i = 1, \ldots, n-1
  \land (t, \sigma_n) \in [[S, V]] \land first(\sigma_n)(p)
  \land (t \Rightarrow \neg last(\sigma_n)(p)) \}
  \bigcup \{ (F, \sigma_1 \sigma_2 \ldots) \mid (T, \sigma_i) \in [[S, V]]
  \land last(\sigma_i) = first(\sigma_{i+1})
  \land first(\sigma_i)(p), i = 1, 2, \ldots \}$

The first set corresponds to the case where the loop is never iterated, the second set to the case where the loop is iterated a finite number of times, and the third set to the case where the loop is iterated an infinite number of times.

- Information hiding:

If $W \subseteq V$ then $[[S, W]]$ is the set of restrictions of state behaviors in $[[S, V]]$ to $W$. Note that $W$ may not contain all free variables of $S$. 

The last rule incidentally handles variable scoping.

2.5.2. A temporal proof system for sequential processes

We now give a temporal proof system based on the behavior model for
sequential processes.

2.5.2.1. Specification

A specification of sequential process $P$ on set $V$ of variables has the form

$$<P>_V R$$

where $R$ is a temporal assertion in which

- the only local free variables are variables in $V$,
- the only global free variable is $t_P$ (the termination bit), and
- there is no local function or predicate symbol.

The interpretation of specification $<P>_V R$ is


2.5.2.2. Proof rules

First of all, we introduce new temporal operators $C$ (combine), $\ast$ (finite iteration), $\infty$ (infinite iteration) and $L$ (last). They are similar to those of [BKP, HKP].

- $\sigma$ satisfies $p \land q$ iff $\sigma = \sigma_1 \circ \sigma_2$, where $\sigma_1$ is eventually constant and satisfies $p$, $\sigma_2$ satisfies $q$, and $\text{last}(\sigma_1) = \text{first}(\sigma_2)$.

- $\sigma$ satisfies $p^\ast$ iff $\sigma = \sigma_1 \circ \ldots \circ \sigma_n$ for some $n$, where $\sigma_1, \ldots, \sigma_n$ are eventually constant and satisfy $p$ and $\text{last}(\sigma_i) = \text{first}(\sigma_{i+1})$, $i = 1, \ldots, n-1$. 
• $\sigma$ satisfies $p^\infty$ iff $\sigma = \sigma_1 \sigma_2^\infty \ldots$, where $\sigma_1, \sigma_2, \ldots$ are eventually constant and satisfy $p$, and $\text{last}(\sigma_i) = \text{first}(\sigma_{i+1})$, $i = 1, 2, \ldots$.

• $\sigma$ satisfies $L\ p$, where $p$ is a classical formula, iff $\sigma$ is eventually constant and its last state satisfies $p$. $L$ is expressible in terms of $\mathcal{C}$.

The proof rules are as follows. We assume that $V$ contains all the variables that appear in the programs.

• $<\text{SKIP}>_V \varnothing (\land_{x \in V} (\Diamond x = x)) \land t_{\text{SKIP}}$

• $<x := e>_V (\Diamond x = x \cup (\Diamond x = e \land \Diamond \varnothing (\Diamond x = x)))
\quad \land \Diamond (\land_{y \neq x} \Diamond y = y) \land t_{x:=e}$

• $<S_1>_V R_1, \ <S_2>_V R_2$

\[ <S_1 ; S_2>_V (R_1[F / t_{S_1}] \land \neg t_{S_1;s_2}) \]
\[ \lor \exists t_{S_2}((R_1[T / t_{S_1}] \lor R_2) \land t_{S_1;S_2} = t_{S_2}) \]

• $<S_k>_V R_k$

\[ <\text{if} \ldots p_k \rightarrow S_k \ldots fi>_V \lor_k \exists t_{S_k}(p_k \land R_k (t_{y\ldots} = t_{S_k})) \]
\[ <S>_\forall R \]

\[
<\text{while } p \text{ do } S>_\forall (\neg p \wedge \Box (\land_{x \in V} \Box x = x) \land t_{\text{while}}) \\
\vee ((p \land R[T / t_S])^\omega \land \neg t_{\text{while}}) \\
\vee ((p \land R[T / t_S])^\omega \subseteq t_S (R \land (t_S \Rightarrow L \neg p) \\
\wedge (t_{\text{while}} = t_S))
\]

\[ <S>_\forall R, R \Rightarrow T, T \text{ an assertion on } W, W \subseteq V \]

\[ <S>_w T \]

where \( R \Rightarrow T \) can be proved using axioms and inference rules for temporal logic, axioms and inference rules for the domain and axioms that characterize state behaviors.

2.5.2.3. Soundness and completeness

The definitions of soundness and completeness are similar to those for the basic proof system. Here, we do not need to have the notion of "expressive relative to", since the primitive processes are \( \text{SKIP} \) and assignment statements, and it is easy to see that the specifications given to them in the proof rules are precise.

To prove soundness and completeness of these rules, one first proves the following non-interference property.

Lemma 2.5.2.3.1: Let \( R \) be an assertion whose only free variables are local variables in \( V \). For any sequences \( \sigma \) and \( \tau \) of states,
\( \sigma|_v = \tau|_v \) implies \( \sigma \) satisfies \( R \) iff \( \tau \) satisfies \( R \).

**Proof:** The proof is by induction on the structure of \( R \).

- \( R \) is an atomic formula. Then \( \sigma \) satisfies \( R \) iff \( R \) is true in \( \text{first}(\sigma) \). But \( \text{first}(\sigma) \) and \( \text{first}(\tau) \) assign the same values to all the terms in \( R \). So \( \sigma \) satisfies \( R \) iff \( \tau \) does.

- \( R \) 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. ■

It is straightforward to check that the proof rules are sound. The proof of completeness is similar to that of Theorem 2.3.1.1, i.e. one proves that the proof rules – except the consequence rule – preserve preciseness of specifications and uses this to prove completeness.

Note that for Hoare-like proof systems, relative completeness is not an appropriate notion for *total correctness*. As shown in [Ap], any totally sound Hoare-like proof system is relatively incomplete. The appropriate notion of completeness in this case is *arithmetic completeness* [Har], which is much more complicated. Our proof system does not suffer from this problem.

**2.5.3. Communicating sequential processes**

To incorporate the sequential program constructs into the basic model, we
define the semantics of a communicating sequential process as a set of combined behaviors. A combined behavior is a triple \((t, \sigma, \tau)\), where \((t, \sigma)\) is a state behavior and \(\tau\) satisfies all the conditions of a communication behavior in the basic model except the condition that the initial trace is empty.

To demonstrate this, we now give a behavior model for a version of CSP [Ho78]. The following program constructs are added to the sequential program constructs:

\[ i \ ? \ x \] (input command), where \(x\) is a variable and \(i\) is a port name.

\[ j \ ! \ e \] (output command), where \(e\) is an expression and \(j\) is a port name.

\[ P_1 \ || \ P_2 \] (parallel command), where \(P_i\)'s are process names.

We also allow input guards in \(if... g_k \rightarrow S_k ... fi\), i.e. for some \(k\), \(g_k = p_k ; i_k \ ? \ x_k\), where \(p_k\) is a Boolean expression. Note that this is slightly different from CSP in that communication is done through named ports, and not directly between processes. It is straightforward to implement the latter using the former, however. As in CSP, the communication is synchronous. Hence the length function \(Rd\) is not needed for the behavior model.

For \(S\) a process on a set \(V\) of variables and a set \(J\) of port names, we let \(\llbracket S, V, J \rrbracket\) denote its set of combined behaviors. If \(V\) contains all free variables and \(J\) contains all port names of \(S\), \(\llbracket S, V, J \rrbracket\) is defined inductively as follows. As before, sometimes the defining set we give is not closed under finite
repetition. The defining sets for the sequential constructs are similar to those
given earlier.

- \([\text{SKIP}, V, J]\) = \{ (T, \sigma, \tau) \mid \sigma, \tau \text{ are constant} \}

- \([x := e, V, J]\) = \{ (T, \sigma, \tau) \mid \tau \text{ is constant} \land \sigma = s_1, s_2, s_2, \ldots \\
  \land s_2 = s_1[x \leftarrow s_1(e)] \}

- \([S_1 ; S_2, V, J]\) = \{ (F, \sigma, \tau) \mid (F, \sigma, \tau) \in [S_1, V, J]\} \\
  \cup \{ (t, \sigma_1 \sigma_2, \tau_1 \tau_2) \mid (T, \sigma_1, \tau_1) \in [S_1, V, J] \\
  \land (t, \sigma_2, \tau_2) \in [S_2, V, J] \\
  \land \text{last}(\sigma_1) = \text{first}(\sigma_2) \\
  \land \text{last}(\tau_1) = \text{first}(\tau_2) \}

The first set corresponds to the case where \(S_1\) does not terminate and the
second set to the case where it terminates.

