JOURNAL OF COMPUTER AND SYSTEM SCIENCES 47, 287–321 (1993)

Nonexpressibility of Fairness and Signaling

DAVID MCALLESTER,* PRAKASH PANANGADEN,† AND VASANT SHANBHOGUE‡

Computer Science Department, Cornell University, Ithaca, New York 14853

In this paper we establish new expressiveness results for indeterminate dataflow primitives. We consider split primitives with three differing fairness assumptions and show that they are strictly inequivalent in expressive power. We also show that the ability to announce internal choices enhances the expressive power of two of the primitives. These results are proved using a very crude notion of observation and thus apply in any reasonable theory of process equivalence. © 1993 Academic Press, Inc.

1. INTRODUCTION

Fairness is regarded as an important property of real systems and there is considerable interest in semantic theories and proof systems for reasoning about fairness [12]. In the present paper we examine the relative expressive power of a variety of fairness primitives and prove new inexpressibility results in the context of asynchronous systems. We prove that three different "split" primitives have different expressive power. We also consider the effect of adding signaling to each primitive. By "signaling," we mean that a primitive has a mechanism for announcing its internal indeterminate decisions. Our investigation is carried out in the context of static dataflow networks, i.e., networks whose structure remains fixed throughout execution.

Our interest in this work stemmed from earlier discoveries by Panangaden and Stark [25, 26, 28] that the fair merge primitive [18] is strictly "more powerful" than other primitives exhibiting unbounded indeterminacy or countable indeterminacy. This showed that one could not classify indeterminate primitives on the degree of branching they embodied. All fair systems include primitives with countable indeterminacy [11]. In the programming model studied by Chandra [4, 11], countable indeterminacy and fairness are equivalent. In the case of asynchronous dataflow networks [16], the analysis is complicated by the fact that a process may receive data from different autonomous processes in an asynchronous fashion. This means that fair merges need to avoid empty data channels as well as make fair

* Currently at MIT.

† Supported in part by NSF Grant DCR-8602072 and by NSERC. Currently at McGill University.

‡ Supported in part by NSF Grant DCR-8602072. Currently at Wichita State University.

choices. Work by Apt and Plotkin [5] shows that the presence of countable indeterminacy in a programming language leads to failures of continuity. The work of Panangaden and Stark shows that there is a breakdown of a monotonicity property that occurs in the case of fair merge.

Having identified monotonicity as a property that differentiates two kinds of countable indeterminacy, we are led to focus attention on monotone primitives. Since semantics of networks including fair merge are notoriously difficult, it is possible that one might develop simpler semantic theories for systems that do exhibit countable indeterminacy but are monotone. We discovered that there are provably inequivalent primitives here too. This paper discusses these primitives and establishes the difference in their expressive power. There appears to be a richer taxonomy of indeterminate primitives than had been suspected earlier. Recently there has been considerable interest in developing semantic theories to handle countable indeterminacy [2, 5, 6, 10, 19, 29]. Our work shows that there are several flavors of countable indeterminacy.

In the rest of this introduction we describe the setting and state the results informally. For our formal results, we will use the notion of traces of networks. Recent work by Jonsson [14] and Russell [31] shows that traces constitute a good abstraction of the detailed operational aspects of network behavior. An automata-theoretic formalism essentially due to Lynch and Tuttle [22] and Stark [34] is presented in Appendix A. We show how one can pass from these automata to traces of the networks, and then reason with traces exclusively after developing the machinery to reason about process equivalence and implementability. We use a very weak notion of process equivalence. The significance of this is that our non-implementability proofs will survive any passage to a more discerning semantic theory. Clearly our positive implementability results are then not of great significance but they do help to classify what can be distinguished at this level of observation.

1.1. *Kahn Networks*

We define an asynchronous dataflow network to be a finite set of autonomous computing agents, represented by the *nodes* of a directed graph, connected by directed arcs, called *channels* or *ports*. The directed arcs coming into a node are called input channels or input ports and those leaving a node are called output channels or output ports. The interconnection structure is fixed throughout execution. Nodes can only "listen" to a single channel at a time. One can think of each node as executing a sequential program. Communication between nodes is effected by the transmission of messages along the channels. The channels are unbounded queues where the sending of a message and the receipt of the message are distinct activities. There is no synchronization on message passing such as in CSP [13] or CCS [24].

We consider abstractions of different schedulers. This leads to three primitives that we call *split* processes. Each can be regarded as a dataflow primitive with an input port and two output ports. Tokens are consumed from the input port and are placed on one or other of the output ports. One can now distinguish between different split primitives based on the fairness properties that they satisfy with respect to choosing between the output channels. The inexpressiveness results here may be considered to be in the same spirit as Stark's investigation into the expressive power of semaphore primitives [33] extended to the dataflow case.

Another inexpressiveness phenomenon at work here arises from *sequentiality*. We consider augmenting the choice primitives with an additional output channel on which a value is output every time a choice is made. This allows other processes in the network to access the choices made. It turns out that this interacts quite delicately with the fairness properties. With a strong fairness assumption, we show that adding signaling does not add to the expressive power, whereas with a weaker fairness assumption, we prove that the primitives that cannot signal are strictly weaker. The proof methods hinge on using the fact that individual processes are sequential in an appropriate sense. One may also view this as an analysis of how information gets dispersed in a network.

