The Expressiveness of Indeterminate Dataflow Primitives

Vasant Shanbhogue
Ph.D Thesis

TR 90-1147
August 1990

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

THE EXPRESSIVENESS OF INDETERMINATE DATAFLOW PRIMITIVES

A Dissertation

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

Vasant Shanbhogue

May 1990

THE EXPRESSIVENESS OF INDETERMINATE DATAFLOW PRIMITIVES

Vasant Shanbhogue, Ph.D.

Cornell University 1990

This thesis establishes that there are different kinds of indeterminacy in an asynchronous distributed computation setting by studying expressiveness and inexpressiveness situations. We use a particular model of asynchronous distributed computation, called the dataflow model. This model very naturally portrays the situation of autonomous computing agents communicating asynchronously to each other along fixed one-way paths called channels. The nature of computation is studied both in an operational setting, and in a more abstract setting, and equivalences are proved between them so that one may freely move between them. We use the operational semantics of Lynch and Stark to describe the operational behaviour of processes in the dataflow model. We show how one can abstract out the low-level operational behaviour and obtain "traces" that are well-suited for reasoning about network behaviour, once the properties of trace sets have been fully described using the operational semantics from which they arise.

We consider several forms of indeterminacy in this context, modeling them as

different fairness guarantees on merge primitives, that try to merge two sequences of values into one, and choice primitives, that split one sequence of values into two. The main contribution here has been to show that there is a surprising hierarchy of different notions of indeterminacy. This cannot simply be described using degree of branching – bounded versus unbounded. We use the trace sets of these primitives for these proofs. The description of this hierarchy clarifies the expressibility situation for indeterminacy in an asynchronous distributed setting. It is our hope that by concentrating on specific properties of agents that are really relevant to their behaviour in any particular system, one would obtain simpler and more convenient semantics to describe this behaviour.

In most of this thesis, we consider static dataflow – a fixed set of processes communicating along a fixed set of channels. In one of the final chapters, we consider recursively defined dataflow networks whose behaviour involves the creation of processes and channels. We prove the equivalence of an operational and an abstract semantics there for determinate networks.

Biographical Sketch

Vasant Shanbhogue was born in Mangalore, India on November 10, 1963. He completed twelve years of schooling in St. Lawrence High School in Calcutta, India. He then enrolled at the Indian Institute of Technology, Kanpur, and graduated in 1986 with a Bachelor of Technology degree in Computer Science. Desiring to pursue an academic career, he accepted admission into the doctoral program at the Computer Science Department at Cornell University. He married a very lovely lady, Chitra, in January 1989, and completed the requirements for a doctoral degree with her support. He will be awarded a PhD in Computer Science by Cornell University in May 1990.

To my parents, my sister Anuradha and my wife Chitra

Acknowledgements

I chiefly wish to thank my advisor, Prakash Panangaden, for initially suggesting my thesis problem, encouraging me in my work, and suffering me during my moody moments. He has always been very enthusiastic about and receptive to new ideas, and at the same time, he has encouraged the sifting of ideas to separate the gold from the chaff. I also wish to thank Robert Constable, Richard Shore and Anil Nerode for serving on my committee.

I wish to thank my wife, Chitra, for being a wonderful life-partner, and supporting me through the dreary days of thesis writing. I thank my friends Samir, Sanjay, Sanjeev, Shyam, Rakesh, Mukta, Desh, Devdatt and Radha for their company and their help in making Ithaca a fun place to be for me. I especially wish to thank Shyam and Sanjeev for their hospitality when I first arrived in Ithaca, and Radha for many stimulating discussions and for carefully reading through my thesis.

Table of Contents

List of Figures

Chapter 1

Introduction

When processes or programs can execute in parallel, we encounter a phenomenon called *concurrency*, arising from the presence of multiple threads of control. This introduces a new dimension to the study of program execution. One fruitful avenue is the study of *network models* to understand the effect of concurrency. In this thesis, we are interested in understanding "computation" in a specific network model, called the *dataflow model.* This is the conceptual framework that we will use for exploring issues that arise with concurrency. Ashcroft and Wadge [WA85] designed the programming language Lucid around the ideas in this model.

A *dataflow network* is defined to be a directed graph, containing a special node $\mathcal{E}$, with autonomous computing agents at all the nodes of the graph except $\mathcal{E}$, such that communication in the network takes place by message transmission along the direction of an arc. Each arc behaves like a unidirectional queue of unbounded size, and message transmission takes place in a FIFO (first in first out) manner with an unbounded but finite delay. The arcs directed from $\mathcal{E}$ to other nodes are called *input arcs*, and the arcs directed to $\mathcal{E}$ from other nodes are called *output arcs.* The node $\mathcal{E}$ represents the environment of the network, and this environment can communicate messages along the input arcs to computing agents in the network as well as receive messages along the output arcs from computing agents in the network. We can equivalently view a dataflow network, by removing the node $\mathcal{E}$, as a directed graph with some extra arcs into the directed graph and out of the directed graph. This is how we will view dataflow networks in the rest of this thesis. The mode of communication along any arc will be asynchronous – the sender of a message and its receiver do not have to synchronize for a message transmission. A node can send a message and then continue its execution, without waiting for the receiver to have actually received the message. We will also refer to the arcs of a dataflow network as *channels.* In the above, we consider a dataflow network to be a fixed network with a fixed set of nodes and arcs, and no new nodes or arcs can get created during computation. This is referred to as *static dataflow,* as opposed to *dynamic dataflow,* in which new nodes and arcs get created during computation. We will consider dynamic dataflow only much later in this thesis. Our dataflow model has also been referred to as *pipeline* dataflow or *stream flow.*

In the first subsection below, we describe some elementary domain theory. In the next subsection, we present Kahn's view [Kah74] of dataflow networks, and a very elegant semantic principle. Kahn's study only involved concurrency, and did not involve indeterminacy. In the following subsections, we introduce the notion of indeterminacy and its importance, and then we will describe and motivate the abstractions of fairness that we study in this thesis.

1.1 Some elementary domain theory

In this subsection, we will define the standard mathematical domains and terminology that arise in the Scott-Strachey approach to programming language semantics. We will need to refer to this terminology when we consider the semantics for our dataflow model. All of this terminology is standard.

A *partially ordered set* is a pair $(S, \sqsubseteq)$ such that S is a set and $\sqsubseteq$ is a reflexive, antisymmetric and transitive relation on S. $(S, \sqsubseteq)$ is said to be *directed* if it is non-empty and for any $x, y \in S$, there exists a $z \in S$ such that $x \sqsubseteq z$ and $y \sqsubseteq z$. $(S, \sqsubseteq)$ is called a *complete partial order (cpo)* if it has a least element, and any directed subset of S has a least upper bound in S. The least element is also referred to as the *bottom element.* We will use the notation $\sqcup X$ for the least upper bound, if it exists, of a subset X of a cpo.

If D is a cpo, then an element $d \in D$ is called *finite* if, for any directed subset X of D, $d \sqsubseteq \sqcup X$ implies that $d \sqsubseteq x$ for some $x \in X$. A cpo D is called *algebraic* if, for any $d \in D$, the set $S_d = \{x | x \sqsubseteq d \text{ and } x \text{ finite}\}$ is directed, and $d = \sqcup S_d$. If, in addition, the set of finite elements is countable, then D is called *ω-algebraic.*

A cpo D is called *bounded-complete* if, for any subset X of D, if X has an upper bound in D, then it has a least upper bound in D.

Given cpos D and E, a function f from D to E is called *monotone* if $x \sqsubseteq y$ in D implies that $f(x) \sqsubseteq f(y)$ in E. A monotone function f is called *continuous* if, for any directed subset X of D, $f(\sqcup X) = \sqcup f(X)$, where $f(X) = \{f(x) | x \in X\}$ and the directedness of X implies the directedness of $f(X)$.

Given cpos D and E, the product $D \times E$ is defined to be the set $\{\langle x, y\rangle | x \in D \text{ and } y \in E\}$. It is a cpo under the componentwise ordering. Finite products of ω-algebraic and bounded-complete cpos are also ω-algebraic and bounded-complete.

1.2 Kahn's Principle

In [Kah74], Kahn formally modeled networks of communicating nodes, as we have described it, with some additional stipulations that we will describe presently. The important point was that Kahn introduced a simple but elegant application of fixed point theory to the study of concurrently executing nodes. He was concerned with a "denotational" modeling of such systems, by which we mean that nodes are described by their actions over entire (possibly infinite) computation periods rather than a step at a time.

Let V be a countable value alphabet. Let V^* denote the set of finite sequences of elements from V, and let V^∞ denote the set of finite and infinite sequences of elements from V. Then V^∞ is an ω-algebraic, bounded-complete cpo under the prefix ordering on sequences. The empty sequence is the bottom element of the cpo. We will refer to the members of V^∞ as streams, and to V^∞ as the domain of streams. We will use the symbol Λ to represent the empty stream, the stream with no elements in it.

If we think of the elements of V as values that may be communicated by one node to another along a communication channel, then we may think of a stream as a possible sequence of communications along some channel. So each channel may be associated with a domain of streams representing all the possible sequences of communications that can happen along that channel. If a channel is directed into a node, then we will refer to it as an *input channel* of the node. We will refer to the product of the domains of streams associated with all the input channels as the *input stream domain* of the node. If a channel is directed out of a node, then we will refer to it as an *output channel* of the node. We will refer to the product of the domains of streams associated with all the output channels as the *output stream domain* of the node. In [Kah74], Kahn dealt only with nodes whose programs

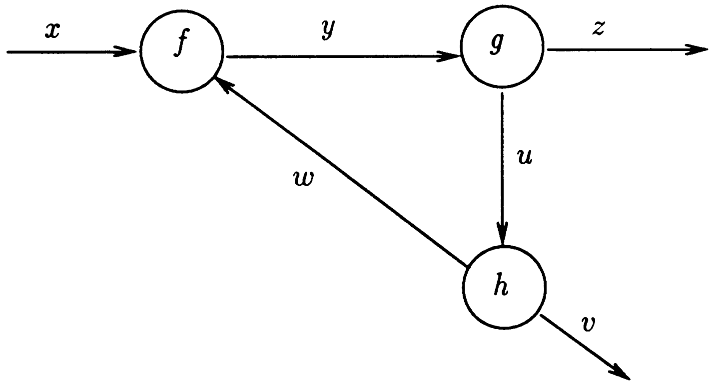

Figure 1.1: A dataflow network

"computed" continuous functions from the input stream domain to the output stream domain. Given a dataflow network with every node A in the network (except the environment $\mathcal{E}$) denoted by a continuous function f_A, and every channel associated with a stream variable, we can write down a set of equations from the syntactic description of the network describing the relationship between these stream variables. A solution of this set of equations is a tuple of values for all the stream variables. The claim is that if the streams along the input arcs of the network are fixed, then the least solution, in the componentwise prefix ordering on tuples of streams, accurately and exactly describes the behaviour of the entire network with the fixed streams at the input arcs. This is *Kahn's principle.* We illustrate this with an example. Figure 1.1 describes a dataflow network, along with the stream variables associated with the channels and the names of the continuous functions denoting the nodes.

We can write down the following equations :

$$\begin{aligned} y &= f(x, w) \\ (z, u) &= g(y) \end{aligned}$$

$$(w, v) = h(u)$$

The equations describe the relationship between the inputs and outputs for every node in the network.

The importance of using fixed-point theory to explain the behaviour of a network is that we now have powerful induction tools to prove properties about networks [Man74].

If the program at a node consists only of some specific kinds of commands, then Kahn observed [Kah74,Kel78] that the node may be denoted by a continuous function as outlined above. The kinds of commands allowed were: (1) internal commands, that operate only on variables internal to a node program, (2) write commands – a write command transmits the value of an internal variable out onto an output channel, and (3) read commands – a read command waits for a value to appear at a specific input channel, and then copies the value into an internal variable. In this way, a node program can be viewed as a sequence of these commands, as in any standard imperative language. This is, by no means, the only way to view node programs. In fact, we can study the behaviour of networks with any particular syntax and semantics of node programs. Keller [Kel78] has proposed extending the above list of commands by new kinds of commands called poll commands, choice commands and extended read commands. These new commands introduce indeterminacy into the language of the node program.

Given a particular syntax and semantics of node programs, we can associate a node with its *input-output relation* – a set of pairs of tuples of streams, describing which streams can result at the output channels of the node for which streams at the input channels of the node as a result of computation of the node. For Kahn's language, the input-output relation is a function.

1.3 Indeterminacy

In the previous section, we mentioned that the language in which node programs are written can be such that the resulting programs exhibit indeterminacy – notably, for a fixed input, a program may have more than one possible output. The input-output relation of such a program is not the graph of some function. The importance of considering indeterminacy lies in the fact that many real systems are indeed indeterminate, and this indeterminacy is observed every day. For example [Kel78], the set of passengers assigned seats on a given flight by an airline reservation system is not purely a function of passengers requesting seats. Also, real-time systems that react to their inputs may react differently depending on the order in which the inputs are perceived. Keller's syntax for programs allows us to write programs that exhibit indeterminacy. Instead of extending Kahn's language for node programs, we will take the view that indeterminacy is introduced by extending the set of node programs considered by Kahn with some specific node programs that capture the essence of different kinds of indeterminacy and fairness. We call these special node programs *indeterminate primitives.*

We now describe a widely studied phenomenon [Bro83,Kel78,Par80,Par82] in "systems programming." Consider a node, which we call *fair merge*, that has two input channels and one output channel. This node waits for values to appear on either of its two input channels, and if there are any such values, then the node reads the values and outputs them onto the output channel. The node waits for values by repeatedly checking each of the two input channels in order for values – this is called *polling* – and reading and outputting a value, if it is present. This behaviour is clearly *time-sensitive*, and the output depends on the rate of arrival of values at the input channels. This kind of phenomenon is described by Keller's extended read command [Keller78] – when the node executes an extended read command for a particular input channel, then either a value is read, if it exists, or a special value '?' is returned. Then the fair merge node can be programmed in Keller's language by a program that loops forever, executing extended read commands at each of the two input channels in order, and outputting the value read if it is not '?'. For any particular stream inputs, the node can then output one of every possible interleaving of the two input streams. Clearly, an analogue of the extended read command is used in real systems, usually by employing a "time-out" mechanism. Equally clearly, this leads to indeterminate behaviour. Many researchers have sought to understand this phenomenon and to formalize semantics of dataflow networks that provide a satisfactory treatment of the fair merge primitive.

Researchers were aware of the distinction between finite indeterminacy and unbounded indeterminacy – the difference between being able to make one of finitely many choices and one of infinitely many choices. In this thesis, we go beyond this distinction, and demonstrate that there is, in fact, an entire hierarchy of different kinds of indeterminacy. These different kinds are related to each other, and the way they are related was not at all clear before we started the work described in this thesis. In the next section, we describe the different primitives we study, as abstracting the different kinds of indeterminacy.

One final point we emphasize is that the differences between these kinds of indeterminacy helps us to understand exactly what kind of indeterminacy a particular set of primitives (or language constructs) possesses, because then one would not make assumptions that these primitives have a "stronger" kind of indeterminacy, and one could restrict one's study to the specific kind of indeterminacy exhibited.

An example of a primitive that we consider in some detail in this thesis is McCarthy's *amb*. McCarthy introduced this primitive to develop a full-fledged programming language *nondeterministic Lisp.* As originally presented by McCarthy, this is a function constant *amb* that has indeterminate behaviour. Specifically, the evaluation of the expression $amb(e_1, e_2)$ terminates with the value of one of e_1 or e_2, whichever terminates first. The evaluation of the expression does not terminate if neither e_1 nor e_2 terminates. An interesting example of the use of *amb* is to consider the recursive definition of a function f by $f(n) = amb(n, f(n+1))$, where n is an integer parameter. Now $f(0)$ is guaranteed to terminate, by the property of *amb*, and moreover, $f(0)$ may terminate with any non-negative integer as value. We express *amb* as a dataflow primitive as follows : the node has two input channels and one output channel. The node polls between the two input channels until it finds the contents of one of the input channels non-empty. Then the node outputs the entire contents of that input channel. For example, if the input streams of the node are 1 and 2, then the output stream is either 1 or 2. If the input streams are 1 and Λ, then the only possible output stream is 1. This behaviour superficially seems very similar to the polling mechanism of *fair merge*, but an important result [PS87,PS88b,PS88a] is that the indeterminacy of *amb* is "strictly weaker" than the indeterminacy of *fair merge.* So we cannot use the properties of *amb* to explain *fair merge.*

1.4 Fairness

This thesis is mainly concerned with the differences between different kinds of fairness. In this endeavour, we define some general properties of networks that give us some insight into the workings of distributed programs, and the ways in which they can be abstracted. The study of fairness is motivated by the study of liveness properties. Although there are some schools of thought that do not encourage the study of fairness, because differences in fairness are usually not "observable" in finite time, we think that it is useful to describe exactly what a system would compute throughout its entire computation. Fairness has been extensively studied in a variety of formalisms, and a number of different varieties of fairness may be found in [Fra86]. We study fairness in the dataflow network context by considering a number of primitives describing different kinds of indeterminacy and different kinds of fairness conditions.

We assume that our countable value alphabet V of values that may be communicated by one node to another in a network includes the non-negative integers. Let us then describe a primitive *anyint* with no input channels and one output channel. The node outputs any positive integer, and does nothing further. This node clearly exhibits unbounded indeterminacy. An iterated version of this primitive, which we will call *anyints*, has no input channels and one output channel, and this node outputs any infinite sequence of positive integers.

Till recently, fairness, as exemplified by fair merge, had been identified with countable indeterminacy [Apt83], and it was believed that a satisfactory theory for countable indeterminacy would provide a satisfactory theory for fair merge. This belief was ill-founded, because it was proved [PS87,PS88b] that the ability of fair merge to read every input value and not get "stuck" at an empty input channel forever waiting for a value cannot be explained by unbounded indeterminacy.

Plotkin's pioneering study of powerdomains for indeterminacy included the observation that it had been specifically designed for bounded indeterminacy and as such excluded the study of fair systems [Plotkin76]. Several people have worked on generalizations of powerdomain techniques that would apply to unbounded indeterminacy [Abr83,AP86,Bro83,Par80,Par82,Plo82], and considerable effort was expended in formalizing semantics of dataflow networks that would include a satisfactory treatment of the fair merge primitive [Bro83,dBKM84,KP85,Par82, Pan85]. All these efforts seemed to take the view that this was the next natural step after Plotkin's work. In this thesis, we will demonstrate that there are many levels of indeterminacy. The differences seem to suggest that it may not be appropriate to study all these levels at the same time, as has been the trend. Furthermore, it suggests that the degree of branching – bounded versus unbounded – is not the best way to classify indeterminacy.

The fairness properties that we now describe fall into two groups – fairness in the selection of input channels to read from, and fairness in the selection of output channels to write to. The first kind of fairness is described by different kinds of "merge primitives." A merge node has two input channels and one output channel, and the primitive reads its input values, possibly not all of them, in some order and outputs them onto the output channel. The second kind of fairness is described by what we call "split primitives." A split node has one input channel and two output channels. It reads every one of its input values and outputs each value on one of the two output channels. So the primitive splits the input stream into two substreams, outputting them on the two output channels.

We describe five merge primitives. The first one is *fair merge*, which we have already described. The important point to note here is that this primitive reads (and outputs) every one of its input values on both its input channels. The second primitive is *angelic merge*. This is an iterated version of McCarthy's *amb*. *Angelic merge* may be partially described by saying that if the input stream at one of the input channels is finite, then every value at the other input channel will be read and output. This implies that *angelic merge* behaves like *fair merge* when both its input streams are finite. Otherwise, it outputs an interleaving of one of the infinite input streams and a prefix (not necessarily proper) of the other input stream. Intuitively, one may think of this primitive as follows : as long as both the input channels have values left to be read, the primitive reads from either one. But if only one of the input channels has values left to be read, then the primitive reads those values. Notice how the choice to read from a non-empty input channel is exactly like *amb*.

The third primitive is called *infinity-fair merge*. It can be partially described by saying that if the input stream at one of the input channels is infinite, then every value at the other input channel will be read and output. This implies that *infinity-fair merge* behaves like *fair merge* when both its input streams are infinite. Otherwise, it outputs an interleaving of one of the finite input streams and a finite prefix of the other input stream. We may also think of this primitive in the following way : the node reads some number of values from one input channel, some number of values from the other input channel, then it again reads some number of values from the first input channel, and some number of values from the second input channel, and so on. The number of values to be read each time from a channel is obtained from an "oracle" that behaves like the *anyints* primitive. This primitive exactly captures (in a way to be made precise later) the unbounded indeterminacy of *anyints*.

We call the fourth primitive *infinity-fair2 merge*. It can also be partially described by saying that if the input stream at one of the input channels is infinite, then every value at the other input channel will be read and output. This implies that *infinity-fair2 merge* behaves like *fair merge* when both its input streams are infinite. Otherwise, it outputs an interleaving of one of the finite input streams and a prefix (not necessarily finite, as in *infinity-fair merge*, and not necessarily proper) of the other input stream. The key difference between this primitive and *infinity-fair merge* is that when at least one of the input streams is finite, *infinity-fair merge* cannot avoid getting "stuck," waiting forever for values at an input channel after reading the entire finite input stream there. On the other hand, *infinity-fair2 merge* may or may not get "stuck" in this manner.

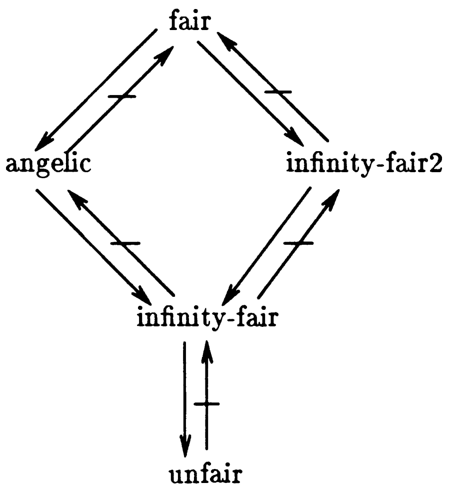

Figure 1.2: The five merges

We call the fifth primitive *unfair merge.* This can be described by saying that it outputs an interleaving of one input stream and a prefix (not necessarily proper) of the other input stream. Intuitively one may think of this primitive as follows : at each step, it makes a decision to read from one of its input channels, and if there is an unread value at that channel, it is read and output, otherwise the primitive gets "stuck."

The relationship between these five primitives is shown in Figure 1.2. An arrow from one primitive to another means that the first primitive "can implement" the second primitive. An arrow with a slash through it from one primitive to another means that the first primitive "cannot implement" the second primitive. Informally, if there is an arrow from A to B, then we can simulate B with A, and if there is an arrow with a slash through it from A to B, then we can prove that it is impossible to simulate B with A.

We will now illustrate the differences between these five primitives by examples. We consider three cases :

(a) the input streams are 1^∞ and 2^∞ : fair merge can output every possible interleaving of 1^∞ and 2^∞. Angelic merge can output every possible stream that fair merge can. In addition, angelic merge can output every possible interleaving of 1^∞ and 2^n for any non-negative integer n, and it can also output every possible interleaving of 1^m and 2^∞ for any non-negative integer m. Infinity-fair merge and infinity-fair2 merge can only output the streams that fair merge can. Unfair merge can only output the streams that angelic merge can.

(b) the input streams are 1^∞ and 2^n for some non-negative integer n : fair merge can output every possible interleaving of 1^∞ and 2^n. Angelic merge can output every possible interleaving of 1^∞ and $2^{n'}$ for every $0 \leq n' \leq n$. Note that this includes every stream that fair merge can output. Infinity-fair merge can output every possible interleaving of 1^m and 2^n for every $m \geq 0$. Infinity-fair2 merge can output all the streams that either fair merge or infinity-fair merge can output. Unfair merge can output all the streams that any of the other merges can output.

(c) the input streams are 1^m and 2^n for some non-negative integers n, m : Fair merge can output every possible interleaving of 1^m and 2^n. Angelic merge can output only the streams that fair merge can. Infinity-fair merge, infinity-fair2 merge and unfair merge can output all the interleavings of 1^m and $2^{n'}$ for every $0 \leq n' \leq n$, and also all the interleavings of $1^{m'}$ and 2^n for every $0 \leq m' \leq m$. Note that this includes the streams that fair merge can output.

We now turn to primitives exhibiting the second kind of fairness alluded to above, namely the split primitives. We first describe three such primitives. The first primitive, which we call *Unfair Split(US)* splits its input stream into two substreams, and it is possibly unfair in the sense that one output channel may receive no values even when the input stream is infinite.

The second primitive is called *Weakly Fair Split(WS)*. This guarantees that each of the two output streams will be non-empty when the input stream is infinite, and it offers no guarantees otherwise.

The third primitive is called *Strongly Fair Split(SS)*. This guarantees that each of the two output streams will be infinite when the input stream is infinite, and it offers no guarantees otherwise.

The difference between these three split primitives is in the guarantees that they make. We will show later that strongly fair split exhibits the same kind of indeterminacy as *anyints* and *infinity-fair merge*.

We notice that each split primitive breaks up its input sequence into two subsequences and outputs one subsequence at one output channel and the other subsequence at the other output channel. The primitive does not "tell" us how it did the break up – which particular input values are output at the first output channel, and which particular input values are output at the second output channel. We therefore now consider split primitives that give us this information. These are the original split primitives above enhanced with an extra output channel, that we will refer to as the *signal* channel. The primitives output a sequence s of 1's and 2's at the signal channel with the length of s being equal to the length of its input sequence, and with the intent that the ith value in s is a 1 if the ith value of the input sequence was output at the first output channel, and the ith value in s is a 2 if the ith value of the input sequence was output at the second output channel. We will refer to these three primitives as *Unfair Split with signal(USS)*, *Weakly Fair Split with signal(WSS)* and *Strongly Fair Split with signal(SSS)*.

1.5 Operational and Denotational Semantics

Two formal styles of semantics that have been developed for programming languages are *operational* semantics and *denotational* semantics. There is a third style of semantics, *axiomatic* semantics, that we do not deal with in this thesis. The operational semantics describes the execution of a program in terms of a sequence of transitions between states. This corresponds to describing an abstract machine and its computation. On the other hand, a denotational semantics works by defining a *meaning function* that maps every syntactic construct to an element of some mathematical domain. Moreover, this semantics must be compositional – the meaning of a syntactic construct can be expressed in terms of the meanings of pieces of the construct. The importance of the operational semantics lies in the fact that it precisely describes the actual working of a machine. But once a close tie-up has been established between the operational and the denotational semantics, we can utilize the compositional nature of the semantics in reasoning about programs, and we can also utilize whatever mathematical tools are available by virtue of working in a mathematical domain (the target of the meaning function).