- \([if\ldots g_k \rightarrow S_k \ldots fi, V, J]\) = \(\bigcup_k B_k\), where

  \(B_k = \{ (t, \sigma, \tau) \mid (S_k, V, J) \mid \text{first}(\sigma)(p_k) \}\)

  if \(g_k = p_k\),

  \(= \{ (t, \sigma, \tau) \mid (\sigma, \tau) = (s_1, u_1), (s_1, u_2), (s_2, u_3), (s_3, u_4), \ldots \\
  \land s_1(p_k) \land u_2 = u_1[\text{In}(i_k) \rightarrow T] \\
  \land \exists d(u_3 = u_2[\text{In}(i_k) \rightarrow F, i_k \rightarrow u_2(i_k).d] \\
  \land s_2 = s_1[x_k \rightarrow d]) \\
  \land (t, \sigma^{(3)}, \tau^{(3)}) \in [S_k, V, J] \}\)

  if \(g_k = p_k ; i_k?x_k\)

When \(g_k = p_k ; i_k?x_k\), \(S_k\) is executed only if \(p_k\) holds and the read opera-
tion \(i_k?x_k\) is successful.
\textbf{while} \( p \text{ do } S, V, J \) = \{ \( (T, \sigma, \tau) \mid \sigma, \tau \text{ are constant} \land \neg \text{first}(\sigma)(p) \} \\
\cup \{ (t, \sigma_1 \ldots \sigma_n, \tau_1 \ldots \tau_n) \mid \\
(T, \sigma_i, \tau_i) \in [S, V, J] \\
\land \text{last}(\sigma_i) = \text{first}(\sigma_{i+1}) \\
\land \text{last}(\tau_i) = \text{first}(\tau_{i+1}) \\
\land \text{first}(\sigma_i)(p), i = 1, \ldots, n-1 \\
\land (t, \sigma_n, \tau_n) \in [S, V, J] \land \text{first}(\sigma_n)(p) \\
\land (t \Rightarrow \neg \text{last}(\sigma_n)(p)) \} \\
\cup \{ (F, \sigma_1 \ldots \sigma_2, \tau_1 \ldots \tau_2) \mid \\
(T, \sigma_i, \tau_i) \in [S, V, J] \\
\land \text{last}(\sigma_i) = \text{first}(\sigma_{i+1}) \\
\land \text{last}(\tau_i) = \text{first}(\tau_{i+1}) \\
\land \text{first}(\sigma_i)(p), i = 1, 2, \ldots \},

The first set corresponds to the case where the loop is never iterated, the second set to the case where the loop is iterated a finite number of times, and the third set to the case where the loop is iterated an infinite number of times.

\textbf{[i \ ? \ x, V, J]} = \{ (F, \sigma, \tau) \mid \sigma \text{ is constant} \land \tau = u_1, u_2, u_2, \ldots \\
\land u_2 = u_1[ln(i) \leftarrow T] \} \\
\cup \{ (T, \sigma, \tau) \mid \\
(\sigma, \tau) = (s_1, u_1), (s_1, u_2), (s_2, u_3), (s_2, u_3), \ldots \\
\land u_2 = u_1[ln(i) \leftarrow T] \\
\land \exists d (u_3 = u_2[ln(i) \leftarrow F, i \leftarrow u_2(i).d] \\
\land s_2 = s_1[x \leftarrow d]) \} \\

The first set corresponds to the case where the read operation is unsuccessful and the second set to the case where it is successful.
\[ \{ (F, \sigma, \tau) \mid \sigma \text{ is constant } \land \tau = u_1, u_2, u_2, \ldots \land u_2 = u_1[Out(j) \leftarrow T] \} \]

\[ \cup \{ (T, \sigma, \tau) \mid \sigma \text{ is constant } \land \tau = u_1, u_2, u_3, u_3, \ldots \land u_2 = u_1[Out(j) \leftarrow T] \land u_3 = u_2[Out(j) \leftarrow F, j \leftarrow u_2(j).s_1(e)] \} \]

The first set corresponds to the case where the write operation is unsuccessful and the second set to the case where it is successful.

- If \( V_1 \cap V_2 = \emptyset \),
\[ [S_1 \parallel S_2, V_1 \cup V_2, J_1 \cup J_2] = \{ (t, \sigma, \tau) \mid \exists (t_1, \sigma|_{V_1}, \tau|_{J_1}) \in [S_1, V_1, J_1], \exists (t_2, \sigma|_{V_2}, \tau|_{J_2}) \in [S_2, V_2, J_2] \text{ such that } t = t_1 \land t_2 \} \]

- Information-hiding:

If \([S, V, J]\) is defined and \( W \subseteq V, L \subseteq J \) then \([S, W, L]\) is the set of restrictions of combined behaviors in \([S, V, J]\) to \(W\) and \(L\). Note that \(W\) and \(L\) may not contain all variables and port names of \(S\).
CHAPTER 3

A PARALLEL FUNCTIONAL LANGUAGE

3.1. Introduction

In [Ba], Backus introduced a *functional style of programming* (FP) in which variable-free programs are built from a set of primitive programs by a small set of combining forms or functionals and by recursive definitions. At the same time, Hoare [Ho78] proposed a *parallel language* (CSP) for communicating sequential processes that has input, output and concurrency as its primitives. We now describe a simple language called *PFL* that incorporates both features. Because of message passing, a purely functional (side-effect free) language cannot be obtained. However, we have kept side-effects to a minimum without making unnatural restrictions on the syntax of the language.

This language incorporates most of the programming constructs introduced in chapter 2 and demonstrates how communication could be handled in a functional language. An interesting feature of the language is *general recursion*, which includes both control-flow recursion (as in sequential programs) and data-flow
recursion. This powerful construct allows us to define processes in PFL that cannot be described in CSP. Another advantage is that it is easier to exploit concurrency in a language without side-effects, as argued in [Ac].

We do not give a complete description of PFL. Rather, we describe the framework of the language and ignore certain details, e.g. the exact set of primitive functions in the language, typing of port variables, etc. We do not restrict our discussion to any single mode of communication, either. Both synchronous and asynchronous communication are allowed.

3.2. Syntax

The syntax of PFL is as follows.

Definition 3.2.1: An object is a

- constant, e.g. 3, T (True), APPLE, or
- port variable, e.g. i, j, k.

Definition 3.2.2: A primitive function is a

- logical function, e.g. $\wedge$, $\vee$, $\neg$, or
- arithmetic function or relation, e.g. $+$, $-$, $<$, even, or
- sequence function, e.g. $\mid\mid$, first, last, head, tail.

For $s$ a finite sequence, first$(s)$ (last$(s)$) is the first (last) element of $s$ and head$(s)$ (tail$(s)$) is the prefix (suffix) of $s$ obtained by removing the last (first)
element. \(|s|\) computes the length of \(s\). \(\blacksquare\)

**Definition 3.2.3:** A *communication function* is

- *in*, or
- *out*

The communication function *in* takes one argument that is a port variable. The communication function *out* takes two arguments: a port variable and a function application (to be defined below). \(\blacksquare\)

\(\text{in}(i)\) means "read the next input at input port \(i\) and return its value". \(\text{out}(j, e)\) means "write the value of \(e\) on output port \(j\) and return this value". Communication function are the only constructs in the language that cause side-effects.

**Definition 3.2.4:** A *function application* is an

- object, or
- primitive function (function application, ..., function application), or
- communication function (function application, ..., function application).

This is called a *communication*.

The number of arguments has to be the same as the arity of the primitive function or communication function. The types of the function parameters and those of the arguments also have to match, e.g. arguments of sequence functions have to be sequences. \(\blacksquare\)
Definition 3.2.5: An *expression* is a side-effect free function application, i.e. one that contains no communication.

A *Boolean expression* is an expression that evaluates to \( T \) or \( F \).

Definition 3.2.6: A *guard* is a

- Boolean expression, or
- communication function (expression, ..., expression), or
- Boolean expression; communication function (expression, ..., expression).

