# On the Kahn Principle and Fair Networks

Stephen Brookes

August 1998

CMU-CS-98-156

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Extended version of a paper presented at MFPS'98.
Submitted to *Theoretical Computer Science*.

This work was partially supported by the Office of Naval Research, under grant number N00014-93-1-0750, and by the National Science Foundation, under grant number CCR-9412980.

**Keywords:** communicating processes, dataflow, networks, denotational semantics, fairness, safety, liveness, non-determinism, parallel programming

## Abstract

The Kahn Principle states that each node in an asynchronous deterministic network computes a continuous function from input histories to output histories, and the behavior of the network can be characterized as a least fixed point. Fairness plays a vital but implicit role: the Kahn Principle is only sound when network execution is assumed to be (weakly) fair. Kahn's model does not extend easily to non-deterministic networks, since the obvious generalization to continuous relations on histories is not compositional. Previous attempts to model non-deterministic networks have sought to remain faithful to Kahn's spirit by retaining some form of continuity assumption; these approaches typically apply only to a limited class of network and do not deal adequately with fairness. We argue that for non-deterministic networks the assumption of continuity is not operationally justifiable, whereas fairness is still vital. We provide a compositional model for fair non-deterministic networks, based on trace sets which can be regarded as history relations "extended in time" to allow for the possibility of interference during execution. For a deterministic network one can extract the Kahn-style history function from the network's trace set, showing that our model is a natural generalization of Kahn's.

# 1 Introduction

Kahn networks [Kah77, KM77] provide an abstract model of the interactive behavior of systems of parallel asynchronous deterministic processes. A network can be viewed as a graph whose nodes represent computing agents and whose arcs represent communication channels. Each node performs some deterministic sequential computation, consuming input and producing output; channels are interpreted as unbounded buffers. Nodes are executed in parallel, subject to the obvious constraints that a node attempting to input from an empty channel must wait until the channel receives some input. The assumption of determinism has an obvious advantage, since it permits the use of an extremely simple and intuitive semantic model that abstracts away from operational details concerning execution order.

Kahn gave an elegant mathematical model of network behavior based on a simple semantic domain of *streams* or *histories*, which represent the actual or potential traffic along a communication channel. When $V$ is the set of data values appropriate for transmission along a channel, the corresponding domain of histories is the set $V^\infty = V^* \cup V^\omega$ of finite and infinite sequences, ordered by *prefix*. Operational intuition then suggests that each node computes a *continuous* function from the histories of its input channels to the histories of its output channels. Kahn's rationale for assuming continuity is based on the following intuitive remarks about the input-output behavior of a node:

- Each output is "caused" by the consumption of a finite amount of input.
- Availability of more input can only provoke more output.
- An infinite output "occurs" only as the limit of its finite prefixes.

Of course an infinite history should be regarded as "potential" rather than having actually occurred, since at any stage in execution only a finite amount of data can have been communicated so far. For example, a "buffer" process carrying data from the set $V$, with a single input channel and a single output channel, computes the identity function from $V^\infty$ to $V^\infty$. If supplied with a longer and longer sequence of input the buffer produces a correspondingly longer and longer sequence of output; the potential availability of infinite input is transformed into the potential for infinite output.

Taken collectively, the nodes of a network compute a tuple of mutually recursive continuous functions, with recursion reflecting feedback cycles in the network's communication graph. The *Kahn Principle* states that the operational behavior of the network corresponds to the input-output function obtained as the *least fixed point* of the corresponding functional [Tar55, Sco82]. This gives rise to a powerful methodology for reasoning about deterministic networks, using standard domain-theoretic fixed-point theorems.

Kahn's approach has several advantages. The programming notation used for nodes and networks is appealingly straightforward. The graphical notation is very intuitive, and gives rise to a simple network "calculus" based on a few natural graph-theoretic operations: *juxtaposition* (parallel composition of disjoint networks), *cascading* (linking the outputs of one network with the inputs of another), and *feedback* (feeding outputs from part of a network back to serve as inputs for another part of the network). Kahn's functional semantics for these network constructs is particularly simple: juxtaposition amounts to forming the product of two input-output functions, cascading amounts to composition of input-output functions, and feedback is handled by introducing a recursively defined history.

Although Kahn did not explicitly describe an operational semantics for networks, so that the validity of his Principle was not formally demonstrated, he did provide informal justification and a series of compelling examples. It was shown later that Kahn's semantics is sound with respect to an operational semantics based on "token-pushing" [Fau82].

## Notation

For illustration we adopt a notation similar to Kahn's, combining a CSP-like syntax for communication primitives with an Algol-like syntax for processes, subject to a few syntactic constraints enforcing determinism[1]. In contrast to CSP [Hoa78], communication is taken to be *asynchronous*: an output can occur "autonomously", but an attempt to input from an empty channel will block until data becomes available. As in Kahn's presentation, we use the keyword **process** rather than **procedure** (as in Algol), and each kind of node is specified as a process definition parameterized over the node's input and

[1]No process is allowed to attempt input simultaneously on two channels; no pair of nodes in a network is permitted to share an input channel or an output channel.

output channel names. Consequently, for the purposes of this paper we need only consider (the analogues of) first-order procedures. We also use local variables where necessary to represent internal data maintained by a node. For example the syntax **local** $h$ **in** $P$ describes a process $P$ equipped with a local variable $h$; the usual scoping conventions apply, so that for instance in

$$(\textbf{local}\ h\ \textbf{in}\ P)\|Q$$

the process $Q$ does not have access to the local variable.

Our notation is (usually implicitly) typed, with $\textbf{chan}[\tau]$ representing the type of channels carrying messages of datatype $\tau$, and $\textbf{var}[\tau]$ representing the type of variables of datatype $\tau$. Datatypes include **int** (integers), **bool** (truth values) and **unit** (the unit type, with sole member $\star$). We let **proc** be the type of "processes". Thus, for instance, a process definition with a single integer input channel and a single integer output channel would be given the procedure type $\textbf{chan}[\textbf{int}] \times \textbf{chan}[\textbf{int}] \to \textbf{proc}$.