We now describe the split primitives that we study. The first primitive, which we call *Unfair Split* (US) splits its input sequence into two subsequences, and it is possible unfair in the sense that one output channel may receive no values even when the input sequence is infinite.

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

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 breakup—which particular input values are output at the first output channel and which particular input values are output at the second output channel. We 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 primitive 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*).

1.2. *Results*

The expressiveness situation that we establish is depicted in Fig. 1. An arrow between two primitives indicates that there exists a finite network built from instances of the first primitive and "ordinary" (essentially sequential and determinate) nodes that implements the input–output behavior of the second primitive. An arrow

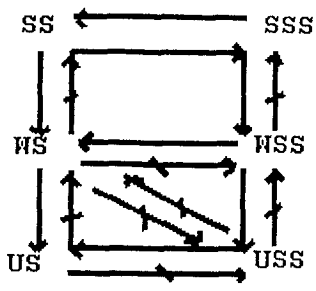

FIG. 1. The relative expressiveness of splits.

with a line through it indicates that we have proven that no such implementation is possible.

Based on experience with semaphores [33], one might expect that US cannot implement WS. It is somewhat more surprising that WS, or even WSS, cannot be somehow "iterated" to implement SS. The second interesting result is that we cannot simulate signals *except* when we have strong fairness. This result is relate to sequentiality in the sense of Kahn and Plotkin [8, 17]; but we do not have a theory of sequential *indeterminate* processes.

2. TRACES OF DATAFLOW NETWORKS

In this paper we work with *traces* of dataflow networks. These are abstractions of computation sequences of the networks. Operationally a dataflow process is described by an automaton equipped with a relation that describes when actions are concurrent. The concurrency structure is the basis of a detailed analysis of the structure of the computations of such networks. Using this, one can define the notion of "completed" or "fair" computation of a network. Furthermore, one can interpret the concurrency relation between actions as corresponding to the ability to permute events in a computation sequence. Finally, one can define what it means to link processes into networks and one can show that such networks are themselves automata equipped with a concurrency relation and apply the theory that has been developed for such automata. We can define certain special types of processes, specifically the *determinate* processes and show that they compute functions. Precise definitions of automata, completed computation sequences, networks, determinacy, input–output relation and permutation of events can be found in Appendix A. It should not be necessary to read the appendix, however, in order to follow the proofs in the paper. Most of the proofs of the lemmas and theorems in this section can be found in Appendix B.

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 [15].

2.1. *Traces and Compositionality*

We now define abstractions of computation sequences and the relationship between the abstractions for a network and the abstractions for its components.

DEFINITION 1. If γ is a computation sequence, then we define $\mathrm{tr}(\gamma)$ to be the subsequence of $\mathrm{ev}(\gamma)$, consisting of all the input and output events in $\mathrm{ev}(\gamma)$.

DEFINITION 2. A trace of a network N of processes is a sequence t of input events and output events of N, such that $t = \mathrm{tr}(\gamma)$ for some completed computation sequence γ of N. We write $\mathrm{Trset}(N)$ for the set of traces of a network N.

In an earlier discussion of network semantics, trace sets were called archives [19]. 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; $t[i]$ will represent the ith event in t.

The following theorem relates the trace set of a network with the trace sets of its individual processes. A more general version of this theorem, relating the trace set of a network with the trace sets of its subnetworks for any arbitrary decomposition of the network into subnetworks was proved in [32]. By a *compatible* set of automata we mean that the automata have disjoint sets of internal actions and their sets of communication actions are arranged so that input for one automaton can be the output of exactly one other automaton.

THEOREM 1. *If the network N is the composition of a compatible set of automata $\{N_i\}$ for i in some finite index set I, then t is a trace of N if and only if $\Pi_{N_i}(t)$ is a trace of N_i for every i.*

It is possible to have processes with different sets of traces, but the same IO-relation. Brock and Ackerman [9] have such an example, but their example uses a powerful primitive, *fair merge*. There are other examples using only finite indeterminacy [31].

2.2. *Sequential Processes*

In [16], 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 [20, 17] 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 use the specialization of this definition to stream domains.

The main reason for our 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 consider the class of sequential processes to constitute that base class of processes. The intuitive idea is to capture the notions of processes with "single threads of control" and no "internal parallelism." Our notion of sequential process expresses this formally. It is possible to relate this notion here, given in terms of traces, to an automata theoretic condition on the transition relation [27].

DEFINITION 3. 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 $\Pi_i(x)$.

One can show that the sequential processes compute sequential functions in the sense of Kahn and Plotkin [20, 17]. We given an example of non-sequentiality.

EXAMPLE 1. *Parallel OR.* This process has two input ports and one output port. It computes the following function POR:

$$\mathrm{POR}(1\,{}^\wedge s, \Lambda) = \mathrm{POR}(\Lambda, 1\,{}^\wedge s') = \mathrm{POR}(1\,{}^\wedge s, 1\,{}^\wedge s') = 1$$

and

$$\mathrm{POR}(0\,{}^\wedge s, 0\,{}^\wedge s') = 0$$

for any sequences of values s, s'. Λ represents the empty sequence here. Then we have

$$\mathrm{POR}(1, \Lambda) = \mathrm{POR}(\Lambda, 1) = \mathrm{POR}(1, 1) = 1, \qquad \mathrm{POR}(0, 0) = 0,$$

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.

2.3. *Causality between Events*

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

DEFINITION 4. 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.