Definition 3.2.7: A *process* is

- process name (port variable, ..., port variable; port variable, ..., port variable)

e.g. `BUFFER(i; j)`. Process names are written in capital letters. The port variables of a process are all distinct. Port variables appearing before (after) ";" are *input* (output) port variables of the process.

Definition 3.2.8: A *network* is a

- function application, or
- process name (sequential call), or
- network; network (sequential composition), or
- \( if \ldots \) guard \( \rightarrow \) network \mid \) guard \( \rightarrow \) network \( \ldots \) \( fi \) (conditional), or
• \( || \) (process, ..., process; process, ..., process) \hspace{1em} \text{(network call).}

In a network call, all the input (output) ports of processes are distinct and all the ports of processes appearing after ";" are linked ports. The processes appearing before (after) ";" are the main (subroutine) components of \( || \) (...). (See Section 2.4.1 for a discussion on main and subroutine components of a network.)

**Definition 3.2.9:** A process definition is

- process = network.

The set of external input ports of the network is a subset of that of the process. If the network contains a network call then the two sets are equal. The same conditions hold for the external output ports. For example,

\[
BUFF1(i; j) = \text{if } T \rightarrow \text{BUFF } \text{fi}
\]

\[
PRIME(i; j) = || (SIEVE(i; k, l), PRINT(k, m; j); PRIME(l; m))
\]

If the left-hand side of a process definition is \( P(...) \) then we require:

- Every \( P(...) \) that occurs in a network call of the right-hand side is a subroutine component.

In a set of process definitions \( P_h(...) = E_h, h = 1, \ldots, n \), we also require the following.

- If process name \( P_k \) occurs in \( E_h \) as a sequential call then processes \( P_h(...) \) and \( P_k(...) \) have the same numbers of input and output ports.
• For every cycle \( k_0, \ldots, k_{m-1} \) such that \( P_{k_{r+1} \mod m} \) or \( P_{k_{r+1} \mod m}(...) \) occurs in \( E_k \) for \( s = 0, \ldots, m-1 \), every occurrence of \( P_{k_{r+1} \mod m}(...) \) in \( E_k \) is a subroutine component of some network call. ■

The condition on (recursive) process definitions ensures that there are only a finite number of processes executing at any time. The corresponding condition on mutually recursive process definitions is for a similar reason.

3.3. Semantics

At any point in time, the value of an input-port variable in a network is the sequence of data that have been read on the port up to that time. Similarly, the value of an output-port variable is the sequence of data that have been written on the port. Thus, for synchronous communication, the value of a linked input-port variable is the same as that of the corresponding linked output-port variable. For asynchronous communication, the former is a prefix of the latter.

In a function application, the arguments are evaluated (executed) only when they are needed, and each time they are needed, from left to right. In \( \text{in}(i) \), the process is blocked until it can read the next input on port \( i \) and return this value. In \( \text{out}(j, e) \), \( e \) is evaluated and, if message passing is asynchronous, the value of \( e \) is sent on port \( j \). For synchronous message passing, the process is blocked and cannot send the value of \( e \) until the receiving process is ready to receive on \( j \).
either case, once the value of \( e \) is sent, \( out(j, e) \) returns this value.

In a sequential call \( \text{process name} \), control is transferred to the process with that name whose input- and output-port variables are replaced by those of the calling process. When and if the called process terminates, control is transferred back to the calling process.

In a sequential composition \( \text{network}_1; \text{network}_2 \), \( \text{network}_1 \) is executed first. If it terminates then \( \text{network}_2 \) is executed.

In a conditional \( \text{if} \ldots \text{guard}_k \rightarrow \text{network}_k \ldots \text{fi} \), a guard is chosen nondeterministically and executed. If the guard is a Boolean expression that evaluates to \( T \), or a communication that succeeds, or a sequential composition of a Boolean expression that evaluates to \( T \) and a communication that succeeds, then the corresponding network is executed. Otherwise another guard is chosen. If the Boolean expressions of all the guards evaluate to \( F \) then the conditional is skipped. Otherwise, if all the guards whose Boolean expressions are \( T \) contain a communication, the process is blocked until one of these communications succeeds. Nondeterminism is achieved in the language by this construct.

In a network call, \( || (\text{process}_1, \ldots ; \text{process}_m, \ldots) \), all the processes are executed concurrently, but the subroutine components are not executed until they are activated (as defined in Definition 2.4.1.2). When and if all the components of the network terminate, the linked-port variables of the network cease to exist and control is transferred back to the calling process.
It is tedious, but not difficult, to give a behavior semantics for PFL that is consistent with the above operational semantics and similar to that for CSP (Section 2.5.3). Since there is no local program variable in PFL, a state behavior component is not needed. Recursion is modeled by a similar technique to that used in modeling recursive network (Section 2.4.2).

3.4. Examples

Example 3.4.1: \(ADD(i_1, i_2; j_1)\) repeatedly reads inputs from \(i_1, i_2\) and writes their sum on \(j_1\).

\[
ADD(i_1, i_2; j_1) = out(j_1, +(in(i_1), in(i_2)));
\]

Example 3.4.2: \(MERGE(i_1, i_2; j_1)\) nondeterministically reads inputs from \(i_1, i_2\) and reproduces them on \(j_1\).

\[
MERGE(i_1, i_2; j_1) = \text{if } in(i_1) \rightarrow out(j_1, last(i_1))
\]
\[
\quad \mid in(i_2) \rightarrow out(j_1, last(i_2))
\]
\[
\quad fi;
\]
\[
MERGE
\]

To take full advantage of concurrency, CSP allows finite arrays of concurrent processes [Ho78, MC]. However, for some networks, unless some restrictions on the size of the inputs to the networks are made, the arrays of processes should be infinite. This problem is overcome in PFL by the use of recursion. Two examples illustrate the approach. We assume synchronous message passing.
Example 3.4.3: Network $FACT(i; j)$ repeatedly reads inputs on port $i$ and produces their factorials on port $j$.

$$FACT(i; j) = \| (CP(i, l; j, k); FACT(k; l))$$

See Figure 3.1.

Network $CP(i, l; j, k)$ consists of three sequential processes

$$CP(i, l; j, k) = \| (IN(i; k, n), BUFF(n; p), OUT(l, p; j))$$

See Figure 3.2.

$IN(i; k, n)$ does the following repeatedly. It reads input $x$ from $i$. If $x > 0$, it writes $x - 1$ on $k$. Then it writes $x$ on $n$.

![Diagram of CP and FACT](image)

**Figure 3.1 FACT**
\[ \text{IN}(i; k, n) = \text{in}(i); \]
\[ \text{if } \text{last}(i) > 0 \rightarrow \text{out}(k, \text{last}(i) - 1) \] \text{fi}; \]
\[ \text{out}(n, \text{last}(i)); \]
\[ \text{IN} \]

\text{BUFF}(n; p) \] is an unbounded buffer.

\[ \text{BUFF}(n; p) = \text{if } \text{in}(n) \rightarrow \text{BUFF} \]
\[ \quad | n| > |p|; \text{out}(p, \text{first}(\text{tail} |p|)(n))) \rightarrow \text{BUFF} \]
\[ \text{fi} \]

\text{OUT}(p, l; j) does the following repeatedly. It reads input \( x \) from \( p \). If \( x \) is 0, it writes 1 on \( j \). Otherwise, it reads input \( y \) from \( l \) and writes \( x \times y \) on \( j \).

\[ \text{OUT}(p, l; j) = \text{in}(p); \]
\[ \text{if } \text{last}(p) = 0 \rightarrow \text{out}(j, 1) \]
\[ \quad | \text{last}(p) > 0 \rightarrow \text{out}(j, \text{last}(p) \times \text{in}(l)) \]
Example 3.4.4: Network $PRIME(i; j)$, given the infinite ascending sequence of odd numbers 3, 5, ... on $i$, produces the infinite ascending sequence of odd primes on $j$.

$$PRIME(i; j) = \| (SIEVE(i; k, l), PRINT(k, m; j); PRIME(l; m))$$

See Figure 3.3.

$SIEVE(i; k, l)$ repeatedly reads inputs from $i$. It reproduces the first input on $k$ and checks subsequent inputs to see if they are divisible by the first input. If they are, it discards them; otherwise it reproduces them on $l$.

$$SIEVE(i; k, l) = out(k, in(i)); SIEVE1$$

Figure 3.3 PRIME
$\text{SIEVE1}(i; k, l) = \text{in}(i);$

    if $\neg\text{divide(first}(i), \text{last}(i)) \rightarrow \text{out}(l, \text{last}(i))$ fi;

$\text{SIEVE1}$

$\text{PRINT}(k, m; j)$ first reads an input from $k$ and reproduces it on $j$, then repeatedly reads inputs from $m$ and reproduces them on $j$.