## An example

To demonstrate Kahn's methodology, consider a family of networks built from "register", "duplicator", and "adder" nodes defined as follows:

$$
\begin{array}{l}
\textbf{process}\ \mathit{reg}(i, o) = \\
\quad \textbf{local}\ x\ \textbf{in} \\
\qquad (o!0;\ \textbf{while true do}\ (i?x;\ o!x)); \\
\textbf{process}\ \mathit{dup}(h, o_1, o_2) = \\
\quad \textbf{local}\ x\ \textbf{in} \\
\qquad \textbf{while true do}\ (h?x;\ o_1!x;\ o_2!x); \\
\textbf{process}\ \mathit{add}(i_1, i_2, o) = \\
\quad \textbf{local}\ x, y\ \textbf{in} \\
\qquad \textbf{while true do}\ (i_1?x;\ i_2?y;\ o!(x+y));
\end{array}
$$

These nodes may be instantiated and linked to form a "sum" network, by joining the output of an *add* node to the input of a *dup* node, joining the second output of the *dup* node to the input of a *reg* node, and joining the output of the *reg* node to the second input of the *add* node, as in Figure 1. The joined channels then become "internal", so that the overall network produced in this manner has a single input channel and a single output channel; this

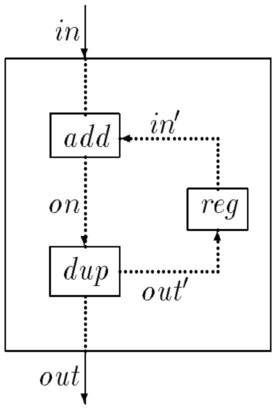


Figure 1: The *sum* network

is indicated in the Figure by the use of dotted lines for internal channels. Using the programming notation the above network can be represented as:

$$
\begin{array}{l}
\textbf{process}\ sum(in, out) = \\
\quad \textbf{local}\ in', on, out'\ \textbf{in} \\
\qquad add(in, in', on) \parallel dup(on, out, out') \parallel reg(out', in')
\end{array}
$$

Note the explicit localization of "internal" channels, and the use of parallel composition. The type of *sum* is $\textbf{chan}[\textbf{int}] \times \textbf{chan}[\textbf{int}] \rightarrow \textbf{proc}$.

Kahn's graphical notation abstracts away from certain syntactic details that become apparent when using the process language. For example, the above network can also be constructed in stages, corresponding to the following three alternative process definitions:

$$
\begin{array}{l}
\textbf{process}\ sum_1 = \\
\quad \textbf{local}\ out',\ in'\ \textbf{in} \\
\qquad (reg \parallel \textbf{local}\ on\ \textbf{in}\ (add \parallel dup)); \\
\textbf{process}\ sum_2 = \\
\quad \textbf{local}\ on,\ out'\ \textbf{in} \\
\qquad (dup \parallel \textbf{local}\ in'\ \textbf{in}\ (reg \parallel add)); \\
\textbf{process}\ sum_3 = \\
\quad \textbf{local}\ in',\ on\ \textbf{in} \\
\qquad (add \parallel \textbf{local}\ out'\ \textbf{in}\ (dup \parallel reg));
\end{array}
$$

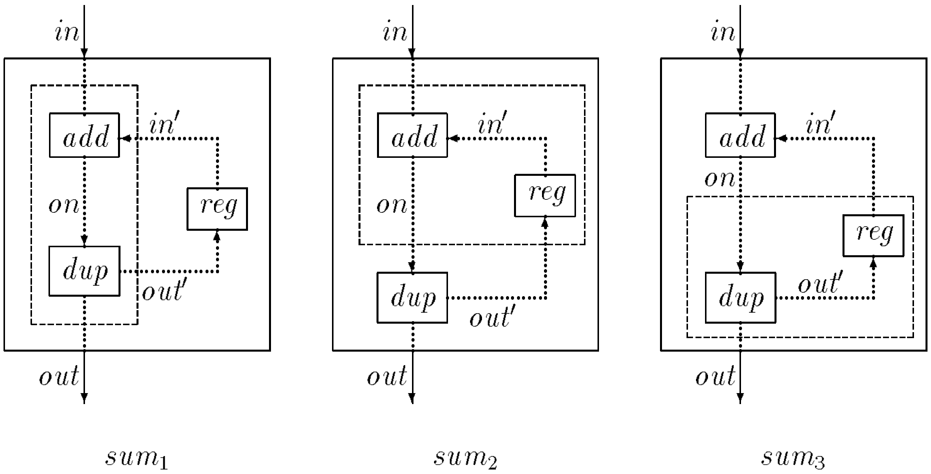


Figure 2: The networks $sum_1$, $sum_2$, and $sum_3$.

These alternative formulations represent what happens when the order of composition is chosen in the three obvious ways. For example in $sum_1$ we first "cascade" *add* onto *dup*, identifying the output channel of *add* with the input channel of *dup*; then cascade the second output of *dup* onto *reg*; then feed the output of the *reg* node back in as the second input to *add*. These three alternatives are displayed in Figure 2, using dashed lines to indicate subnetwork structure. Obviously these four networks ought to be behaviorally equivalent, and any reasonable semantic model should make this clear.

Following Kahn's approach, the functional behavior of the nodes is described as follows, using the standard list-manipulation primitives. For each node we specify a continuous function from input histories to output histories.

The equation defining the behavior of the *add* node is:

$$\begin{array}{rcl} F_{add}(in, in') & = & \textbf{if } in = \epsilon \vee in' = \epsilon \textbf{ then } \epsilon \textbf{ else} \\ & & \quad (hd(in) + hd(in')) :: F_{add}(tl(in), tl(in')) \end{array}$$

Here $in$ and $in'$ range over $V_{\mathrm{int}}^{\infty}$, and :: is the usual infix "cons" operator. This equation thus reflects the intuition that an *add* node waits for its two input channels to produce an integer, whereupon it consumes them, outputs their sum, and recurses. Actually this function definition is itself recursive, but it is easy to see that the least fixed point of the corresponding functional is the intended function $F_{add}$, representing the correct operational behavior.