We then have the following lemma that tells us that the causal set captures all the dependencies between the events of a trace.

LEMMA 1. *If t is a trace of a sequential process 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.*

The proof is in Appendix B.

2.4. *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. The failure of this property leads to problems with limit closure of sequences. In order to eliminate this, we define "scheduled" traces and work with scheduled trace sets of processes, instead of the trace sets. Roughly speaking, in the scheduled traces every component of a network is scheduled to process in a cyclic fashion if it has an enabled event. Using scheduled traces rather than the full trace set does not change the basic theory because, as we will show for the processes we 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. In the rest of this section, by networks, we mean compositions of compatible sets of processes.

An important closure property we use in subsequent sections is the following:

DEFINITION 5. 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 a simple buffer process 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 6. A *scheduled trace* t of a network N is a network trace such that for every process P in N, if t_P is the projection of t onto P, and P has m ports, then for every $i>0$, if $t_P[i]$ is an output event and $t_P[(i-m)\cdots(i-1)]$ does not contain any output events at the same port, then $t_P[1\cdots(i-m-1)]^\wedge t_P[i]^\wedge t_P[(i-m)\cdots(i-1)]^\wedge t_P[(i+1)\cdots]$ is not a trace of P.

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*. The following is clear from the definition.

LEMMA 2. *The projection of a scheduled trace t of a network N onto a process P of the network is a scheduled trace of P.*

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. The following lemma says that the scheduled trace set of the processes that we work with are indeed prefix limit closed.

LEMMA 3. *The scheduled trace set of a determinate process is prefix limit closed.*

The proof is in Appendix B.

DEFINITION 7. An infinite chain $\beta_1 \sqsubseteq \beta_2 \sqsubseteq \cdots$ of sequences is said to be *eventually increasing* if it is non-decreasing and there is no i, such that $\forall j \geqslant i$, $\beta_j = \beta_i$.

We use this definition in Lemma 4 and in subsequent sections when we discuss the other split primitives.

LEMMA 4. *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.*

The proof is in Appendix B.

So far we have concentrated on the scheduled trace set and shown that it has a nice property, prefix limit closure, that will be important to us in the next section. 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 behaviors of a process or network. This is done by defining a scheduling operation on traces that uses the causal dependency between events. The details are in Appendix C.

3. Inexpressibility Results

We have described our model of computation, and 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 split processes that were defined in the Introduction.

3.1. *Implementability*

In this subsection, we establish our notions of when one level is stronger than another and when one level is weaker than another. We describe the rules by which we can construct networks of processes in the appendices. We now establish the definitions for *implementability* and *non-implementability.*

Definition 8. 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 9. 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 10. 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.

We will be concerned with proving failure of weak implementability, since this is a stronger notion than failure of strong implementability. 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 N_1 can or cannot strongly implement (weakly implement) another process 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 11. Processes N_1 and N_2 are said to be *at the same expressiveness level* if they can strongly implement each other.

Definition 12. A process 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.

3.2. *Oracles*

We now present some implementability results, and we use some special processes called oracles to present these results.

DEFINITION 13. 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. Let us define the three oracles that we will be most concerned with and are intimately related to the split primitives. 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 network shown in Fig. 2 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 repeats the above. When the oracle chosen is O_{us}, then we strongly implement unfair split and unfair split with signal. When the oracle is 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.

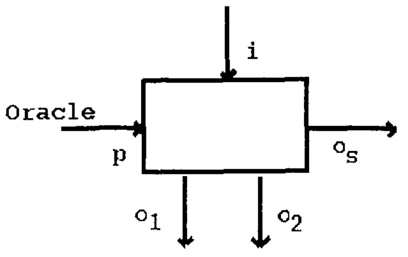

FIG. 2. Split from an oracle.

THEOREM 2. *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.*

3.3. *Some Strong Implementability Results*

We will now describe the expressiveness picture in Fig. 1, in which an arrow from one process to another indicates that the first process can strongly implement the second.

THEOREM 3. *Strongly fair split can strongly implement strongly fair split with signal.*

Proof. Figure 3 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, 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 4. *O_{ss} can strongly implement weakly fair split with signal, and hence O_{ws}.*

Proof. The same figure 2 illustrates how we can strongly implement weakly fair split with signal from O_{ss}. The determinate process P is different: it first 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. If it decides that both streams are to be infinite, then it uses the sequence of 1's and 2's at port p to decide which input values should be output at port o_1 and which input values should be output at port o_2. ▌

THEOREM 5. *Weakly fair split can strongly implement unfair split.*

Proof. Let Σ^+ be the infinite sequence $1\^2\^3\^\cdots$ of all positive integers. When this is the input to a weakly fair split, the first value on the first output

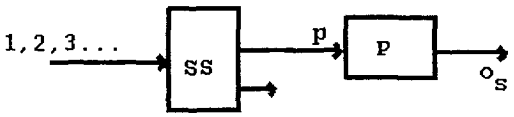

FIG. 3. SSS from SS.

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 6. *O_{ws} can strongly implement O_{us}.*

Proof. A similar proof as the above. ▌

3.4. *Inexpressibility 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 14. 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 15. 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 5. *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 Koenig's lemma.