We will describe an abstract machine as a particular kind of automaton. It has a set of events, that is partitioned into the set of input events, the set of output events and the set of internal events. A transition is a change of state of the automaton due to the occurrence of an event. The transitions corresponding to input events will be called input transitions, the transitions corresponding to output events will be called output transitions, and the rest of the transitions will be called internal transitions. We then abstract away from the internal transitions of the machine. We use *traces* as abstractions of node program behavior. By traces, we simply mean some specific sequences of input and output events.

Associating a node program with a set of traces comprises a denotational semantics. The main point of the denotational semantics is that it is compositional — the semantics of a network can be expressed in terms of the semantics of subnetworks. Moreover, the denotational semantics should abstract away irrelevant operational detail. Traces have been studied at various levels of mathematical rigor and have a rich mathematical theory [AR88]. It would have been preferable if we could have used purely extensional properties like the input-output relation but the well-known Brock-Ackerman example shows that the input-output relation computed by a network is not compositional [BA81]. Jim Russell [Rus89] has shown that the input-output relation is not compositional even in the presence of only bounded indeterminacy.

It turns out that traces are not only compositional but are *fully abstract* [Jon89,Kok88,Rus89], and we will present a proof of this. The operational semantics that we describe is closely modeled on work of Stark [PS88b,Sta87, Sta89b] and also on the input-output automata formalism of Lynch and Tuttle [LT87]. Regarding the full abstractness result, it is important to be aware that we (and Kok and Jonsson) view the complete output streams in response to given (possibly infinite) input streams to be observable. This is, of course, a rather liberal view of observability. It allows us to differentiate between programs that cannot be differentiated on the basis of finite execution sequences. For example, we would be able to differentiate between a program that outputs any finite length stream of 1's and a program that only outputs an infinite stream of 1's. Rabinovich and Trakhtenbrot have carried out a study of the semantics of dataflow networks with only *finite observations* permitted [RT88,RT89].

Chapter 2

Operational Semantics of Dataflow Processes

As we mentioned in the introduction, the operational semantics describes the execution of a program in terms of a sequence of transitions between states. This is very close to the actual low-level implementation and execution of the program with transitions corresponding to atomic commands that the underlying machine architecture supports. We do not wish to get tied down to any particular machine, so we wish to define an abstract machine that is always in one of some fixed set of states. Command execution of the machine is represented by events that cause transitions between states. Since this machine should be general, it should also support parallel execution of commands, and then the notion of fairness comes in – if there are two infinite sequences of commands that the machine can execute in parallel, then we should make sure that both the infinite sequences of commands are executed by the machine. This leads us to the necessity of defining legal or fair computations. This work is based on the work of Lynch [LT87,LS89] and E. W. Stark [Sta87,Sta89b].

Kahn's language for processes in his 1974 paper restricted processes to be "strictly sequential" — they could only execute one command at a time. So there can be no parallel execution within a process. Faustini [Fau82] searched for a more general description of processes – his aim was to be able to represent processes with multiple threads of control, rather than simply the processes with single threads of control as Kahn had. For example, consider a *double identity* process with two input channels and two output channels, that simply reads all values on its first input channel and outputs all of them on its first output channel, and it also reads all values on its second input channel and outputs all of them on its second output channel. This has two separate threads of control. On the other hand, we choose a description of processes that can be specialized to Kahn's and Faustini's descriptions, but it is sufficiently general for us to represent not only processes with multiple threads of control, but also indeterminate processes and also networks of processes.

Recall that we said that command execution of a process is represented by events, each of which cause a transition from one state to another. Each process will have a set of events associated with it, and we represent concurrency by a binary relation on events, that tells us when two events may happen in different orders but having the same end effect. This is the abstraction of describing when two commands can be executed in parallel. This binary relation, which we will call the *concurrency relation*, is axiomatized via equations that express the fact that the order of execution of concurrent events can be permuted, thus capturing the causal independence of concurrent events.

2.1 Port Automata

We now formally describe computing agents as automata, that can receive values at "input ports" and output values at "output ports." We use the term "port" instead of "channel" to emphasize that this is where an automaton interfaces with its environment. The set of events of an automaton comes equipped with a concurrency relation, that describes which pairs of events are causally independent and can be permuted in execution sequences.

Definition 1. A **concurrent alphabet** is a set E, equipped with a symmetric, irreflexive binary relation $\|_E$, called the **concurrency relation**.

This concept is used in trace theory [AR88,Maz86] to obtain an algebraic theory of traces. We will call events related by the concurrency relation *concurrent*. Let V be a set of **data values** called the *value alphabet*. Throughout this paper, we will assume a fixed countable value alphabet. We refer to V^∞ as the domain of streams. We use the term "stream" interchangeably with the term "value sequence."

We now describe the notion of an automaton that can execute events. The input and output events are described as (port,value) pairs. The rest of the events need not be of this form.

Definition 2. A **port automaton** is a tuple

$$M = (E, Q, A)$$

where

- E is a concurrent alphabet of **events**, and *Inp* and *Out* are disjoint subsets of E, called the sets of **input events** and **output events**, respectively. $Inp = P^{in} \times V$, and $Out = P^{out} \times V$, for some disjoint finite sets P^{in} and

P^{out}. The elements of P^{in} are called **input ports**, and the elements of P^{out} are called **output ports**. The elements of $E \backslash (Inp \cup Out)$ are called **internal events.**

- Q is a set of **states**, and $q^{\iota} \in Q$ is a distinguished **initial state**.
- A is a transition function that maps each pair of states q, r in Q to a subset $A(q,r)$ of $E \cup \{\epsilon\}$. ϵ, a special event not in E, is called the **identity event.**

satisfying the following conditions :

(Disambiguation) $r \neq r'$ implies $A(q,r) \cap A(q,r') = \emptyset$.

(Identity) $\epsilon \in A(q,r)$ iff $q = r$.

(Receptivity) For any state q and any input event a, there exists a state r such that $a \in A(q,r)$.

(Commutativity) For any state q and any events a, b, if $a \| b$, $a \in A(q,r)$ and $b \in A(q,s)$, then there exists a state p such that $a \in A(s,p)$ and $b \in A(r,p)$.

This definition is similar to the definitions of a *port automaton* and an *input-output automaton* due to Stark in [LS89,PS88b], and is closely related to the *input-output automata* of Lynch and Tuttle [LT87]. Disambiguation states that from a particular state, an event cannot take the automaton to two different states.

A basic property of systems is that they cannot control what their inputs are. They may, of course, ignore their inputs but they cannot determine their inputs, which are supplied by the external environment. To express this we would also like to have input events always "enabled." We can make the notion of enabling precise by saying that event a is *enabled* at state q if $a \in A(q,r)$ for some state r. The intent of input events is to represent arrival of data on input ports. The arrival of data on input ports should not be dependent on the state, and so, for any state and for any event corresponding to a value arriving on an input port, there is a new state corresponding to the value having arrived. This is captured by *receptivity.*

If two events are concurrent, i.e. are related by the concurrency relation, and if both of them are enabled in a particular state, then the execution of any one of these two events does not disable the other, and moreover, the execution of these events in either order results in the same final state. This is captured by *commutativity.*

The *transitions* of an automaton are the triples (q, a, r) with $a \in A(q, r)$. We will denote the transition (q, a, r) by $q \xrightarrow{a} r$. The transition $q \xrightarrow{\epsilon} q$ is called an *identity transition*, and is denoted by id_q.

One should note that there is a difference between the notions of *event* and *transition.* A transition describes two states and an event such that when the event is executed in the first state, the second state is reached. An event may execute in different states. For example, an "$x := x + 1$" event may be executed in a state in which x is 3, as well as in a state in which x is 4. But they will correspond to different transitions.

Definition 3. A **computation sequence** γ is a finite or infinite sequence of transitions of the form

$$q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \ldots$$

The *domain dom(γ)* of γ is the state q_1. A computation sequence is said to be *initial* if *dom(γ)* is the distinguished start state q^ι. Two computation sequences γ and δ are *coinitial* if *dom(γ) = dom(δ).*

In earlier work [PS88b,PSS90], automata in which inputs cannot disable other events were considered. These are called *monotone* automata, and these are port automata satisfying the following additional property.

Definition 4. *(Non-Disabling Inputs)* If e is an input event at an input port, then $e \| e'$ for any event e' that is not an input event at the same port.

As Example 2 below shows, not all port automata satisfy this property, and since we would like to be able to represent automata as in Example 2, we will not restrict our discussion to monotone automata in this thesis.

We will now give two examples of automata. We will use $^\wedge$ as an infix operator for representing concatenation of sequences.

Example 1. *Buffer* : This automaton has one input port and one output port. It reads values and outputs them, guaranteeing to read and output all values that arrive on the input port.

Let the set of states Q be V^*. A state here represents the contents of the input port. The initial state is Λ. Let the set of input events *Inp* be $\{i\} \times V$ and the set of output events *Out* be $\{o\} \times V$. Then the set of events E is *Inp*$\cup$*Out*.

We now define .e transition relation, using v to represent a member of V. $A(q,r) = \{(i,v)\}$ iff $r = q^\wedge v$. $A(q,r) = \{(o,v)\}$ iff $q = v^\wedge r$. $A(q,q) = \{\epsilon\}$.

Every event in *Inp* is concurrent with every event in *Out*, and ϵ is concurrent with any other event.

Example 2. *Poll* : This automaton has one input port and one output port. It repeatedly checks its input port for data. If a data value is present, then it is read and output. If not, a special value $\star$ is output.

Let Q and *Inp* be the same as for the previous example. Let the set of output events *Out* be the set in the previous example, together with an extra event $(o,\star)$.

Besides the transitions in the previous example, there is an extra transition, and we redefine $A(q,q)$. $A(q,q) = \{\epsilon,(o,\star)\}$ if $q = \Lambda$, and $A(q,q) = \{\epsilon\}$ otherwise. The only new thing added to the concurrency relation of the previous example is that ϵ is concurrent with the new event $(o,\star)$.

Notice that input events are not concurrent with $(o, \star)$, so that arrival of input disables the output of a $\star$. So the input here has the power to interrupt. This automaton is not monotone.

We end this section with some notation. We will refer to events of the form (p, v) as p-events. Also, we denote the value component v of an event $e = (p, v)$ by *value*(e). We also extend the definition of *value* to sequences – *value*$((p, v_1)(p, v_2)\ldots)$ is $v_1 v_2 \ldots$. For any computation sequence

$$\sigma = q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \ldots$$

we define $ev(\sigma)$, the *sequence of events of* σ to be $a_1 a_2 \ldots$. We will use the symbol Π as a projection operator on sequences of events. Given any sequence t of events and a set S of ports, we will use $\Pi_S(t)$ to represent the subsequence of t consisting of the p-events in t for all ports p in S. If S is a singleton set $\{p\}$, then we will use the notation $\Pi_p(t)$ instead of $\Pi_{\{p\}}(t)$. If t is the sequence of events of a computation sequence σ, then we also write $\Pi_S(\sigma)$ to mean the same thing as $\Pi_S(t)$, and $\Pi_p(\sigma)$ to mean the same thing as $\Pi_p(t)$. To project out the p-events of a sequence t, we define $\Pi_{\sim p}(t)$ to be the subsequence of t consisting of all the events of t except the p-events. We will also use the symbol $\sqsubseteq$ for the prefix relation between sequences of events. When we compare the projections of a sequence of events onto different ports, then we will follow the convention of implicitly applying *value* to the projections. We will use the notation $t[i]$ to represent the ith event in a sequence t of events, and the notation $\gamma[i]$ to represent the ith transition in a computation sequence γ.

2.2 Completed computations and the input-output relation

In this section, we describe which computation sequences of automata we view as "completed," i.e. cannot be extended further. Once we establish this, we then show how we can abstract the input-output behaviour of an automaton from its completed computation sequences.

We first describe the notion of a *history*. Let P be the set of input ports and output ports of an automaton. A *history over* P is defined to be a function from P to V^{∞}. Then for any computation sequence σ, we can define a history H_σ by letting $H_\sigma(p)$ be $value(\Pi_p(\sigma))$. Similarly, for any sequence $t \in (P \times V)^{\infty}$, we can define a history H_t by letting $H_t(p)$ be $value(\Pi_p(t))$. We denote the restriction of H_σ to the input ports by H_σ^{in}, and call it the *input port history* corresponding to σ. We denote the restriction of H_σ to the output ports by H_σ^{out}, and call it the *output port history* corresponding to σ.

We now describe the computation sequences that we consider as "completed." To do this, we extend the prefix ordering on computation sequences to include the concurrency information in the concurrency relation. A finite computation sequence γ is a *prefix* of a computation sequence δ, and we write $\gamma \preceq \delta$, iff there exists a computation sequence ξ with $\gamma\xi = \delta$. We define *permutation equivalence* to be the least congruence $\sim$, respecting concatenation, on the set of finite computation sequences of an automaton such that whenever $a \| b$, the computation sequences $q \xrightarrow{a} r \xrightarrow{b} p$ and $q \xrightarrow{b} s \xrightarrow{a} p$ are $\sim$-related. We define the *permutation preorder* relation $\precsim$ on finite computation sequences of A as the transitive closure of $\preceq \cup \sim$. Define $\simeq = \precsim \cap \succsim$. It is an easy lemma that for γ, δ finite, $\gamma \precsim \delta$ iff $\exists \xi$ such that $\gamma\xi \simeq \delta$. One observation we can make is that if $\gamma \precsim \delta$, then the multiset of events in γ is a subset of the multiset of events in δ. Another lemma is that for γ, δ finite, if $\gamma \underset{\sim}{\sqsubset} \delta$ and the multiset of events in γ is contained in the multiset of events in a prefix δ' of δ, then $\gamma \underset{\sim}{\sqsubset} \delta'$. We can now extend the permutation preorder relation to infinite computation sequences by defining $\gamma \underset{\sim}{\sqsubset} \delta$ iff for every finite $\gamma' \preceq \gamma$, there exists a finite $\delta' \preceq \delta$, such that $\gamma' \underset{\sim}{\sqsubset} \delta'$. We define $\simeq = \underset{\sim}{\sqsubset} \cap \underset{\sim}{\sqsupset}$ for infinite computation sequences also.

We would like a notion of "completed" computation sequence, in which all events that could happen at any state have either happened or been disabled.

Definition 5. A computation sequence γ is called **completed** if it is either finite and no non-input event is enabled at its end, or it is infinite and there is no event e enabled at every state in $\gamma[i..]$ and concurrent with every event in $\gamma[i..]$ for some i.

This turns out to be identical to $\underset{\sim}{\sqsubset}$-maximality for a particular input, as the lemma below will show. Whenever we talk about $\underset{\sim}{\sqsubset}$-maximality, we will actually mean maximality for a particular input.

For the port automata considered in [PS88b], no two non-input events were concurrent. In this paper, the notion of a *fair* initial computation sequence was introduced. For single automata, this notion is equivalent to our notion of completion. We will consider networks of automata presently, and for networks of monotone automata as in [PS88b], the fair computation sequences coincide with the completed computation sequences, which coincide with the $\underset{\sim}{\sqsubset}$-maximal computation sequences. Moreover, the projections of maximal computation sequences onto individual automata in the network are themselves maximal. When we consider networks of general port automata as in this thesis, this is no longer the case. So, as we will discuss in Section 2.5, we will give a different notion of completion for networks of automata. For networks of monotone automata however [Sta90], the projections of a maximal computation sequence onto individual automata are again maximal. This means that even though our above notion of completion is equivalent to maximality for general automata, this notion can be extended to networks of monotone automata but not to networks of general automata. This is unfortunate, but unavoidable.

Lemma 1. A computation sequence γ is $\underset{\sim}{\sqsubset}$-maximal among all the computation sequences having the same input port history as γ iff it is completed.

Proof : ($\Rightarrow$) Suppose γ is finite and there is a non-input event enabled at the end of γ. Let the new computation sequence be called γ'. Then $\gamma \preceq \gamma'$, and so $\gamma \underset{\sim}{\sqsubset} \gamma'$. But since the length of γ' is bigger than that of γ, $\gamma' \not\underset{\sim}{\sqsubset} \gamma$. Therefore γ is not maximal among the computation sequences having the same input as γ.

Suppose γ is infinite and there is an event e enabled at every state in $\gamma[i..]$ and concurrent with every event in $\gamma[i..]$ for some i. Then e does not occur in $\gamma[i..]$ as the concurrency relation is irreflexive. Let $\gamma = q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \ldots q_i \xrightarrow{a_i} \ldots$. Then $q_i \xrightarrow{e} q'_i$ for some state q'_i. By commutativity, there are states $q'_{i+1}, q'_{i+2}, \ldots$ such that $q_{i+m} \xrightarrow{e} q'_{i+m}$ for all $m > 0$, and $q'_{i+m} \xrightarrow{a_{i+m}} q'_{i+m+1}$ for all $m \geq 0$ (see figure).

$$\begin{tikzcd}
q_1 \arrow[r, "a_1"] & q_2 \arrow[r, "a_2"] & \circ\ \circ\ \circ\ q_i \arrow[r, "a_i"] \arrow[dr, "e"] & q_{i+1} \arrow[r, "a_{i+1}"] \arrow[dr, "e"] & {} \\
& & & q'_i \arrow[r, "a_i"] & q'_{i+1} \arrow[r, "a_{i+1}"] & {}
\end{tikzcd}$$

Let γ' be the computation sequence $q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \ldots q_i \xrightarrow{e} q'_i \xrightarrow{a_i} \ldots$. We claim that $\gamma \underset{\sim}{\sqsubset} \gamma'$ and $\gamma' \not\underset{\sim}{\sqsubset} \gamma$. The multiset of events in $q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \ldots q_i \xrightarrow{e} q'_i$ is not a subset of the multiset of events in any prefix of γ because e does not occur in $\gamma[i..]$. Therefore $\gamma' \not\underset{\sim}{\sqsubset} \gamma$. Also, for any prefix $\gamma[1..i{+}m]$ of γ, the prefix $q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \ldots q_i \xrightarrow{e} \ldots q_{i+m+1}$ of γ' is $\underset{\sim}{\sqsupset} \gamma[1..i{+}m]$. Therefore $\gamma \underset{\sim}{\sqsubset} \gamma'$, and so γ is not maximal among all computation sequences with the same input as γ.

($\Leftarrow$) Suppose γ is not maximal. If γ is finite and $\gamma \precsim \gamma'$ but $\gamma' \not\precsim \gamma$, then $\gamma\xi \simeq \gamma'$ for some nonempty ξ. Since γ' contains no more input events than γ, ξ consists only of non-input events, and so there is a non-input event enabled at the end of γ. Therefore γ is not completed.

If γ is infinite and $\gamma \precsim \gamma'$ but $\gamma' \not\precsim \gamma$, then there is a smallest finite prefix $\gamma'[1..i]$ of γ' such that $\gamma'[1..i] \not\precsim \gamma$. Then we claim that the event e in $\gamma'[i]$ is enabled at the end of $\gamma[1..i']$ for some i' and at every state in $\gamma[(i'+1)\ldots]$ and is concurrent with every event in $\gamma[(i'+1)\ldots]$. Since $\gamma'[1..(i-1)] \precsim \gamma[1..i']$ for some i', $\gamma[1..i'] \simeq \gamma'[1..(i-1)]\xi$ for some ξ. Since $\gamma \precsim \gamma'$, for every $m \geq (i'+1)$, there is some $i'' > i$, such that $\gamma'[1..(i-1)]\xi\gamma[(i'+1)..m] \simeq \gamma[1..m] \precsim \gamma'[1..i'']$. Therefore e is enabled at the end of $\gamma[1..i']$ and at every state thereafter, and is concurrent with every event in $\gamma[(i'+1)\ldots]$. ∎

We could think of the preorder $\precsim$ as the prefix ordering in which concurrency information has been encoded. It is quite pleasant to be able to state completedness as a maximality property of computation sequences.

Example 3. Consider an automaton with one input port i and one output port o. The event (o, m) is enabled if and only if m values have arrived at the input port. Output events do not commute with input events. Consider the infinite computation sequence $\gamma = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \ldots$ consisting only of input events. q_i is the state after i values have arrived at the input port, and q_0 is the initial state. Then the event (o, m) is enabled at q_m. So there is a non-input event enabled at every state, but it is a different non-input event for every state. There is no sensible way of "completing" γ, because there is a different non-input event enabled at different states of γ, and they do not commute with the input events. So we would accept γ as a completed computation sequence.

We now give a similar example in which the same event is continuously enabled.

Example 4. Consider an automaton with one output port o and only two internal events e_1 and e_2. There are only three states q_1, q_2, q_3. q_1 is the initial state. The transitions are completely described by $A(q_1, q_1) = \{\epsilon, e_1\}$, $A(q_1, q_2) = \{e_2\}$, $A(q_2, q_3) = \{(o, v)\}$ where v is some fixed value, $A(q_2, q_2) = \{\epsilon\}$ and $A(q_3, q_3) = \{\epsilon\}$. There is an infinite sequence γ of e_1-transitions from the initial state, and the event e_2 is continuously enabled, following which an output would be enabled. But the event e_2 is not concurrent with e_1, and the automaton is really making an indeterminate choice at each step without any fairness constraints. So we would accept γ as a completed computation sequence.

This example makes clear the distinction between completion and continuous enabling.

Now we can describe the *input-output relation* of an automaton. This describes the input-output behaviour – says which outputs are possible for which inputs. This is the most abstract that we can get because function semantics cannot be used for indeterminate networks.

Definition 6. The **input-output relation** of an automaton is the set of all pairs $(H_\sigma^{in}, H_\sigma^{out})$ with σ being a completed computation sequence of the automaton.

We can also equivalently consider the input-output relation to be a set of pairs of tuples of streams, the first tuple of each pair consisting of streams at the input ports and the second tuple consisting of streams at the output ports. We will also refer to the input-output relation as the *IO-relation.*

2.3 Moves of Computation Sequences

We now formalize some of the implications of commutativity and permutation equivalence for computation sequences.

Definition 7. A **move** of a computation sequence γ is a pair $(i, i+1)$ such that $\gamma[i] = q \xrightarrow{a} r$ and $\gamma[i+1] = r \xrightarrow{b} p$ and there exists a state s such that $q \xrightarrow{b} s$ and $s \xrightarrow{a} p$.

Recall that we defined $\precsim$ to be the transitive closure of the union of the prefix preorder $\prec$ with the permutation equivalence relation $\sim$, and we defined $\simeq = \precsim \cap \succsim$. It then follows from the definition of $\simeq$ that if γ, δ are finite computation sequences and $\gamma \simeq \delta$, then there is a finite sequence of moves that can transform γ to δ.

Definition 8. A **move transformation** of a computation sequence γ is any sequence of moves that involves any particular event occurrence in γ only finitely often.

The proviso, about moving any particular event occurrence only finitely often, is present because we do not want to consider sequences of moves for which event occurrences may get "lost." For example, consider a computation sequence γ consisting of an output transition followed by infinitely many input transitions. Every one of the infinitely many input transitions can be flipped with the output transition. The result of this infinite sequence of moves would not contain the output transition at all.

Lemma 2. If γ is a computation sequence, η is a move transformation of γ, and δ is the result of the move transformation on γ, then $\delta \simeq \gamma$.

Proof : If η is finite, then it involves only a finite prefix π of γ, and if the result of applying η to π is π', then $\pi \simeq \pi'$, and therefore, $\gamma = \pi\xi \simeq \pi'\xi = \delta$ for some ξ.

If η is infinite, then let π be a finite prefix of δ. Let η' be the smallest finite prefix of η such that the rest of the moves in η do not involve the event occurrences in π. Let every event occurrence in π and every event occurrence involved by η' be in the prefix ξ of γ. If π' is the result of applying η' to ξ, then $\xi \simeq \pi'$, and this must extend π. Therefore $\pi \preceq \pi' \simeq \xi$, thus proving $\delta \sqsubseteq \gamma$.

To prove that $\gamma \sqsubseteq \delta$, let π be a finite prefix of γ. Let ξ be the smallest prefix of δ containing all the event occurrences in π. There must be a smallest prefix η' of η such that the rest of the moves in η do not involve events in ξ. We claim that $\pi \sqsubseteq \xi$. Let π' be the smallest prefix of γ, extending π, and ξ' be the smallest prefix of δ, extending ξ, involved by η'. Then $\pi' \simeq \xi'$. Then $\pi \preceq \pi' \simeq \xi'$. Therefore $\pi \sqsubseteq \xi'$. Since $\xi \preceq \xi'$ and every event occurrence of π is in ξ, $\pi \sqsubseteq \xi$, thus proving that $\gamma \sqsubseteq \delta$. ∎

Corollary 1. If γ is a maximal computation sequence, and η is a move transformation, then the result of applying η to γ is also maximal.

Lemma 3. If γ is a computation sequence of an automaton, then it can be $\sqsubseteq$-extended to a completed computation sequence with the same input as in γ.

The proof uses Zorn's lemma, and is similar to the one in [PS88b]. Briefly, we can show that every chain of computation sequences $\gamma_1 \sqsubseteq \gamma_2 \sqsubseteq \ldots$, such that $\gamma \sqsubseteq \gamma_i$ for every i, has a lub. Hence, by Zorn's lemma, the set of all computation sequences δ, such that $\gamma \sqsubseteq \delta$, has a lub and this is maximal.

Corollary 2. If γ is a finite computation sequence of an automaton A, then it can be extended to a completed computation sequence with the same input as in γ.

Proof : Let δ be a completed computation sequence such that $\gamma \precsim \delta$. Since γ is finite, there is a finite prefix δ' of δ, such that $\gamma \precsim \delta'$. Therefore $\gamma\xi \simeq \delta'$ for some ξ. Hence there is a sequence of moves that transforms δ' to $\gamma\xi$, and therefore transforms δ to a completed computation sequence extending γ. ∎

2.4 Determinate Automata

Earlier, we defined the input-output relation for an automaton. If this relation turns out to be the graph of a function – i.e. for any input, there is a unique output associated with it in the input-output relation – then the automaton is said to *compute* that function. We now describe a subclass of the class of automata that compute functions.

The following definition is from [Sta87].

Definition 9. An automaton is **determinate** if it satisfies the following condition : $b||b'$ whenever b, b' are distinct non-input events both enabled at some state.