The equations for *dup* and *reg* nodes are simpler: *dup* merely copies its input onto both of its output channels, so we will use the same equation for both output channels; *reg* outputs an initial zero and thereafter copies input to output. Letting *on* and *out'* range over the domain of integer histories, we therefore have:

$$\begin{array}{rcl} F_{dup}(on) & = & on \\ F_{reg}(out') & = & 0 :: out' \end{array}$$

Next we examine the network for *sum*, taking into account the channel linkages between nodes, leading to the following "network equations":

$$\begin{array}{rcl} on & = & F_{add}(in, in') \\ out & = & F_{dup}(on) \\ out' & = & F_{dup}(on) \\ in' & = & F_{reg}(out') \end{array}$$

By substitution and algebraic manipulation we can then extract the following recursive formulation for *out* as a function of *in*:

$$out \quad = \quad \textbf{if } in = \epsilon \textbf{ then } \epsilon \textbf{ else } hd(in) :: F_{add}(tl(in), out)$$

Kahn's semantics thus predicts that for a particular input history *in*, the output produced on channel *out* is the least fixed point of the continuous functional

$$G(in) = \lambda out.\ \textbf{if } in = \epsilon \textbf{ then } \epsilon \textbf{ else } hd(in) :: F_{add}(tl(in), out)$$

It can then be shown, by analyzing the least fixed point of this function for fixed values of *in*, that when *in* is a finite sequence $[v_1, \ldots v_n]$ the output is the sequence $[v_1, (v_1 + v_2), \ldots, (v_1 + \cdots + v_n)]$ of "prefix sums", and when *in* is infinite the output is the corresponding infinite sequence of prefix sums[2]. In particular, when *in* is empty so is *out*. Note that the causality relationship

[2]The proof for finite input is by induction on the length of *in*, using the fact that the least fixed point of $G(in)$ is the limit of the sequence built by iterating the application of $G(in)$ to the empty history. The case when *in* is infinite then follows by continuity.

between inputs and outputs is accurately represented by this simple functional description, in that the length of the output is always equal to the length of the input – each input triggers the availability of the next output[3].

The three alternative decompositions of the sum network each give rise to slightly different sets of functional equations, but in each case the equation defining *out* has the same least fixed point. This property relies on some elementary fixed point theory, in particular using Bekic's Theorem on the replacement of a simultaneous mutually recursive definition by a nested sequence of single recursions, so that one can "solve" the network equations in any order. This property, although seemingly obvious at an intuitive level, is actually important in justifying Kahn's use of graphical notation: even though the notation is syntactically ambiguous, the same graph representing many alternative "concrete" networks differing in the order in which the nodes are combined, the notation is semantically unambiguous.

As the above example shows, Kahn's semantics can be used to prove nontrivial properties of networks. In particular, although the semantics describes only complete (potential) histories, it still supports analysis of many *safety* and *liveness* properties. For example, it follows from the above analysis that the *sum* network satisfies the safety property that the output forms a nondecreasing sequence of integers, and the liveness property that if the input is infinite then so is the output.

## Operational considerations

Kahn's Principle can be interpreted as stating that the least fixed point characterization of network behavior is *operationally justified*, in that the input-output behavior predicted by the fixed-point construction is an accurate abstraction from operational behavior. Of course the reason why such an abstraction is desirable is obvious: to avoid having to reason about details of scheduling and timing that are beyond the programmer's control and would anyway complicate matters excessively. Although Kahn did not formally specify an operational semantics for networks it seems to have been gener-

[3]Actually the use of the word "always" is slightly misleading, since the functional description only characterizes the "final" output history that will be produced *eventually* when the network is supplied with a given input sequence. At various stages during execution the length of the output produced so far might lag behind the number of input items consumed so far, but the input-output history function is insensitive to such details.

ally accepted as almost obvious that Kahn's intuitions were sound. Indeed it has been shown that Kahn's model is consistent with a form of "token-pushing" operational semantics [Fau82], and for non-deterministic networks various models have been suggested and connections have been established with operational semantics based on I/O automata [Sta89, LS89].

Kahn tacitly assumed the existence of an operational semantics based on a *weakly fair* execution [Par79], so that every node that has not yet terminated will eventually be given a chance to run[4]. Any reasonable scheduling strategy, such as round-robin, has this property; assuming fairness thus allows us to abstract away from the details of any particular scheduler and this weak form of fairness is a valid abstraction from "realistic" network implementation. The soundness of Kahn's model relies on the fact that the input-output function of a deterministic network is independent of scheduling details, provided fairness is assumed. It can be shown that Kahn's semantics of deterministic networks is consistent with a standard operational semantics of networks, in which parallel composition is interpreted as fair interleaving. (See Appendix A for the relevant transition rules.)

Without this fairness assumption the Kahn Principle ceases to be valid, since it becomes impossible to justify the use of continuous *functions* to represent the abstract behavior of networks. For example, consider the prefix-sum network discussed earlier. If implemented unfairly, by a scheduler which fixates on the duplicator node, no output will ever get produced, contradicting the predicted functional behavior. If we abstract over *all* possible schedulers, including unfair ones, the best we can say about this network is that it computes a *relation* on histories, since more than one output history is possible for a given input history.

# 2 Limitations of Kahn's model

We next identify three major limitations: Kahn's model is *too abstract* for many purposes, lacking discriminatory power in many cases where there is a good operational reason to distinguish between processes; Kahn's model

[4]Kahn commented that "a parallel program can be safely simulated on a sequential machine, provided the scheduling algorithm is fair enough, i.e. it eventually attributes some more computing time to a process which wants it". He also remarked that if the scheduler is unfair the program might produce "less output than what could be expected".

applies only to *deterministic* systems; and the semantics lacks *homogeneity*.

## Too abstract

Kahn's semantics ignores many potentially significant attributes of a network's behavior, by focussing entirely on the relationship between complete input histories and complete output histories. To illustrate this consider the following "buffer" processes:

$$\begin{aligned}\textbf{process } \mathit{buff}(i,o) &= \textbf{local } x \textbf{ in while true do } (i?x; o!x);\\ \textbf{process } \mathit{buffs}(i,o) &= \textbf{local } h \textbf{ in } (\mathit{buff}(i,h) \parallel \mathit{buff}(h,o))\end{aligned}$$

The one-place buffer node $\mathit{buff}(i,o)$ clearly operates by repeatedly absorbing a single input datum and then outputting it, whereas the network $\mathit{buffs}(i,o)$ obtained by linking two such nodes in a chain, connected by a local channel, behaves like an unbounded finite buffer capable of absorbing an arbitrary number of inputs before outputting. Yet both compute the same input-output function, the identity function on $V^\infty$. The point here is that Kahn's semantics is too abstract to support reasoning about the stimulus-response attributes of a process. Although the one-place buffer node cannot absorb a second input item before it has emitted the first this property is not reflected in the node's input-output function. By identifying these two processes as semantically equivalent the model is incapable of distinguishing between them in any context, despite their operationally distinct characteristics.

## Determinism

Kahn's model is applicable only to deterministic networks. To ensure determinism Kahn imposed certain syntactic constraints, notably:

- at any given time, each node is either computing, or waiting for input on *one* of its input channels;
- each node is sequential.

Consequently, no node can ever be waiting for data to arrive on more than one input channel. For similar reasons nodes are not permitted to share output channels. These constraints were enforced syntactically, for instance by restricting the use of input and output inside nodes and forbidding parallel composition inside a node.