$\text{PRINT}(k, m; j) = \text{out}(j, \text{in}(k)); \text{PRINT1}$

$\text{PRINT1}(k, m; j) = \text{out}(j, \text{in}(m)); \text{PRINT1}$
CHAPTER 4

A SYNTHESIS SYSTEM FOR NETWORKS OF PROCESSES

4.1. Introduction

Program synthesis is the systematic derivation of a program from a given specification. In [MW80, MW82], a deductive system for the synthesis of sequential applicative (side-effect free) programs is presented. In this system, the synthesis of a program is regarded as a theorem-proving problem; the desired program is constructed as a by-product of the proof. This approach makes it possible to combine techniques of nonclausal resolution, transformation rules, mathematical induction, etc. within a single deductive system.

Here, we follow that approach in synthesizing networks of communicating processes. We restrict ourselves to networks in which all the component processes are deterministic and message transmission among them is asynchronous. For this type of network, an elegant fixed-point semantics has been developed by Kahn [Ka]. (This semantics does not work for more general types of network, unfortunately.) A process (network) is described as a continuous function from tuples
of input sequences to tuples of output sequences. The function describing a network can be obtained from those of the component processes by solving a set of recursive equations. By the continuity of the functions, a solution always exists.

4.2. Networks of asynchronous deterministic processes

We briefly describe a model of networks of asynchronous deterministic processes, based on Kahn's idea [Ka].

4.2.1. Syntax

A process, as depicted in Figure 4.1, has a finite number of input ports and output ports associated with it. The names of these ports are all distinct.

Networks of processes are formed by linking some input ports of a process to some output ports of another process, or of itself, in a one-to-one manner. The name of a linked port is that of the output port. The ports of a network are required to have distinct names. See Figure 4.2.

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

Networks can be formed using three primitive operations:

- \textit{Disjoint parallel composition}

  //\((P_1, P_2)\) is the network obtained from \(P_1\) and \(P_2\) by running them in parallel. See Figure 4.3.
• *Sequential composition*

$S_{j_1, \ldots, j_m}^i(P_1, P_2)$ is the network obtained from $P_1$ and $P_2$ by linking output ports $j_1, \ldots, j_m$ of $P_1$ to input ports $i_1, \ldots, i_m$ of $P_2$. $P_1$ and $P_2$ are required to be distinct, i.e. no cycle is formed. See Figure 4.4.
Cycle formation

$C_j(P)$ is the process obtained from $P$ by linking output port $j$ to input port $i$ of $P$. See Figure 4.5.

These three primitive operations are sufficiently general to form any network that can be represented by a directed graph.
4.2.2. Semantics

Let $D$ be a domain of data values. Let $S$ be the set of all sequences, finite or infinite, of elements in $D$. Let $\subseteq$ be the prefix relation on $S$, i.e. $s \subseteq t$ iff $s$ is a prefix of $t$.

It is easy to see that $(S, \subseteq)$ is a cpo (complete partial order), i.e. $\subseteq$ is a partial ordering on $S$, there exists a least element in $S$, denoted by $\bot$, and every increasing sequence $s_1 \subseteq s_2 \subseteq \ldots$ in $S$ has a lub (least upper bound). A function $f: X \rightarrow Y$, where $X$ and $Y$ are cpo's, is continuous if $\text{lub}_{n \rightarrow \infty} \{ f(s_n) \} = f(\text{lub}_{n \rightarrow \infty} \{ s_n \})$ for every increasing sequence $\{ s_n \}$. A function with more than one arguments is continuous iff it is continuous in each of its arguments.

Semantically, a process with input ports $i_1, \ldots, i_m$ and output ports $j_1, \ldots, j_n$ is a continuous function from $S^m$ to $S^n$, where $S^m$ denotes the direct product of $m$ copies of $S$. Equivalently, the process is a set of $n$ continuous functions, each mapping $S^m$ to $S$. Continuity is a very natural property for deterministic processes to have. It simply means that

- A process produces more outputs when given more inputs.
- A finite number of outputs depends on a finite number of inputs.

It is well-known that any continuous function $f: T \rightarrow T$, where $T$ is a cpo, has a least fixed point (see [Ka]). A fixed point of $f$ is an element $d$ of $T$ such that $d = f(d)$. A least fixed point of $f$ is the least element of the set of fixed points of $f$. This least fixed point is in fact $\text{lub}_{n \rightarrow \infty} \{ f^n(\bot) \}$. Viewed as functions,
• \((P_1, P_2)\) is the direct product of \(P_1\) and \(P_2\). That is, if \(P_1: S^m \to S^n\) and \(P_2: S^h \to S^l\), then \((P_1, P_2): S^{m+h} \to S^{n+l}\) and
\[
((P_1, P_2)(x_1, \ldots, x_m, y_1, \ldots, y_h) = (P_1(x_1, \ldots, x_m), P_2(y_1, \ldots, y_h))
\]

• \(S^{i_1, \ldots, i_k}_{j_1, \ldots, j_k}(P_1, P_2)\) is a functional composition of \(P_1\) and \(P_2\). More precisely, let \(P_1\) be a set of functions \(f_1, \ldots, f_n: S^m \to S\), \(P_2\) be \(g_1, \ldots, g_l: S^h \to S\). Then \(S^{i_1, \ldots, i_k}_{j_1, \ldots, j_k}(P_1, P_2)\) is the set of functions \(f'_{k+1}, \ldots, f'_n, g'_{i_1}, \ldots, g'_{i_l}: S^{m+h-k} \to S\), where

- (a) \(f'_t(x_1, \ldots, x_m, y_{k+1}, \ldots, y_{h}) = f_t(x_1, \ldots, x_m)\), for \(t = k+1, \ldots, n\), and

- (b) \(g'_t(x_1, \ldots, x_m, y_{k+1}, \ldots, y_{h}) = g_t(f'_1(x_1, \ldots, x_m), \ldots, f'_k(x_1, \ldots, x_m), y_{k+1}, \ldots, y_{h})\), for \(t = 1, \ldots, l\).

• \(C^i_{j_1, \ldots, j_k}(P)\) is obtained by taking the least fixed point of a recursive equation. More precisely, let \(P\) be a set of functions \(f_1, \ldots, f_n: S^m \to S\). Let \(z_0\) be the least solution (with respect to \(\sqsubseteq\)) of the equation \(z = f_t(x_1, \ldots, x_{k-1}, z, x_{k+1}, \ldots, x_m)\). Because of the continuity of the function, such a least solution always exists. Then \(C^i_{j_1, \ldots, j_k}(P)\) is the set of functions \(g_1, \ldots, g_{l-1}, g_{l+1}, \ldots, g_n: S^{m-1} \to S\), where
\[
g_t(x_1, \ldots, x_{k-1}, x_{k+1}, \ldots, x_m) = f_t(x_1, \ldots, x_{k-1}, z_0, x_{k+1}, \ldots, x_m),
\]
for \(t = 1, \ldots, l-1, l+1, \ldots, n\).
By giving the above semantics to processes, we restrict our attention to networks of deterministic processes, in which the message transmission is asynchronous, i.e. a process can send a message as soon as it is ready without having to wait for other processes to receive it.

### 4.3. Program synthesis

The problem of synthesizing networks of processes is as follows.

Let there be given a finite set of processes \( \{ P_i \} \), which we call primitive processes. For any process \( P \) with input ports \( i_1, \ldots, i_m \) and output ports \( j_1, \ldots, j_n \), a specification of \( P \) is a first-order formula of the form

\[
\forall x_1 \ldots \forall x_m \exists z_1 \ldots \exists z_n (R \Rightarrow S),
\]

where the free variables of \( R \) and \( S \) are among \( x_1, \ldots, x_m \) and \( x_1, \ldots, x_m, z_1, \ldots, z_n \), respectively. The interpretation of this is:

Given any input sequences \( x_1, \ldots, x_m \) on ports \( i_1, \ldots, i_m \) that satisfy \( R \), \( P \) produces output sequences \( z_1, \ldots, z_n \) on ports \( j_1, \ldots, j_n \) that satisfy \( S \).