Intuitively, a determinate automaton does not exhibit "internal indeterminacy" – the only possible indeterminate choices that the automaton makes occur between input event transitions. The following theorem and lemma were proved in [Sta87].

Theorem 1. Determinate automata compute functions. Moreover, a function f is computed by a determinate automaton iff f is a continuous function.

Lemma 4. Suppose A is a determinate automaton. Then for each input x, there is a unique completed computation sequence, upto $\simeq$-equivalence, having input x. Moreover, if input x' extends x, then for any completed computation sequence γ with input x and any completed computation sequence γ' with input x', $\gamma \precsim \gamma'$.

We now state two lemmas describing how the relative ordering of events may be preserved between $\precsim$-related computation sequences of determinate automata.

Lemma 5. If γ, δ are computation sequences with $\gamma \precsim \delta$, then there is a move transformation η of δ, such that it transforms δ to a computation sequence δ' such that the event occurrences in γ occur in the same order in δ' as in γ.

Proof : If γ is finite, then $\gamma \precsim \delta'$ for some finite prefix δ' of δ. So $\gamma\xi \simeq \delta'$ for some ξ, and so there is a finite sequence of moves that transforms δ' to $\gamma\xi$, and hence transforms δ to an extension of γ.

We now consider the case when γ is infinite. We inductively define finite sequences of moves η_i, such that the result of applying $\eta_1{}^\wedge \ldots {}^\wedge\eta_i$ to δ is a computation sequence δ_i with a prefix $\delta_i[1..l_i]$ containing the events in $\gamma[1..i]$ in the same order in which they appear in $\gamma[1..i]$ and possibly other events not in γ. We will choose l_i so that $\delta_i[l_i + 1]$ is an event occurrence in γ. Suppose we have already defined $\eta_1{}^\wedge \ldots {}^\wedge\eta_i$. Let e be the event occurrence in $\gamma[i + 1]$. Let this be the event occurrence in $\delta_i[k], k > l_i$. Since $\gamma \precsim \delta \simeq \delta_i$, and $\delta_i[1..k]$ contains all the event occurrences in $\gamma[1..i+1]$, $\gamma[1..i+1] \precsim \delta_i[1..k]$. Then by the determinacy of the automaton, e is enabled at every state in $\delta_i[(l_i + 1)..(k - 1)]$, and e commutes with all the events in $\delta_i[(l_i + 1)..(k - 1)]$. Therefore we define η_{i+1} to be $(k-1, k), \ldots (l_i+1, l_i+2)$. The required move transformation is then $\eta_1{}^\wedge\eta_2{}^\wedge \ldots$. That this is indeed a move transformation follows from construction, because the events of $ev(\gamma)$ are clearly moved finitely often, and for any other event $\delta[m]$ of δ, it does not get moved any more once all the events of $ev(\gamma)$ in $\delta[1..m]$ have stopped moving. ∎

Corollary 3. If γ and δ are computation sequences with $\gamma \simeq \delta$, then there is a move transformation that transforms γ to δ.

Lemma 6. If γ, δ are computation sequences with $\gamma \precsim \delta$, then there is a move transformation η of γ, such that it transforms γ to a computation sequence γ' such that the event occurrences in γ' occur in the same order in δ as in γ'.

Proof : Similar to Lemma 5. ∎

2.5 Networks of Automata

We now describe how we can build networks of automata by collecting together individual automata and then linking ports together. We first describe the construction of finite networks, consisting of finitely many automata. We allow the use of three operations – aggregation, feedback and output hiding. These operations will capture the notions of network composition as suggested by Kahn in [Kah74].

Aggregation involves taking two networks with disjoint sets of ports and "keeping them side by side" to obtain a new network. Feedback (see Figure 2.1) involves identifying an output port p_1 of the network with an input port p_2 of that network, so that the port p_2 can no longer be observed by the environment and can no longer serve as an input port. The values appearing at p_2 will be those that are output at p_1. One stipulation that we make is that the output port p_1 and the input port p_2 not be ports of the same automaton in the network. Our last operation, output hiding, removes an output port, so that the environment can no longer observe values at this hidden port.

We can define the first two operations as special cases of the *composition* of a finite *compatible* set of automata.

Definition 10. If I is a finite index set, then a set $\mathcal{S} = \{M_i : i \in I\}$ of automata is said to be **compatible** if

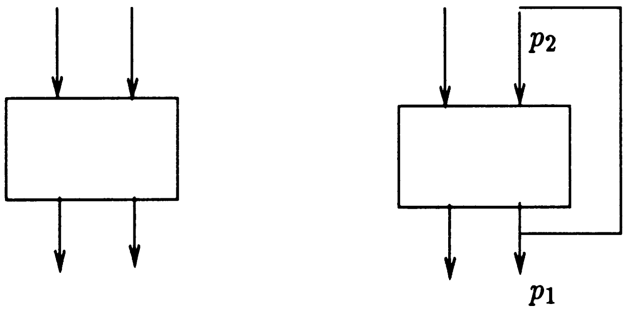

Figure 2.1: Feedback

- for all $i, j \in I$ such that $i \neq j$ we have $(E_i \backslash (\mathrm{Inp}_i \cup \mathrm{Out}_i)) \cap (E_j \backslash (\mathrm{Inp}_j \cup \mathrm{Out}_j)) = \emptyset$, that is, the sets of internal events of any pair of automata are disjoint, and,
- for any port name, at most two automata may have that name in common, and in that case, it must be the name of an output port of one automaton and an input port of the other automaton,

where $M_i = (E_i, Q_i, A_i)$, and Inp_i is the set of input events of M_i, and Out_i is the set of output events of M_i.

The shared port names represent ports that will get connected when the set of automata are composed together. We will then obtain a *network* automaton. The input ports of the network will be all the input ports of the M_i's, excluding those that are shared. The output ports of the network will be all the output ports of the M_i's.

Definition 11. The **composition** of a compatible set $\mathcal{S}$ of automata is the automaton $\prod M_i = (E, Q, A)$, where

- $E = \cup E_i$, with $a \| b$ iff $a \|_i b$ for all $i \in I$ such that both a and b are in E_i.

- $Out = (\cup Out_i)$, and $Inp = (\cup Inp_i) \backslash (\cup Out_i)$,
- $Q = \prod_{i \in I} Q_i$
- $q^\iota = (q_i^\iota : i \in I)$
- $e \in A((q_i : i \in I), (r_i : i \in I))$ iff for all $i \in I$, either $e \notin E_i$ and $r_i = q_i$, or else $e \in E_i$ and $e \in A_i(q_i, r_i)$.

The definition of $\|$ above implies that events of distinct automata, that do not share any ports, are concurrent, because they are not both in the event set of any single automaton. We now note that the above definition includes the definition of *aggregation*, which is what happens when two automata do not share any ports, and the definition of *feedback*, which is what happens when two automata share a port name and it is the output port of one automaton and an input port of the other automaton. We now explicitly define *output hiding.*

Definition 12. If $\mathcal{A}$ is an automaton with input ports P^{in} and output ports P^{out}, and S is a subset of P^{out}, then the **output hiding** of S in $\mathcal{A}$ results in the automaton with input ports P^{in} and output ports $P^{out} \backslash S$ with exactly the same sets of events and states and the same transition relation as $\mathcal{A}$.

When we compose two automata with a shared port name p, p being an output port of one automaton and an input port of the other automaton, then the two automata connected in this manner may execute a single event in the composed automaton, but this might correspond to an output event of one of them and an input event of the other. For example, suppose A and B are the two automata sharing port name p, that is p is an output port of A, but an input port of B. Then (p, v) is an output event for A, but an input event for B. For the composed automaton, this corresponds to the emission of value v by A at its output port p **and** the arrival of v at the input port p of B. By defining composition in this way, we do not have to worry about liveness conditions, to ensure that values output by A at p will eventually arrive at the input port p of B.

The difference between a network of automata and a single automaton is that we can recover the structure of the individual automata in the network by appropriate projections. A network can be thought of as an automaton, coming with a predefined decomposition. One may, of course, specify a large automaton without giving such a decomposition.

With each component automaton M_i, we associate restriction functions ρ_i from states of the network to states of M_i, and α_i from events of the network to events of M_i. ρ_i is defined by $\rho_i((q_i : i \in I)) = q_i$, and α_i is defined by $\alpha_i(a) = a$, if $a \in E_i$, and $\alpha_i(a) = \epsilon$ otherwise. Then we can define the **restriction** $\Pi_{M_i}(\gamma)$ of a computation sequence $\gamma = q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \ldots$ of the network to a component automaton M_i by $\rho_i(q_1) \xrightarrow{\alpha_i(a_1)} \rho_i(q_2) \xrightarrow{\alpha_i(a_2)} \ldots$ with the identity transitions collapsed. We can similarly define $\Pi_{N'}(\gamma)$, the restriction of a computation sequence γ to a subnetwork N'.

We now come to an important point about the construction of networks from individual automata. When feedback is utilized as a construction mechanism, an output port and an input port get connected, the input port gets hidden and is no longer observable by the external environment, but the output port is not hidden. We emphasize here that the shared output ports must get hidden in the "real" networks that are built, and so an output port that has been connected to an input port cannot really serve as an output port of the final network. The reason that we have chosen to describe composition, and in particular, feedback, in the above manner is that we will be representing the "meanings" of networks of automata in terms of the events at their input and output ports. This means that, if we define feedback so as to hide both the connected input and output ports, we cannot represent the events on the "internal channels" arising from feedback in the meanings of networks, and it turns out that when we prove inexpressibility results later on by reasoning about the network computations, we really need to represent these events on the shared ports. This forced us to define composition as we have done. Since we have the output hiding operation, we can also define the "real" networks to be those in which all the shared output ports created by feedback must get hidden using output hiding. We can view the shared output ports created by feedback to really be "pseudo-output" ports, and the networks thus created really being "pseudo-networks." Henceforth, we will drop the prefix "pseudo," and refer to all the networks and pseudo-networks as simply networks, bearing in mind that some output ports may implicitly be pseudo-output ports, "real" networks do not have pseudo-output ports, and pseudo-networks may only arise as an intermediate construction in proofs.

We can define *history, input port history* and *output port history* corresponding to computation sequences of networks, just as we did for computation sequences of single automata. We recall that, for a single automaton, the completed computation sequences were the $\precsim$-maximal ones. We could now define *permutation equivalence* and the *permutation preorder* $\precsim$ as we did for single automata, and then talk about the $\precsim$-maximal computation sequences. But we argue that this maximality condition is not a reasonable definition of completion for networks in general. We illustrate this point of view by an example.

Example 5. Consider the network in Figure 2.2, consisting of a buffer automaton, as in example 1, and a poll automaton, as in example 2. Suppose we took maximality of computation sequences as defining completion. Then the computation sequence starting with an input event at the input port of the buffer, followed by an infinite sequence of $\star$ outputs by the poll, is a maximal computa-

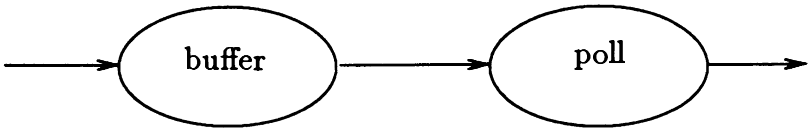

Figure 2.2: A Buffer and a Poll

tion sequence, because even though the output event for buffer (input event for poll) is enabled at every state following the read event of the buffer, it does not commute with a $\star$ output. But this computation sequence does not really make sense as a completed sequence, because we want the buffer to output all its input values. In other words, the projection of the network computation sequence onto the ports of the buffer is not a completed computation sequence for the buffer.

The following definition, however, captures the proper notion of completion.

Definition 13. A computation sequence γ for a network of automata is **completed** if, for every automaton M in the network, the restriction of γ to M is a completed computation sequence for M.

Intuitively, this definition makes sense, because we would like to call those computation sequences of the network "completed," in which every component automaton gets chances to execute and exhibits a completed computation sequence in every computation of the network.

Just as we defined the input-output relation for a single automaton earlier, we can now define the following :

Definition 14. The **input-output relation** of a network of automata is the set of all pairs $(H_\sigma^{in}, H_\sigma^{out})$ with σ being a completed computation sequence of the network, H_σ^{in} being the input port history corresponding to σ, and H_σ^{out} being the output port history corresponding to σ.

In this chapter, we have presented a view of processes and networks as automata, and a view of computations of processes as sequences of events of these automata. In the next chapter, we will discuss how we can abstract the internal events away from this description, and the modification to the view of processes and networks that is necessary if our abstract representation is to satisfy intuitive and natural properties.

Chapter 3

Traces of Dataflow Networks

A computation sequence describes the sequence of states and the sequence of events, including internal events, that cause the state transitions. We now abstract away from states and internal events, and only consider sequences of events on the input and output ports of a network. It turns out that this has exactly the right amount of information to encode observable equality in all network contexts. At first glance, it appears as if there is a significant degree of temporal information encoded in sequences of events. It turns out, however, that the set of these sequences is closed under certain event permutations that essentially "wash away" the ordering information.

3.1 Observability and Buffering

It is clear that, for an "external" observer of an automaton, all that can be observed is a sequence of input events at each input port of the automaton, and a sequence of output events at each output port of the automaton. Due to the asynchronous nature of communication between automata, there may be an arbitrary delay between the emission of a value at an output port and the observation of that value. So it is not possible for an observer to determine the order of emission of values on *different* ports,

To reason about processes, we would like to abstract away from computation sequences by considering only the input events and output events, as these are the events that interface the process with the external world. But the definition of automata do not allow any two events on different ports to be commuted. That means that if there is a computation sequence γ with event e on output port p occurring before an event e' on some other output port p', then there may be no computation sequence with e' preceding e and the rest of γ unchanged. But, as far as the external observer is concerned, he should not be able to distinguish the automaton in which e always occurs before e' from the automaton in which e may precede e', or e' may precede e. So, as far as the external observer is concerned, these two automata should have the same set of "traces" – abstractions from computation sequences.

For this reason, we will describe **processes** as automata with buffers attached to each of the input and output ports. As we shall see, this will allow us to commute events on different ports. Adding buffers to the description of a process will also make the arbitrary delay between the emission of a value and its observation explicit in computation sequences of the network.

3.2 Processes

We will describe processes as automata with buffers attached to each of the input and output ports of the automaton. A buffer was described as a specific automaton in example 1. We now formally describe what we mean by a "buffered automaton."

Definition 15. A **process** is the composition of

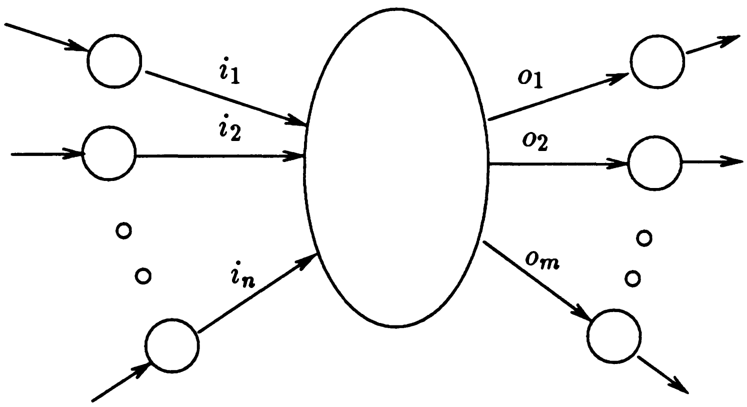

Figure 3.1: The Structure of a Process

(i) an automaton, called the **central automaton** of the process, with input ports $i_1, i_2, \ldots i_n$ and output ports $o_1, o_2, \ldots o_m$, and

(ii) $m + n$ buffers, called the **process buffers**, n of them having output ports $i_1, i_2, \ldots i_n$ respectively, and the remaining m of them having input ports $o_1, o_2, \ldots o_m$ respectively. The rest of the ports of the buffers are disjoint from the set $\{i_1, i_2, \ldots i_n, o_1, o_2, \ldots o_m\}$.

with the ports $\{i_1, i_2, \ldots i_n, o_1, o_2, \ldots o_m\}$ hidden.

Figure 3.1 shows a process; we may also refer to this as a buffered automaton.

By our definition of completed computation sequences, we ensure that every value input to the process will arrive at an input port of the central automaton, and every value output by the central automaton will be output by the process.

Just as we defined a compatible set of automata in the previous section, we can similarly define a compatible set of processes – the set consisting of the automata that make up these processes must be compatible. The *composition* of a compatible set of processes is the network obtained by composing the set of network automata representing the individual processes. A network of processes may also be obtained by hiding some of the output ports of the composition of a compatible set of processes.

3.3 Trace sets

We can describe the behaviour of a network by describing its input-output behaviour – say which outputs are possible for which inputs. This is the most abstract that we can get because Kahn's function semantics cannot be used for indeterminate networks. We will define this input-output behaviour as the *input-output relation*. We will take the view that two networks are to be viewed "observably equal" if they have the same input-output relation. The input-output relation of a network of processes will simply be the input-output relation of the associated network of automata.

Since "buffering" an automaton does not affect the input-output relation, we have a process with the input-output relation of *poll* (example 2). But note that we do not consider "arbiters," which can distinguish between the order of arrival of values at distinct input ports. This is because, for the processes we consider, we cannot distinguish between the order of arrival of values at distinct input ports, because the central automaton of the process may see the values in either order, as the buffers at the input ports of the process may produce the corresponding output events in either order.

The input-output relation semantics for processes and networks fails to be compositional. The reason for this is that the input-output relation simply describes which sequences of values can be obtained at the output ports for given sequences of values at the input ports. So this view describes the *entire* output when given the *entire* input. It does not describe which particular values in the output sequences *really depended* on the presence of which particular values in the input sequences. Briefly, there is no causality information between events encoded in the input-output relation. One way to express causality information is to consider sequences of input and output events.

Definition 16. If γ is a computation sequence, then we define $tr(\gamma)$ to be the subsequence of $ev(\gamma)$, consisting of all the input and output events in $ev(\gamma)$.

Definition 17. A **trace** of a network N of processes is a sequence t of input events and output events of N, such that $t = tr(\gamma)$ for some completed computation sequence γ of N. We write $Trset(N)$ for the set of traces of a network N.

In an earlier discussion of network semantics, trace sets were called archives [KP86]. Presented in this way, the traces appear as an abstraction of computation sequences that were defined using an operational formalism. The important point is that we can define composition rules directly on trace sets, and this allows us to build up trace sets of complex networks structurally. If t is a sequence of events and P is a process or a network, $\Pi_P(t)$ will represent the subsequence of t consisting of all the events on the input and output ports of P in t.

In a network N of processes, an output port of a process A in the network may have the same name p as an input port of a process B, possibly the same as A, in the network. This port is then either an "internal port" of the network or a pseudo-output port of the network. It cannot be an input port of the network. We would like to prove a theorem that the trace set of a network may be described in terms of the trace sets of its "subnetworks." We would like to view a subnetwork to be the composition of a subset of the processes in N, and in addition, if such a subset of processes contains both the processes A and B as above, then we would like to be able to view the input port of B as an input port of the subnetwork.

So we will take the view that the output port p of A and the input port p of B may be renamed to distinct names p_{out} and p_{in} in the subnetwork.

We define the notion of corresponding events in a trace or a computation sequence.

Definition 18. In a trace or computation sequence of a network with two of its ports being p_1 and p_2, a p_1-event occurrence e_1 and a p_2-event occurrence e_2 are said to be *corresponding events* if e_1 is the ith event on port p_1, and e_2 is the ith event on port p_2 for some $i > 0$.

We use the predicate "precedes" to indicate temporal precedence between events in a trace.

Definition 19. $\mathit{precedes}(p_1, p_2, t) =_{def}$ for every finite prefix of t, $\mathit{value}(\Pi_{p_2}(t))$ is a prefix of $\mathit{value}(\Pi_{p_1}(t))$.

Informally, this means that every p_2-event occurrence is preceded by a corresponding p_1-event occurrence with the same value. This predicate is useful for defining the trace sets of networks built using feedback.

Theorem 2. Consider any decomposition of a network N into subnetworks $\{N_i\}$ for i in some finite index set I, where the N_i are not necessarily individual processes. Then the set of traces $Trset(N)$ of the network is exactly the set of sequences $\Pi_N(t)$, such that

(i) t is a sequence of events at the input and output ports of the N_i's, distinguishing between an output port and an input port even if they have the same name p by calling them p_{out} and p_{in} respectively,

(ii) for all $i \in I$, $\Pi_{N_i}(t)$ is a trace of N_i, and

(iii) for every port p that is an output port of N_i, as well as an input port of N_j, i not necessarily distinct from j, $\mathit{precedes}(p_{out}, p_{in}, t)$ and $\Pi_{p_{out}}(t) = \Pi_{p_{in}}(t)$.

Proof : Let t be a sequence of events, satisfying conditions (i), (ii) and (iii) above. We show that $\Pi_N(t)$ is a trace of N. Let $t_i = \Pi_{N_i}(t)$. For each t_i, there is a completed computation sequence γ_i of N_i, such that $t_i = tr(\gamma_i)$. Then we can dovetail among the computation sequences $\{\gamma_i : i \in I\}$ to obtain a computation sequence γ, such that t is the subsequence of $ev(\gamma)$ consisting of all the input and output events of the N_i. Now, using condition (iii) in the statement of the theorem, and by the property of non-disabling inputs of processes, we obtain a new computation sequence γ'' in which for every shared port p, the output events at p_{out} immediately precede the corresponding input events at p_{in}. Replacing each such adjacent pair of events at p_{out} and p_{in} by a single p-event, we obtain a computation sequence γ' of the network N in which the output events on the shared port are also the input events on the shared port. Any automaton M in the network is in one of the subnetworks, say N_i, and so the restriction of γ' to M is the same as the restriction of γ_i to M. Since the latter is completed, because γ_i is a completed computation sequence of N_i, so is the restriction of γ' to M. Therefore γ' is a completed computation sequence of the whole network, and so $tr(\gamma')$ is a trace and it is exactly the restriction of t to the events on the input and output ports of the network.

Let t' be a trace of N corresponding to the completed computation sequence γ of the network, composed of subnetworks N_i. The restriction γ_i of γ onto any subnetwork N_i is then a completed computation sequence of N_i by the definition of a completed computation sequence of a network. Let t'' be the subsequence of $ev(\gamma)$ consisting of all the input and output events of the N_i in $ev(\gamma)$. Now, for every port that is shared by an N_i and an N_j, i not necessarily distinct from j, replace an event on that port in t'' by two events – an output event on the output port followed by an input event on the input port. The new event sequence t is a sequence of events such that its restriction to the events of any subnetwork N_i

is a trace of that subnetwork, and moreover, the restriction of t to events on the input and output ports of the network is the same as the trace $t' = tr(\gamma)$. ∎

It is possible to have processes with different sets of traces, but the same *IO*-relation. Brock and Ackerman [BA81] have such an example, but their example uses a powerful primitive, *fair merge*. There are other examples using only finite indeterminacy [Rus90].

We will use vector notation to talk about collections of ports. We project traces onto vectors of ports and assume that we get a vector of event sequences. Thus, we may write, "a trace t when projected onto ports $\vec{a}$ produces the sequences $\vec{\alpha}$." This notation allows us to write many projections at once and also allows us to name the components of the vectors as the need arises.

We described the construction of networks in the previous chapter. We will use the notation $N \| M$ to represent the network built by aggregation from the two subnetworks N and M. We will use the notation $loop(p_1, p_2, N)$ to represent the network built by identifying an output port p_1 of the network N with the input port p_2 of N. Lastly, the traces of a network built by hiding some output ports of another network are obtained by simply taking the traces of the original network and projecting out the events on the hidden output ports. We use the notation $N \backslash S$ for the network N with the ports S of N hidden.

Theorem 3.

$$\text{Trset}(N \| M) = \{t \in (P \times V)^{\infty} | \Pi_N(t) \in \text{Trset}(N) \wedge \Pi_M(t) \in \text{Trset}(M)$$

where P is the set of input and output ports of N and M.

Theorem 4. $Trset(loop(p_1, p_2, N))$ consists of every sequence $\Pi_{\sim p_2}(t)$ such that $t \in Trset(N)$ and $precedes(p_1, p_2, t)$ and $(\Pi_{p_1}(t) = \Pi_{p_2}(t))$.

The statement of the theorem expresses the idea that in the network with the feedback, the events on p_2 arise from events on p_1.

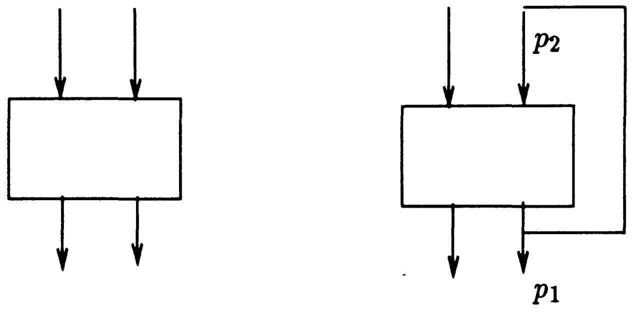

Figure 3.2: Loop formation

Theorem 5. Suppose S_O is a subset of the set of output ports of a network N. Then

$$Trset(N \backslash S_O) = \{\Pi_{\sim S_O}(t) | t \in Trset(N)\}$$

All these proofs follow directly from Theorem 2.

We will use these theorems in proofs by structural induction that need to consider all finite networks that can be built out of some set of primitive processes.

3.4 Closure Properties of Trace Sets

As we noted in an earlier section, we have a particular notion of observability in mind. It is not possible for an external observer to determine the order in which values are output at different output ports. Also, if it is possible to observe a sequence of input and output events with a particular output event immediately preceding a particular input event, then it must be possible to observe the same sequence of events except that the input event is now observed before the output event. This motivated us to define processes as "buffered automata." We will now prove that trace sets of processes and networks of processes are closed under our notion of observation.

Definition 20. A **move** of a trace t is a pair $(i, i+1)$ such that either

(i) both $t[i], t[i+1]$ are input events at different ports, or

(ii) both $t[i], t[i+1]$ are output events at different ports, or

(iii) $t[i]$ is an output event and $t[i+1]$ is an input event.

We say that the second event **moves left** over the first event, and both events **participate** in the move.

Definition 21. A **move transformation** of a trace is defined as a sequence of moves, such that it involves any particular event occurrence in the trace only finitely often.

The proviso ensures that no event of the original trace is "lost" in the result. For example, if a trace consists of an output event followed by infinitely many input events, then we do not consider the sequence of moves that moves all the input events before the output event. The result of applying such a sequence of moves would not contain the output event at all and need not be a trace.