As Kahn realized, non-deterministic networks arise naturally in practice. For example, if we allow the sharing of an output channel it becomes possible to design a non-deterministic "merge" network capable of merging two input channels into a single output channel:

$$
\begin{array}{l}
\textbf{process}\ \mathit{merge}(\mathit{left}, \mathit{right}, \mathit{out}) = \\
\quad \textbf{local}\ x, y\ \textbf{in} \\
\quad\quad \phantom{\|}\textbf{while true do}\ (\mathit{left}?x;\ \mathit{out}!x) \\
\quad\quad \|\,\textbf{while true do}\ (\mathit{right}?y;\ \mathit{out}!y)
\end{array}
$$

Similarly, if we allow sharing of input channels one can design a "spraying" network that splits the input from one channel onto two output channels:

$$
\begin{array}{l}
\textbf{process}\ \mathit{spray}(\mathit{in}, \mathit{left}, \mathit{right}) = \\
\quad \textbf{local}\ x\ \ \mathrm{in} \\
\quad\quad \phantom{\|}\textbf{while true do}\ (\mathit{in}?x;\ \mathit{left}!x) \\
\quad\quad \|\,\textbf{while true do}\ (\mathit{in}?x;\ \mathit{right}!x)
\end{array}
$$

And if we allow a node to use a channel for both input and output it becomes possible to specify a bi-directional pipeline network[5]:

$$
\begin{array}{l}
\textbf{process}\ \mathit{pipe}(a, b) = \\
\quad \textbf{local}\ x, y\ \textbf{in} \\
\quad\quad \textbf{while true do}\ (a?x \| b?y;\ \ a!y \| b!x)
\end{array}
$$

Moreover, the prefix-sum network discussed earlier can easily be recast as a non-deterministic network, replacing the addition and duplication nodes by the following non-deterministic variants:

$$
\begin{array}{l}
\textbf{process}\ \mathit{dup}'(h, o_1, o_2) = \\
\quad \textbf{local}\ x\ \textbf{in} \\
\quad\quad \textbf{while true do}\ (h?x;\ \ (o_1!x \parallel o_2!x)); \\
\textbf{process}\ \mathit{add}'(i_1, i_2, o) = \\
\quad \textbf{local}\ x, y\ \textbf{in} \\
\quad\quad \textbf{while true do}\ ((i_1?x \parallel i_2?y);\ \ o!(x + y));
\end{array}
$$

[5]Of course in this example it no longer makes sense to characterize each stream as being either an input stream or an output stream.

We then obtain the non-deterministic network

$$
\begin{array}{l}
\textbf{process } sum'(in, out) = \\
\quad \textbf{local } in', on, out' \textbf{ in} \\
\quad\quad add'(in, in', on) \parallel dup'(on, out, out') \parallel reg(out', in')
\end{array}
$$

This network violates Kahn's syntactic restrictions, since the $add'$ node waits on two input channels simultaneously, and both $add'$ and $dup'$ nodes involve parallel composition. Yet it is intuitively obvious that the network still behaves deterministically, computing the same input-output function as before. Of course this cannot be shown within Kahn's functional framework. This provides a simple example in which a network built from non-deterministic components may still exhibit deterministic behavior.

It would be useful to extend Kahn's ideas to incorporate non-deterministic systems, but clearly continuous input-output functions no longer suffice. The actual behavior of a non-deterministic network may depend on scheduling details, in that the output produced from a given input stream may depend on the order in which individual nodes get executed, and such a network cannot properly be viewed as computing a *function* from input streams to output streams. Given the desire to abstract away from scheduling details, it makes sense instead to view each of these nodes as computing a *relation* on histories. For example, assuming a fair scheduler, the *merge* node should be regarded as computing the *fairmerge* relation [Par79], i.e. the set of all triples $(\alpha, \beta, \gamma)$ over $V^\infty$ such that $\gamma$ is an interleaving of all of $\alpha$ with all of $\beta$. Unfortunately the obvious relational generalization of Kahn's model lacks compositionality, as we will discuss later [BA81]. We mention this here simply to point out that Kahn's limitation to deterministic systems is inherent in his approach and cannot easily be overcome with a minor modification.

## Lack of homogeneity

Kahn's semantics treats nodes and networks in disparate ways [Kah77]. Each node determines a *flowchart*, from which its input-output function is extracted, and the input-output behavior of a network is then obtained by forming a mutually recursive family of functional equations and taking the least fixed point. This is a two-stage construction: first consider the individual nodes (in an essentially operational manner), then analyze the entire network (in a denotational manner). The separation into phases follows the same pattern as the syntactic constraints built into Kahn's framework: nodes execute sequential programs for which flowcharts can be used, whereas networks involve parallelism and do not correspond to flowcharts.

Perhaps it is less clear why one cannot simply treat nodes and networks on a more equable basis, by giving a denotational (compositional) description of the input-output semantics of the sequential programming language used "inside" nodes. It turns out that this is impossible, because the input-output semantics of a sequential composition $N_1; N_2$ cannot be deduced from the input-output semantics of $N_1$ and $N_2$. A simple example shows this: the input-output functions of $i?x$ and **skip** coincide (both equalling $\lambda\rho \in V^{\infty}.\epsilon$), but the input-output functions of $i?x; o!0$ and **skip**; $o!0$ differ (only the latter maps the empty input sequence to $[0]$).

This lack of homogeneity is aesthetically unattractive, since it would be more natural to treat nodes and networks on exactly the same semantic footing, and this is required in order to perform hierarchical network analysis. Indeed, for this very reason Kahn suggested that it is sometimes desirable to treat a subnetwork as a single node. However, taken literally and exploited in full generality this is inconsistent with the constraints imposed to ensure determinism: a subnetwork built from deterministic nodes may fail to satisfy the determinism constraints. For example, consider the network obtained by juxtaposing two disjoint one-place buffers:

$$\mathit{buff}(i_1, o_1) \parallel \mathit{buff}(i_2, o_2),$$

where we assume for simplicity that each channel has the same type $\mathbf{chan}[\tau]$. Each buffer node by itself is deterministic, and the network built this way is perfectly well-behaved, computing the identity function on $V_\tau^{\infty} \times V_\tau^{\infty}$. However, this network obviously waits for input on *two* channels, because of the use of parallel composition. When viewed as a single node it therefore violates the original intention that nodes be sequential and that nodes wait on at most one channel at a time.

Rather than dismiss these issues as minor quibbles we feel that they indicate that lack of homogeneity is a serious methodological problem.