For example, the process that iteratively reads a value on port \( i \) and reproduces it on port \( j \) satisfies the specification

\[
\forall x \exists z (T \Rightarrow z = x)
\]

The \textit{synthesis} problem is
Given a specification $T$, form a network from the primitive processes that satisfies $T$, if $T$ is *satisfiable* (i.e. if there exists a network that satisfies $T$).

4.4. The deductive approach

4.4.1. Deductive tableaus

We now describe the *deductive tableau method* [MW80, MW82].

The basic structure is the *deductive tableau*, which consists of a set of rows. Each row contains either an *assertion* or a *goal*, and optional associated *output entries*. For example,

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
<th>outputs $f(s)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s = z$</td>
<td></td>
<td>$z$</td>
</tr>
</tbody>
</table>

Given a specification $\forall x_1 \ldots \forall x_m \exists z_1 \ldots \exists z_n (R \Rightarrow S)$, we construct the initial tableau
where $R[...s_i... / ...x_i...]$ denotes the formula obtained from $R$ by replacing every free occurrence of $x_i$ by $s_i$. The $s_i$ are constants obtained by removing the universal quantifiers of the $x_i$ through skolemization, and $z_1, ..., z_n$ are free variables.

Typically, in addition to the input condition $R(s)$, the initial assertions of the tableau include axioms for the theory under consideration (e.g. integers, sequences, ...) and the underlying logic.

The intuition behind the deductive tableau method is as follows.

Associated with each tableau containing assertions $A_1, ..., A_m$ and goals $G_1, ..., G_n$ is a formula

$$(4.4.1.1) \quad (\forall \bar{y} A_1 \land \ldots \land \forall \bar{y} A_m) \Rightarrow (\exists \bar{y} G_1 \lor \ldots \lor \exists \bar{y} G_n)$$

where $\bar{y}$ denotes all the free variables in the $A_i$'s and the $G_j$'s. (In the terminology of [MW80, MW82], this formula is the "meaning" of the tableau.)

By giving a constructive proof for the formula (4.4.1.1), we obtain a program in the target language that satisfies the original program specification. The proof is carried out by modifying the initial tableau. This is done carefully so that, at all times, the output entries record the construction of the proof, and
(4.4.1.2) if some instance of an assertion (goal) is false (true) then the corresponding instances of the output entries satisfy the original program specification. (An instance of a row is obtained by replacing free variables of the row with terms.)

The intuition behind (4.4.1.2) is that an instance of the formula (4.4.1.1) is true iff the corresponding instance of some of its assertions (goals) is false (true), and whenever it is true, we want the corresponding instances of the output entries satisfy the original program specification. Therefore, if a row of a tableau is of the form

\[
\begin{array}{c|c|c}
 & T & t \\
\hline
\end{array}
\]

or

\[
\begin{array}{c|c|c}
F & & t \\
\hline
\end{array}
\]

where \( t \) consists entirely of symbols from the target programming language, then the formula (4.4.1.1) is proved and \( t \) is the desired program.

It follows from (4.4.1.1) and (4.4.1.2) that the goals of the tableau have a tacit disjunction between them while the assertions have a tacit conjunction. The free variables of the goals are implicitly existentially quantified while those of the assertions are implicitly universally quantified. Because of the implicit
quantifiers, the free variables in a row are dummy variables and can be systematically replaced by new variables without changing the meaning of the tableau. The distinction between assertions and goals helps to make our deduction easier to understand but does not increase the logical power of the deductive system. If we delete a goal (assertion) from the tableau and add its negation as a new assertion (goal), we obtain an equivalent tableau. This property is known as duality.

A tableau can be modified with the use of deduction rules. These rules preserve (4.4.1.2) at all times. Deduction rules can be used to

- add new rows to the tableau, or
- create new tableaux.

The creation of new tableaux creates auxiliary program specifications whose solutions constitute a solution for the original specification. This corresponds to creating subgoals for proving the goal (4.4.1.1).

Definition 4.4.1.1: A tableau is terminal if it contains a terminal row. A terminal row is a row of the form

\begin{equation}
\begin{array}{c|c|c}
 & T & t \\
\end{array}
\end{equation}

or
where $t$ consists entirely of symbols from the target programming language, or

A row that contains references to other tableaus and the only symbols in the output expression not from the target programming language are the headings of output columns of other tableaus. (This will become clear when we introduce the disjoint parallel composition rule.)

Intuitively, a tableau is terminal if a constructive proof is found for the formula associated with the tableau or if the proof can be constructed from those of other tableaus.

**Definition 4.4.1.2:** The *result* of a terminal tableau is

- The output expression of a terminal row if it is of the first two types, or
- The output expression of a terminal row of the third type with every heading of the output column of a tableau replaced by the result of that tableau.

**Definition 4.4.1.3:** The deduction process *terminates* when

- The original tableau contains a terminal row of the form (4.4.1.3) or (4.4.1.4), or
The original tableau contains a terminal row of the form (4.4.1.5), and any tableau that is referred to by this row is a terminal tableau.

The result of the original tableau when (and if) the deduction process terminates is the desired program (process).

Similar definitions apply when the tableaus have more than one output columns.

4.4.2. Deduction rules

Following [MW80, MW82], we outline the logical structure of the system without considering the strategic aspects of how deductions are directed. For ease of exposition, we do not consider deduction rules for dealing with special relations like equality and equivalence. This can be incorporated into the system, just as in [MW82]. Some of the deduction rules developed in [MW80] for sequential programs still apply. We briefly describe them here. They are the splitting rules, transformation rules and resolution rules.

Splitting rules

These rules allow us to decompose an assertion or goal into its logical components.
**and-split**

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F \land G$</td>
<td></td>
<td>$t$</td>
</tr>
<tr>
<td>$F$</td>
<td></td>
<td>$t$</td>
</tr>
<tr>
<td>$G$</td>
<td></td>
<td>$t$</td>
</tr>
</tbody>
</table>

This means that if rows matching those above the double line are present in the tableau, then the corresponding rows below the double line may be added. By duality, we also have

**if-split**

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$F \Rightarrow G$</td>
<td>$t$</td>
</tr>
<tr>
<td>$F$</td>
<td></td>
<td>$t$</td>
</tr>
<tr>
<td>$G$</td>
<td></td>
<td>$t$</td>
</tr>
</tbody>
</table>

**Transformation rules**

First of all, we need to discuss the topics of substitution and unification — see [Ni]. They are used to match certain subexpressions of a set of expressions. This is often necessary for proving theorems involving quantified formulas. For example, to infer $E_2(A)$ from $\forall x(E_1(x) \Rightarrow E_2(x))$ and $E_1(A)$, it is necessary to find the
substitution "A for x" to make $E_1(A)$ and $E_1(x)$ identical.

Definition 4.4.2.1: A term is a variable symbol, a constant symbol or a functional expression of the form function symbol (term, ..., term).

Definition 4.4.2.2: A substitution is a set of ordered pairs \( \{t_i/v_1, ..., t_n/v_n\} \), where the \( t_i \) are terms and the \( v_j \) are variables such that no \( v_k \) occurs in \( t_k \).

Definition 4.4.2.3: A substitution instance \( E\sigma \) of expression \( E \) using substitution \( \sigma \) is the expression obtained by substituting terms \( t_i \) for variables \( v_i \) throughout \( E \), where \( t_i/v_i \in \sigma \).

For example, if

\[
E = P[x, f(y), B] \\
\sigma = \{g(z)/x, A/y\}
\]

then

\[
E\sigma = P[g(z), f(A), B]
\]

Definition 4.4.2.4: The composition of two substitutions \( \sigma_1 \) and \( \sigma_2 \), denoted by \( \sigma_1 \sigma_2 \), is the substitution obtained from \( \sigma_1 \) by applying \( \sigma_2 \) to the terms of \( \sigma_1 \) and then adding any pairs of \( \sigma_2 \) having variables not occurring among the variables of \( \sigma_1 \).

For example,

\[
\{g(x, y)/z\} \{A/x, B/y, C/w, D/z\} = \{g(A, B)/z, A/x, B/y, C/w\}
\]
It can be shown that applying $\sigma_1$ and $\sigma_2$ successively to an expression $E$ is the same as applying $\sigma_1\sigma_2$ to $E$. That is,

$$(E\sigma_1)\sigma_2 = E(\sigma_1\sigma_2)$$

It can also be shown that the composition of substitutions is associative:

$$\sigma_1(\sigma_2\sigma_3) = (\sigma_1\sigma_2)\sigma_3$$

**Definition 4.4.2.5:** A set of expressions $\{E_i\}$ is **unifiable** if there exists a substitution $\sigma$ such that $E_1\sigma = E_2\sigma = \ldots$ $\sigma$ is said to be a **unifier** of $\{E_i\}$.