THEOREM 7. *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. We fix the input stream to be $1\,\hat{}\,2\,\hat{}\,3\ldots$. Then the first output port c of N, corresponding to the first output port of the WS process being implemented, 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 $1\,\hat{}\,2\,\hat{}\,3\ldots$. 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 $T(S)$. By Lemma 4, every path in the tree is a network trace, and so it has an output event on c. We prune every path at the first output event on that path on port c. Moreover, by Lemma 5, $T(S)$ is finitely branching. Therefore the pruned tree is a finitely branching tree with no infinite paths. By Koenig'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 Koenig's lemma arguments by themselves do not seem sufficient.

THEOREM 8. *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 up to this countable choice.

DEFINITION 16. 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 $t[1 \cdots i]$.

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

DEFINITION 17. 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 6. *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 $t[1 \cdots i]$. ▮

DEFINITION 18. 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 7. *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 8. *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 9. *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 8, 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 4. Let t_i be the projection of $t[1 \cdots i]$ 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 ouput 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 ouput porst 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 $j \geqslant i$, $t_j = t_i = t_P$. Let t' be a scheduled trace of the network, such that $t[1 \cdots (i+m)]$ 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 19. 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 20. 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. Let one of the output ports, corresponding to an output port of the SS process being implemented, be c. We fix the input stream to be $1 \wedge 2 \wedge 3 \wedge \ldots$. 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 9. 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 Koenig'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 $s[1], s[2], \ldots, s[i-1]$, 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 $T_1, T_2, \ldots$. 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. ∎

3.5. *Inexpressibility of Signaling*

In this subsection we explore the nonexpressability arising from the sequentiality of the individual processes. Understanding sequentiality is a fundamental concern in the semantics of modern programming languages [20, 30]. 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 21. 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 22. 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 10. *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 an 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 9. *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 10. *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 23. 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 11. *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 10, except for the following case. $\langle c, n\rangle$ has an earliest occurrence time equal to k in C'_{sI}, 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 it 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 10. ▌

Proof of Theorem 10. 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 6. 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 11, they are all determined.

As in Lemma 7, 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 the next section, we presented our main results proving the split primitives to be of differing expressive power.

4. Conclusions

We have examined the expressiveness situations that arise with a variety of fairness primitives 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. 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 primitives, their behaviors will satisfy different properties, and we cannot always hope to simulate the effect of one primitive using another. We also saw how the expressive power of primitives varies when internal choices are made explicit.

It is known, largely through work by Abramsky [3] that one can give an elegant fixed point semantics to networks containing strongly fair split or infinite fair merge. The point is that such primitives can be thought of as being "oracle driven," that is, the indeterminacy arises from an external source that resolves choices in a manner independent of the input. One can model such networks as sets of functions. Russell [31] has shown that such sets of functions are fully abstract if one enforces the proper closure conditions on such sets. The oracle view does not apply to networks containing, for example, fair merge. Our results in this paper say how the expressive power varies with the output of the oracle.

Finally, we feel that the significance of the signaling phenomenon is that there are some subtle interactions between indeterminacy and sequentiality. We have not formulated a definition of what sequential or nonsequential might mean in the presence of indeterminacy. We hope that subsequent investigations will lead to a satisfactory definition.

APPENDIX A: Operational Semantics of Dataflow Processes

In this appendix we describe the operational semantics of dataflow networks in terms of automata equipped with a notion of concurrent transition. The development here is essentially due to Lynch and Tuttle [21, 22] and Stark [34, 35]. We have included it in order to make the paper reasonably self-contained. The operational semantics describes the execution of a program in terms of a sequence of transitions between states. 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.

A.1. *Port Automata*

We now formally describe processes or 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 24. 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 [1, 23] to obtain an algebraic theory of traces. We call events related by the concurrency relation *concurrent.* Let V be a set of *data values* called the *value alphabet.* Throughout this paper, we 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 25. A *monotone 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. $\mathit{Inp} = P^{\text{in}} \times V$, and

$Out = P^{\mathrm{out}} \times V$, 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*. (p, v_1) and (p, v_2) are not concurrent for any p, v_1, v_2. The elements of $E \backslash (Inp \cup Out)$ are called *internal events*.

- Q is a set of *states*, and $q^I \in Q$ 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 $E \cup \{\varepsilon\}$. ε, a special event not in E, is called the *identity event*,

satisfying the following conditions:

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

Identity. $\varepsilon \in A(q, r)$ 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 states q, r, s 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 $b \in A(r, p)$.

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.

Output delay. For any states q, r, s and any events a, b, if $a \in A(q, r)$, $b \in A(r, s)$, a is an output event at an output port and b is not an output event at the same port, then there exists a state p such that $b \in A(q, p)$ and $a \in A(p, s)$.

This definition is similar to the definitions of a *port automaton* and an *input–output automaton* due to Stark in [21, 28] and is closely related to the *input–output automata* of Lynch and Tuttle [22]. 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 the 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*. Further, an input event should not be able to disable other events that were enabled before this input event. This is captured by the property of non-disabling inputs.

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 intent of output events is the detection of data at output ports by the environment. The property of output delay says that events following an output event are, in general, not dependent in any way on the output event, and so they could even happen before the output event. This idea is similar to, but a restricted form of, an axiom in Bednarczyk's asynchronous systems [7].

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 $q \xrightarrow{\varepsilon} q$ 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 26. A *computation sequence* γ is a finite or infinite sequence of transitions of the form

$$q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \cdots .$$

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

We will now give an example of an automaton. We will use ^ as an infix operator for representing concatenation of sequences.