The main theorem here shows that trace sets contain less temporal information than appears to be the case at first sight.

Theorem 6. The trace set of a network of processes is closed under move transformations.

To prove this theorem, we will first prove that single moves preserve tracehood.

Lemma 7. If t is a trace of a network N and $(i, i+1)$ is a move of t, then the result $t[1..(i-1)]^\wedge t[i+1]^\wedge t[i]^\wedge t[(i+2)...]$ of applying the move to t is a trace of N.

Proof : Let γ be a completed computation sequence of the network such that $tr(\gamma) = t$. Let the events $t[i], t[i+1]$ correspond to the transitions $\gamma[j], \gamma[k]$ respectively.

If $t[i]$ is an input event, then $t[i+1]$ must also be an input event by the definition of a move. Also they are input events of different buffer automata B and B' of the network, and so they commute. $t[i+1]$ must be enabled at the end of $\gamma[1..(j-1)]$ since input events are always enabled. All the events in $\gamma[(j+1)..(k-1)]$ are internal events of the network, and so they are either output events of B' or events of other automata, and hence $t[i+1]$ commutes with all of them. Therefore $(k-1,k), \ldots (j, j+1)$ is a legal sequence of moves of the computation sequence γ. Let the resulting computation sequence be called γ'. Then $\gamma'[j], \gamma'[j+1]$ correspond to the events $t[i+1], t[i]$ respectively. Let $\gamma'[l], l > (j+1)$ be the output event of B that outputs the value in $\gamma'[j+1]$. Let l' be the minimum of l and the index of the first non-internal transition in γ' after $\gamma'[j+1]$. The sequence of events $ev(\gamma'[j+2..l'])$ is enabled at the end of $\gamma'[1..j]$ and the event $t[i]$ (in $\gamma'[j+1]$) commutes with all of them. Therefore $(j+1, j+2), \ldots (l'-2, l'-1)$ is a legal sequence of moves of the computation sequence γ'.

If $t[i]$ is an output event, then it is the output event of some buffer automaton B in the network. The events in $\gamma[(j+1)..k]$ are all either input events of B or events of other automata in the network. In either case, $t[i]$ commutes with them. Moreover this sequence of events is enabled at the end of $\gamma[1..(j-1)]$. So $(j, j+1), \ldots (k-1, k)$ is a legal sequence of moves of the computation sequence γ.

By Lemma 2, the resulting computation sequence in either of the two cases above must be completed. ∎

Proof of Theorem 6: Let t be a trace of the network corresponding to the completed computation sequence γ. Let $m_1, m_2, \ldots$ be a move transformation for t. For each m_i, we define a finite sequence η_i of computation sequence moves. Let κ_0 be the original computation sequence γ, and η_0 be the empty sequence of moves. Let κ_i be the result of applying all the moves in $\eta = \eta_1, \eta_2, \ldots \eta_{i-1}$ in order to κ_0. For every move m_i, we define η_i to be the sequence of moves as in the previous lemma.

We now prove that the entire sequence $\eta_1{}^\wedge\eta_2{}^\wedge \ldots$ involves each event occurrence in γ only finitely often. The non-internal event occurrences of γ exactly correspond to the event occurrences of t. By the definition of a move transformation, these are involved in moves only finitely often. For any internal event occurrence e in γ, we will refer to the closest non-internal event occurrences, if they exist, to the left and right of e as the left and right trace-neighbours of e, respectively. By the construction of η, e gets moved only if both its trace-neighbours exist. Moreover, every time e is moved, it is as a result of performing the trace move involving its two trace-neighbours. If e is close to only finitely many different event occurrences of t during the operation of η, then the movement of e infinitely often would imply that at least of these event occurrences of t gets moved infinitely often, which is not possible. So e must get moved only finitely often.

If e is close to infinitely many different event occurrences of t during the operation of η, then we achieve a contradiction as follows. There are two cases :

(1) e never gets moved over its left trace-neighbour. Let L be the set of event occurrences of t to the left of e in γ. Then, by the construction of η, the left trace-neighbour of e is always a member of L. So there are infinitely many event occurrences of t to the right of e getting moved over some member

of L. This contradicts the definition of a transformation. (We note here that it is the requirement that the left trace-neighbour of e always be a member of L that prompted us to define η as we did. For e to have a left trace-neighbour not in L, an input event would have to move left over the current left trace-neighbour, then move right over it, but not move right over e. But by our construction of η, this moving right over the current left-trace neighbour would entail moving right over e.)

(2) e does get moved over its left trace-neighbour at some point in the transformation. Then the left trace-neighbour must have been an output event occurrence o. Thereafter, o always remains to the right of e, and so, if e has to be close to infinitely many event occurrences of t, then infinitely many event occurrences of t must move over o, thus moving o infinitely often, contradicting the definition of a move transformation.

Thus e must get moved only finitely often.

Let δ be the result of applying $\eta = \eta_1{}^\wedge\eta_2{}^\wedge\ldots$ to γ. The above shows that the projection η_A of η onto the projection $\Pi_A(\gamma)$ of γ onto any component automaton A in the network moves any particular event occurrence of $\Pi_A(\gamma)$ only finitely often. Therefore, by Lemma 2, $\Pi_A(\gamma) \simeq \Pi_A(\delta)$, the result of applying η_A to $\Pi_A(\gamma)$. Since $\Pi_A(\gamma)$ was maximal by the completedness of γ, $\Pi_A(\delta)$ is also maximal, and hence δ is a completed computation sequence of the network. Therefore, $tr(\delta)$, which is exactly the result of applying the move transformation to t, is a trace of the network. ∎

3.5 Determinate processes and their properties

In this section, we define the determinate processes arising from the determinate automata of the previous chapter. We consider a specialization of these processes to the "sequential" processes. Finally, we consider properties of the trace sets of sequential processes

Definition 22. A process is called **determinate** if the central automaton of the process is a determinate automaton.

3.5.1 Sequential processes

In [Kah74], Kahn gave a denotational semantics for dataflow networks by modeling processes by continuous functions between finite products of domains of streams. Subsequently, Kahn and Plotkin [Cur86] introduced a general class of domains, called *concrete domains*, that generalized the stream domains originally used by Kahn, and permitted a general definition of a *sequential* function. Since we are working with dataflow and domains of streams here, we will use the specialization of this definition to stream domains.

The main reason for introducing sequential processes is that when we describe our implementability and non-implementability results, they have to be relative to a base class of processes, i.e. when we build networks, we are allowed to use copies of processes in this base class. We will consider the class of sequential processes to constitute that base class of processes. Kahn's original intention [Kah74, Kel78] was to consider processes with "single threads of control" and no "internal parallelism," and this corresponds to the intuitive idea of normal sequential programs. Our notion of sequential process includes this, and hence strengthens our non-expressibility results.

Definition 23. A process P is called **sequential** if it is determinate, and the function f computed by it satisfies the following property : suppose $f(x) = y$, and o is an output port of P. If there exists an $x' \sqsupset x$ such that $\Pi_o(f(x'))$ extends

$\Pi_o(y)$, then there is an input port i of P such that whenever $x' \sqsupset x$ is such that $\Pi_o(f(x'))$ extends $\Pi_o(y)$, then also $\Pi_i(x')$ extends $\Pi_i(x)$.

Such a function is called a *sequential function.* In [PSS90], it was shown that there was a certain class of "sequential automata" that exactly computes all the sequential functions. We give an example of non-sequentiality.

Example 6. *Parallel OR* : This process has two input ports and one output port. It computes the following function POR : POR$(1^\wedge s, \Lambda)$ = POR$(\Lambda, 1^\wedge s')$ = POR$(1^\wedge s, 1^\wedge s')$ = 1, and POR$(0^\wedge s, 0^\wedge s')$ = 0 for any sequences of values s, s'. Then we have POR(1,Λ) = POR(Λ,1) = POR(1,1) = 1, POR(0,0) = 0, and the output is Λ otherwise. If we think of 0 and 1 representing false and true respectively, then the process outputs true if it gets a true on either of its input channels, and it outputs false if it gets a false on both of its input channels. Intuitively, the computation of this automaton must proceed in a "parallel" fashion, because it has to produce an output when either of the input ports receives a 1.

It is the sequential processes that constitute the base class of processes allowed in the building of networks. The notion of a determinate process is a very general one, and encompasses many different kinds of computations – sequential and parallel. The notion of sequentiality as we have defined it here generalizes the well-understood notion that has been used to write programs with single threads of control since the invention of early programming languages.

3.5.2 Causality between events

We first define a notion of causality for events in a trace of a determinate automaton.

Definition 24. If γ is a completed computation sequence of a determinate automaton A, then the **causal set** of γ is the set of all pairs (e_1, e_2) of event occurrences in γ such that

(i) e_1, e_2 are events at the same port, and e_1 precedes e_2 in γ, or

(ii) e_1 is an input event, e_2 is an output event, and e_1 precedes e_2 in every completed computation sequence with the same input as γ.

Definition 25. If t is a trace of a sequential process, then the **causal set** of t is the set of all pairs (e_1, e_2) of event occurrences in t such that

(i) e_1, e_2 are events at the same port, and e_1 precedes e_2 in t, or

(ii) e_1 is an input event, e_2 is an output event, and e_1 precedes e_2 in every trace with the same input as t.

In what follows, we will assume that P is a sequential process composed of a determinate automaton A with input ports $i_1, ..i_n$ and output ports $o_1, ..o_m$, and $m+n$ buffer automata, n of them having input ports i'_j and output ports i_j, $1 \leq j \leq n$, and m of them having input ports o_k and output ports o'_k, $1 \leq k \leq m$.

Lemma 8. If t is a trace of P for some input and some output, and γ is a completed computation sequence of A with the same input and output, then t and γ have the same causal sets upto the bijections between the input ports of P and input ports of A, and between the output ports of P and the output ports of A.

Proof : If (e_1, e_2) is not in the causal set of γ, and $e_1 = (i_j, v_1)$ and $e_2 = (o_k, v_2)$, and e'_1, e'_2 are the corresponding events $(i'_j, v_1), (o'_k, v_2)$ of P, then there is a completed computation sequence of P in which the corresponding event e'_2 precedes e'_1. Therefore there is a trace of P in which e'_2 precedes e'_1, and hence (e'_1, e'_2) is not in the causal set of t.

Suppose $e'_1 = (i'_j, v_1)$ and $e'_2 = (o'_k, v_2)$ are events in t, and (e'_1, e'_2) is not in the causal set of t. If (e_1, e_2) was in the causal set of γ, then every completed computation sequence of A with the same input as γ has e_1 preceding e_2. Therefore, for every completed computation sequence of P, e'_1 precedes e'_2, contradicting the assumption that (e'_1, e'_2) is not in the causal set of t. ∎

If we define $\precsim$ on computation sequences of a determinate process, it turns out that the completed computation sequences of the process are exactly the maximal ones. So we can define move transformations on completed computation sequences of determinate processes, and claim, by Theorem 2, that the result is completed, since move transformations preserve maximality. The proof of the notion that completed computation sequences of a determinate process are exactly the maximal ones is analogous to the proof in [PS88b]. The idea is that since all the automata in a process have the non-disabling inputs property, the projection of a computation sequence onto each automaton is maximal iff the entire computation sequence itself is maximal.

We now describe a property of sequential processes.

Lemma 9. If a sequential process P has an output history H^{out} for an input history H^{in} with finitely many input events, and if γ is a finite computation sequence of P containing all the input events in H^{in} and i output events at output port p, i being less than the length of $H^{out}(p)$, then there is a computation sequence extending γ in which the next non-internal event after the events in γ is $(p, H^{out}(p)[i+1])$.

Proof : If this event $e = (p, H^{out}(p)[i+1])$ is enabled at the end of γ, then we are done. If not, then we know, by Lemma 2, that there is an extension δ of γ with the same input history H^{in} and containing e. The only non-internal events not in γ and preceding e in δ must then be output events. Since output events of processes commute with all following events in a computation sequence, there is a sequence of moves that move these output events forward past e. ∎

Lemma 10. If t is a trace of P, and t' is any linearization of the events in t such that for every pair (e_1, e_2) in the causal set of t, e_1 precedes e_2 in t', then t' is a trace of P.

Proof : Let γ be a completed computation sequence of P such that $tr(\gamma) = t$. We define a move transformation on γ to obtain a completed computation sequence γ' by Lemma 2, such that $tr(\gamma') = t'$, hence proving our lemma. We will define finite sequences η_i of moves such that the result of applying $\eta_1{}^\wedge\eta_2{}^\wedge \ldots {}^\wedge\eta_i$ on γ is a computation sequence γ_i such that $tr(\gamma_i)$ has $t'[1..i]$ as prefix.

We define γ_0 to be γ. We will define η_i by induction. Suppose we have defined $\eta_1, \ldots \eta_{i-1}$, and suppose $tr(\gamma_{i-1}[1..r])$ is $t'[1..(i-1)]$, and $\gamma_{i-1}[r+1]$ is an input or output event. Suppose $t'[i]$ is an input event, and let it be $\gamma_{i-1}[r'], r' > r+1$. Then we define η_i to be $(r'-1, r'), \ldots (r+1, r+2)$.

Suppose $e = t'[i]$ is an output event. We would like to prove that the input in $\gamma_{i-1}[1..r]$ can cause the output of e. Suppose this is not the case. If there is a single input event e' in $\gamma_{i-1}[r+1..r']$, then (e', e) is not in the causal set of t. Therefore, there is some input I extending that in $\gamma_{i-1}[1..r]$, but not containing e', that can produce the output event e. By the definition of sequentiality, I must contain some event at the same port as e', and hence must be the event e' by the consistency of inputs. This is a contradiction. We can also achieve a contradiction when there are multiple input events in $\gamma_{i-1}[(r+1)..r']$ by using induction.

Therefore, by Lemma 9, there is a computation sequence $\kappa = \gamma_{i-1}[1..r]^\wedge\xi$, extending $\gamma_{i-1}[1..r]$, in which the next non-internal event is e, and ξ ends in this event. By Lemma 4, κ can be extended to a completed computation sequence with the same input as in γ, and therefore $\kappa \precsim \gamma_{i-1}[1..r']$. Therefore $\xi \precsim \gamma_{i-1}[(r+1)..r']$. Then $\xi\xi' \simeq \gamma_{i-1}[(r+1)..r']$, and there is a sequence of moves, which we take to be η_i, on $\gamma_{i-1}[(r+1)..r']$ that transforms it to $\xi\xi'$.

The fact that $\eta_1{}^\wedge \ldots$ is a move transformation follows from the fact that a non-internal event is clearly only moved finitely often, and an internal event gets moved only as long as all the non-internal events to its left in γ have not stopped moving, which they do in finite time. ∎

3.6 Adequacy and Full Abstraction of Trace Semantics

We now give a strong justification of our use of trace semantics by showing that it is both adequate and fully abstract. The present proof is inspired by the work of Joost Kok [Kok87] and is essentially the same proof as that reported by Bengt Jonsson [Jon89]. Bengt Jonsson had the first proof of full abstraction of trace semantics in his thesis [Jon87]. Jim Russell [Rus89]'s results on full abstraction, proved independently from the above authors, applies in more general situations.

We first describe when two networks of processes are *operationally equal.* To do this, we need the notion of a *network context*, and the notion of when it is legal to substitute a network into a network context. Informally, a network context is a network with a hole in it, with the hole being associated with two disjoint sets of ports I and O and it is legal to substitute a network into the hole if the network has I as its set of input ports, and O as its set of output ports.

Definition 26. Two networks N_1 and N_2 of processes are said to be **operationally equal**, if, for every network context $\mathcal{C}[\ ]$ of processes, the networks $\mathcal{C}[N_1]$ and $\mathcal{C}[N_2]$ have the same input-output relation.

Definition 27. A semantics for networks of processes is said to be **adequate**, if, whenever two networks are equal under the semantics, they are also operationally equal.

Definition 28. A semantics for networks of processes is said to be **fully abstract**, if the semantics is adequate, and moreover, whenever two networks are operationally equal, then they are also equal under the semantics.

In other words, a fully abstract semantics equates two networks of processes iff they are operationally equal, i.e. there is no network context of processes that can differentiate between them with respect to the input-output relation. We note here that the above definitions reflect our philosophy, that has pervaded throughout this thesis, that it is the input-output relation that is "observable" by an external observer, and if two networks are to be distinguished by the external observer, then they should have different input-output relations.

The following lemma states that if two networks have the same trace sets then their input-output relations are equal.

Lemma 11. If two networks N_1 and N_2 have the same trace sets, then they have the same input-output relation.

Proof : Suppose $(H_\sigma^{in}, H_\sigma^{out})$ is in the input-output relation for N_1 corresponding to a completed computation sequence σ of N_1. Then $tr(\sigma)$ is a trace of N_1, and hence a trace of N_2. So there is a completed computation sequence σ' of N_2, such that $t = tr(\sigma')$. Then since $H_\sigma^{in} = H_t^{in} = H_{\sigma'}^{in}$, and $H_\sigma^{out} = H_t^{out} = H_{\sigma'}^{out}$, $(H_\sigma^{in}, H_\sigma^{out})$ is also in the input-output relation for N_2. ∎

The next lemma states that trace equality is a congruence. We know from the Brock-Ackerman anomaly [BA81] that input-output relation equality is not a congruence.

Lemma 12. If two networks N_1 and N_2 have the same trace sets, then for every network context $\mathcal{C}[\,]$, the networks $\mathcal{C}[N_1]$ and $\mathcal{C}[N_2]$ have the same trace sets.

Proof : Consider $\mathcal{C}[N_1]$ to be a network composed of the following subnetworks : N_1, and the rest of the network context, call it N_c. Suppose t is a trace of $\mathcal{C}[N_1]$. Then, by Theorem 2, there is a sequence t' of events on the input and output ports of the subnetworks, such that the restriction of t' to any subnetwork is a trace of that subnetwork, and the restriction of t' to the input and output ports of the whole network is exactly t. Since N_1 and N_2 have the same set of traces by assumption, and we can consider $\mathcal{C}[N_2]$ to be composed of N_2 and N_c, just as we did for $\mathcal{C}[N_1]$, t' is also a sequence of events on the input and output ports of these subnetworks, such that the restriction to every subnetwork is a trace of that subnetwork. Then, using Theorem 2 once more, we conclude that the restriction of t' to the input and output ports of $\mathcal{C}[N_2]$ is a trace of $\mathcal{C}[N_2]$. But this restriction is exactly t. Therefore t is a trace of $\mathcal{C}[N_2]$. Hence every trace of $\mathcal{C}[N_1]$ is a trace of $\mathcal{C}[N_2]$, and vice-versa. ∎

Adequacy follows almost immediately.

Theorem 7. The semantics associating each network of processes with its trace set is adequate.

Proof : Suppose two networks N_1 and N_2 have the same trace sets. Then, by the previous lemma, for every context $\mathcal{C}[\ ]$, $\mathcal{C}[N_1]$ and $\mathcal{C}[N_2]$ have the same trace sets. Then it follows by Lemma 11, that $\mathcal{C}[N_1]$ and $\mathcal{C}[N_2]$ have the same input-output relation. ∎

We now concentrate on our proof of full abstraction. We will use the closure properties of trace sets, that we proved in the previous subsection, in this proof. We recall that the main requirement for this proof is to describe a network context that can distinguish between processes having different trace sets by making the networks have different input-output relations. It is quite surprising that, for these purposes, one can define a simple *uniform* context – the context

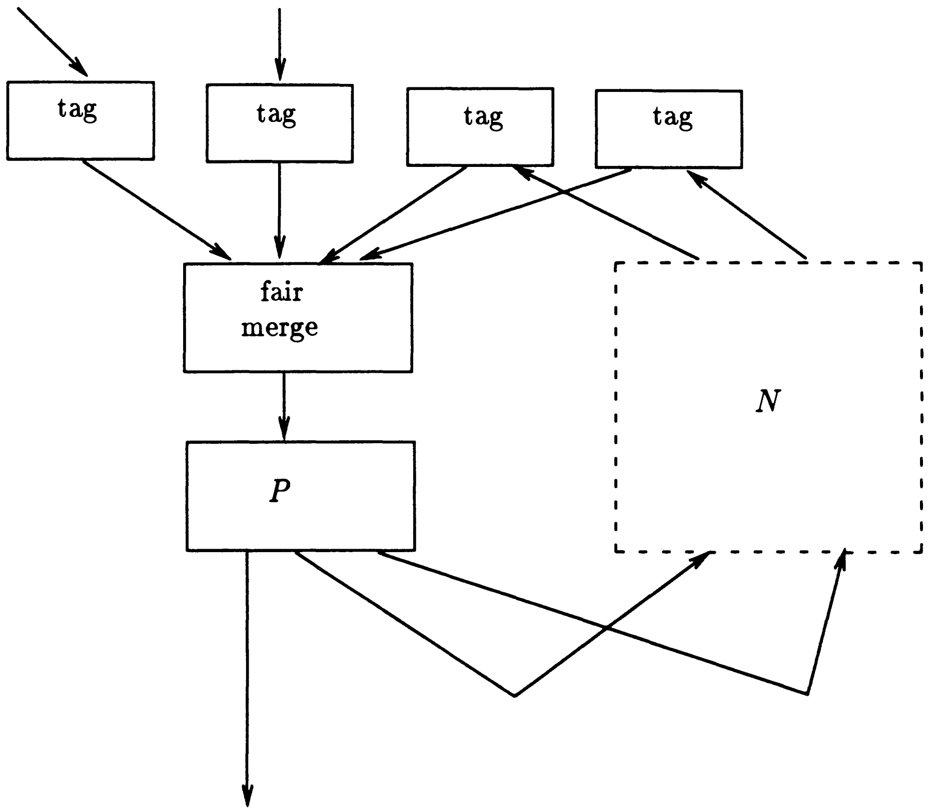

Figure 3.3: The Context $\mathcal{D}_{2,2}$

is independent of the processes that are to be distinguished. Actually this is not strictly correct, because the context depends on the number of input ports and the number of output ports of the processes being distinguished, but that is all. Let the context $\mathcal{D}_{2,2}$ be as shown in Figure 3.3. We will use this to differentiate between two networks of processes, both having two input ports and two output ports, when they have different trace sets. The generalization to m input ports and n output ports is straightforward, and we define a family of contexts $\mathcal{D}_{m,n}$. All that we need do is use a multiway fair merge, one with more than two input ports, instead of the ordinary fair merge in $\mathcal{D}_{2,2}$ with two input ports. A multiway

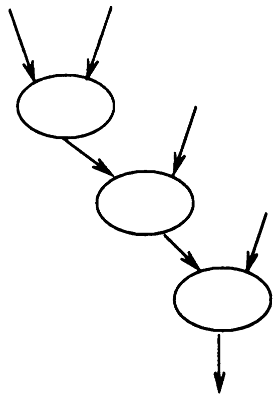

Figure 3.4: 4-way fair merge using 2-way fair merges

fair merge can easily be built by using many copies of the two-way fair merges and building a tree-like network, as in Figure 3.4. A formal proof of this claim may be found in [Sha90]. We now describe the context $\mathcal{D}_{m,n}$. We refer to the annotations in Figure 3.3 in the description that follows. There is a bijection $\mathcal{F}$ between the set of m input ports of $\mathcal{D}_{m,n}[N]$ and the set of m input ports of N. If H is a history over the input ports of $\mathcal{D}_{m,n}[N]$, then $\mathcal{F}(H)$ will represent the corresponding history over the input ports of N. Informally, each tagger process simply reads values from its input port, tags them with its input port name, and outputs them. The process P reads a value v and checks the tag p. If p is the name of an output port of N, then P simply outputs the value v onto the output port of the network. If p is the name of an input port of the network, then P strips the tag off v and outputs the stripped value v' onto the appropriate input port $\mathcal{F}(p)$ of N. P also tags the value v' with the port name $\mathcal{F}(p)$ and outputs the tagged value onto the output port of the network. Formal descriptions of the automata and trace sets for these processes is straightforward.

We will refer to the context $\mathcal{D}_{m,n}$ simply as $\mathcal{D}$, leaving the subscripts m, n

implicit. We now note that every output value of the network $\mathcal{D}[N]$ contains a tag which is an input or output port name of N. Hence an output stream of the network may be construed as an event sequence of N.

Definition 29. If $e = (p, v)$ is an event, then $\tau(e)$ is defined as the value representing v tagged by the port name p.

If $t = e_1, e_2, \ldots$ is a sequence of input or output events of a network, then we will refer to the sequence $\tau(e_1), \tau(e_2), \ldots$ by $\tau(t)$.

Lemma 13. For every trace t of N for some input history H of N, $\tau(t)$ is a possible output stream of $\mathcal{D}[N]$ for the input history $\mathcal{F}^{-1}(H)$ of $\mathcal{D}[N]$.

Proof : We consider the network $\mathcal{D}[N]$ as the composition of three subnetworks : N, the process P with input port p_I and an output port p_O that is also the output port of $\mathcal{D}[N]$, as well as all the input ports of N as output ports, and the rest of $\mathcal{D}[N]$, call it N', composed of the fair merge and the tagger processes. Let t be a trace of N for some input history H of N. We shall construct a sequence t' of events on the input and output ports of the subnetworks, such that t' restricted to the events of any subnetwork is a trace of that subnetwork, $\Pi_N(t') = t$, and $\Pi_{p_O}(t') = \tau(t)$. Then, by Lemma 2, $\Pi_{\mathcal{D}[N]}(t')$ is a trace of $\mathcal{D}[N]$, and hence $\tau(t)$ is a possible output stream of $\mathcal{D}[N]$ for the input history $\mathcal{F}^{-1}(H)$.

For every input event occurrence $e = (p, v)$ in t, we replace it by the sequence $\mathcal{F}^{-1}(p), (p_I, \tau(e)), (p_O, \tau(e)), e$. We will refer to these four events as e^1, e^2, e^3 and e^4 respectively. For every output event occurrence $e = (p, v)$ in t, we replace it by the sequence $e, (p_I, \tau(e)), (p_O, \tau(e))$. We will refer to these three events as e^1, e^2 and e^3 respectively. The new sequence of events thus obtained is the desired t'. By Lemma 2, $\Pi_{\mathcal{D}[N]}(t')$ is a trace of $\mathcal{D}[N]$ for the input history $\mathcal{F}^{-1}(H)$, and this trace has $\tau(t)$ as the corresponding output stream $\Pi_{p_O}(t)$. ∎

Lemma 14. For any input history H for the network of processes $\mathcal{D}[N]$, every possible output stream of $\mathcal{D}[N]$ is $\tau(t)$ for some trace t of N for the input history $\mathcal{F}(H)$ of N.