A unifier $\tau$ of $\{E_i\}$ is the **most general unifier** if, for any unifier $\sigma$ of $\{E_i\}$, there exists a substitution $\alpha$ such that

$$\{E_i\}\sigma = \{E_i\}\tau\alpha \quad \blacksquare$$

Intuitively, the most general unifier of a set of expressions is the simplest substitution that unifies the expressions.

Transformation rules instantiate variables and enable assertions or goals to be derived from others. Suppose that we have a transformation rule of the form

$$r \Rightarrow s \text{ if } P,$$

which means that, in any formula, subexpression $r$ is equal to subexpression $s$ and the former can be replaced by the latter, provided that the condition $P$ holds.

Then we have
<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F$</td>
<td></td>
<td>$t$</td>
</tr>
<tr>
<td>$P\theta \Rightarrow F\theta[s\theta / r\theta]$</td>
<td></td>
<td>$t\theta$</td>
</tr>
</tbody>
</table>

where $F$ contains a subexpression $r'$ and $\theta$ is the most general unifier of $r$ and $r'$.

By duality, a similar rule is

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F$</td>
<td></td>
<td>$t$</td>
</tr>
<tr>
<td>$P\theta \land F\theta[s\theta / r\theta]$</td>
<td></td>
<td>$r\theta$</td>
</tr>
</tbody>
</table>

Transformation rules can be logical rules relating two logical expressions, or they can be domain dependent rules relating two expressions in the target programming language, or a logical expression in the specification language to one in the target programming language.

Resolution rules

These rules serve similar purposes as transformation rules. They perform case analysis on the truth of subsentences of the assertions or goals of the tableau to produce new assertions or goals. The version of resolution used here does not require the sentences to be in conjunctive normal form.
**AA-resolution**

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F$</td>
<td>$F$</td>
</tr>
<tr>
<td>$G$</td>
<td>$G$</td>
</tr>
</tbody>
</table>

$F\theta[T / P_1\theta] \lor G\theta[F / P_2\theta]$ for some subexpression $P_1$ and $P_2$, where $\theta$ is the most general unifier of $P_1$ and $P_2$. By duality, we also have

**GG-resolution**

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F$</td>
<td>$F$</td>
</tr>
<tr>
<td>$G$</td>
<td>$G$</td>
</tr>
</tbody>
</table>

$F\theta[T / P_1\theta] \land G\theta[F / P_2\theta]$}

**GA-resolution**

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F$</td>
<td>$F$</td>
</tr>
<tr>
<td>$G$</td>
<td>$G$</td>
</tr>
</tbody>
</table>

$F\theta[T / P_1\theta] \land \neg G\theta[F / P_2\theta]$
**AG-resolution**

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F$</td>
<td></td>
</tr>
<tr>
<td>$G$</td>
<td>$-F\theta[T/P_1\theta] \land G\theta[F/P_2\theta]$</td>
</tr>
</tbody>
</table>

Furthermore, if exactly one of the initial rows has an output expression $t$ then the resulting row has $t\theta$ as the output expression. Unlike the case of sequential programs, the resolution rules cannot be applied if both of the initial rows have (distinct) output expressions. The reason is that if-then-else is not a primitive construct of the language for network formation.

The above rules are sufficient for weakening the assertions, for strengthening the goals, and for synthesizing the sequential composition operator in our language. (Recall that sequential composition is just like composition of functions.)

### 4.4.2.1. Examples

In the examples in this chapter, $s$, $t$... denote sequences and $a$, $b$... denote elements (of sequences). In this example, we assume that the data domain is $\text{Bool } (= \{T, F\})$.

Let *and* and *not* be two primitive processes defined as follows.
• \( \text{and}(s, t)(n) = s(n) \land t(n) \)

• \( \text{not}(s)(n) = \neg s(n) \)

These definitions can be used both as assertions and as transformation rules. Suppose we want to synthesize a network that reads two input sequences \( s \) and \( t \) and produces an output sequence \( y \) such that \( y(n) = s(n) \lor t(n) \). To solve this, we first form the tableau

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
<th>outputs ( f(s) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T )</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>( y(n) = s(n) \lor t(n) )</td>
<td>( y )</td>
</tr>
</tbody>
</table>

Apply some obvious logical transformation rules

| \( y(n) = \neg(s(n) \lor t(n)) \) | \( y \) |
| \( y(n) = \neg(\neg s(n) \land \neg t(n)) \) | \( y \) |

Applying the definitions of \( \text{and} \) and \( \text{not} \) as transformation rules, we obtain

| \( y(n) = \neg(\text{not}(s)(n) \land \text{not}(t)(n)) \) | \( y \) |
| \( y(n) = \neg\text{and}(\text{not}(s), \text{not}(t))(n) \) | \( y \) |
Apply the GA-resolution rule, using the assertion \( x = x \) —assuming it is in our general knowledge— and the new goal

<table>
<thead>
<tr>
<th>( T )</th>
<th>( \text{not}(\text{and}(\text{not}(s), \text{not}(t))) )</th>
</tr>
</thead>
</table>

The deduction process terminates. The required network is depicted in Figure 4.6.

**Disjoint parallel composition rule**

\[
\begin{align*}
\text{not} & \quad \text{and} & \quad \text{not} & \quad y \\
\text{not} & \quad \text{and} & \quad \text{not} & \quad y \\
\end{align*}
\]

Figure 4.6
We need an additional rule for synthesizing the disjoint parallel composition operator. The rule is as follows.

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
<th>outputs $f(\overline{s})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1(\overline{s}_1) \wedge P_2(\overline{s}_2)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\wedge R_1(\overline{s}_1, y_1, \ldots, y_k)$</td>
<td>$R_2(\overline{s}<em>2, y</em>{k+1}, \ldots, y_n)$</td>
<td>$(y_1, \ldots, y_n)$</td>
</tr>
<tr>
<td>$\wedge R_2(\overline{s}<em>2, y</em>{k+1}, \ldots, y_n)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$Tableau_1 \wedge Tableau_2$</td>
<td></td>
<td>$/\left(g(\overline{s}_1), h(\overline{s}_2)\right)$</td>
</tr>
</tbody>
</table>

where $\overline{s}_1 = (s_1, \ldots, s_i)$ and $\overline{s}_2 = (s_{i+1}, \ldots, s_m)$. Two new tableaus are created:

**Tableau_1**

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
<th>outputs $g(\overline{s}_1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1(\overline{s}_1)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\wedge R_1(\overline{s}_1, y_1, \ldots, y_k)$</td>
<td></td>
<td>$(y_1, \ldots, y_k)$</td>
</tr>
</tbody>
</table>

**Tableau_2**

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
<th>outputs $h(\overline{s}_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_2(\overline{s}_2)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\wedge R_2(\overline{s}<em>2, y</em>{k+1}, \ldots, y_n)$</td>
<td></td>
<td>$(y_{k+1}, \ldots, y_n)$</td>
</tr>
</tbody>
</table>
The meaning of all this is: when Tableau$_1$ and Tableau$_2$ terminate, their results will replace $g(\tilde{s}_1)$ and $h(\tilde{s}_2)$ in the term $//(g(\tilde{s}_1), h(\tilde{s}_2))$. This is the result of the original tableau. The intuition behind this rule is as follows: if the input and output ports of a process $f$ can be partitioned into two sets that act independently of each other, then $f$ could be the disjoint parallel composition of two processes $g$ and $h$. It is easy to see that the rule is correct.

4.4.2.2. Examples

Let $id$, $fstwo$ and $plus$ be three primitive processes defined as follows.