EXAMPLE 2. *Buffer*. This automaton has one input port and one output port. It reads values and outputs them, guaranteeing to read and output of 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 the transition relation, using v to represent a member of V. $A(q, r) = \{(i, v)\}$ iff $r = q \,\hat{}\, v$. $A(q, r) = \{(o, v)\}$ iff $q = v \,\hat{}\, r$. $A(q, q) = \{\varepsilon\}$.

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

We end this subsection with some notation. We 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 $\mathrm{value}(e)$. We also extend the definition of *value* to sequences—*value*$((p, v_1)(p, v_2)...)$ is $v_1 v_2 ...$. For any computation sequence

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

we define $\mathrm{ev}(\sigma)$, the *sequence of events* of σ to be $a_1 a_2 ...$. We 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 $\Pi_p(t)$. 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 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.

A.2. *Completed Computation and the Input–Output Relation*

In this subsection, 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 behavior of an automaton from its completed computation sequences.

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 $\gamma \preccurlyeq \delta$, 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 $\sqsubseteq$ on finite computation sequences of A as the transitive closure of $\preccurlyeq \cup \sim$. Define $\simeq\, = \,\sqsubseteq \cap \sqsupseteq$. It is an easy lemma that for γ, δ finite, $\gamma \sqsubseteq \delta$ iff $\exists \xi$ such that $\gamma\xi \simeq \delta$. One observation we can make is that if $\gamma \sqsubseteq \delta$, then the multiset of events in γ is contained in the multiset of events in δ. Another lemma is that for γ, δ finite, if $\gamma \sqsubseteq \delta$ and the multiset of events in γ is contained in the multiset of events in a prefix δ' of δ, then $\gamma \sqsubseteq \delta'$. We can now extend the permutation preorder relation to infinite computation sequences by defining $\gamma \sqsubseteq \delta$ iff for every finite $\gamma' \preccurlyeq \gamma$, there exists a finite $\delta' \preccurlyeq \delta$, such that $\gamma' \sqsubseteq \delta'$. We define $\simeq\, = \,\sqsubseteq \cap \sqsupseteq$ 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. We would like to say that a finite computation sequence γ is *not completed* if there is a non-input event enabled at its end. We would like to say that an infinite computation sequence γ is *not completed* if there is a suffix of γ and an event e such that e is enabled at every state in the suffix and commutes with every event in the suffix. Intuitively, the event e can happen at any point in the suffix but does not do so. Completedness turns out to be identical to $\sqsubseteq$-maximality for a particular input [32]. So we will take *maximality* to be the definition of *completedness*. Whenever we talk about $\sqsubseteq$-maximality, we will actually mean maximality for a particular input. We could think of the preorder $\sqsubseteq$ 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.

DEFINITION 27. A computation sequence γ of an automaton is said to be *completed* if is $\precsim$-maximal among all computation sequences with the same input as γ.

Let us now 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.

Now we can describe the *input–output relation* of an automaton. This describes the input–output behavior—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 28. The *input–output relation* of an automaton is the set of all pairs $(H_\sigma^{\text{in}}, H_\sigma^{\text{out}})$ 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*.

A.3. *Moves of Computation Sequences*

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

DEFINITION 29. 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 30. 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 12. *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 $\pi \preccurlyeq \pi' \simeq \xi$, thus proving $\delta \precsim \gamma$.

To prove that $\gamma \precsim \delta$, 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 $\pi \precsim \xi$. Let π' be the smallest prefix of γ that extends π and is involved by η'. Then $\pi' \simeq \xi$. Then $\pi \preccurlyeq \pi' \simeq \xi$. Therefore $\pi \precsim \xi$, thus proving that $\gamma \precsim \delta$. ▮

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 13. *If γ is a computation sequence of an automaton, then it can be $\precsim$-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 [28]. Briefly, we can show that every chain of computation sequences $\gamma_1 \precsim \gamma_2 \precsim \dots$, such that $\gamma \precsim \gamma_i$ for every i and all the γ_i have the same input as γ, has a lub. Hence, by Zorn's lemma, the set of all computation sequences δ, such that $\gamma \precsim \delta$, 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. ▮

A.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. The following definition is from [34].

DEFINITION 31. An automaton is *determinate* if it satisfies the following condition: $b \parallel b'$ 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 choice that the automaton makes occur between input event transitions. The following theorem and lemma were proved in [34].

THEOREM 11. *Determinate automata compute functions. Moreover, a function f is computed by a determinate automaton iff f is a continuous function.*

LEMMA 14. *Suppose A is a determinate automaton. Then for each input x, there is a unique completed computation sequence, up to $\simeq$-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', $\gamma \lesssim \gamma'$.*

A.5. *Networks of Automata*

We now describe how we can build networks of automata by collecting together individual automata and then linking ports together.

DEFINITION 32. 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

- for all $i, j \in I$ such that $i \neq j$ we have $(E_i \backslash (Inp_i \cup Out_i)) \cap (E_j \backslash (Inp_j \cup Out_j)) = \emptyset$; 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 33. The *composition* of a compatible set $\mathcal{S}$ of automata is the automaton $\Pi M_i = (E, Q, A)$, where

- $E = \bigcup E_i$, with $a \parallel b$ iff $a \parallel_i b$ for all $i \in I$ such that both a and b are in E_i.
- $Out = (\bigcup Out_i)$, and $Inp = (\bigcup Inp_i) \backslash (\bigcup Out_i)$,

- $Q = \Pi_{i \in I} Q_i$
- $q^\iota = (q_i^\iota : i \in I)$
- $e \in A((q_i : i \in I), (r_i : i \in I)$ 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 explicitly define *output hiding.*

Definition 34. If $\mathscr{A}$ 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 $\mathscr{A}$ results in the automaton with input ports P^{in} and output ports $P^{\text{out}} \backslash S$ with exactly the same sets of events and states and the same transition relation as $\mathscr{A}$.