Proof : Suppose s is an output stream of $\mathcal{D}[N]$ for input history H. Then there is a completed computation sequence γ of $\mathcal{D}[N]$ for that input history, such that the corresponding output stream is s. Let the restriction of γ to the subnetwork N be the completed computation sequence γ' of N, and let the corresponding trace of N be t'. We will construct a move transformation, such that, by the closure of trace sets under move transformations, when this move transformation is applied to t', then we obtain a trace t of N for the input history $\mathcal{F}(H)$, and $\tau(t) = s$.

Every value output by $\mathcal{D}[N]$ is a value tagged by an input or output port of N. Thus the output stream s is $\tau(t)$ for some sequence of events t of N. It is also easy to see that, for every input and output event occurrence e in t', $\tau(e)$ is a value in s, and moreover, every value in s is $\tau(e)$ for some event occurrence in t'. This is straightforward by the property of fair merge that it outputs every value input to it.

We claim that t is the desired trace of N. To construct the required move transformation, transforming t' into t, we use the following notation – κ_0 is t', and κ_i is the result of applying a sequence of moves η_i to κ_{i-1}, or, equivalently, applying the sequence of moves $\eta_1{}^\wedge\eta_2{}^\wedge \ldots \eta_i$ to κ_0. We describe the construction inductively, maintaining the invariant that t and κ_i agree on the first i events. Suppose $\eta_1 \ldots \eta_i$ have already been constructed.

(i) The $(i+1)$th events in κ_i and t are identical. Then define η_{i+1} to be the empty sequence of moves.

(ii) The $(i+1)$th events in κ_i and t are not identical. Then the $(i+1)$th event in t must be the $(i+k)$th event in κ_i for some $k > 1$. Then we define η_{i+1} to be the sequence $(i+k-1, i+k), (i+k-2, i+k-1), \ldots (i+1, i+2)$.

We need to check that the moves above are legal. We first observe that if e_1 and e_2 are two events of t' at the same input or output port of N, then e_1 precedes e_2 in t' iff the corresponding event occurrences occur in the same order in t. This is an easy proof, noting that if both e_1 and e_2 were input events, then N must have executed the event occurrence for e_1 before the event occurrence for e_2, hence P must have output $\tau(e_1)$ before $\tau(e_2)$. And if both e_1 and e_2 were output events, then the fair merge process outputs $\tau(e_1)$ before $\tau(e_2)$, and hence P outputs $\tau(e_1)$ before $\tau(e_2)$.

We also observe that if e_1 is an input event of t', and e_2 is an output event of t', and e_1 precedes e_2 in t', then the corresponding event occurrences occur in the same order in t. This is because $(p_I, \tau(e_1))$ precedes e_1 in the computation sequence, and e_2 precedes $(p_I, \tau(e_2))$ in the computation sequence, and therefore $(p_I, \tau(e_1))$ precedes $(p_I, \tau(e_2))$, and hence P outputs $\tau(e_1)$ before $\tau(e_2)$.

Let e_2 be the $(i+k)$th event in κ_i in case(ii). The sequence in case(ii) would be a sequence of moves if e_2 is an input event occurrence at port p, and if the $(i+1)$th to $(i+k-1)$th events in κ_i are all output events or input events at ports other than p. If this is not the case, then there is some event e_1 at the port p among the $(i+1)$th to $(i+k-1)$th events in κ_i. But this would mean that the event occurrence corresponding to e_2 precedes that corresponding to e_1 in t, contradicting earlier observation.

If e_2 is an output event occurrence at the port p and the $(i+1)$th to $(i+k-1)$th events in κ_i are all output events, then a similar argument as above shows that none of the $(i+1)$th to $(i+k-1)$th events in κ_i are output events at the port p. Suppose there is an input event e_1 among the $(i+1)$th to $(i+k-1)$th events in

κ_i. Then e_1 precedes e_2 in κ_i, and hence in t' because the relative order of these event occurrences has not been affected by the moves $\eta_1{}^\wedge\eta_2{}^\wedge\ldots\eta_i$. Therefore, by earlier observation, the event occurrence corresponding to e_1 must precede that corresponding to e_2 in t, and this is not the case, thus giving us a contradiction.

So now we have a sequence of moves $\eta_1{}^\wedge\eta_2{}^\wedge\ldots$. It is clear that this satisfies the definition of a move transformation. Moreover, applying the sequence of moves to t' produces t, because for every $i > 0$, the prefix of t of length i is exactly the prefix of length i of the result of applying $\eta_1{}^\wedge\eta_2{}^\wedge\ldots\eta_i$ to t', and this is exactly the prefix of length i of the result of applying the entire move transformation to t', because moves in η_{i+1} onwards do not touch the first i events of κ_i. ∎

The main theorem now follows easily.

Theorem 8. (Full Abstraction) Two networks N_1 and N_2 of processes have the same trace sets iff they are operationally equal.

Proof : The "only if" direction followed from the adequacy theorem 7.

For the "if" direction, suppose N_1 and N_2 have different trace sets. Then there is an input history H of N_1 and N_2 for which t is a trace of N_1, but not a trace of N_2. Then, by the previous lemmas, $\mathcal{D}[N_1]$ has $\tau(t)$ as a possible output stream for the input history $\mathcal{F}^{-1}(H)$, but $\mathcal{D}[N_2]$ does not, thus differentiating the input-output relations of $\mathcal{D}[N_1]$ and $\mathcal{D}[N_2]$. Therefore N_1 and N_2 are not operationally equal. ∎

In this chapter, we described traces as an abstraction from computation sequences. We also described sequential processes, that we use as the base set of processes for inexpressibility proofs, and proved some of their properties that we will use in the next chapter. The next chapter deals with these inexpressibility proofs.

Chapter 4

Inexpressibility Results

In previous chapters, we have described our model of computation, including both an operational semantics and a compositional trace set semantics. We now wish to understand the different "levels" of indeterminacy that arise in this model. These levels will be described in later sections using the merge and split processes that were defined in the introduction.

4.1 Inexpressibility

In this section, we will establish our notions of when one level is stronger than another, and when one level is weaker than another. We will also describe the general strategy that we will use to prove that one level is weaker than another – to prove the inexpressibility of one level by another.

We have already described the rules by which we can construct networks of processes. We now establish the definitions for *implementability* and *non-implementability*.

Definition 30. A set S of processes can **implement** a relation R if there is a finite network N, built out of copies of processes in S, such that R is the input-output relation of N.

Definition 31. A set S of processes can **weakly implement** a process or a network M if there is a finite network N, built out of copies of processes in S, such that N and M have the same input-output relation.

Definition 32. A set S of processes can **strongly implement** a process or a network M if there is a finite network N, built out of copies of processes in S, such that N and M have the same trace set.

To prove that a set S of processes cannot strongly implement a process M, it suffices, by lemma 11, to show that the set S of processes cannot weakly implement the process M. Therefore it suffices to show that the set S of processes cannot implement the input-output relation of M. This is how we will prove inexpressibility results.

The general strategy that we will follow to prove that a set S of processes cannot implement a relation R is the following :

1. define a predicate ϕ on trace sets of processes and networks that is true for every process in S,

2. define a predicate ψ on input-output relations of processes and networks, such that for any network N of processes, if ϕ is true for the trace set of N, then ψ is true for the input-output relation of N,

3. prove that the predicate ϕ is true for every finite network built out of processes in S, and

4. prove that ψ does not hold for the relation R.

So we need to define appropriate predicates ϕ and ψ, and the proof that ϕ is true for every finite network N built out of processes in S should be a structural induction proof on the structure of the network N. This proof will use the compositionality of the trace set semantics.

In all our implementability and non-implementability proofs, we will always assume that we can use copies of processes from a specific base set of processes. These are the sequential processes described earlier.

We emphasize that whenever we say that a process or network N_1 can or cannot strongly implement (weakly implement) another process or network N_2, then we really mean that the set of processes consisting of all sequential processes and N_1 cannot strongly implement (weakly implement) N_2.

Definition 33. Networks N_1 and N_2 are said to be **at the same expressiveness level** if they can strongly implement each other.

Definition 34. A network N_1 is said to be **more expressive** than N_2 if N_1 can strongly implement N_2, but N_2 cannot strongly implement N_1.

We will also say that N_2 is **less expressive** than N_1 in the above definition.

4.2 The Merge Hierarchy

In this section, we study properties of the merge processes defined in the introduction, and prove that the different merge processes actually form different levels in a hierarchy of expressiveness. In the next section, we will describe how the split processes also form a hierarchy of expressiveness, and how the two hierarchies connect with each other very nicely.

The properties that we define and study in this section are two different notions of "monotonicity." We will refer to these as "Hoare-monotonicity" and "Smyth-monotonicity," and these are properties of trace sets. We will prove that Smyth-monotonicity of trace sets is preserved under network composition, but it turns out to be exceedingly difficult to prove that Hoare-monotonicity of trace sets is preserved under network composition. So a stronger property of trace sets, that implies Hoare-monotonicity, needs to be proved. This is a "continuity" kind of property.

4.2.1 Hoare-monotonicity

The intuition behind the definition of Hoare-monotonicity is that, in a monotone network, i.e. satisfying the non-disabling input property (definition 4), arrival of values at input ports cannot disable output of values that were already enabled before the arrival of the values at the input ports. This property was also used by Panangaden and Stark[PS88b] in an operational setting to prove that there is no monotone network that can implement the input-output relation of fair merge. We will formulate this property as a property of trace sets, and we will also describe a stronger property - continuity - that is preserved by network composition. The proof is in [PS]. We will just describe here how this implies that angelic merge cannot weakly implement fair merge. In fact, our proof is more general and shows that no set of Hoare-monotone and continuous processes can weakly implement fair merge.

Definition 35. Let t_1 and t_2 be two traces of a network N. Then the **relative order of events in** t_1 is said to be preserved in t_2 if the following two conditions hold :

(i) for every port p of N, $\Pi_p(t_1) \sqsubseteq \Pi_p(t_2)$, and

(ii) for every pair e_1, e_2 of event occurrences in t_1, if e_1 is the ith event on port p_1, and e_2 is the jth event on port p_2, then e_1 precedes e_2 in t_1 iff the ith event on port p_1 in t_2 precedes the jth event on port p_2 in t_2.

Definition 36. Let N be a network with input ports $\vec{a}$ and output ports $\vec{b}$. $Trset(N)$ is said to be **Hoare-monotone** if, for any trace $t \in Trset(N)$ with $\Pi_{\vec{a}}(t) = \vec{\alpha}$, and for any $\vec{\alpha}' \sqsupseteq \vec{\alpha}$, there is a trace $t' \in Trset(N)$ with $\Pi_{\vec{a}}(t') = \vec{\alpha}'$, with $\Pi_{\vec{b}}(t) \sqsubseteq \Pi_{\vec{b}}(t')$, and with the relative ordering of the events in t preserved in t'.

This says that, given a trace, we can always extend the input, and there will be some trace that represents the response of the network to the new input and in this new trace, the sequence of values on each port will be an extension of or equal to the sequence of values seen before. Clearly we cannot expect that every response will be an extension (in the above sense) of or equal to the old response since the networks are indeterminate. This definition captures the idea that if there was an enabled output then adding new input will not disable this output.

Definition 37. $Trset(N)$ is said to be **continuous** if it is Hoare-monotone, and satisfies the following property. Suppose that $\{t_i | i < \omega\}$ is a set of traces of N such that for any port p, $\Pi_p(t_i)$ is a chain in the prefix ordering, and the relative order of events is preserved along this chain. Then there is a trace $t \in Trset(N)$ such that, for any port p we have $\Pi_p(t) = \sqcup \Pi_p(t_i)$ and the relative order of events in t_i is preserved in t for every i.

The proof that a network composed of continuous components has to be continuous may be found in [PS87,PS]. It is also proved in [PS87,PS] that hiding output ports of a network preserves Hoare-monotonicity.

We now define a predicate on the input-output relations of networks, such that Hoare-monotonicity of the trace set of a network implies the truth of this predicate on the input-output relation of that network. We will give the same name Hoare-monotonicity to this predicate.

Definition 38. The input-output relation of a network N is said to be **Hoare-monotone**, if whenever for some input $\vec{\alpha}$, there is an output $\vec{\beta}$, and $\vec{\alpha'} \sqsupseteq \vec{\alpha}$, then there is a possible output $\vec{\beta'} \sqsupseteq \vec{\beta}$.

Note that since we are only talking of the input-output relation here, we do not have requirements as for Hoare-monotonicity of trace sets, such as the preservation of relative order of events in traces.

The trace set of angelic merge can be described as follows. If an angelic merge process has input ports i_1, i_2 and output port o, then the trace set of the process consists of all sequences $t \in (\{i_1, i_2, o\} \times V)^\infty$ such that $\Pi_o(t)$ can be broken up into two subsequences s_1, s_2 such that

(i) $value(s_1)$ is a prefix of $value(\Pi_{i_1}(t))$ and $value(s_2)$ is a prefix of $value(\Pi_{i_2}(t))$,

(ii) if $\Pi_{i_1}(t)$ is finite, then $value(s_2) = value(\Pi_{i_2}(t))$,

(iii) if $\Pi_{i_2}(t)$ is finite, then $value(s_1) = value(\Pi_{i_1}(t))$, and

(iv) for every prefix t' of t, the prefix of s_1 in t' is a prefix of $\Pi_{i_1}(t')$, and the prefix of s_2 in t' is a prefix of $\Pi_{i_2}(t')$.

We prove in [PS] that this trace set is continuous. This leads to the main theorem.

Theorem 9. Fair merge cannot be weakly implemented by any finite network of continuous processes, including angelic merge.

Proof : As proved in [PS87,PS], the trace set of any finite network built out of continuous processes by aggregation, feedback and output hiding is Hoare-monotone, and hence the input-output relation is also Hoare-monotone. But fair merge does not have a Hoare-monotone input-output relation. To see this, consider the case where the input sequences of fair merge are 1^∞ and Λ, the empty sequence. Then 1^∞ is the only possible output sequence. Now if the sequence on the second input port is extended to 2, then the output sequence must include the value 2, and no such output sequence is an extension of 1^∞. ∎

We prove in [PS], using Lemma 5 that every determinate process has a continuous trace set. Therefore every process in our base set of processes, the sequential processes, has a continuous trace set, leading to the following corollary.

Corollary 4. Angelic merge cannot weakly implement fair merge.

4.2.2 Smyth-monotonicity

Hoare-monotonicity described the property that given a trace t, if we consider an increased input, then there is a trace t' with that increased input and an equal or increased output, and further the relative ordering of events in t is preserved in t'. We now define a "similar" property. Informally, this property says that given a trace t, if we consider a decreased input, then there is a trace t' with that decreased input and an equal or decreased output, and further the relative ordering of events in t' is preserved in t.

Definition 39. $Trset(N)$ is said to be *Smyth-monotone* if for any trace t with input sequences $\vec{\alpha}$ and output sequences $\vec{\beta}$, if $\vec{\alpha'} \sqsubseteq \vec{\alpha}$, then there is a trace t' with input $\vec{\alpha'}$ and output $\vec{\beta'}$ and $\vec{\beta'} \sqsubseteq \vec{\beta}$ and the relative order of events in t' is preserved in t.

The next lemmas establish that Smyth-monotonicity is preserved by network composition.

Lemma 15. Suppose N_1 and N_2 are Smyth-monotone networks. Then their aggregate N is also Smyth-monotone.

Proof : Let t be a trace of N. Then t is an interleaving of a trace t_1 of N_1 and a trace t_2 of N_2. Let $\vec{\alpha}$ and $\vec{\beta}$ be the input sequences and output sequences respectively in t_1. Let $\vec{\gamma}$ and $\vec{\delta}$ be the input sequences and output sequences respectively in t_2. Let $\vec{\alpha'} \sqsubseteq \vec{\alpha}$ and $\vec{\gamma'} \sqsubseteq \vec{\gamma}$. Then, by Smyth-monotonicity of N_1 and N_2, there are traces t_3 and t_4, of N_1 and N_2 respectively, such that the relative order of events in t_3 and t_4 are preserved in t_1 and t_2 respectively. Then an appropriate interleaving of t_3 and t_4 is a trace of N, the relative order of events of which are preserved in t. ∎

Lemma 16. If N is a Smyth-monotone network with an output port p_1 and an input port p_2, then the network $loop(p_1, p_2, N)$ is Smyth-monotone.

Proof : Let M be the network $loop(p_1, p_2, N)$. We recall that

$$Trset(M) = \{\Pi_{\sim p_2}(t) | t \in Trset(N) \wedge precedes(p_1, p_2, t) \wedge (\Pi_{p_1}(t) = \Pi_{p_2}(t))\}$$

Let t' be a trace of M with $\vec{\alpha}$ and $\vec{\beta}$ as the input sequences and output sequences respectively. There is a trace t of N, such that $t' = \Pi_{\sim p_2} t$ and $precedes(p_1, p_2, t)$ and $\Pi_{p_1}(t) = \Pi_{p_2}(t)$. Let $\delta = value(\Pi_{p_1}(t))$. Then the input sequences in t are $\vec{\alpha}; \delta$. Let $\vec{\alpha'} \sqsubseteq \vec{\alpha}$. By Smyth-monotonicity of N, there is a trace r_1 of N, such that the input sequences are $\vec{\alpha'}; \delta$ and $value(\Pi_{p_1}(r_1)) = \delta_1 \sqsubseteq \delta$ and the relative order of events in r_1 are preserved in t.

If $\delta = \delta_1$ then we are done because then $\Pi_{\sim p_2} r_1$ is the desired trace of M. If not, we iterate as outlined below until we finally obtain the desired trace. At each stage of the iteration, we construct new traces such that the relative order of events in newer traces is preserved in the older traces.

The iteration proceeds as follows: If $\delta \neq \delta_1$ then δ_1 is a proper prefix of δ. This implies that $\vec{\alpha'}; \delta_1$ is a prefix of $\vec{\alpha'}; \delta$. So, by Smyth-monotonicity of N, there is a trace r_2 with input sequences $\vec{\alpha'}; \delta_1$ and $value(\Pi_{p_1}(r_2)) = \delta_2 \sqsubseteq \delta_1$ and $precedes(p_1, p_2, r_2)$.

If we repeat the above step, we will reach $\delta_i = \delta_{i+1}$ and then we will be done. This procedure will terminate because the prefix ordering is a well-ordering and δ_i must certainly hit $\emptyset$ and stabilize if it does not stabilize earlier. ∎

Lemma 17. Any network obtained by hiding some output ports of a Smyth-monotone network is Smyth-monotone.

Proof : Let N be a Smyth-monotone network, and let M be the network obtained by hiding some set S of output ports of N. Let t be a trace of M. Let the input be $\vec{\alpha}$ and the output be $\vec{\beta}$. By theorem 5, there is a trace t' of N such that $t = \Pi_{\sim S}(t')$. The output in t' is $\vec{\beta};\vec{\gamma}$ where $\vec{\gamma}$ consist of the sequences of events at the ports in S. Let $\vec{\alpha'} \sqsubseteq \vec{\alpha}$. Then by the Smyth-monotonicity of N, there is a trace $s' \in Trset(N)$ with input $\vec{\alpha'}$ and output $\vec{\beta'};\vec{\gamma'}$, such that the relative order of events in s' is preserved in t'. Then $\Pi_{\sim S}(s')$ is a trace of M with the desired property. ∎

We now define a predicate on input-output relations that is true for any network whose trace set has the Smyth-monotonicity property. We will refer to this predicate as Smyth-monotonicity too.

Definition 40. The input-output relation of a network N is said to be **Smyth-monotone**, if whenever for some input $\vec{\alpha}$, there is an output $\vec{\beta}$, and $\vec{\alpha'} \sqsubseteq \vec{\alpha}$, then there is a possible output $\vec{\beta'} \sqsubseteq \vec{\beta}$.

Lemma 18. Angelic merge does not have a Smyth-monotone input-output relation.

Proof : Let the input sequences at the two ports of angelic merge be 1^∞ and $2^\wedge 3$. Then one possible output is $2^\wedge 3^\wedge 1^\infty$. Now consider the input sequences 1^∞ and 2 respectively at the two input ports. The only possible outputs are

$1^i{}^\wedge 2^\wedge 1^\infty$ for $i \geq 0$, and 1^∞. $2^\wedge 3^\wedge 1^\infty$ is not an extension of 1^∞ or $1^i{}^\wedge 2^\wedge 1^\infty$ for any i. ∎

We can describe the trace set of infinity-fair merge as follows. If an infinity-fair merge process has input ports i_1, i_2 and output port o, then the trace set of the process consists of all sequences $t \in (\{i_1, i_2, o\} \times V)^\infty$ such that $\Pi_o(t)$ can be broken up into two subsequences s_1, s_2 such that

(i) *value*(s_1) is a prefix of *value*($\Pi_{i_1}(t)$) and *value*(s_2) is a prefix of *value*($\Pi_{i_2}(t)$),

(ii) if $\Pi_{i_1}(t)$ is infinite, then *value*(s_2) = *value*($\Pi_{i_2}(t)$),

(iii) if $\Pi_{i_2}(t)$ is infinite, then *value*(s_1) = *value*($\Pi_{i_1}(t)$),

(iv) if either of $\Pi_{i_1}(t)$ or $\Pi_{i_2}(t)$ is finite, then $\Pi_o(t)$ is finite, and either *value*(s_1) = *value*($\Pi_{i_1}(t)$) or *value*(s_2) = *value*($\Pi_{i_2}(t)$), and

(iv) for every prefix t' of t, the prefix of s_1 in t' is a prefix of $\Pi_{i_1}(t')$, and the prefix of s_2 in t' is a prefix of $\Pi_{i_2}(t')$.

To see that this trace set is Smyth monotone, let t be a trace of infinity-fair merge. $\Pi_o(t)$ can be broken up into s_1, s_2 as above. Let α, β be sequences of events that are prefixes of $\Pi_{i_1}(t), \Pi_{i_2}(t)$ respectively. We can delete all the input events not in α or β from t, and we also delete all the corresponding output events in s_1 and s_2. The result is a trace t' of infinity-fair merge, and the relative order of events in t' is preserved in t. We now prove our main theorem.

Theorem 10. Angelic merge cannot be weakly implemented by any finite network of Smyth-monotone processes, including infinity-fair merge.

Proof : Smyth-monotonicity of trace sets is preserved under aggregation, feedback and output hiding. Therefore the trace set of any finite network of Smyth-monotone processes must be Smyth-monotone, and hence the network must have

$$\begin{array}{c} \text{fair} \\ \downarrow \not\uparrow \\ \text{angelic} \\ \downarrow \not\uparrow \\ \text{infinity-fair} \end{array}$$

Figure 4.1: A Straight-line Picture of Merges

a Smyth-monotone input-output relation. But angelic merge does not have a Smyth-monotone input-output relation. Therefore no such finite network can weakly implement angelic merge. ∎

By lemma 6, it follows that determinate processes have Smyth-monotone trace sets. Therefore, every process in our base set of processes, the sequential processes, has a Smyth-monotone trace set, thus implying the following corollary.

Corollary 5. Infinity-fair merge cannot weakly implement angelic merge.

4.2.3 Some strong implementability results

The results in the previous subsection show that infinity-fair merge cannot weakly implement angelic merge, and that angelic merge cannot weakly implement fair merge. Eugene Stark showed how one could strongly implement infinity-fair merge using angelic merge [Sta90]. We will now show how one could implement angelic merge using fair merge, thus providing a very nice straight-line picture of expressibility (figure 4.1).

Figure 4.2 shows the network that strongly implements angelic merge. The process P is a determinate process whose behaviour can be described as follows : it reads values from its first input port and outputs the values that are not

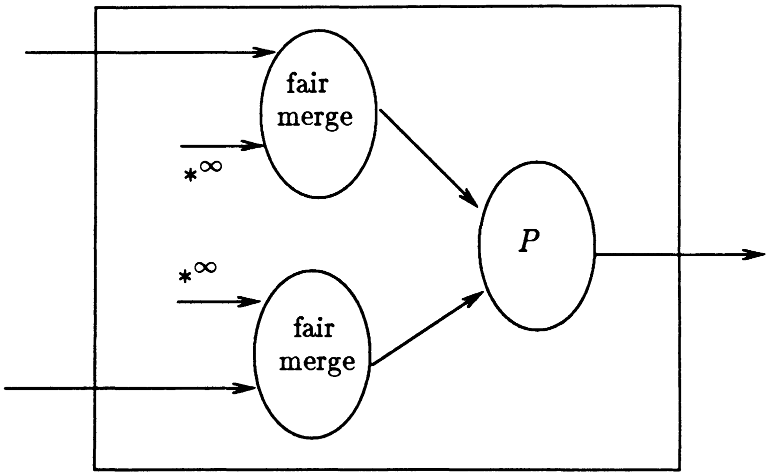

Figure 4.2: Angelic Merge from Fair Merge

$*$, until it reads two successive $*$'s, in which case it switches to reading from its second input port and reads and outputs values that are not $*$ until it reads two successive $*$'s, in which case it switches back to the first input port and continues as above. We could formally describe this as an automaton, but the behaviour should be clear from the given description.