# 3 Generalizing Kahn

As we indicated above, a non-deterministic network can be regarded as computing an *input-output relation*. However, it is well known that Kahn's Principle does not immediately generalize to the relational setting in the obvious way, because the input-output relation of a non-deterministic network cannot be defined compositionally. We summarize briefly the classic Brock-Ackerman anomaly that demonstrates the problem [BA81].

## The Brock-Ackerman anomaly

Let us write $str[\![N]\!]$ for the history relation computed by network $N$. Consider the following pair of nodes, with input channel $i$ and output channel $o$:

$$\begin{array}{lcl}\textbf{process } P_1(i,o) & = & \textbf{local } x,y \textbf{ in } (i?x; o!x; i?y; o!y) \\ \textbf{process } P_2(i,o) & = & \textbf{local } x,y \textbf{ in } (i?x; i?y; o!x; o!y)\end{array}$$

Clearly these nodes compute different relations, because $P_1$ needs only a single input datum to trigger its first output but $P_2$ needs two inputs. Thus

$$str[\![P_1]\!] \neq str[\![P_2]\!].$$

Now consider the following context $S[-]$, with a "hole" into which we may plug $P_1$ or $P_2$:

$$\begin{array}{lcl}S[-] & = & \textbf{local } on, on', i \textbf{ in} \\ & & \qquad double(in, on) \parallel double(in', on') \parallel merge(on, on', i) \parallel [-]\end{array}$$

where *merge* is the merge process defined earlier and *double* is given by:

$$\textbf{process } double(in, on) \;=\; \textbf{local } z \textbf{ in } (in?z; on!z; on!z)$$

The networks formed by substituting $P_1$ and $P_2$ into this context are shown in Figure 3. Neither of the networks $S[P_1]$ or $S[P_2]$ produces any output unless it receives input. If one or more input items are available on either input channel the *double* nodes ensure that the internal $P_k$ node receives at least two inputs, thus masking the difference between $P_1$ and $P_2$, so that

$$str[\![S[P_1]]\!] = str[\![S[P_2]]\!].$$

Now consider the following context:

$$
\begin{array}{rcl}
T[-] & = & \textbf{local } o, in', out' \textbf{ in} \\
 & & \quad \mathit{spread}(o, out, out') \parallel \mathit{plus1}(out', in') \parallel [-]
\end{array}
$$

where the nodes *spread* and *plus1* are given by

$$
\begin{array}{rcl}
\textbf{process } \mathit{spread}(o, out, out') & = & \textbf{local } z \textbf{ in} \\
 & & \quad \textbf{while true do } (o?z; out!z; out'!z)
\end{array}
$$

$$
\begin{array}{rcl}
\textbf{process } \mathit{plus1}(out', in') & = & \textbf{local } z \textbf{ in} \\
 & & \quad \textbf{while true do } (out'?z; in'!(z+1))
\end{array}
$$

Plugging in $S[P_1]$ or $S[P_2]$ into this context produces networks $T[S[P_1]]$ and $T[S[P_2]]$, also shown in Figure 3. These networks have *different* input-output behavior, i.e.

$$
\mathit{str}[\![T[S[P_1]]]\!] \neq \mathit{str}[\![T[S[P_2]]]\!].
$$

For instance, suppose the network $T[S[P_1]]$ is supplied with a single input value 5 on channel *in*. This value will pass through the first *double* node, then through *merge*, through $P_1$ and through *spread* to become the first output on channel *out*. The *spread* node will also send a 5 to *plus1*, causing a 6 to appear on $in'$, and the second *double* node can thus pass a 6 on to the *merge* node. The merge node now has a choice of consuming either the second 5 or this 6. Consequently the network can output a 5 followed either by a 5 or a 6. However, if $T[S[P_2]]$ is supplied with a single input value 5 its $P_2$ node cannot produce output until it has received a second input; thus eventually (by fairness) the *merge* node must consume both of the 5's produced by the first *double* node, and the only possible output of the network begins with 5 followed by 5.

Thus we have two networks with the same history relation but which induce different history relations when used as components in a larger network. The conclusion is that the input-output relation of a non-deterministic network cannot be computed compositionally from the input-output relations of its components. Note also that this failure of compositionality cannot be sidestepped by banishing or limiting the use of the relevant program construct: the problem occurs with the fundamental network primitives (juxtaposition, feedback and cascading).

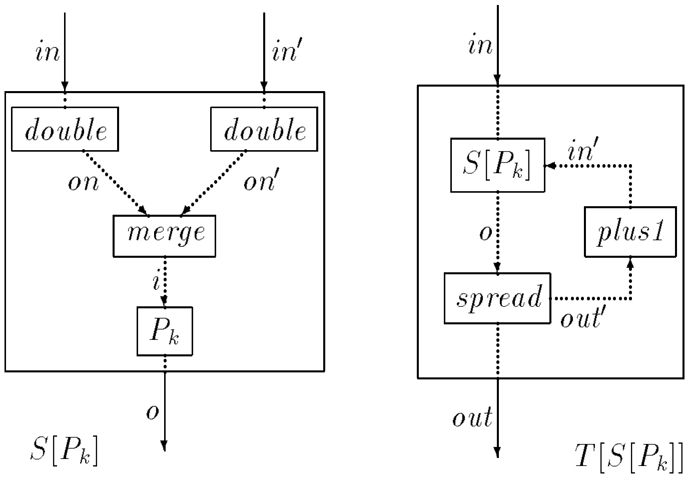


Figure 3: The networks $S[P_k]$ and $T[S[P_k]]$ (for $k = 1, 2$)

This anomaly led to the search for an appropriate compositional semantics generalizing Kahn's model to incorporate non-determinism. A variety of models has been proposed, including hiatons [Fau82], scenarios [BA81], I/O automata [Sta89, RT89], and sets of continuous functions [Abr90]. Various trace-theoretic models have also been proposed, including [KP84, Kok87, Jon89, Rus90]. Although each of these models attempted to stay faithful to Kahn's spirit, typically retaining some form of continuity assumption, none is as simple and elegant as Kahn's original model. Moreover these approaches have achieved only limited success, usually being incapable of properly modelling fairness. Rather than reviewing the details of these previous models we will re-examine the operational rationale behind Kahn's original model and show that in the non-deterministic setting the rationale ceases to be justifiable.