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 ouput 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 $\alpha_i(a) = \varepsilon$ otherwise. Then we can define the *restriction* $\Pi_{M_i}(\gamma)$ of a computation sequence $\gamma = q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \dots$ of the network to a component automaton M_i by $\rho_i(q_1) \xrightarrow{\alpha_i(a_1)} \rho_i(q_2) \xrightarrow{\alpha_i(a_2)} \dots$ with the identity transitions collapsed.

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. Intuitively, we would call a computation sequence of a network completed, if its projection onto every individual automaton is completed. This turns out to be equivalent to maximality of the computation sequence, so we will again define completedness to be maximality.

Definition 35. A computation sequence γ of a network is said to be completed if it is $\precsim$-maximal among all computation sequences having the same input as γ.

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

DEFINITION 36. The *input–output* relation of a network of automata is the set of all pairs $(H^{\text{in}}_{\sigma}, H^{\text{out}}_{\sigma})$ with σ being a completed computation sequence of the network.

In this section, 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. We will henceforth use the terms "process" and "automaton" interchangeably. In the next section, we will discuss how we can abstract the internal events away from this description.

APPENDIX B: PROOFS ABOUT TRACES

In this appendix we give in full some of the proofs that we omitted in the main text of Section 2.

DEFINITION 1. If γ is a computation sequence, then we define $\text{tr}(\gamma)$ to be the subsequence of $\text{ev}(\gamma)$, consisting of all the input and output events in $\text{ev}(\gamma)$.

DEFINITION 2. A *trace* of a network N of processes is a sequence t of input events and output events of N, such that $t = \text{tr}(\gamma)$ for some completed computation sequence γ of N. We write $\text{Trset}(N)$ for the set of traces of a network N.

THEOREM 1. *If the network N is the composition of a compatible set of automata $\{N_i\}$ for i in some finite index set I, then t is a trace of N if and only if $\Pi_{N_i}(t)$ is a trace of N_i for every i.*

Proof. Let t be a sequence of events such that $\Pi_{N_i}(t)$ is a trace of N_i for every i. We show that 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 $t_i = \text{tr}(\gamma_i)$. 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 $\text{ev}(\gamma)$ consisting of all the input and output events of the N_i. Since each γ_i is completed, so is γ, and so t is a trace of N.

Let t be a trace of N corresponding to the completed computation sequence γ of the network. The projection γ_i of γ onto any automaton N_i is then a completed computation sequence of N_i. Therefore $\Pi_{N_i}(t) = \text{ev}(\Pi_{N_i}(\gamma))$ is a trace of N_i. ▌

We need a particular property of sequential processes, and the proof of Lemma 1 depends on this property.

LEMMA 15. *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 $H^{\text{out}}(p)$, then there is a computation sequence extending γ in which the next non-internal event after the events in γ is $(p, H^{\text{out}}(p)[i+1])$.*

Proof. If this event $e=(p, H^{\text{out}}(p)[i+1])$ is enabled at the end of γ, then we are done. If not, then we know, by Corollary 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 by the property of output delay, there is a sequence of moves that move these output events forward past e. ▌

The following is the proof of Lemma 1.

LEMMA 1. *If t is a trace of a sequential process 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 $\text{tr}(\gamma)=t$. We define a move transformation on γ to obtain a completed computation sequence γ' by Lemma 12, such that $\text{tr}(\gamma')=t'$, hence proving our lemma. We will define finite sequences η_i of moves such that the result of applying $\eta_1{}^\wedge\eta_2{}^\wedge\cdots{}^\wedge\eta_i$ on γ is a computation sequence γ_i such that $\text{tr}(\gamma_i)$ has $t'[1\cdots i]$ as prefix.

We define γ_0 to be γ. We will define η_i by induction. Suppose we have defined $\eta_1, ..., \eta_{i-1}$, and suppose $\text{tr}(\gamma_{i-1}[1\cdots r])$ is $t'[1\cdots(i-1)]$, 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 $(r'-1, r'), ..., (r+1, r+2)$.

Suppose $e=t'[i]$ is an output event. We would like to prove that the input in $\gamma_{i-1}[1\cdots r]$ can cause the output of e. Suppose this is not the case. If there is a single input event e' in $\gamma_{i-1}[r+1\cdots r']$, then (e', e) is not in the causal set of t. Therefore, there is some input $I \sqsubseteq H^{\text{in}}(\gamma)$ extending that in $\gamma_{i-1}[1\cdots r]$, 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 $\gamma_{i-1}[(r+1)\cdots r']$ by using induction.