4.2.4 A variation of Infinity-fair merge

We have already noted that infinity-fair merge is both Hoare-monotone and Smyth-monotone, angelic merge is Hoare-monotone but not Smyth-monotone, and fair merge is neither. Infinity-fair2 merge complements this picture very nicely because we will show that it is Smyth-monotone but not Hoare-monotone. This immediately gives us, by the lemmas earlier in this section, the theorems that angelic merge cannot weakly implement infinity-fair2 merge, infinity-fair2 merge cannot weakly implement angelic merge, infinity-fair merge cannot weakly implement infinity-fair2 merge, and infinity-fair2 merge cannot weakly imple-

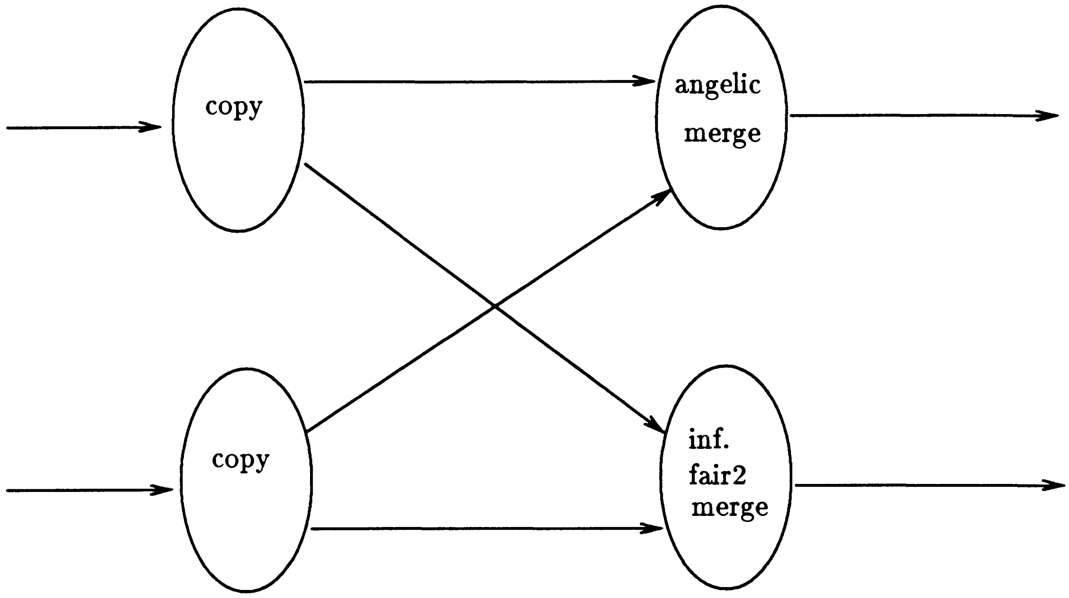

Figure 4.3: Neither Hoare-monotone nor Smyth-monotone

ment fair merge. Some implementability results are that fair merge can strongly implement infinity-fair2 merge, and infinity-fair2 merge can strongly implement infinity-fair merge. We now prove that infinity-fair2 merge and angelic merge, even together, cannot implement fair merge. This is not obvious because, by combining these two processes, we can build networks that are neither Hoare-monotone nor Smyth-monotone – see figure 4.3 for a simple example. In this example, the *copy* process simply outputs a copy of its input stream onto both its output ports. Then the presence of angelic merge causes the input-output relation of the network not to be Smyth-monotone, and the presence of infinity-fair2 merge causes the input-output relation of the network not to be Hoare-monotone. Due to the presence of such examples, it is not clear that fair merge, which is neither Hoare-monotone nor Smyth-monotone, cannot be implemented.

Theorem 11. Angelic merge and infinity-fair2 merge cannot weakly implement fair merge.

Proof : Suppose there is a finite network N of angelic merge and infinity-fair2 merge processes and sequential processes that weakly implements fair merge. We note that the trace set of infinity-fair merge is a subset of the trace set of infinity-fair2 merge. Therefore if we obtain a network N' by replacing every infinity-fair2 merge process in N by an infinity-fair merge process, then N' weakly implements a total subset of the input-output relation of fair merge. By earlier lemmas, N' has a Hoare-monotone trace set, and hence a Hoare-monotone input-output relation. Consider the input sequences 1^∞ and Λ at the two input ports of the network N'. Since the network weakly implements a total subset of the input-output relation of fair merge, the output has to be 1^∞. Now we consider the extension of the input sequence at the second input port to 2. Again, since the network weakly implements a subset of the input-output relation of fair merge, the output has to be of the form $1^{i\wedge}2^{\wedge}1^\infty$ for some $i \geq 0$. But none of these sequences extend 1^∞, violating the fact that the input-output relation of N' is Hoare-monotone. ∎

4.3 The Split Hierarchy

We now turn our attention to the duals of merge processes. These are the split processes that split any sequence of values into two subsequences. Informally, just as the merge processes "gathered information," the split processes "disperse information." The different kinds of guarantees that can be offered about the splitting gives rise to our split hierarchy.

4.3.1 Oracles

Definition 41. An **oracle** is a process with no input ports and one output port p, such that the process can output any one of the sequences in some specific set $S \subseteq V^{\infty}$ of sequences at the output port p.

One example of an oracle is a process with no input ports and one output port, at which the process can output any positive integer. Our results then show that in our dataflow model of computation, there are many provably different kinds of oracles. These results tie in very well with the merge hierarchy of the previous section, as we shall see.

Let us define the three oracles that we will be most concerned with in this section. Each of these three oracles will be described by their sets of possible outputs, which will be sequences of 1's and 2's.

1. the set O_{us} of all infinite sequences of 1's and 2's,
2. the set O_{ws} of all infinite sequences of 1's and 2's containing at least one 1 and at least one 2, and
3. the set O_{ss} of all infinite sequences of 1's and 2's containing infinitely many 1's and infinitely many 2's.

We will also refer to the oracle processes corresponding to these sets of outputs by O_{us}, O_{ws} and O_{ss} respectively.

We can now strongly implement any one of the six split processes by using the appropriate oracle in the networks shown in 4.4 and either suppressing or not suppressing the output at o_s. The process P reads a value v from its input port i. If there are no values to be read, it waits until there is a value to be read. It then reads a value v' from the oracle output. If $v' = 1$, then it outputs v at port o_1. If $v' = 2$, then it outputs v at port o_2. It also outputs v' at o_s and then

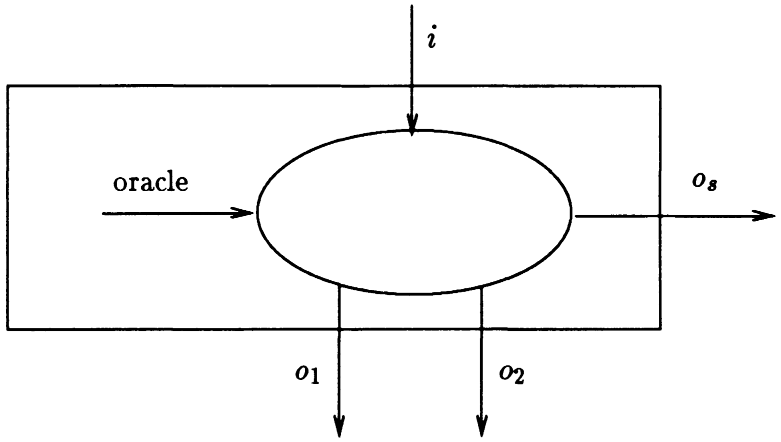

Figure 4.4: Split from an oracle

repeats the above. When the oracle chosen is O_{us}, then we strongly implement unfair split and unfair split with signal. When the oracle chosen is O_{ws}, then we strongly implement weakly fair split and weakly fair split with signal. When the oracle chosen is O_{ss}, then we strongly implement strongly fair split and strongly fair split with signal.

It is also clear that unfair split with signal can strongly implement O_{us}, weakly fair split with signal can strongly implement O_{ws}, and strongly fair split with signal can implement O_{ss}. This is because the signal output port of these split processes can be used to *manufacture* the oracle outputs at their signal output channels by making the input sequences of these split processes be any infinite sequence, say 1^{∞}. Therefore the following theorem follows.

Theorem 12. Unfair split with signal and O_{us} are at the same expressiveness level. Weakly fair split with signal and O_{ws} are at the same expressiveness level. Strongly fair split with signal and O_{ss} are at the same expressiveness level.

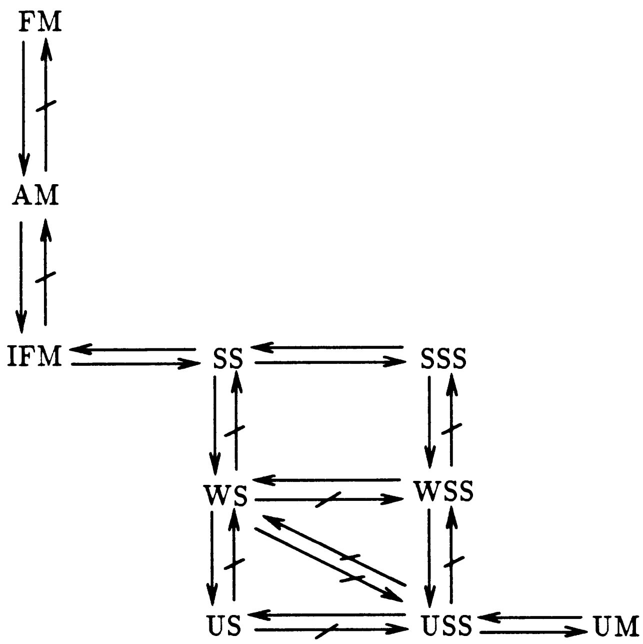

Figure 4.5: The Merge and Split Hierarchies

4.3.2 Some Strong Implementability Results

We will now describe the expressiveness picture in figure 4.5, in which an arrow from one process to another indicates that the first process can strongly implement the second.

Theorem 13. Strongly fair split can strongly implement strongly fair split with signal.

Proof : The figure 4.6 illustrates how we can manufacture the oracle O_{ss} from strongly fair split. The determinate process P receives an infinite increasing sequence of positive integers at port p such that the sequence has an infinite increasing complement. P outputs an infinite sequence of 1's and 2's at port o_s,

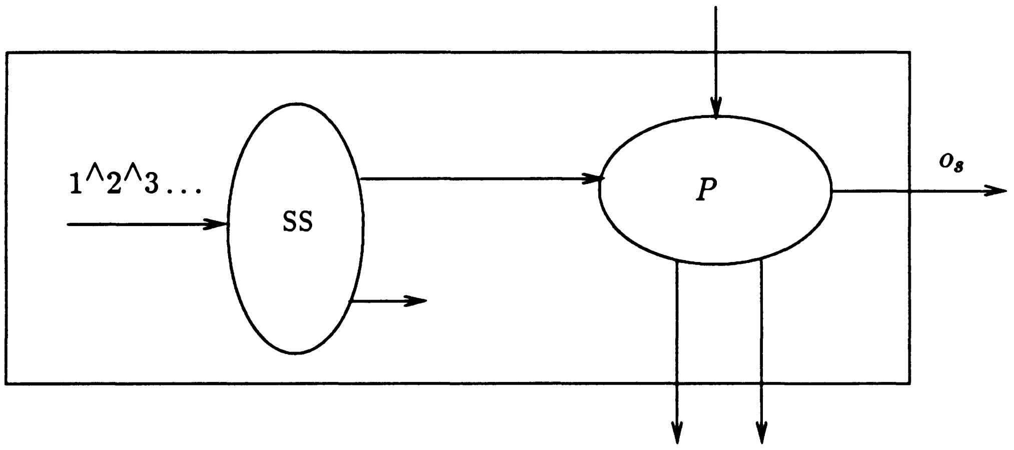

Figure 4.6: SSS from SS

such that the ith value in the sequence is a 1 if and only if P reads the value i from port p.

The oracle O_{ss} can now be used to strongly implement strongly fair split with signal, as described earlier. ∎

This implies that strongly fair split, strongly fair split with signal and O_{ss} are all at the same expressiveness level.

Theorem 14. O_{ss} can strongly implement weakly fair split with signal, and hence O_{ws}.

Proof : The same figure 4.6 illustrates how we can strongly implement weakly fair split with signal from strongly fair split. The determinate process P is different : it uses the sequence at port p to decide whether either of the output streams should be finite, and if a stream should be finite, to decide which elements of the input stream should comprise the finite output stream. ∎

Theorem 15. Weakly fair split can strongly implement unfair split.

Proof : Let Σ^+ be the infinite sequence $1^\wedge 2^\wedge 3\ldots$ of all positive integers. When this is the input to a weakly fair split, the first value on the first output channel could be any positive integer. A determinate process can obtain arbitrary positive integers from a finite number of weakly fair splits in this way. Then the determinate process can use these arbitrary integers to decide whether one of the output streams of the unfair split process being implemented should be empty, in which case the entire input stream should comprise the other output stream. Otherwise the process just uses a weakly fair split process to decide the distribution of input stream elements. ∎

Theorem 16. O_{ws} can strongly implement O_{us}.

Proof : A similar proof as the above. ∎

Finally, we now describe the connection between the merge hierarchy and the split hierarchy. We will show that infinity-fair merge and O_{ss} are at the same expressiveness level.

Lemma 19. O_{ss} and infinity-fair merge can strongly implement each other.

Proof : Figure 4.7 shows how O_{ss} can strongly implement infinity-fair merge. The determinate process P reads the oracle output. Every time it reads a 1, it reads a value, if it exists, from its first input port and outputs it. Every time it reads a 2, it reads a value, if it exists, from its second input port and outputs it. If a value is not available to be read at the appropriate input port, then P waits until such a value becomes available.

If an infinity-fair merge process is given the streams 1^∞ and 2^∞ as input, then this implements O_{ss}. ∎

We can also show that *unfair merge*, that we have not yet placed in the hierarchies, is at the same expressiveness level as unfair split with signal.

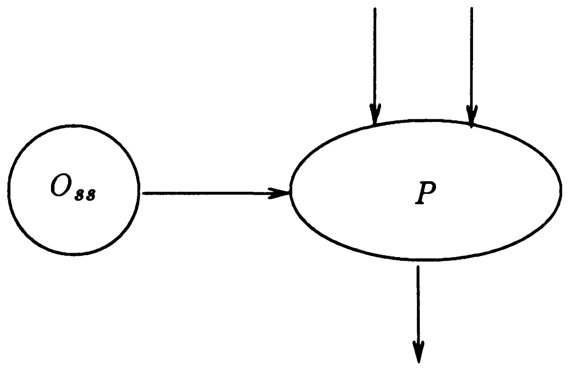

Figure 4.7: Infinity-fair merge from O_{ss}

Lemma 20. Unfair split with signal and unfair merge are at the same expressiveness level.

Proof : If an unfair split with signal process is given an infinite input stream, then the output at the signal port p is any stream of 1's and 2's. Hence this implements O_{us}. Then a determinate process with two input ports, besides p, and one output port, can use the stream at p to decide which input port to read from next, and then wait at that input port until there is a value available to be read.

We can give the input streams 1^∞ and 2^∞ to an unfair merge process, and this implements the oracle O_{us}, and hence can implement unfair split with signal.

∎

4.3.3 The collapse of a small hierarchy of weakly fair splitters

We required the weakly fair splitter process to guarantee to output at least one value on each of its output ports when the input is infinite. In this subsection, we answer the question : Why can't a hierarchy of weakly fair splitters be defined as follows — a weakly j-fair splitter guarantees at least j values on each of its output ports, when the input is infinite. Note that a weakly 1-fair splitter is exactly the weakly fair split process.

The answer to the question is that all weakly j-fair splitters, $j \geq 1$, can strongly implement each other. We now prove this claim.

Theorem 17. A weakly 1-fair splitter is at the same expressiveness level as a weakly j-fair splitter, for all positive integers j.

Proof : Let Σ^+ be the infinite sequence $1^\wedge 2^\wedge 3\ldots$ of all positive integers, and let this be the input to a weakly j-fair splitter. A couple of determinate processes can each read one element from the two output ports of the weakly j-fair splitter. At least one value is ensured on each output port. The processes then ignore the next $j-1$ values on each output, and use the rest of the output streams of the weakly j-fair splitters to simulate a weakly fair split process.

For the other way, we use enough weakly fair split processes (at most $2j$ needed) with input streams Σ^+ to choose the positions in the input sequence of $j-1$ elements on either output. This is possible because from each weakly fair split process, we can obtain an unbounded positive integer. Then we use a weakly fair split process to obtain the rest of the output. ∎

4.3.4 Considering a particular subset of the trace set – the scheduled trace set

In traces, if an event is "enabled" at some point in the trace, it need not occur in the trace within a bounded number of steps from that point. Failure of this property leads us to define "scheduled" trace, and work with scheduled trace sets of processes, instead of the trace sets. This will not be a very drastic decision, because, as we will show for the processes we now consider, for any trace t, there is a scheduled trace with the same set of events as t – in particular, with the same input and output as t. Intuitively, a trace is scheduled if every process in the network is guaranteed to "make progress" in a bounded number of steps.

Definition 42. A set Σ of traces of a network N of processes is said to be **prefix-limit-closed** if for an arbitrary infinite sequence t, if every finite prefix of t is a prefix of some trace in Σ, then t is a trace of N.

We cannot expect the set of all traces of a process to be prefix-limit-closed, because even if we take the simple buffer process (example 1), there are traces in which arbitrarily many values arrive at the input port before any value is output. So every prefix of an infinite sequence of input events can be extended to a trace, but the infinite sequence of input events is certainly not a trace.

The problem is that asynchrony allows an arbitrary amount of input to arrive before output is produced. We therefore define the notion of a scheduled trace, in which enabled output events happen within a fixed bounded number of steps. The set of scheduled traces of sequential processes, unfair split processes and unfair split with signal processes turn out to be prefix-limit-closed.

Definition 43. A **scheduled trace** t of a network N is a network trace such that for every subnetwork M of processes, if t_M is the projection of t onto M, and M has m ports, then for every $i > 0$, if $t_M[i]$ is an output event and $t_M[(i-m)..(i-1)]$ does not contain any output events at the same port, then $t_M[1..(i-m-1)]^\wedge t_M[i]^\wedge t_M[(i-m)..(i-1)]^\wedge t_M[(i+1)...]$ is not a trace of M.

The idea here is that every enabled event that continues to remain enabled must happen in the trace within the next m events of the point where it is first enabled. So if an output event is enabled at some point in the trace, it could not have been enabled m events ago. The set of scheduled traces of a network will be called its **scheduled trace set**.

We now need to ensure that for every trace with some particular sequences of input events and sequences of output events, there is a scheduled trace with the same input and output, so that we can use the scheduled trace set to represent the behaviours of a process or network. For these purposes, we will define a "scheduling operation" on traces that yield scheduled traces.

We will use $\langle p, n\rangle$ to refer to the nth event on port p in a trace. We will use the notation $t[i]$ for the ith element of a sequence t, $t[1..m]$ for the prefix of t consisting of the first m events of t, and $t[m...]$ for the suffix of t starting from $t[m]$.

Definition 44. Suppose p is a port of a network N, and t is a trace of N. A pair $\langle p, n\rangle$ is said to **occur** at time i in trace t if $t[i]$ is the nth event on port p.

We also say that $\langle p, n\rangle$ occurs in t if it occurs at some time in t. Note that a pair is not the same as an event. A pair represents an event in a trace, and the same pair may represent different events in different traces.

We first describe a causality relation on events of a trace, that will represent the "causal" order between events in a trace, and that is well-founded, antisymmetric and transitive. We then prove that every linearization of this relation is a trace, and we will then take some particular linearizations to be scheduled traces. We first define a relation $\prec_1$, and obtain the desired relation $\prec$ as its reflexive and transitive closure.

Definition 45. For a trace t of a network N, let T be the set of all traces of N containing exactly the events in t. Then $t[i] \prec_1 t[j]$ if $t[i]$ is an input event, $t[j]$ is an output event, the events $t[i], t[j]$ are represented by the pairs $\langle p, n\rangle$ and $\langle p', n'\rangle$ respectively, and either

(i) the event corresponding to the pair $\langle p, n\rangle$ precedes the event corresponding to the pair $\langle p', n'\rangle$ in every trace in T, or

(ii) $t[i]$ is the mth input event in t of a US or USS process in N, and $t[j]$ is the mth output event in t (including the events at both the non-signal output ports) of that process.

Definition 46. $\prec = (\prec_1)^*$

Lemma 21. $\prec$ is well-founded, antisymmetric and transitive.

Proof : By the definition of $\prec_1$, $t[i] \prec_1 t[j]$ implies that $i < j$. Now if $t[i] \prec t[j]$, i.e. $t[i] = t[i_1] \prec_1 t[i_2] \prec_1 \ldots \prec_1 t[i_k] = t[j]$, then $i = i_1 < i_2 < \ldots < i_k = j$. Therefore $i \leq j$. By the antisymmetry and well-foundedness of $\leq$ on positive integers, it immediately follows that $\prec$ is well-founded and antisymmetric. Moreover, $\prec$ is clearly transitive, as it is the transitive closure of $\prec_1$. ∎

We shall sometimes denote the $\prec$ associated with trace t by $\prec_t$.

Lemma 22. If t is a trace of a sequential process P, then any linearization of $\prec_t$ is a trace of P.

Proof : Let t' be a linearization of $\prec_t$. For every pair (e_1, e_2) in the causal set of t, e_1 precedes e_2 in t'. Therefore, by Lemma 10, t' is a trace of P. ∎

We now describe the trace set of unfair split again. Let the unfair split process have an input channel i and two output channels o_1, o_2. Then its trace set consists of all sequences $t \in (\{i, o_1, o_2\} \times V)^\infty$ such that $\Pi_i(t)$ can be broken up into two subsequences s_1, s_2 such that

(i) $\mathit{value}(s_1) = \mathit{value}(\Pi_{o_1}(t))$ and $\mathit{value}(s_2) = \mathit{value}(\Pi_{o_2}(t))$,

(ii) for every prefix t' of t, $\mathit{value}(\Pi_{o_1}(t'))$ is a prefix of the value sequence in the prefix of s_1 in t' and $\mathit{value}(\Pi_{o_2}(t'))$ is a prefix of the value sequence in the prefix of s_2 in t'.

Lemma 23. If t is a trace of any split process, then any linearization of $\prec_t$ is a trace of P.

Proof : This is clear by the definition of the trace sets of split processes. ∎

Throughout this section, whenever we refer to networks, we will assume that they are obtained as compositions of compatible sets of processes without output hiding.

Lemma 24. If t is a trace of a finite network of sequential processes and split processes, then any linearization of $\prec_t$ is a trace of the network.

Proof : Let t' be a linearization of $\prec_t$. Let t_P and t'_P be the projections of t and t' respectively onto a process P. Then, t_P is a trace of P, because t is a network trace. t'_P has the same set of events as t_P. We will now show that t'_P is a trace of P, and hence conclude, by Theorem 2, that t' is a network trace. We first prove that t'_P is a linearization of $\prec_{t_P}$. If T_P is the set of all traces of the process P with the same set of input and output events as t_P, and T is the set of all traces of the network with the same input and output events as t, then the projections of traces in T onto process P is a subset of T_P. Since T_P determines $\prec_{t_P}$, a subset of T_P determines a relation $\prec'$ that contains $\prec_{t_P}$. Since $\prec'$ is the restriction of $\prec_t$ to the events of t_P, and t'_P is a linearization of $\prec'$, t'_P is also a linearization of $\prec_{t_P}$, and hence is a trace of P by Lemmas 22,23. Therefore the projection of t' onto every process in the network is a trace of the process, and therefore, t' is a network trace. ∎

To schedule a trace, we dovetail among the sequences of input events and output events at the various ports of the network, making sure at each step, that when an event is considered to be the next event in the new trace, then all its predecessors in the partial order have already been considered.

Definition 47. A **port order** of a finite network with m ports is defined to be a total ordering $p_0, p_1, p_2, \dots p_{m-1}$ of the m ports of the network.

Definition 48. For any finite network with m ports, and any port order $\mathbf{o} = p_0, p_1, p_2 \dots p_{m-1}$, the **scheduling operation** $S_{\mathbf{o}}$ is defined as follows : Then $S_{\mathbf{o}}(t)$ is a total ordering of the events of t, such that the following holds. $S_{\mathbf{o}}(t)[1]$ is an event on the first of the ports in $\mathbf{o}$ such that it has no predecessor in $\prec_t$. The existence of such an event is guaranteed by well-foundedness of $\prec_t$. If $S_{\mathbf{o}}(t)[i]$ is an event corresponding to port p_k, then $S_{\mathbf{o}}(t)[i+1]$ is an event on the first of the ports $p_{k+1 \mathbf{\,mod\,} m}, \dots p_k$ such that each of its $\prec_t$-predecessors is in $S_{\mathbf{o}}(t)[1..i]$.

Lemma 25. If t is a trace of a finite network N of sequential processes and split processes, then $S_{\mathbf{o}}(t)$, for any port order $\mathbf{o}$, is a scheduled trace of N.

Proof : Let $t' = S_{\mathbf{o}}(t)$. Then t' is a linearization of $\prec_t$, and hence, by Lemma 24, it is a network trace. Suppose t'_M is the projection of t' onto a subnetwork M with m ports, and suppose $t'_M[i]$ is an output event of M. $t'_M[1..(i-m-1)]^\wedge t'_M[i]^\wedge t'_M[(i-m)..(i-1)]^\wedge t'_M[(i+1)...]$ cannot be a trace of M, because otherwise the behaviour of the scheduling operation would be violated. Hence t' is a scheduled trace. ∎

Lemma 26. The projection of a scheduled trace t of a network N onto a process P of the network is a scheduled trace of P.

Proof : Clear.

The point of the scheduling operation has been to ensure that, in our arguments, we can represent processes and networks by sets of traces that are prefix-limit-closed. We would definitely like the scheduled trace sets of sequential processes to be prefix-limit-closed, and we will prove this fact.

Lemma 27. The scheduled trace set of a determinate process is prefix-limit-closed.

Proof : Let A be a determinate process, and let t be an infinite sequence of input and output events of A, such that every finite prefix of t can be extended to a scheduled trace of A. We need to prove that t is then a trace of A. We will construct a completed computation sequence γ of A such that $tr(\gamma) = t$, and hence conclude that t is a trace of A. We will construct γ in stages. Having constructed the finite computation sequence γ_i at the end of the ith stage, we will describe how to properly extend γ_i to obtain a finite computation sequence γ_{i+1} at the end of the $(i+1)$th stage. If $tr(\gamma_i) = t[1..j]$, and $t[j+1]$ is an input event, then we define γ_{i+1} to be γ_i followed by the input event transition corresponding to $t[j+1]$ followed by the output event of the buffer automaton corresponding to $t[j+1]$. If $t[j+1]$ is an output event, then let C be the central automaton of the process A, and B be the buffer automaton whose output event $t[j+1]$ is. Let e represent the input event occurrence of B that corresponds to the output event $t[j+1]$. If there is a transition for e in γ_i, then we define γ_{i+1} to be γ_i followed by an output event transition corresponding to $t[j+1]$, which is enabled, because e has already occurred. If there is no transition for e in γ_i, then by the determinacy of C, by lemma 4, we can extend $\Pi_C(\gamma_i)$ to a computation sequence that contains the event e. Let δ be the extension of $\Pi_C(\gamma_i)$ ending in the transition corresponding to e in this computation sequence. We then define γ_{i+1} to be γ_i followed by all the transitions in δ, followed by the output event corresponding to $t[j+1]$.