## Is continuity a fair assumption?

We argued above, echoing Kahn, that it is operationally reasonable to model the behavior of a deterministic node as a continuous function from input streams to output streams. A key point in rationalizing the decision to assume continuity was that a deterministic process capable of generating an infinite output sequence must in fact generate successively longer finite prefixes of its output from successively longer prefixes of its input. If we imagine running the process repeatedly from the start, each time supplying a longer portion of the input sequence, we would expect to observe a correspondingly longer portion of the output, since the node is deterministic and therefore executing the same code in each run. In the limit, if an infinite supply of input is available the node would eventually produce the entire infinite output. However, if the process is non-deterministic this argument ceases to be valid, since there is no longer any guarantee that the process behaves the same way in different runs when supplied with the same (or longer) input sequence. Thus the rationale for assuming continuity is no longer justified in the non-deterministic setting.

Another example helps to make this point. Consider the following three kinds of buffer-like node, assuming that each channel has type **chan**[**int**]:

$$\begin{array}{rcl}
\textbf{process}\ \mathit{buff}(i,o) & = & \textbf{local}\ x\ \textbf{in} \\
 & & \quad \textbf{while true do}\ (i?x; o!x) \\[1ex]
\textbf{process}\ \mathit{buff}'(i,o) & = & \textbf{local}\ x\ \textbf{in} \\
 & & \quad \textbf{while true do}\ (\textbf{skip or}\ (i?x; o!x)) \\[1ex]
\textbf{process}\ \mathit{buff}_*(i,o) & = & \textbf{local}\ x, n\ \textbf{in} \\
 & & \quad n{:=}?; \textbf{for}\ i{:=}1\ \textbf{to}\ n\ \textbf{do}\ (i?x; o!x)
\end{array}$$

Here $n{:=}?$ is assumed to be a random assignment setting $n$ to an arbitrary non-negative integer. Intuitively *buff* – as discussed earlier – is a conventional one-place buffer, and is obviously deterministic; $\mathit{buff}'$ is a non-deterministic node that keeps making a choice either to behave like a buffer for one step or to "skip"; and $\mathit{buff}_*$ chooses an arbitrary finite bound on the number of times it will behave like a buffer, after which it stops inputting. Non-determinism in these latter two cases means that the node has more than one possible execution, and for a given input history the output depends on which execution occurs. For $\mathit{buff}'$ it is clear that the output will always be a *prefix* of the input, and that for each input sequence there is an execution that faithfully outputs all of the input, even if the input is infinite. For $\mathit{buff}_*$ again the output is a prefix of the available input, but only a *finite* sequence of output will be produced. The stream relations computed by these nodes are, correspondingly:

$$\begin{array}{rcl} str[\![\mathit{buff}(i,o)]\!] & = & \{(\rho,\rho) \mid \rho \in V^{\infty}\} \\ str[\![\mathit{buff}'(i,o)]\!] & = & \{(\rho,\sigma) \mid \sigma \leq \rho \ \&\ \rho,\sigma \in V^{\infty}\} \\ str[\![\mathit{buff}_*(i,o)]\!] & = & \{(\rho,\sigma) \mid \sigma \leq \rho \ \&\ \rho \in V^{\infty} \ \&\ \sigma \in V^*\}. \end{array}$$

We write $\sigma \leq \rho$ to indicate that $\sigma$ is a prefix of $\rho$.

The important point to note here is that in the third case the relation is *not* continuous, since the presence of the input-output pair $(0^k, 0^k)$ for all $k \geq 0$ does not imply the presence of $(0^\omega, 0^\omega)$. Indeed, there is no operational justification for forcing the input-output relation of $\mathit{buff}_*(i,o)$ to contain $(0^\omega, 0^\omega)$, because each of the "approximations" $(0^k, 0^k)$ represents a behavior observed along a *different* computation, and no single computation exists along which, if infinitely many 0's were available as input, infinitely many outputs would also occur. Contrast this with $\mathit{buff}'(i,o)$, which *does* have a computation involving infinite input and output, since it is possible for the node to keep choosing the "active" branch. If we chose to enforce continuity, thereby equating $\mathit{buff}_*$ with $\mathit{buff}'$, we would be forced to ignore the fact that these two nodes are not operationally equivalent.

# 4 A model for fair networks

We have already mentioned the fundamental role played by fairness in the operational underpinnings of Kahn's semantics. For non-deterministic networks fairness is, of course, still fundamental, again providing us with a way to abstract away from irrelevant scheduling details. We propose a model of *fair networks*, in which nodes are (possibly non-deterministic) computing agents, communicating asynchronously on buffered channels, executing fairly. We allow full use of sequential and parallel composition, both at the node level and at the network level. We allow sharing of input- or output-channels, and we even allow channels to be used in a bi-directional manner. We treat nodes and networks homogeneously, so that from now on we will only use the neutral term "process".

Our model is an adaptation of *transition traces* [Bro93] to incorporate asynchronous communication, along lines sketched in [Bro97]. Each process denotes a *trace set*, amounting intuitively to an "input-output relation extended in time"; we do not impose any continuity constraint. The trace set of an entire network is obtained by *fair parallel composition* of the trace sets of its nodes. We provide a fixed-point characterization of fair parallel composition, thus making good on our claim that we maintain the spirit of Kahn's Principle. The trace set of a recursively defined network is also characterized as a fixed-point. Our semantics is operationally justified in that the traces of a network correspond precisely to fair executions of the network, as prescribed by a standard operational semantics outlined in Appendix A.

Our approach is compositional: in particular, sequential composition amounts to concatenation of trace sets, and we no longer need to treat nodes and networks separately. Trace semantics is useful for safety and liveness analysis, as well as general analysis of the stimulus-response behavior of networks. Fairness is often a vital assumption in liveness arguments, and our model is well suited for this purpose. Because our semantics is homogeneous we support hierarchical analysis. We can also handle dynamic networks, in which the number of active processes changes as the network evolves.

Moreover, for networks in which each channel is used unambiguously by each node either for input or for output we can extract an *input-output relation* from the network's trace set. In the case of a deterministic network this coincides with the graph of the input-output function predicted by Kahn's semantics, so that we do indeed obtain a true generalization of Kahn's semantics.

We provide here only a sketch of the main semantic ideas and definitions. For fuller details the reader should consult [Bro93, Bro96, Bro97]. Even without detailed analysis of the semantic definitions it should be possible to understand the general concepts and see how our semantics works out when applied to the examples under discussion.