- $id$ has one input port and one output port.

$$id(s) = s$$

- $fstwo$ has one input port and one output port.

$$fstwo([ ]) = [ ]$$

$$fstwo([a]) = [a]$$

$$fstwo(s) = [s(1), s(2)], \text{ if } |s| \geq 2$$

- $plus$ has two input and one output ports.

$$plus(s, t)(n) = s(n) + t(n)$$

Now, suppose that we want to synthesize a network that, given input sequences $r, s, t$ satisfying $|s|, |t| \geq 2$, produces output sequences $y, z$ satisfying

$$y = r$$

$$z = [s(1) + t(1), s(2) + t(2)]$$
To solve this, we first form the tableau

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
<th>outputs $f(r, s, t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>s</td>
<td>\geq 2 \land</td>
</tr>
<tr>
<td></td>
<td>$y = r \land$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$z = [s(1) + t(1), s(2) + t(2)]$</td>
<td>(y, z)</td>
</tr>
</tbody>
</table>

It is clear from the tableau that $y$ is independent of $s$ and $t$, and $z$ is independent of $r$. Thus the disjoint parallel composition rule can be applied. We obtain a new row

| $Tableau_1 \land Tableau_2$ | $// (g(r), h(s, t))$ |

and new tableaus

$Tableau_1$

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
<th>outputs $g(r)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$y = r$</td>
<td>$y$</td>
</tr>
</tbody>
</table>
\textit{Tableau}_2

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
<th>outputs $h(s, t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>s</td>
<td>\geq 2 \land</td>
</tr>
</tbody>
</table>

Apply the GA-resolution rule to \textit{Tableau}_1, and use the definition of \textit{id} as an assertion, we obtain

\[
\begin{array}{c|c|c}
\text{T} & \text{id}(r) & \text{z} \\
\end{array}
\]

Apply a simple transformation to \textit{Tableau}_2, using the definition of \textit{plus}

\[
\begin{array}{c|c|c}
\text{z} = [\text{plus}(s, t)(1), \text{plus}(s, t)(2)] & \text{z} \\
\end{array}
\]

Apply another transformation, using the definition of \textit{fstwo}

\[
\begin{array}{c|c|c}
|\text{plus}(s, t)| \geq 2 \land z = \text{fstwo(plus}(s, t)) & \text{z} \\
\end{array}
\]

Assume that this assertion is in our general knowledge:

\((|s| \geq m \land |t| \geq m) \Rightarrow |\text{plus}(s, t)| \geq m\)

Applying the AA-resolution rule and using this assertion and the initial assertion in \textit{Tableau}_2, we obtain
|\text{\textit{plus}}(s, t)| \geq 2

Now apply the GA-resolution rule, using this new assertion and the most recently created goal, to obtain

\begin{center}
\begin{tabular}{|c|c|}
\hline
\multicolumn{2}{|c|}{z = \textit{fstwo}(\textit{plus}(s, t))} \\
\hline
\end{tabular}
\end{center}

Apply the GA-resolution rule, using the assertion \( x = x \)

\begin{center}
\begin{tabular}{|c|c|}
\hline
\multicolumn{2}{|c|}{T} \\
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{|c|}
\hline
\textit{fstwo}(\textit{plus}(s, t)) \\
\hline
\end{tabular}
\end{center}

The desired network is depicted in Figure 4.7.

\textbf{Cycle formation rule}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure47.png}
\caption{Figure 4.7}
\end{figure}
This rule tells us when the desired process could be a cycle formation of another process.

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
<th>outputs $f(\bar{s})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q(\bar{s})$</td>
<td>$R(\bar{s}, y_1, \ldots, y_n, x)$</td>
<td>$(y_1, \ldots, y_n)$</td>
</tr>
<tr>
<td>$Tableau_1$</td>
<td>$C_{j_{n+1}}^{i_{m+1}}(g)(\bar{s})$</td>
<td></td>
</tr>
</tbody>
</table>

where $x$ is a variable and $\bar{s} = (s_1, \ldots, s_m)$. $g$ is a new process that has one input port and one output port more than $f$. $i_{m+1}$ and $j_{n+1}$ are $g$'s new ports that have new constant $s_{m+1}$ and variable $y_{n+1}$ as their values. A new tableau is created for $g$ that expresses the constraints on $s_{m+1}$ and $y_{n+1}$.

$Tableau_1$

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
<th>outputs $g(\bar{s}, s_{m+1})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q(\bar{s})$</td>
<td>$s_{m+1} = y_{n+1} \Rightarrow$ $R(\bar{s}, y_1, \ldots, y_{n+1})$</td>
<td>$(y_1, \ldots, y_{n+1})$</td>
</tr>
</tbody>
</table>

Intuitively, this rule states that if a goal contains a variable $x$ that is not an output variable, then $x$ could be the value of a linked port of the desired network, i.e. the network is a cycle formation of another one.
Lemma 4.4.2.1: The cycle formation rule is correct.

Proof: Let \( P \) be a process having \( f_1, \ldots, f_{n+1} \) as its set of functions such that, given any input sequences \( s_1, \ldots, s_{m+1} \) satisfying \( Q(s_1, \ldots, s_m) \), \( P \) produces output sequences \( y_1, \ldots, y_{n+1} \) satisfying \( s_{m+1} = y_{n+1} \Rightarrow R(s_1, \ldots, s_m, y_1, \ldots, y_{n+1}) \), i.e. \( P \) is a solution of Tableau\(_1\). Let \( \bar{s} \) denote \( (s_1, \ldots, s_m) \), it follows that

\[
(4.4.2.1) \quad s_{m+1} = f_{n+1}(\bar{s}, s_{m+1}) \Rightarrow R(\bar{s}, f_1(\bar{s}, s_{m+1}), \ldots, f_{n+1}(\bar{s}, s_{m+1}))
\]

Now consider \( C'_{n+1}(P) \), as depicted in Figure 4.8. We show that it is the solution of the original tableau.

Let \( z_0 \) be the least solution of the equation \( z = f_{n+1}(\bar{s}, z) \). Since \( f_{n+1} \) is continuous, such a solution always exists. Replacing \( s_{m+1} \) in \( (4.4.2.1) \) with \( z_0 \), we have

![Figure 4.8](image-url)
\(z_0 = f_{n+1}(\vec{s}, z_0) \Rightarrow R(\vec{s}, f_1(\vec{s}, z_0), \ldots, f_{n+1}(\vec{s}, z_0)),\)

which is equivalent to

\(T \Rightarrow R(\vec{s}, f_1(\vec{s}, z_0), \ldots, f_{n+1}(\vec{s}, z_0)),\) i.e.

\((4.4.2.2) \quad R(\vec{s}, f_1(\vec{s}, z_0), \ldots, f_{n+1}(\vec{s}, z_0))\)

By the semantics given earlier, \(C_{n+1}^{*}(P)\) is the set of functions \(g_1, \ldots, g_n,\)

where

\(g_i(\vec{s}) = f_i(\vec{s}, z_0), \text{ for } t = 1, \ldots, n.\)

This fact, together with \((4.4.2.2),\) gives

\(R(\vec{s}, g_1(\vec{s}), \ldots, g_n(\vec{s}), f_{n+1}(\vec{s}, z_0)),\) i.e.

\(\exists x R(\vec{s}, g_1(\vec{s}), \ldots, g_n(\vec{s}), x)\)

So, if \(C_{n+1}^{*}(P)\) is given inputs \(\vec{s}\) satisfying \(Q(\vec{s})\), it produces outputs \(z_1, \ldots, z_n\)

satisfying \(\exists x R(\vec{s}, z_1, \ldots, z_n, x)\), i.e. it is the solution of the original tableau.

The cycle formation rule is correct. ■

4.5. Examples

By convention, we let \(a + s(0) = a\) for any sequence \(s\), i.e. \(s\) starts from \(s(1)\). Similarly, \(\sum_{i=1}^{0}(\text{any expression}) = 0.\)

Example 4.5.1: Let \textit{fanout} and \textit{plusD} be two primitive processes defined as follows.
• fanout has one input and two output ports.

\[ \text{fanout}(s) = (\text{fanout}_1(s), \text{fanout}_2(s)) \]

\[ \text{fanout}_i(s) = s, \text{ for } i = 1, 2. \]

• plusl has two input and one output ports.

\[ \text{plusl}(s, t)(n) = s(n) + t(n-1) \]

Now, suppose that we want to synthesize a network that reads a sequence \([a_1, a_2, a_3, ...]\) and produces the sequence \([a_1, a_1 + a_2, a_1 + a_2 + a_3, ...]\). To solve this, we first form the tableau

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
<th>outputs ( f(s) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( y(n) = \sum_{i=1}^n s(i) )</td>
<td></td>
<td>( y )</td>
</tr>
</tbody>
</table>

By a simple algebraic transformation, we obtain a new row

| \( y(n) = s(n) + \sum_{i=1}^{n-1} s(i) \) |       | \( y \)            |

Introduce a new variable \( x \) through an algebraic transformation

| \( y(n) = s(n) + x(n-1) \land x(n) = \sum_{i=1}^n s(i) \) |       | \( y \)            |
Apply the cycle formation rule

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
& \textbf{Tableau}_1 & \textbf{C}_{j_2}^i(g)(s) \\
\hline
\end{tabular}
\end{table}

\textbf{Tableau}_1

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
<th>outputs $g(s, t)$</th>
</tr>
</thead>
</table>
| $T$         | $t = z \Rightarrow$
\quad $(y(n) = s(n) + t(n-1) \land$
\quad $z(n) = \sum_{i=1}^{n} s(i))$ | $(y, z)$ |

where $t$ is a constant denoting the input sequence on port $i_2$ and $z$ is a variable denoting the output sequence on port $j_2$.