We now claim that the computation sequence γ' obtained in this way can be $\precsim$-extended to a completed computation sequence γ without adding any output events. If not, then for some $\precsim$-extension γ'' of γ', there is an output event that is enabled at every state of some suffix of γ'', and commutes with all the events in that suffix. Suppose this output event is enabled at the end of $\gamma''[1..k]$ and at every state thereafter. But since t has infinitely many events, and every prefix of t is a scheduled trace, the output event must occur in γ'' in finite time by the definition of a scheduled trace. We thus get a contradiction, thus proving our claim. ∎

Definition 49. An infinite chain $\beta_1 \sqsubseteq \beta_2 \sqsubseteq \ldots$ of sequences is said to be **eventually increasing** if it is non-decreasing and there is no i, such that $\forall j \geq i, \beta_j = \beta_i$.

Lemma 28. For any finite network N of sequential processes, unfair split processes and unfair split with signal processes, the scheduled trace set is prefix-limit-closed.

Proof : Suppose N has m ports, and t is an infinite sequence that is not a trace of N but every prefix of t is a prefix of a scheduled trace of N. Then the projection t_P of t onto some process P is not a trace of P. Let t_i be the projection of $t[1..i]$ onto P. Then each t_i is a prefix of a scheduled trace of P by the previous lemma, because t_i is a prefix of the projection of some scheduled trace onto P.

Case 1 : The t_i's form an eventually increasing sequence. Then, by prefix-limit-closure of the scheduled trace set of P, t_P is a trace of P, contradicting assumption.

Case 2 : for some i, for all $j \geq i$, $t_j = t_i = t_P$. Let t' be a scheduled trace of the network, such that $t[1..(i+m)]$ is a prefix of t'. Therefore the projection t'_P of t' onto P has $t_{i+m} = t_i$ as a prefix. Since t_i is not a trace of P and t'_P is a trace of P, t'_P must contain an output event e such that $t_i{}^\wedge e$ is a prefix of a trace of P, violating the definition of a scheduled trace for t'.

Thus t is a network trace, and hence the scheduled trace set is prefix-limit-closed. ∎

4.3.5 Nonexpressibility of Fair Split

Our main theorem here states that there is no network consisting of sequential processes and WSS process that weakly implements SS. We consider a network that supposedly weakly implements SS. We express the set of scheduled traces of the network as the union of a countable family of trace sets. We show that the traces in each member of the family is prefix-limit-closed. We build a tree representation of the traces in each family. We quotient the tree by contracting all edges that do not correspond to events at output ports. Each quotiented tree is finitely branching. Finally we diagonalize to exhibit a possible output sequence of strongly fair split that is not produced by any trace of the network.

First we establish the required definitions and lemmas.

Definition 50. If S is a set of traces of a network, then $T(S)$ is the tree whose nodes are finite prefixes of traces in S and such that prefix s' is a child of prefix s iff $s' = s^\wedge e$ for some event e. We assume that each edge is labeled with the last event of the prefix associated with the descendant node.

We note that the set of sequences corresponding to the paths in the tree from the root is not necessarily equal to the set of traces S. All that can be said is that, for every sequence corresponding to a path in the tree from the root, every prefix of this sequence is a prefix of a trace in S.

Definition 51. A process P is said to be **finitely branching** if for any finite sequence of events t that is a prefix of a trace of P and not itself a trace, there are only finitely many output events e such that $t^\wedge e$ is a prefix of a trace of P.

Note that there are clearly infinitely many input events that can be the next event after the sequence of events t. The definition restricts the number of output events that can be the next event.

We note that all sequential and split processes are finitely branching.

Lemma 29. If S is the trace set of a network N of finitely branching processes for a fixed input, then $T(S)$ is finitely branching.

Proof : Suppose s is a prefix of a trace in S. Then the next event of the trace could be an input event on any of the finitely many input ports, or an output event for some process in the network. There are finitely many of these too, because there are finitely many processes, and each process is finitely branching. Therefore, s has only finitely many children in $T(S)$. So every vertex in the tree has finitely many children, i.e. the tree is finitely branching. ∎

The following theorem follows easily from Kœnig's Lemma.

Theorem 18. No network of sequential processes and unfair split with signal processes can weakly implement weakly fair split.

Proof : Suppose there is a finite network N of sequential processes and unfair split with signal processes that weakly implements WS. Then N has one input port and two output ports, corresponding to those for a WS process. We fix the input stream to be $1^\wedge 2^\wedge 3\ldots$. Then the first output port c, say, of N is guaranteed to have at least one value output on it. This value can be any positive integer.

Let S consist of the scheduled traces of N for the input $1^\wedge 2^\wedge 3\ldots$. Every trace has a scheduling with respect to any port order. Therefore, for every possible output sequence of the network onto port c, there is a scheduled trace in S that outputs that sequence on c. We consider the tree $T(S)$, and we prune every path at the first output event on that path on port c. By lemma 28, every path in the tree is a network trace, and so has an output event on c. Moreover, by lemma 29, $T(S)$ is finitely branching. Therefore the pruned tree is a finitely branching tree with no infinite paths. By Kœnig's lemma, the tree is finite. So there are finitely many leaves, i.e. finitely many possibilities for the first output event on port c. This means that the network does not weakly implement weakly fair split – contradiction. ∎

The next theorem shows that strongly fair split cannot be weakly implemented by a weakly fair split even with a signal. The proof requires a diagonalization argument – cardinality or Kœnig's lemma arguments by themselves do not seem sufficient.

Theorem 19. No network of sequential processes and WSS processes can weakly implement SS.

In order to prove this theorem we need several definitions and lemmas.

We make explicit the fact that WS embodies a countable choice. First we give a definition for events that lead upto this countable choice.

Definition 52. Let N be any network and let t be any sequence of events in that network. i is said to be a **split initiation time** for t if there is a WS or WSS process P in N and a non-signal output port c of P such that either

1. $t[i]$ is the first output event on c in t, or,
2. $t[i]$ is an input event of P and there is no output event on port c in $t[1..i]$.

Such a $t[i]$ is called an **initiation event.**

Definition 53. Let s be a finite sequence of events of a network N. An **initiation-free extension** of s is a sequence t such that s is a prefix of t and such that all initiation events in t occur in the prefix s.

The next lemma follows from the definition of weakly fair split.

Lemma 30. For any trace t of a network, there exists some finite prefix s of t such that t is an initiation-free extension of s.

Proof : We show that each WS or WSS process in the network has a last split initiation time. There are two cases.

Let t' be the projection of t onto the ports of a WS or WSS process.

(i) there are output events on both the output ports (both the non-signal output ports in the case of a WSS process). In that case, if $t[m]$ is the first output event on the first output port, and $t[m']$ is the first output event on the second output port, then the last split initiation time for this process is $\max(m, m')$.

(ii) there are no output events on one of the output channels. Then there must be only finitely many input events in t'. Let the last of these be $t[m]$. Also let the first output event on the other output channel be $t[m']$. Then the last split initiation time for this process is $\max(m, m')$.

Since there are finitely many WS or WSS processes, if i is the maximum of their last split initiation times, then t is an initiation-free extension of $t[1..i]$. ∎

Definition 54. Let N be a network and s a finite sequence. We define C_s to be the set of all scheduled traces of the network, that are initiation-free extensions of s.

Lemma 31. For any finite network N, there are countably many sets of the form C_s, for s any finite sequence of events of the network.

Proof : Each event is a pair, and there are countably many of these, assuming that there are countably many values that may be transmitted at a port. Therefore, there are countably many finite sequences of events, and so there are countably many sets of the form C_s. ∎

Note that even though every member of C_s is an initiation-free extension of s, it is not obvious that every path in $T(C_s)$ is a member of C_s. So the following lemma is required.

Lemma 32. For any finite network N, and any sequence of events s of the network, any path in $T(C_s)$ is an initiation-free extension of s.

Proof : Let t be a path in $T(C_s)$. Since every trace in C_s starts with the prefix s, t must also start with the prefix s. It follows from the definition of an initiation time that if i is an initiation time in t, and a trace t' is identical to t up to and including the ith event, then the ith event is also an initiation event in the trace t'. Hence this ith event is in s. Since every prefix of t is a prefix of some trace t' in C_s, t cannot contain any initiation-events other than those in the prefix s. ∎

Lemma 33. For any network N of sequential processes, weakly fair split processes, and weakly fair split with signal processes, and for any finite sequence of events s, C_s is prefix-limit-closed.

Proof : Let t be a sequence such that every prefix of t is a prefix of some member of C_s. By lemma 32, t is an initiation-free extension of s. We must show that t is a network trace.

Suppose t is not a network trace. Then the projection t_P of t onto some process P of the network is not a trace of P. We will proceed as in lemma 28. Let t_i be the projection of $t[1..i]$ onto process P. Then each t_i is a prefix of a scheduled trace of P.

Case 1 : The t_i's form an eventually increasing sequence. If P is not a weak split or a weak signal split process, then, by prefix-limit-closure of the scheduled trace set of P, t_P is a trace of P, contradicting the assumption.

If P is a WS or WSS process, then t_P must be an infinite sequence, containing infinitely many input events and infinitely many output events. Since this is not a trace of P, it must be the case that all the output events are on the same output port of P, contradicting the requirement that there be output events on both the output ports if the input is infinite. This means that the projection of t onto process P has infinitely many input events for P, and all of these are initiation events for t. This contradicts the fact that s is finite, and t is an initiation-free extension of s.

Case 2 : for some i, for all $j \geq i$, $t_j = t_i = t_P$.Let t' be a scheduled trace of the network, such that $t[1..(i+m)]$ is a prefix of t'. Therefore the projection t'_P of t' onto P has $t_{i+m} = t_i$ as a prefix. Since t_i is not a trace of P and t'_P is a trace of P, t'_P must contain an output event e such that $t_i{}^\wedge e$ is a prefix of a trace of P, violating the definition of a scheduled trace for t'.

Thus t is a network trace and so, C_s is prefix-limit-closed. ∎

Definition 55. The **complement** of an increasing infinite sequence s of positive integers is defined to be the increasing sequence of all those positive integers that are not in the sequence s.

We now define a quotienting operation on trees that conceals events that are not output events at a fixed port.

Definition 56. Let T be a tree in which the edges are labeled with events from a network N. Let c be a port of N. We define the **quotient** of T with respect to c, written T/c, to be the tree obtained by contracting every edge in T that is not an output event on c.

Proof of Theorem 2: Suppose there is a network N of sequential processes and WSS processes that weakly implements strongly fair split. Then the network has one input port and two output ports. Let one of the output ports be c. We fix the input stream to be $1^\wedge 2^\wedge 3$.... Then, at c, N can output any increasing infinite sequence of positive integers, whose complement is also an increasing infinite sequence of positive integers.

Since the scheduling operation can be applied to any trace to obtain a scheduled trace, every possible output sequence of N onto port c is output by some scheduled trace.

Let S be the scheduled trace set of N for the fixed input. We divide S into subclasses C_s, as defined earlier. We then obtain a countable family of trees $T(C_s)/c$.

We claim that each tree $T(C_s)/c$ is finitely branching. Every path in $T(C_s)$ has infinitely many output events at port c, since every path in the tree is a network trace by lemma 33. Consider any node n of the tree such that the prefix associated with that node ends in an output event on c. These are exactly the nodes that remain after the quotienting. We prune every path from n at the first output event on c on the path. Since the tree $T(C_s)$ is finitely branching the pruned tree below n is also finitely branching and has no infinite paths. By Kœnig's lemma, the tree is finite. So there are finitely many leaves. Thus in the quotiented tree, n has finitely many children corresponding to the finitely many leaves of the above pruned tree.

We name the quotiented trees $T(C_s)/c$ by $T_1, T_2, \ldots$. Each path in any of these trees must correspond to an infinite increasing sequence of positive integers. To obtain a contradiction, we construct, by diagonalization, an infinite increasing sequence of positive integers with infinite complement, that will be in none of these trees. Since every tree, T_i, is finitely branching, every level of each T_i has finitely many nodes. Hence, there is a maximum positive integer that occurs at that level. Let this maximum positive integer for the jth level in the ith tree be called $M_{i,j}$. We define $s[1]$, the first element of the sequence being constructed, to be any positive integer greater than $M_{1,1}$, say $M_{1,1} + 1$. Having fixed the elements $s[1], s[2], \cdots, s[i-1]$, we define $s[i]$ to be any positive integer greater than $\max\{M_{i,i}, s[i-1]+1\}$. This is certainly an infinite increasing sequence. Moreover, between any two consecutive elements $s[i-1]$ and $s[i]$ of the sequence, there is at least one positive integer not in the sequence, namely $s[i-1]+1$. So the sequence has an infinite complement. But this sequence is not in any of the trees $T_1, T_2, \cdots$. This is because, for any i, the ith element of the sequence is greater than $M_{i,i}$, and this is the greatest integer at the ith level of T_i.

This means that there is an infinite increasing sequence of positive integers with infinite complement, that is not a possible output sequence at c. Hence the network could not have weakly implemented strongly fair split. ∎

4.3.6 Nonexpressibility of Signaling

In this section we explore the nonexpressibility arising from the sequentiality of the individual processes. Understanding sequentiality is a fundamental concern in the semantics of modern programming languages [Cur86,Plo77]. Our results in this section may be viewed as a first step towards understanding how sequentiality interacts with indeterminacy. The main theorem states that one cannot obtain a split with a signal from an ordinary split. The point is that the signal port is guaranteed to have as many values output on it as there are inputs. Unfair split has no output ports on which a stipulated number of values are guaranteed to appear. The only processes for which one could guarantee that a certain number of values would be output at a particular output port is a sequential process. In this case, however, the output values are determined by the input values. We show that this argument extends to networks composed of split processes and sequential processes. It turns out that the theorem holds for weakly fair split as well but not for strongly fair split. Thus the result is quite delicate and depends on the level of fairness we consider.

Definition 57. Suppose c is a port of a network N. Let R be a subset of the trace set of N. We say that a pair $\langle c, n\rangle$ is **guaranteed in** R if it occurs in every trace in R.

Definition 58. Suppose c is a port of a network N. Let R be a subset of the traces of N. We say that a pair $\langle c, n\rangle$ is **determined in** R if

$$\forall t_1, t_2 \in R.\ \langle c, n\rangle \text{ occurs at } i \text{ in } t_1 \text{ and at } j \text{ in } t_2$$

$$\Rightarrow t_1[i] = t_2[j].$$

The following is the central lemma of this section.

Lemma 34. For any network N of sequential processes and unfair split processes, if R is the set of all network traces with a particular input I, then every pair $\langle c, n\rangle$ that is guaranteed in R is determined in R.

Proof: The proof proceeds by induction on the earliest occurrence of a guaranteed pair. Suppose $\langle c, n\rangle$ occurs at time 1 in a trace t. Then clearly $n = 1$. Also c has to be either the output port of a sequential process or an input port of the network. In the first case it is clearly determined by determinacy of the sequential process, and in the second case, it is determined since we are considering a fixed input.

Suppose the guaranteed pair $\langle c, n\rangle$ has an earliest occurrence time equal to k in R. Suppose that the lemma holds for all guaranteed pairs that have an earliest occurrence time less than k in R. Suppose that this pair is not determined in R. Then there are two traces g and h differing at the pair $\langle c, n\rangle$. Since they differ, c cannot be an input port of the network. Because the pair $\langle c, n\rangle$ is guaranteed, c cannot be an output port of an unfair split process. Thus c must be the output port of a sequential process A. Without loss of generality, we can assume g to be the trace in which $\langle c, n\rangle$ occurs at time k. Let the sequence $\Pi_A(g)$ be s and the sequence $\Pi_A(h)$ be s'. Let $\langle c, n\rangle$ occur at times i and j in s and s' respectively. Let $\Pi_I^g(s)$ be the guaranteed input in s. Since every event in $\Pi_I^g(s)$ has a earliest occurrence time less than k, they are all determined. Therefore, if $\Pi_I^g(s)$ can produce the output event $\langle c, n\rangle$, then it must be determined too, contradicting our assumption that the pair $\langle c, n\rangle$ is not determined. So $\Pi_I^g(s)$ cannot produce the output event $\langle c, n\rangle$. By sequentiality, there is an input port of A that must get extended for the output at c to get extended. This means that there is a guaranteed input event in s other than those in $\Pi_I^g(s)$. But $\Pi_I^g(s)$ contains all the guaranteed input events in s, giving us a contradiction. Therefore $\langle c, n\rangle$ must be determined. ∎

Theorem 20. No finite network of sequential processes and unfair split processes can weakly implement unfair split with signal.

Proof : Suppose there is a finite network N showing this implementation. Let c be the signal output port in this implementation. Let the input stream to the network be a single element, and suppose R is the set of network traces with this particular input. Then at least one event is guaranteed at port c in every trace in R. Moreover, it is the case that this first event at port c could be $\langle c, 0\rangle$ or $\langle c, 1\rangle$. This contradicts the earlier lemma. ∎

The following theorem is the extension to the case where we allow weakly fair split instead of unfair split.

Theorem 21. No finite network of sequential processes and weakly fair split processes can weakly implement unfair split with signal.

In order to prove this theorem, we need several definitions and lemmas.

Definition 59. Let N be a finite network and s a finite sequence of events. We define $C'_{s,I}$ to be the set of all traces of the network for a particular input I, that are initiation-free extensions of s.

Lemma 35. For any network N of sequential processes and weakly fair split processes, every pair $\langle c, n \rangle$ that is guaranteed in $C'_{s,I}$ is determined in $C'_{s,I}$.

Proof : The proof proceeds exactly as in lemma 34, except for the following case. $\langle c, n \rangle$ has an earliest occurrence time equal to k in $C'_{s,I}$, and all pairs with earliest occurrence times less than k are guaranteed by the induction hypothesis. Suppose that this pair is not determined in $C'_{s,I}$. Then there are two traces g and h differing at this pair. We consider the case where c is an output port of a weakly fair split process. In that case, $n = 1$, because only one event is guaranteed at an output port of a weakly fair split process. Therefore this is an initiation event, and so must be in s. Hence g and h cannot disagree on $\langle c, n \rangle$ because both g and h have the same prefix s, and this contradicts the supposition that the pair is not determined.

The rest of the cases are exactly as in lemma 34. ∎

Proof of theorem 21 : Suppose there is a finite network that is supposed to implement WSS. Let c be the signal output port in N. Let the input stream to N be some infinite stream I. This guarantees that the output stream on c is infinite for every network trace.

Every network trace is in some class $C'_{s,I}$, as in lemma 30. Moreover, every trace in $C'_{s,I}$ has the same output at port c. This is because, since the input I is infinite, there are infinitely many events guaranteed at port c, and by Lemma 35, they are all determined.

As in Lemma 31, there are countably many such classes $C'_{s,I}$, and so for the input I, there are at most countably many different outputs at port c. But, by the definition of an unfair split process with signal, for an infinite input, there are uncountably many output stream possibilities for the signal output port. This means that the network N does not weakly implement an unfair split process with signal. ∎

In this chapter, we described our main results. We looked at hierarchies of merge and split primitives and compared their expressiveness in the context of static dataflow.

Chapter 5

Kahn's Principle for recursive determinate networks

So far, we have been considering networks that do not "evolve" – a network is made up of a fixed set of automata, interconnected in a fixed manner. The set of automata and the manner of interconnection do not change in a particular network. This is called *static dataflow* in contrast to *dynamic dataflow* in which the number of automata and the manner of interconnection can change during computation. In this chapter, we consider the semantics of recursive definitions of networks, by which we mean a definition of a network in terms of itself. We will formalize the notion of "finite unwindings" of the recursive definitions in an operational semantics, and then prove Kahn's principle by showing that functional semantics exactly matches the operational semantics. This work is based on work by Lynch and Stark [LS89], who proved Kahn's principle for static determinate networks. We would like to emphasize that Kahn's principle has been proved earlier by a number of researchers using a number of different operational semantics. Faustini [Fau82] has presented a proof of Kahn's principle in his thesis.

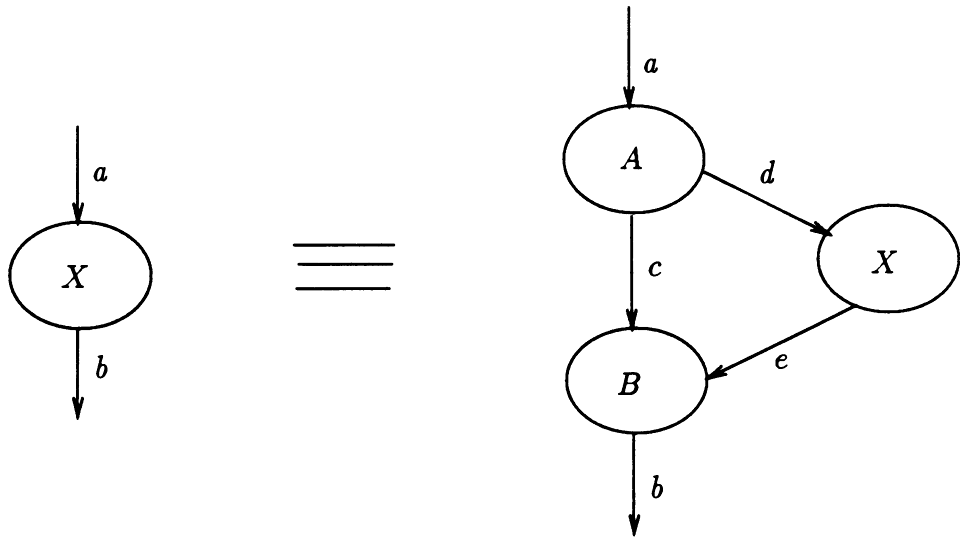

Figure 5.1: A recursive definition

In [Bro87], Broy describes a denotational semantics for networks with recursive definitions, but he does not compare it to an operational semantics. In [dB85], de Bruin and Böhm give a language for defining processes as well as dynamic networks, and give a denotational semantics for that language. They do not consider an operational semantics, however. On the other hand, we do not give a language for defining processes, and we take processes to be the building blocks of networks. We consider an operational semantics and show its correspondence to Kahn's functional semantics. We present our proof because we feel, as do Lynch and Stark [LS89], that the port automata semantics is a convenient operational semantics for dataflow networks. We will deal with determinate automata only, because Kahn's principle and functional semantics do not apply to indeterminate automata. Henceforth, in this chapter, every automaton that we consider will be determinate.

We will first give a simple example of a recursive definition. Let A and B be some fixed automata, and X be defined as in figure 5.1. X is an automaton with one input port a and one output port b. It can "expand" or "unwind" to the network shown in the figure, and then automata A, B can execute enabled events. The network has a copy of X, so it again has the capability to "expand." Notice that the names of the input port and the output port in the network are different from those of the original X. This is because the copy of X is really a copy, and the names a, b are bound to the input and output port of the network respectively. We also need to describe the events that cause the expansion of networks. We will therefore enhance the definition of port automata to include the notion of *expansion events.*

Given a set of input ports P^{in} and a set of output ports P^{out}, we define the "least" automaton having these ports as follows :

Definition 60. If P^{in} and P^{out} are disjoint sets of port names, then $\perp_{P^{in},P^{out}}$ is the automaton with

(i) input ports P^{in} and output ports P^{out},

(ii) states : all functions from P^{in} to V^*,

(iii) transitions : for every $p \in P^{in}$ and $v \in V$, $f \xrightarrow{(p,v)} g$ iff $g(p) = f(p)^\wedge v$ and $g(p') = f(p')$ if $p' \neq p$.

Briefly, for this automaton, the only possible events are input events caused by values arriving at its input ports from its environment. It has no internal or output events enabled at any state. We will use the notation $\perp$, when the subscripts are clear from the context.

Let $X = \mathcal{N}(X)$ be a recursive definition. Let P^{in} be the input ports of X, and P^{out} be the output ports of X. Then the automaton that should represent

X should be able to behave like the corresponding least automaton $\perp_{p_{in}, p_{out}}$. It should have all the transitions and states of $\perp_{p_{in}, p_{out}}$. Moreover, its description should also include an *expansion event* that is enabled at any state, and commutes with every other event. Once the expansion event is executed, it should now be able to behave like the expanded network $\mathcal{N}(\perp)$, which is the network $\mathcal{N}(X)$, with $\perp$ substituted for X. More expansion events should also now be enabled corresponding to the copies of X in $\mathcal{N}(X)$. We recall that the state of a network of automata is a tuple of states, one for each automaton in the network. Therefore an expansion event should cause a transition between a state of a network N and a state of the network resulting from N by the expansion event. The first expansion event should cause a transition from any state of $\perp$ to an appropriate state of $\mathcal{N}(\perp)$.

In our example, an expansion event causes a transition from state f of $\perp$ to a state $(q_A, q_B^\iota, q_\perp^\iota)$, where q_A is the state resulting from applying all the events in f to the initial state of A – this state is unique by commutativity, q_B^ι is the initial state of B, and $q_\perp^\iota$ is the initial state of $\perp$ in $N(\perp)$.

Since each expansion of X causes more copies of X to come into existence, and these copies can themselves expand causing still more copies of X to come into existence, the automaton for X should be able to behave like the "union" of all the automata resulting from any finite number of expansion events.

For ease of exposition, we will assume in what follows that $\mathcal{N}(X)$ has a single occurrence of X. So every time an expansion event is executed, there is a single new copy of X that may expand. The generalization to having finitely many copies of X in $\mathcal{N}(X)$ is tedious, but straightforward, involving the introduction of multiple new expansion events every time an expansion event is executed. Let $\mathcal{N}^2(\perp)$ represent the network with $\mathcal{N}(\perp)$ substituted for X in $\mathcal{N}(X)$. Let $\mathcal{N}^3(\perp)$ represent the network with $\mathcal{N}^2(\perp)$ substituted for X in $\mathcal{N}(X)$, and so on. Strictly speaking, the copy of X in $\mathcal{N}(X)$ has a different set of input and output ports than X, so we cannot simply carry out the substitutions for the copy of X. Let R^{in} be the input ports of X in $\mathcal{N}(X)$, and let R^{out} be the output ports of X in $\mathcal{N}(X)$. We assume the existence of a function τ that takes an automaton A with input ports P^{in} and output ports P^{out} and produces an automaton $\tau(A)$ with input ports R^{in} and output ports R^{out}, such that it bijectively maps the ports P^{in} to the ports R^{in}, the ports P^{out} to the ports R^{out}, the states of A to the states of $\tau(A)$, the internal events of A to the internal events of $\tau(A)$, such that $q \xrightarrow{e} q'$ iff $\tau(q) \xrightarrow{\tau(e)} \tau(q')$. In other words, τ just "renames" all the ports, states and events. We also require τ to satisfy the property that if an automaton A has more events, more states and more transitions, in addition to those of A, then $\tau(A')$ has more events, more states and more transitions than $\tau(A)$, in addition to those of $\tau(A)$. Such a *renaming* function can be easily defined by simply requiring it to tag events and states to obtain new events and states.