## States

A key feature in our framework is the way we model state. A *state* of a network is a tuple $w = (\bar{v}, \bar{\rho})$ giving the values of all non-local variables used in the network and the current contents of channels. We assume that each variable and channel is typed, and we let $V_\tau$ be the set of data values of type $\tau$. Since channels are modelled as unbounded queues the contents of a channel of type **chan**$[\tau]$ will belong to the set $V_\tau^*$ of finite sequences. A typical *state set* is thus a cartesian product of form $W = (V_1 \times \cdots V_m) \times (V_{m+1}^* \cdots \times V_n^*)$. Our semantic definitions are parameterized in terms of the choice of current state set. This permits smooth handling of local variable declarations [Rey81, Ole82]. For each type $\theta$ (such as **proc**, **var**$[\tau]$, and **chan**$[\tau]$) and each state set $W$ we define a set of meanings of type $\theta$ "over" $W$.

## Variables and channels

A variable of type $\tau$ can be modelled as an *acceptor*, a function *acc* of type $V_\tau \to (W \to W)$ describing the effect of assignment to the variable, together with an *expression value*, i.e. a function *val* of type $W \to V_\tau$ describing the (state-dependent) current value of the variable. This is exactly as in the Reynolds/Oles semantics of Idealized Algol [Rey81, Ole82].

Similarly a channel can be modelled as a *put* operation, i.e. a function of type $V_\tau \to (W \to W)$ describing the effect of "sending" to the channel, together with a *get* operation, a function of type $W \to (V_\tau \times W)\mathit{option}$ describing the effect of "receiving" from the channel. We use the mathematical analog here of the ML "option" type constructor; thus for any state $w$ attempting a *get* will either produce $\mathit{some}(v, w')$ or *none*. In the former case $v$ is the "next" remaining item in the channel's queue, and $w'$ is the state produced by removal of that item. The *none* case occurs when the channel queue is empty.

For example, over state set $W \times V^*$ the channel corresponding to the second component is represented by the pair $(\mathit{put}, \mathit{get})$ with the functions

$$\begin{array}{rcl} \mathit{put} & : & V \to W \times V^* \to W \times V^* \\ \mathit{get} & : & W \times V^* \to (V \times (W \times V^*))\mathit{option} \end{array}$$

given by

$$\begin{array}{rcl} \mathit{put}(v)(w, \rho) & = & (w, \rho v) \\ \mathit{get}(w, \epsilon) & = & \mathit{none} \\ \mathit{get}(w, v\rho) & = & \mathit{some}(v, (w, \rho)) \end{array}$$

for all $\rho \in V^*$, $w \in W$, and $v \in V$. A "put" appends the new item at the end of the current queue and a "get" removes the first item of the current queue. Note that we use the notation $\rho v$ or $v\rho$ as appropriate.

## Processes

A *trace* of a process is a finite or infinite sequence of state changes

$$\langle w_0, w'_0\rangle\langle w_1, w'_1\rangle \ldots \langle w_n, w'_n\rangle \ldots$$

representing a fair interactive computation; each *step* $\langle w_i, w'_i\rangle$ models a finite sequence of atomic actions executed by the process, and each "external" change from $w'_i$ to $w_{i+1}$ represents a state change caused by the process's "environment". With our view of state and channels, communication causes a state change. We model an attempt to input from an empty channel as a "busy wait", i.e. an infinite trace consisting entirely of stuttering steps.

A process denotes a *trace set*, intuitively a total relation on states, extended in time to allow for the potential for interference. Such a trace set specifies a complete recipe for predicting all possible fair interactive computations of a process. Trace sets are *closed* under stuttering and mumbling, so that for instance

$$\begin{array}{ll} \alpha\beta \in t \ \&\ w \in W \ \Rightarrow \alpha\langle w, w\rangle\beta \in t & \textit{stuttering} \\ \alpha\langle w, w'\rangle\langle w', w''\rangle\beta \in t \ \Rightarrow\ \alpha\langle w, w''\rangle\beta \in t & \textit{mumbling} \end{array}$$

We write $T^{\dagger}$ for the closure of $T$, defined as the smallest closed set of traces containing $T$ as a subset[6].

Closed trace sets form a domain, ordered by reverse inclusion. This ordering can be regarded as a measure of non-determinism: the least element is the set of all traces, corresponding to the most non-deterministic process of all.

Although traces themselves – as a form of sequence – form a domain under the prefix ordering, our domain of closed trace sets is *not* constructed as a *powerdomain* [Smy78] over a domain of traces. The use of powerdomains would cause difficulties with the incorporation of fairness and in any case a simpler model serves our purposes.

Note also that we do *not* assume that the trace sets denoted by processes are *prefix-closed* or *closed under limit*, in contrast to most traditional semantic models of CSP-like languages [Ros98]. This is because we use a trace to represent an *entire* computation: incomplete or partial traces do not occur

[6] The closure conditions specified above generalize in the obvious way to allow stuttering and mumbling at infinitely many positions in an infinite trace.

in our trace sets. Enforcing closure under limit, so that an infinite trace is deemed to be present if each of its finite-length approximants is present, would cause difficulty with fairness. By working with "complete" traces we are able to avoid this problem.

## Environments

An *environment* over state set $W$ is a finite mapping from identifiers to variables over $W$, and channel identifiers to channels over $W$; each variable and channel corresponds to a component of the state. For the most part in this presentation we will suppress details of binding and scope, since the ideas can be conveyed more simply by factoring out these book-keeping details when we discuss specific examples.

## Semantic definitions

We now define the trace semantics of processes. For simplicity we assume that *expressions* (such as $x + y$) are evaluated atomically, cause no side-effects, and always terminate. It would be straightforward to adapt the semantic definitions to allow for fully general expression evaluation, as in [Bro93].

Whenever $P$ is a process, $W$ is a set of states, and $u$ is an environment mapping the free identifiers of $P$ into variables and channels over $W$, the trace set

$$traces[\![P]\!]Wu$$

is defined as follows, by structural induction on $P$. In each case it is to be understood that the trace set being defined also includes all traces obtained by stuttering and mumbling from traces mentioned explicitly. In cases where $W$ and $u$ can be assumed known we may refer simply to $traces[\![P]\!]$.

- The process **skip** has traces of form $\langle w_0, w_0\rangle \dots \langle w_k, w_k\rangle$, i.e. finite stuttering, reflecting termination without changing the state, regardless of interruption. These traces are obtainable from singleton stuttering traces by closure, so that $traces[\![\textbf{skip}]\!]Wu = \{\langle w, w\rangle \mid w \in W\}^\dagger$.
- If $h$ denotes the channel $(put, get)$ and $x$ denotes the variable $(acc, val)$,

the input command $h?x$ has traces of form

$$\begin{array}{ll} \langle w, acc(v)w'\rangle & \text{where } get(w) = some(v, w') \\ \langle w_0, w_0\rangle \ldots \langle w_k, w_k\rangle \ldots & \text{where } \forall k \geq 0.\ get(w_k) = none \end{array}$$

- When $h$ denotes the channel $(put, get)$ the output command $h!v$ has traces of form $\langle w, put(v)w\rangle$.