Therefore, by Lemma 15, there is a computation sequence $\kappa=\gamma_{i-1}[1\cdots r]^\wedge\xi$, extending $\gamma_{i-1}[1\cdots r]$, in which the next non-internal event is e, and ξ ends in this event. By Lemma 14, κ can be extended to a completed computation sequence with the same input as in γ, and therefore $\kappa \precsim \gamma_{i-1}[1\cdots r']$. Therefore $\xi \precsim \gamma_{i-1}[(r+1)\cdots r']$. Then $\xi\xi' \simeq \gamma_{i-1}[(r+1)\cdots r']$, and there is a sequence of moves, which we take to be η_i, on $\gamma_{i-1}[(r+1)\cdots r']$ that transforms it to $\xi\xi'$.

The fact that $\eta_1{}^\wedge...$ 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. ▮

The following is the basic lemma 3 about scheduled traces.

LEMMA 3. *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 we 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 $\mathrm{tr}(\gamma_i) = t[1 \cdots j]$ 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 $t[j+1]$. If $t[j+1]$ is an output event, then by the determinacy of A and by Corollary 2, there is an extension γ_i' of γ_i with the same input as γ_i and containing the event $t[j+1]$ following the events in γ_i. All output events, if any, preceding $t[j+1]$ and occurring after γ_i in γ_i' can be moved forward past the event $t[j+1]$ by the property of output delay. We then choose γ_{i+1} to be that prefix of the resulting computation sequence that ends in the event $t[j+1]$.

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 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 $\gamma[1 \cdots k]$ 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, proving our claim. ▮

The following is the proof of Lemma 4.

LEMMA 4. *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 $t[1 \cdots i]$ onto P. Then each t_i is a prefix of a scheduled trace of P 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 the assumption.

Case 2. For some i, for all $j \geqslant i$, $t_j = t_i = t_P$. Let t' be a scheduled trace of the network, such that $t[1 \cdots (i+m)]$ 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. ▮

APPENDIX C: THE SCHEDULING OPERATION

In Section 2, we defined the scheduled trace set of a network. 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 behaviors 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, $t[1 \cdots m]$ for the prefix of t consisting of the first m events of t, and $t[m \cdots]$ for the suffix of t starting from $t[m]$.

DEFINITION 37. 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 38. 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 of a process P in N, $t[j]$ is an output event of P, 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 39. $\prec = (\prec_1)^*$.

LEMMA 19. $\prec$ 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., $t[i] = t[i_1] \prec_1 t[i_2] \prec_1 \cdots \prec_1 t[i_k] = t[j]$, then $i = i_1 < i_2 < \cdots < i_k = j$. Therefore $i \leqslant j$. By the antisymmetry and well-foundedness of $\leqslant$ 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 sometimes denote the $\prec$ associated with trace t by $\prec_t$.

LEMMA 20. *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 1, 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) value(s_1) = value$(\Pi_{o_1}(t))$ and value(s_2) = value$(\Pi_{o_2}(t))$,

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

LEMMA 21. *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. ∎

LEMMA 22. *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 1, 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 20 and 21. 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 40. A *port order* of a finite network with m ports is defined to be a total ordering $p_0, p_1, p_2, ..., p_{m-1}$ of the m ports of the network.

DEFINITION 41. For any finite network with m ports, and any port order $\mathbf{o} = p_0, p_1, p_2, ..., p_{m-1}$, 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 $p_{k+1} \bmod_m, ..., p_k$ such that each of its $\prec_t$-predecessors is in $S_{\mathbf{o}}(t)[1 \cdots i]$.

LEMMA 23. *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 22, 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. $t'_M[1 \cdots (i-m-1)]^\wedge t'_M[i]^\wedge t'_M[(i-m) \cdots (i-1)]^\wedge t'_M[(i+1) \cdots]$ cannot be a trace of M, because otherwise the behavior of the scheduling operation would be violated. Hence t' is a scheduled trace. ∎

ACKNOWLEDGMENTS

We thank Dexter Kozen for active encouragement and Keshav Pingali for useful discussions. The research reported here was supported by NSF Grant CCR-8817909 to Cornell University and an NSERC grant to McGill University.

REFERENCES

1. I. J. AALBERSBERG AND G. ROZENBERG, Theory of traces, *Theoret. Comput. Sci.* **60**, No. 1 (1988), 1–82.
2. S. ABRAMSKY, On semantic foundations for applicative multiprogramming, *in* "Proceedings, Tenth International Conference on Automata, Languages and Programming," (J. Diaz, Ed.), pp. 1–14, Springer-Verlag, New York, 1983.
3. S. ABRAMSKY, A generalized Kahn's principle, *in* "Proceedings, Fifth Workshop on Mathematical Foundations of Programming Semantics" (M. Mislove, Ed.), Lecture Notes in Computer Science, Springer-Verlag, New York, 1990.
4. K. R. APT AND E. R. OLDEROG, Proof rules and transformations dealing with fairness, *Sci. Comput. Programming* **3** (1983), 65–100.
5. K. R. APT AND G. D. PLOTKIN, Countable nondeterminism and random assignment, *J. Assoc. Comput. Mach.* **33**, No. 4 (1986), 724–767.