The operational semantics of the recursive definition is the network automaton resulting from the execution of every possible expansion event, such that the resulting network is "fully unwound" and does not have any copy of X in it. This definition is equivalent to the *union* of countably many automata, corresponding to the finite unwindings of the recursive definition, together with expansion events. $\perp$ is the automaton corresponding to no unwindings. $\mathcal{N}(\tau(\perp))$ is the automaton corresponding to one unwinding. $\mathcal{N}(\tau(\mathcal{N}(\tau(\perp))))$ is the automaton corresponding to two unwindings, and so forth. We will refer to these automata respectively as $\perp, \mathcal{N}(\perp), \mathcal{N}^2(\perp)$, and so on. We note that there is an injection from the states, events and transitions of $\mathcal{N}^i(\perp)$ to the states, events and transitions of $\mathcal{N}^{i+1}(\perp)$. We take the union of the states, events and transitions of all these automata, and then add expansion events $e_0, e_1, e_2, \ldots$, where the expansion event e_i causes the expansion from $\mathcal{N}^i(\perp)$ to $\mathcal{N}^{i+1}(\perp)$. The event e_i is enabled in every state of $\mathcal{N}^i(\perp)$, and the transitions corresponding to this event are defined as follows : let $(q, q', \ldots, f)$ be a state of $\mathcal{N}^i(\perp)$ with f being the state of the $\perp$ automaton that expands by e_i, and $q, q', \ldots$ being the states of the rest of the automata in $\mathcal{N}^i(\perp)$. The expansion causes the replacement of the $\perp$ automaton by a network of automata. Let $(r, r', \ldots)$ be the unique state of this network that it can reach by executing all the events in f, starting from the initial state. Then e_i causes a transition to the state $(q, q', \ldots, r, r', \ldots)$ of $\mathcal{N}^{i+1}(\perp)$. We will refer to this automaton as $\mathcal{O}(X)$. In $\mathcal{O}(X)$, everything that can be expanded has been expanded.

Let $\mathcal{N}(X)$ be the composition of a set of automata M_i and a copy R of X having input ports R^{in} and output ports R^{out}. Let P^{in} be the set of input ports of X, and hence of the network $\mathcal{N}(X)$. Let P^{out} be the set of output ports of X, and hence of $\mathcal{N}(X)$. Let P be the set of all the ports in $\mathcal{N}(X)$, which include the input and output ports of all the M_i's and R. Then the (functional) behaviour of the network can be described as a function from a history on the input ports H^{in} to a history on the set of all ports P. Let F_i be the function from histories over P^{in} to histories over P computed by the network $\mathcal{N}^i(\perp)$. We note that since $\mathcal{N}^{i+1}(\perp)$ has more states, events and transitions than $\mathcal{N}^i(\perp)$, every completed computation sequence of $\mathcal{N}^i(\perp)$ is a computation sequence of $\mathcal{N}^{i+1}(\perp)$, and hence can be $\underset{\sim}{\sqsubseteq}$-extended to a completed computation sequence. This means that $F_i \sqsubseteq F_{i+1}$. We will first prove that the function $Fun(\mathcal{O}(X))$ computed by $\mathcal{O}(X)$ from histories over P^{in} to histories over P is equal to the least upper bound of the F_i's.

Lemma 36. $Fun(\mathcal{O}(X)) = \sqcup F_i$.

Proof : Since the states, events and transitions of $\mathcal{N}^i(\perp)$ are also present in $\mathcal{O}(X)$, every completed computation sequence of $\mathcal{N}^i(\perp)$ is a computation sequence of $\mathcal{O}(X)$, and hence can be extended to a completed computation sequence of $\mathcal{O}(X)$. Therefore, each F_i is $\sqsubseteq Fun(\mathcal{O}(X))$. Therefore $\sqcup F_i \sqsubseteq Fun(\mathcal{O}(X))$.

We now prove that $Fun(\mathcal{O}(X)) \sqsubseteq \sqcup F_i$. For some input history H^{in} of $\mathcal{O}(X)$, let γ be a completed computation sequence. Let $\gamma[1..j]$ have the expansion events $e_0, \ldots e_{i-1}$. Then since expansion event e_k is enabled at any state in $\mathcal{N}^k(\perp)$, the sequence of expansion events $e_0, \ldots e_{i-1}$ is enabled at the initial state. Then we can define a sequence of moves to obtain a computation sequence $\gamma' \simeq \gamma[1..j]$, and γ' consists of the expansion events $e_0, \ldots e_{i-1}$, followed by a computation sequence of $\mathcal{N}^i(\perp)$. Let H_j be the history over ports P induced by $\gamma[1..j]$ or γ'. Then $H_j \sqsubseteq F_i(H^{in}) \sqsubseteq \sqcup F_i(H^{in})$. Then $Fun(\mathcal{O}(X))(H^{in}) = \sqcup H_j \sqsubseteq \sqcup F_i(H^{in})$. ∎

Kahn [Kah74] advocated finding the least solution of a certain set of equations. We can recast this, as in [LS89], as obtaining the least fixed point of a certain functional, as described below.

Suppose H^{in} is a history on the input ports P^{in}. Also suppose that the input-output behaviour of the automaton M_i is f_i, a function from histories on the set of input ports of M_i to histories on the set of output ports of M_i. Let P_i^{in} be the input ports of M_i, P_i^{out} be the output ports of M_i. We define a functional $\Phi : [Hist(P^{in}) \rightarrow Hist(P)] \rightarrow [Hist(P^{in}) \rightarrow Hist(P)]$ by

(1) $\Phi(f)(H^{in})|P^{in} = H^{in}$

(2) $\Phi(f)(H^{in})|P_i^{out} = f_i(f(H^{in})|P_i^{in})$,

(3) $\Phi(f)(H^{in})|R^{out} = \tau(f(\tau^{-1}(f(H^{in})|R^{in}))|P^{out})$.

Lemma 37. Φ is continuous, and hence has a least fixed point $\mu\Phi$.

Proof : Continuity of Φ follows easily from the continuity of the f_i's. Therefore Φ has a least fixed point. ∎

We will first prove a lemma about the functions computed by the finite unwindings. This will be used in the main theorem that follows.

Lemma 38. For $j > 1$, $\tau(F_{j-1}(\tau^{-1}(F_j(H^{in})|R^{in}))|P^{out}) = F_j(H^{in})|R^{out}$.

Proof : Let σ be a completed computation sequence of $\mathcal{N}^j(\perp)$ for the input given by H^{in}. Then, by the definition of completion, the projection γ' of γ onto the subnetwork $\tau(\mathcal{N}^{j-1})$ is a completed computation sequence. The input in γ' is given by $F_j(H^{in})|R^{in}$, and the output is given by $F_j(H^{in})|R^{out}$. Since $\mathcal{N}^{j-1}(\perp)$ computes the function F_{j-1}, using the bijection τ, $F_{j-1}(\tau^{-1}(F_j(H^{in})|R^{in}))|P^{out} = \tau^{-1}(F_j(H^{in})|R^{out})$, from which the lemma follows. ∎

Theorem 22. (Kahn Principle for Dynamic Networks) $Fun(\mathcal{O}(X)) = \mu\Phi$.

Proof : We first show that $Fun(\mathcal{O}(X))$ is a fixed point of Φ.

Case 1: $\Phi(Fun(\mathcal{O}(X))(H^{in})|P^{in} = H^{in} = Fun(\mathcal{O}(X))(H^{in})|P^{in}$.

Case 2: $\Phi(Fun(\mathcal{O}(X))(H^{in})|P_i^{out} = f_i(Fun(\mathcal{O}(X))(H^{in})|P_i^{in})$
$= f_i(\sqcup_j F_j(H^{in})|P_i^{in}) = \sqcup_j f_i(F_j(H^{in})|P_i^{in})$ by continuity of f_i
$= \sqcup_j F_j(H^{in})|P_i^{out} = (\sqcup_j F_j)(H^{in})|P_i^{out} = Fun(\mathcal{O}(X))(H^{in})|P_i^{out}$

Case 3: $\Phi(Fun(\mathcal{O}(X))(H^{in})|R^{out}$
$= \tau(Fun(\mathcal{O}(X))(\tau^{-1}(Fun(\mathcal{O}(X))(H^{in})|R^{in}))|P^{out})$
$= \tau(\sqcup_{j>0}F_j(\tau^{-1}(\sqcup_{k>0}F_k(H^{in})|R^{in}))|P^{out})$
$=$ (by continuity of τ, τ^{-1} and F_j) $\sqcup_{j>0,k>0}\, \tau(F_j(\tau^{-1}(F_k(H^{in})|R^{in}))|P^{out})$
$=$ (proved below) $\sqcup_{i>0}\, F_i(H^{in})|R^{out} = Fun(\mathcal{O}(X))(H^{in})|R^{out}$.

We have $F_1(H^{in})|R^{out} =\perp$, because R^{out} is the set of output ports of a $\perp$ process in $\mathcal{N}(\perp)$. Also, by above lemma, for $i > 1$, $F_i(H^{in})|R^{out}$
$= \tau(F_{i-1}(\tau^{-1}(F_i(H^{in})|R^{in}))|P^{out}) \sqsubseteq \sqcup_{j>0,k>0}\tau(F_j(\tau^{-1}(F_k(H^{in})|R^{in}))|P^{out})$.

Therefore, $\sqcup_{i>0}F_i(H^{in})|R^{out} \sqsubseteq \sqcup_{j>0,k>0}\tau(F_j(\tau^{-1}(F_k(H^{in})|R^{in}))|P^{out})$.

Now, if $j < k$, then $\tau(F_j(\tau^{-1}(F_k(H^{in})|R^{in}))|P^{out})$
$\sqsubseteq \tau(F_{k-1}(\tau^{-1}(F_k(H^{in})|R^{in}))|P^{out}) = F_k(H^{in})|R^{out} \sqsubseteq \sqcup_{i>0}F_i(H^{in})|R^{out}$.

If $j \geq k$, then $\tau(F_j(\tau^{-1}(F_k(H^{in})|R^{in}))|P^{out})$
$\sqsubseteq \tau(F_j(\tau^{-1}(F_{j+1}(H^{in})|R^{in}))|P^{out}) = F_{j+1}(H^{in})|R^{out} \sqsubseteq \sqcup_{i>0}F_i(H^{in})|R^{out}$.

Therefore $Fun(\mathcal{O}(X))$ is a fixed point of Φ.

Now to prove that $Fun(\mathcal{O}(X))$ is the least fixed point, since $Fun(\mathcal{O}(X)) = \sqcup F_j$, it is enough to show that for every $j > 0$ and for all input histories H^{in}, $F_j(H^{in}) \sqsubseteq \mu\Phi(H^{in})$.

We prove this by induction on j. Suppose this is true for $F_1, F_2, \ldots F_{j-1}$, $j > 0$. We will prove the statement for F_j.

Let σ be a completed computation sequence of $\mathcal{N}^j(\perp)$ for input given by H^{in}. Then the history induced by σ is $F_j(H^{in})$. Each finite prefix π_k of σ induces a history H_{π_k} on the ports P, and $F_j(H^{in})$ is equal to the lub of the H_{π_k}'s. So it suffices to show that every $H_{\pi_k} \sqsubseteq \mu\Phi(H^{in})$.

The proof is by induction on k. The base case $k = 0$ is obvious. For the inductive step, suppose that $H_{\pi_k} \sqsubseteq \mu\Phi(H^{in})$. Let $\pi_{k+1} = \pi_k a$.

Case 1: a is an internal transition. Then $H_{\pi_{k+1}} = H_{\pi_k} \sqsubseteq \mu\Phi(H^{in})$.

Case 2: a is an input transition. Then $H_{\pi_{k+1}}(p) = H_{\pi_k}(p) \sqsubseteq \mu\Phi(H^{in})(p)$ for every output port p. And $H_{\pi_{k+1}}(p) \sqsubseteq H^{in}(p) = \mu\Phi(H^{in})(p)$ for every input port p.

Case 3: a is an output transition, corresponding to an output port of M_i. Then $H_{\pi_{k+1}}|P_i^{out} \sqsubseteq f_i(H_{\pi_{k+1}}|P_i^{in}) = f_i(H_{\pi_k}|P_i^{in}) \sqsubseteq f_i(\mu\Phi(H^{in})|P_i^{in}) = \mu\Phi(H^{in})|P_i^{out}$, since $\mu\Phi$ is a fixed point of Φ.

Case 4: a is an output transition, corresponding to an output port of $M_i^!$. The projection γ of π_{k+1} onto the subexpression $\mathcal{N}^{j-1}(\perp)$ is a prefix of a computation sequence of that automaton. Therefore $H_{pi_{k+1}}|R^{out} = \tau(H_\gamma|P^{out} \sqsubseteq \tau(F_{j-1}(H_\gamma|P^{in}))) = \tau(F_{j-1}(\tau^{-1}(H_{pi_{k+1}}|R^{in})))$
$= \tau(F_{j-1}(\tau^{-1}(H_{pi_k}|R^{in}))) \sqsubseteq \tau(\mu\Phi(\tau^{-1}(\mu\Phi(H^{in})|R^{in})))$ using the inductive hypotheses on $j-1$ and k. This is simply $\mu\Phi(H^{in})|R^{out}$ by the fact that $\mu\Phi$ is a fixed point of Φ. ∎

This chapter gives a short description of an extension of the operational semantics for static dataflow to dynamic dataflow. The suitability of this operational semantics is evidenced by the naturality of its definition and by the fact that it satisfies Kahn's principle. We discussed only determinate networks, but we believe that a similar construction can be described to obtain operational semantics for indeterminate networks.

Chapter 6

Conclusions

In this thesis, we have studied the expressiveness and inexpressiveness situations that arise due to the presence of different kinds of indeterminacy in an asynchronous distributed computation setting. We used a particular model of asynchronous distributed computation, called the dataflow model. This model very naturally portrays the situation of autonomous computing agents communicating asynchronously with each other. We used the operational semantics of Lynch and Stark to describe the operational behaviour of processes in the dataflow model. We showed how one could abstract out the low-level operational behaviour and obtain traces that are well-suited for reasoning about network behaviour, once the properties of trace sets have been fully described using the operational semantics from which they arise. We modeled autonomous computing agents in a way that was general enough to make our intuitive notions of observation in asynchronous systems precise, and also allowed us to express the different notions of indeterminacy that arose in this study. Our notion of observation translated into closure properties of trace sets, that we proved using the operational semantics.

The main contribution here has been to show that there is a surprising hierarchy of different notions of indeterminacy. This cannot simply be described using degree of branching – bounded versus unbounded. In fact we have shown that depending on what fairness guarantees we demand of our schedulers, their behaviours will satisfy radically different properties, and we cannot always hope to simulate the effect of one scheduler using another. Regarding the merge primitives, not only have we shown that the three merge primitives fair merge, angelic merge and infinity-fair merge are at different expressiveness levels, but also we have done so by describing some very interesting monotonicity properties – Hoare-monotonicity and Smyth-monotonicity. This leads to the hope that if one concentrates on the specific properties and aspects of the agents in any particular system that are really relevant to their behaviour, then one would obtain simpler and more convenient semantics to describe this behaviour.

Our operational semantics has been based on the work of Lynch and Stark [LT87,LS89], who developed the notion of input-output automata to give an operational description of computing agents. Gene Stark [Sta90] has gone on to classify different notions of indeterminacy as we have done, except that he does it from an operational point of view by describing different classes of automata for different notions of indeterminacy. In [Sta90], he describes the class of *semi-determinate automata*, that contains infinity-fair merge but does not contain angelic merge. In [Sta89a], he describes a compositional relational semantics for indeterminate dataflow networks, where the "relation" is not simply the input-output relation, but has added structure.

There has been a lot of related work in the area of indeterminacy in dataflow. Several treatments of unbounded indeterminacy may be found [Abr83,AP86, Bro83,Par80,Par82,Plo82], and there are several attempts to give a satisfactory treatment of the fair merge primitive [Bro83,Bro87,dBKM84,KP85,Par82, Pan85]. The work in this thesis somewhat clarifies the situation by distinguishing between different notions of indeterminacy. We also proved that trace semantics is fully abstract for dataflow networks. Fully abstract semantics emerged only recently [Jon87,JK88,Jon89,Kok88,Rus89,RT88]. The proof in this thesis was proved independently but is essentially the same as in [Jon89], and is based on insights provided by [Kok88]. Kok's semantics used infinite sequences of finite sequences of values, called finite word streams. The idea was to assume "check-points" in runs of the networks. He proved that this semantics was fully abstract for networks with angelic merge in them, but not fair merge.

Rabinovich and Trakhtenbrot [RT88] have a different notion of observation. Instead of viewing the entire input-output relation to be observable, they view only finite pieces of it to be observable. Of course, this immediately identifies primitives that have identical finite behaviours but can be distinguished by observing their entire infinite behaviour. They show that prefix-closed sets of finite sequences of input and output events provide a fully abstract semantics for their notion of observation.

Jim Russell [Rus90] has contributed a lot to this area. He has made a detailed study of a generalization of the class of dataflow networks implementable with infinity-fair merge. He calls these oraclizable networks, and he has given a Kahn-style principle for such networks. He has also shown that trace semantics is fully abstract not only for networks with fair merge, but also for networks that only have finite indeterminacy. His comparison between Egli-Milner monotone (both Hoare-monotone and Smyth-monotone) primitives and oraclizable networks sheds new light onto the understanding of the monotonicity conditions that we described in this thesis.

In [Bro87], Broy describes a denotational semantics for dataflow networks. His semantics associates each primitive with a set of continuous functions together with an "admissibility" predicate that determines whether the behaviour of a particular function for some particular input is a legal or admissible behaviour of the primitive. He does not consider any operational semantics and so does not consider the relationship of his denotational semantics to an operational semantics.

The study of indeterminate computation is not restricted to dataflow. Milner's CCS [Mil80] and Hoare's CSP [Hoa85] are process algebras that describe a different model of communicating processes. Critchlow and Panangaden [CP89] have carried out an investigation on the different kinds of "delay operators" in synchronous CCS. These operators have the same flavour as our merge primitives.

There has been recent interest in higher-order process calculi, in which processes themselves may be passed as values. An interesting extension of our study would be to consider the different notions of indeterminacy for models of dataflow in which "higher-order values" may be passed along channels.

Our dataflow model dealt with *asynchronous* communicating processes. This meant that processes did not have to undergo any "handshaking" to communicate with each other. We could also consider *synchronous* communicating processes by requiring two processes to go through a handshaking protocol every time one of them wishes to send a value to another. In that case, we believe that the operational and trace semantics described in this thesis can also be used for the study of synchronous systems.

Finally, we believe that this study has given us valuable insights into concurrent behaviour and the role of feedback in networks, and this should prove useful in studying other styles of programming languages such as concurrent logic programming languages.

Bibliography

[Abr83] S. Abramsky. On semantic foundations for applicative multiprogramming. In J. Diaz, editor, *Proceedings of the Tenth International Conference On Automata, Languages And Programming*, pages 1–14, New York, 1983. Springer-Verlag.

[AP86] K. R. Apt and G. D. Plotkin. Countable nondeterminism and random assignment. *Journal Of The ACM*, 33(4):724–767, 1986.

[AR88] I. J. Aalbersberg and G. Rozenberg. Theory of traces. *Theoret. Comput. Sci*, 60(1):1–82, 1988.

[BA81] J. D. Brock and W. B. Ackerman. Scenarios: A model of nondeterminate computation. In *Formalization of Programming Concepts*, pages 252–259, 1981. LNCS 107.

[Bro83] M. Broy. Fixed point theory for communication and concurrency. In *Formal Description of Programming Concepts II*, pages 125–148. North-Holland, 1983.

[Bro87] M. Broy. Semantics of finite and infinite networks of concurrent communicating agents. *Distributed Computing*, 2:13–31, 1987.

[CP89] C. Critchlow and P. Panangaden. The expressive power of delay operators in sccs. Technical Report TR 89-998, Cornell University, May 1989.

[Cur86] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Research Notes in Theoretical Computer Science. John Wiley and Sons, 1986.

[dB85] A. deBruin and W. Böhm. The denotational semantics of dynamic networks of processes. *ACM Transactions on Programming Languages and Systems*, 7(4):656–679, 1985.

[dBKM84] J. W. deBakker, J. A. Bergstra, J. W. Klop, and J.-J. Ch. Meyer. Linear time and branching time semantics for recursion with merge. *Theoretical Computer Science*, 34:134–156, 1984.

[Fau82] A. A. Faustini. *The Equivalence of an Operational and a Denotational Semantics for Pure Dataflow.* Ph.D. dissertation, The University of Warwick, 1982.

[Fra86] N. Francez. *Fairness.* Springer-Verlag, 1986.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Series in Computer Science. Prentice-Hall International, London, 1985.

[JK88] B. Jonsson and J. Kok. Comparing dataflow models. Manuscript, 1988.

[Jon87] B. Jonsson. *Compositional Verification of Distributed Systems.* Ph.D. dissertation, Uppsala University, 1987.

[Jon89] B. Jonsson. A fully abstract trace model for dataflow networks. In *Proceedings of the Sixteenth Annual ACM Symposium On Principles Of Programming Languages*, 1989.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74*, pages 993–998. North-Holland, 1974.

[Kel78] R. M. Keller. Denotational models for parallel programs with indeterminate operators. In *Formal Description of Programming Concepts*, pages 337–366. North-Holland, 1978.

[Kok87] J. Kok. A fully abstract semantics for dataflow nets. In *Proceedings of Parallel Architectures And Languages Europe 1987*, pages 351–368, Berlin, 1987. Springer-Verlag.

[Kok88] J. Kok. Dataflow semantics. Technical Report CS-R8835, Centre for Mathematics and Computer Science, August 1988.

[KP85] R. M. Keller and P. Panangaden. Semantics of networks containing indeterminate operators. In *Proceedings of the 1984 CMU Seminar on Concurrency*, pages 479–496, 1985. LNCS 197.

[KP86] R. M. Keller and P. Panangaden. Semantics of digital networks containing indeterminate operators. *Distributed Computing*, 1(4):235–245, 1986.

[LS89] N. A. Lynch and E. W. Stark. A proof of the kahn principle for input/output automata. *Information and Computation*, 1989.

[LT87] N. A. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical Report MIT/LCS/TR-387, M. I. T. Laboratory for Computer Science, April 1987.

[Man74] Z. Manna. *Mathematical Theory of Computation.* McGraw Hill, 1974.

[Maz86] A. Mazurkiewicz. *Advanced Course in Petri Nets*, volume 255 of *LNCS*, chapter Trace Theory, pages 279–324. Springer-Verlag, 1986.

[Mil80] R. Milner. *A Calculus for Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[Pan85] P. Panangaden. Abstract interpretation and indeterminacy. In *Proceedings of the 1984 CMU Seminar on Concurrency*, pages 497–511, 1985. LNCS 197.

[Par80] David Park. On the semantics of fair parallelism. In *Proceedings of the Winter School on Formal Software Specification*, pages 504–526, New York, 1980. Springer-Verlag. Lecture Notes In Computer Science 86.

[Par82] D. Park. The "fairness problem" and non-deterministic computing networks. In *Proceedings of the Fourth Advanced Course on Theoretical Computer Science, Mathematisch Centrum*, pages 133–161, 1982.

[Plo77] Gordon Plotkin. Lcf considered a programming language. *Theoretical Computer Science*, 5(3):223–256, 1977.

[Plo82] G. D. Plotkin. A powerdomain for countable nondeterminism. In *Proceedings of the Ninth ICALP.* Springer-Verlag, 1982. LNCS 140.

[PS] P. Panangaden and V. Shanbhogue. On the expressive power of indeterminate primitives. To appear in *Information and Computation.*

[PS87] P. Panangaden and V. Shanbhogue. On the expressive power of indeterminate primitives. Technical Report 87-891, Cornell University, Computer Science Department, November 1987.

[PS88a] P. Panangaden and V. Shanbhogue. Mccarthy's amb cannot implement fair merge. In *Proceedings of the Eighth FSTTSC Conference*, pages 348–363, 1988. LNCS 338.

[PS88b] P. Panangaden and E. W. Stark. Computations, residuals and the power of indeterminacy. In Timo Lepisto and Arto Salomaa, editors, *Proceedings of the Fifteenth ICALP*, pages 439–454. Springer-Verlag, 1988. Lecture Notes in Computer Science 317.

[PSS90] P. Panangaden, V. Shanbhogue, and E. W. Stark. Stability and sequentiality in dataflow networks. In *Proceedings of the Seventeenth*

ICALP, 1990. Also TR 89-1055, Computer Science Department, Cornell University.

[RT88] A. Rabinovich and B. A. Trakhtenbrot. Nets of processes and dataflow. In *Proceedings of ReX School on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency.* Springer-Verlag, 1988. LNCS.

[RT89] A. Rabinovich and B. A. Trakhtenbrot. Nets and data flow interpreters. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 164–174, 1989.

[Rus89] J. R. Russell. Full abstraction for nondeterministic dataflow networks. In *Proceedings of the 30th Annual Symposium of Foundations of Computer Science*, pages 170–177, 1989.

[Rus90] J. Russell. *Full Abstraction and Fixed-point Principles for Indeterminate Computation.* Ph.D. dissertation, Cornell University, 1990.

[Sha90] V. Shanbhogue. The relationship between multiway and two-way fair merges. In *ISLIP 90*, 1990.

[Sta87] E. W. Stark. Concurrent transition system semantics of process networks. In *Proceedings Of The Fourteenth Annual ACM Symposium On Principles Of Programming Languages*, pages 199–210, 1987.

[Sta89a] E. W. Stark. Compositional relational semantics for indeterminate dataflow networks. Manuscript in Preparation, 1989.

[Sta89b] E. W. Stark. Concurrent transition systems. *Theoretical Computer Science*, pages 221–269, 1989.

[Sta90] E. W. Stark. On the relations computable by a class of concurrent automata. In *Proceedings Of The Fifteenth Annual ACM Symposium On Principles Of Programming Languages*, New York, 1990. IEEE.

[WA85] W. W. Wadge and E. A. Ashcroft. *Lucid, the dataflow programming language.* Academic Press, 1985.