Since outputs are often expressed in terms of inputs and seldomly in terms of other outputs, we change some of the occurrences of $z$ into $t$ and apply algebraic transformation rules

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
& $(y(n) = s(n) + t(n-1) \land$
\quad $z(n) = s(n) + \sum_{i=1}^{n-1} s(i))$ | $(y, z)$ |
\hline
\end{tabular}
\end{table}
\[
\begin{array}{|c|c|}
\hline
n = z \Rightarrow \\
(y(n) = s(n) + t(n-1) \land \\
z(n) = s(n) + z(n-1)) & (y, z) \\
\hline
\end{array}
\]

Apply the if-split rule

\[
\begin{array}{|c|c|}
\hline
t = z \\
y(n) = s(n) + t(n-1) \land \\
z(n) = s(n) + t(n-1) & (y, z) \\
\hline
\end{array}
\]

Using the definitions of \textit{plusI} and \textit{fanout} as transformation rules, we obtain

\[
\begin{array}{|c|c|}
\hline
y(n) = \text{plusI}(s, t)(n) \land z(n) = \text{plusI}(s, t)(n) & (y, z) \\
\hline
y = \text{plusI}(s, t) \land z = \text{plusI}(s, t) & (y, z) \\
\hline
y = \text{fanout}_1(\text{plusI}(s, t)) \land z = \text{fanout}_2(\text{plusI}(s, t)) & (y, z) \\
\hline
\end{array}
\]

Apply the GA-resolution rule, using the assertion \(x = x\)
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>$(\text{fanout}_1(\text{plusI}(s, t)), \text{fanout}_2(\text{plusI}(s, t)))$</td>
</tr>
<tr>
<td>$T$</td>
<td>$\text{fanout}(\text{plusI}(s, t))$</td>
</tr>
</tbody>
</table>

The deduction process terminates. The desired process is depicted in Figure 4.9.

**Example 4.5.2:** Let *and*, *not*, *fanout* and *addT* be primitive processes, where *and*, *not* and *fanout* are as before, and

- $\text{addT}(s)(1) = T$

  $\text{addT}(s)(n+1) = s(n)$

*addT* adds the element $T$ to the head of the input sequence.

![Diagram](image)  

**Figure 4.9**
Suppose we want to synthesize a network that reads an input sequence \( s \) and produces an output sequence \( y \) such that

\[
y(1) = F
\]
\[
y(n+1) = (s(n+1) \land \neg y(n))
\]

To solve this, we first form the tableau

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
<th>outputs ( f(s) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| \( y(1) = F \land y(n+1) = (s(n+1) \land \neg y(n)) \) | \( y \) |}

We first introduce a new variable through an algebraic transformation

\[
y(1) = (s(1) \land F)
\land y(n+1) = (s(n+1) \land \neg x(n+1))
\land x(n+1) = y(n)
\]

The natural thing to do now is to put \( y(1) \) in the same form as \( y(n+1) \)

\[
y(1) = (s(1) \land \neg x(1))
\land y(n+1) = (s(n+1) \land \neg x(n+1))
\land x(1) = T \land x(n+1) = y(n)
\]

\[ y \]
Apply the cycle formation rule

\[
\begin{array}{|c|c|c|}
\hline
Tableau_1 & & C_{j2}(g)(s) \\
\hline
\end{array}
\]

\textit{Tableau}_1

<table>
<thead>
<tr>
<th>assertions</th>
<th>goals</th>
<th>outputs $g(s, t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>$t = z \Rightarrow (y(1) = (s(1) \land \neg z(1))$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\land y(n+1) = (s(n+1) \land \neg z(n+1))$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\land z(1) = T \land z(n+1) = y(n))$</td>
<td>$(y, z)$</td>
</tr>
</tbody>
</table>

Apply another algebraic transformation rule

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>$(y, z)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t = z \Rightarrow (y(n) = (s(n) \land \neg z(n))$</td>
<td>$\land z(1) = T \land z(n+1) = y(n))$</td>
<td>$(y, z)$</td>
</tr>
</tbody>
</table>

Since outputs are often expressed in terms of inputs and seldomly in terms of other outputs, we change some of the occurrences of $z$ into $t$

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>$(y, z)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t = z \Rightarrow (y(n) = (s(n) \land \neg t(n))$</td>
<td>$\land z(1) = T \land z(n+1) = y(n))$</td>
<td>$(y, z)$</td>
</tr>
</tbody>
</table>
\[ t = z \Rightarrow (y(n) = (s(n) \land \neg t(n))) \]
\[ \land z(1) = T \land z(n+1) = (s(n) \land \neg t(n))) \]

\[(y, z)\]

Apply the if-split rule and the definition of \textit{not}

<table>
<thead>
<tr>
<th>[ t = z ]</th>
<th>[(y, z)]</th>
</tr>
</thead>
</table>
| \[ y(n) = (s(n) \land \neg t(n)) \]  
\[ \land z(1) = T \land z(n+1) = (s(n) \land \neg t(n))) \] | \[(y, z)\] |
| \[ y(n) = (s(n) \land \text{not}(t)(n)) \]  
\[ \land z(1) = T \land z(n+1) = (s(n) \land \text{not}(t)(n)) \] | \[(y, z)\] |

Apply the definitions of \textit{and}, \textit{addT} and \textit{fanout}

| \[ y(n) = \text{and}(s, \text{not}(t))(n) \]  
\[ \land z(1) = T \land z(n+1) = \text{and}(s, \text{not}(t))(n) \] | \[(y, z)\] |
| \[ y = \text{and}(s, \text{not}(t)) \]  
\[ \land z = \text{addT}(\text{and}(s, \text{not}(t))) \] | \[(y, z)\] |
| \[ y = \text{fanout}_1(\text{and}(s, \text{not}(t))) \]  
\[ \land z = \text{addT}(\text{fanout}_2(\text{and}(s, \text{not}(t)))) \] | \[(y, z)\] |

Finally, apply the GA-resolution rule, using the assertion \( x = x \)
The deduction process terminates, and the desired network is depicted in Figure 4.10.
CHAPTER 5

DISCUSSION

5.1. Model and proof system

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 trace-based models [BA, Br, Ho83, P] but is more general and expressive. It 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. [BKP, MP81a, MP81b, MP83]. The proof system is also compositional. Soundness and relative completeness proofs of the proof system are straightforward.

What remains to be done is to find a sound and complete set of axioms for the new temporal operators that we introduced in the proof system for sequential processes. The technique can also be extended to deal with the shared memory model of concurrency. Another direction for future research is to apply the proof technique to the verification of network protocols.

124
5.2. Language

We have shown how features of functional and concurrent languages can be combined into a parallel functional language. The language is simple and yet more expressive than Hoare's CSP. This is due to PFL's general recursion scheme. PFL also demonstrates how communication could be handled in a functional language. We have kept side-effects to a minimum without making unnatural restrictions on the syntax of the language.

One direction for future research is to develop a formal proof system for the language. Our proof technique given in Chapter 2 could be used for this purpose. Another interesting problem is to add the notion of types to PFL and to implement the language.

5.3. Program synthesis

We have also presented a deductive system for synthesizing networks of processes. We laid down the logical foundation without considering the strategic aspects of how deductions are directed. The framework described relies on a theorem-proving approach and follows closely that of [MW80, MW82] for sequential programs.

What we can do next is to consider the strategic aspects of how deductions are directed. This is necessary if we want to implement the deductive system.
Another problem is to extend the system to deal with recursive networks.

A possible different approach is to find synthesis techniques that use our model and temporal proof system in Chapter 2 as the basis (instead of Kahn's semantics). This will make the synthesis system more expressive and general.
REFERENCES


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