6. R. J. BACK, A continuous semantics for unbounded non-determinism, *Theoret. Comput. Sci.* **23**, No. 2 (1983), 187–210.
7. M. BEDNARCZYK, "Categories of Asynchronous Systems," Ph.D. thesis, University of Sussex, October 1987.
8. G. BERRY, P. L. CURIEN, AND J. J. LEVY, Full abstraction for sequential languages; the state of the art, *in* "Algebraic Methods in Semantics" (M. Nivat and J. Reynolds, Ed.), Chap. 3, pp. 89–132, Cambridge Univ. Press, Cambridge, UK, 1985.
9. J. D. BROCK AND W. B. ACKERMAN, Scenarios: A model of non-determinate computation, *in* "International Colloquium on Formalization of Programming Concepts" (J. Diaz and I. Ramos, Eds.), pp. 252–259, Lect. Notes in Comput. Sci., Vol. 107, Springer-Verlag, New York, 1981.
10. M. BROY, Fixed point theory for communication and concurrency, *in* "Formal Description of Programming Concepts II," p. 125–148, North-Holland, Amsterdam, 1983.
11. A. CHANDRA, Computable non-deterministic functions, *in* "Proceedings, 19th Annual Symposium of Foundations of Computer Science," pp. 127–131, IEEE, New York, 1978.
12. N. FRANCEZ, "Fairness," Springer-Verlag, New York/Berlin, 1986.
13. C. A. R. HOARE, "Communicating Sequential Processes," Series in Computer Science, Prentice–Hall International, London, 1985.
14. B. JONSSON, "Compositional Verification of Distributed Systems," Ph.D. thesis, Uppsala University, 1987.
15. B. JONSSON, Fully abstract trace semantics for dataflow networks, *in* "Proceedings, Sixteenth Annual ACM Symposium on Principles of Programming Languages, 1989."
16. G. KAHN, The semantics of a simple language for parallel programming, *in* "Information Processing 74," p. 993–998. North-Holland, Amsterdam, 1977.
17. G. KAHN AND G. PLOTKIN, "Domaines concrets," Rapport IRIA-LABORIA 336, 1978.
18. R. M. KELLER, Denotational models for parallel programs with indeterminate operators, *in* "Formal Description of Programming Concepts" (E. Neufeld, Ed.), pp. 337–366, North-Holland, Amsterdam, 1978.
19. R. M. KELLER AND P. PANANGADEN, Semantics of digital networks containing indeterminate operators, *Distrib. Comput.* **1**, No. 4 (1986), 235–245.
20. P. L. CURIEN, "Categorical Combinators, Sequential Algorithms and Functional Programming," Research Notes in Theoretical Computer Science, Wiley, New York, 1986.
21. N. A. LYNCH AND E. W. STARK, A proof of the Kahn principle for input/output automata, *Inform. and Comput.* **82**, No. 1 (1989), 81–92.
22. N. A. LYNCH AND M. TUTTLE, "Hierarchical Correctness Proofs for Distributed Algorithms," Technical Report MIT/LCS/TR-387, MIT Laboratory for Computer Science, April 1987.
23. A. MAZURKIEWICZ, Advanced course in Petri nets, *in* "Lecture Notes in Computer Science," Vol. 255, pp. 279–324, Springer-Verlag, New York/Berlin, 1986.
24. R. MILNER, "A Calculus for Communicating Systems," Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, New York/Berlin, 1980.
25. P. PANANGADEN AND V. SHANBHOGUE, "On the Expressive Power of Indeterminate Primitives," Technical Report 87–891, Computer Science Department, Cornell University, November 1987; *Inform. and Comput.*, to appear.
26. P. PANANGADEN AND V. SHANBHOGUE, Mccarthy's amb cannot implement fair merge, *in* "Proceedings, Eighth FSTTSC Conference," Lecture Notes in Comput. Sci., Vol. 338, pp. 348–363, Springer-Verlag, New York/Berlin, 1988.
27. P. PANANGADEN, V. SHANBHOGUE, AND E. W. STARK, Stability and sequentiality in dataflow networks, *in* "Proceedings, Seventeenth International Colloquium On Automata Languages and Programming" (M. S. Paterson, Ed.), pp. 308–321, Lecture Notes in Computer Science, Vol. 443, Springer-Verlag, New York/Berlin, 1990.
28. P. PANANGADEN AND E. W. STARK, Computations, residuals and the power of indeterminacy, *in* "Proceedings, Fifteenth International Colloquium on Automata Languages and Programming" (T. Lepisto and A. Salomaa, Eds.), pp. 348–363, Lecture Notes in Computer Science, Vol. 317, Springer-Verlag, New York/Berlin, 1988.

29. D. PARK, The "fairness problem" and non-deterministic computing networks, *in* "Proceedings, Fourth Advanced Course on Foundations of Computer Science-Distributed Systems" (J. de Bakker and L. van Leeuwen, Eds.), pp. 133–161, Mathematisch Centrum, Amsterdam, 1982.
30. G. D. PLOTKIN, Lcf considered a programming language, *Theoret. Comput. Sci.* **5**, No. 3 (1977), 223–256.
31. J. R. RUSSELL, On oracleizable networks and Kahn's principle, *in* "Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, 1990."
32. V. SHANBHOGUE, "The Expressiveness of Indeterminate Dataflow Primitives," Ph.D. thesis, Cornell University, 1990.
33. E. W. STARK, Semaphore primitives and starvation-free mutual exclusion, *J. Assoc. Comput. Mach.* **29**, No. 4 (1982), 1049–1072.
34. E. W. STARK, Concurrent transition system semantics of process networks, *in* "Proceedings, Fourteenth Annual ACM Symposium on Principles of Programming Languages," 1987, pp. 199–210.
35. E. W. STARK, Concurrent transition systems, *Theoret. Comput. Sci.* **64** (1989), 221–269.