- Sequential composition corresponds to concatenation of traces:

  $$traces[\![P_1; P_2]\!]Wu = \{\alpha_1\alpha_2 \mid \alpha_1 \in traces[\![P_1]\!]Wu \ \&\ \alpha_2 \in traces[\![P_2]\!]Wu\}^\dagger$$

- Parallel composition corresponds to fair merging of traces

  $$\begin{array}{rcl} traces[\![P_1 \| P_2]\!]Wu & = & \{\gamma \mid \exists\alpha \in traces[\![P_1]\!]Wu,\ \beta \in traces[\![P_2]\!]Wu. \\ & & \quad (\alpha, \beta, \gamma) \in fairmerge_{W\times W}\}^\dagger, \end{array}$$

  where for each set $A$ the relation $fairmerge_A \in \mathcal{P}(A^\infty \times A^\infty \times A^\infty)$ is given by

  $$fairmerge_A \ = \ both^* \cdot one \ \cup \ both^\omega$$

  where

  $$\begin{array}{rcl} both & = & \{(\alpha, \beta, \alpha\beta), (\alpha, \beta, \beta\alpha) \mid \alpha, \beta \in A^+\} \\ one & = & \{(\alpha, \epsilon, \alpha), (\epsilon, \beta, \beta) \mid \alpha, \beta \in A^\infty\} \end{array}$$

  Here we have used the obvious generalizations of concatenation and finite and infinite iteration to triples of traces, and to sets of triples. $A^+$ is the set of non-empty finite sequences over $A$. Thus in particular every triple $(\alpha, \beta, \gamma) \in fairmerge_A$ for which both $\alpha$ and $\beta$ are infinite can be decomposed into the form

  $$\begin{array}{l} \alpha = \alpha_1\alpha_2 \ldots \\ \beta = \beta_1\beta_2 \ldots \\ \gamma = \alpha_1\beta_1\alpha_2\beta_2 \ldots \end{array}$$

  or its symmetric variant $\gamma = \beta_1\alpha_1\beta_2\alpha_2 \ldots$, each $\alpha_i$ and $\beta_i$ being non-empty finite sequences. The case where one or more of $\alpha$ and $\beta$ is finite has a similar decomposition. This formal specification of *fairmerge* clearly corresponds precisely to the fair merge operation mentioned earlier in the *merge* example.

The *fairmerge* relation can also be characterized as the least fixed point of the functional

$$\lambda R.\, both \cdot R \cup one,$$

where $R$ ranges over the lattice of relations over $A^\infty \times A^\infty \times A^\infty$, ordered by reverse inclusion. The least element of this lattice is the universal relation, and the fixed point can be calculated as an intersection.

- The traces of

  $$\textbf{local } h : \textbf{chan}[\tau] \textbf{ in } P$$

  over state set $W$ are obtained by projection onto $W$ from the traces of $P$ over $W \times V_\tau^*$, in a suitably expanded environment, along which the local channel is initially empty and its contents are never changed across step boundaries. In the expanded environment $h$ is bound to the channel value $(put, get)$ whose operations refer to the $V_\tau^*$ component of the expanded state set $W \times V_\tau^*$.

  This definition generalizes in an obvious manner to **local** $h = \rho$ **in** $P$, in which the initial value assumed for the local channel is the given sequence $\rho \in V_\tau^*$.

  The semantics of **local** $x : \textbf{var}[\tau]$ **in** $P$ is defined in a similar manner.

- For simplicity we assume that recursive process definitions are syntactically *guarded*, in that each occurrence of the recursive process name is preceded by a communication or some other atomic action[7]. For each state set $W$ such a recursive process definition determines a guarded continuous function $F$ on trace sets over $W$, and denotes the (closure of the) least fixed point of this function. Recall that trace sets over $W$ form a domain under reverse set inclusion, with least element the set of all traces $(W \times W)^\infty$. The fixed point can therefore be calculated by forming the intersection of the sets $F^n((W \times W)^\infty)$ for $n \geq 0$.

[7]This constraint can be removed, allowing arbitrary recursive definitions, if we insert a "semantic guard" in the form of an initial stuttering step. The mechanics of this approach are explained in [Bro96]. Note that for technical reasons it is necessary to take the fixed point of a functional over *arbitrary* trace sets, and then to form the closure. This guarantees that a divergent process definition such as **process** $div =$ **skip**; $div$ is given the correct denotation $\{\langle w, w\rangle \mid w \in W\}^\omega$, i.e. infinite stuttering, rather than $(W \times W)^\omega$.

For example, assume that the state set is $V^* \times V^*$ and that $a$ and $b$ correspond to the first and second components respectively. The recursive process definition

$$\mathbf{process}\ B = \mathbf{local}\ x\ \mathbf{in}\ (a?x; b!x; B),$$

essentially the one-place buffer example discussed earlier, determines the guarded function

$$F(t) = (a?\epsilon)^\omega \ \cup \ \bigcup_{v \in V} a?v; b!v; t,$$

where we write

$$\begin{array}{rcl} a?\epsilon & = & \{\langle(\epsilon, \sigma), (\epsilon, \sigma)\rangle \mid \sigma \in V^*\} \\ a?v & = & \{\langle(v\rho, \sigma), (\rho, \sigma)\rangle \mid \rho, \sigma \in V^*\} \\ b!v & = & \{\langle(\rho, \sigma), (\rho, \sigma v)\rangle \mid \rho, \sigma \in V^*\}. \end{array}$$

The least fixed point of this function is, as expected by the intended operational behavior, the trace set given by:

$$\{a?v; b!v \mid v \in V\}^\omega \ \cup \ \{a?v; b!v \mid v \in V\}^*(a?\epsilon)^\omega.$$

- For a conditional command we define

  $$traces[\![\mathbf{if}\ B\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2]\!] \ = \ [\![B]\!]_{\text{true}}; traces[\![P_1]\!] \cup [\![B]\!]_{\text{false}}; traces[\![P_2]\!],$$

  where $[\![B]\!]_{\text{true}} = \{\langle(w, w)\rangle \mid w \in W \ \& \ [\![B]\!]w = \text{true}\}$. Here we write $[\![B]\!] : W \rightarrow V_{\text{bool}}$ for the semantic function for boolean expressions, which is assumed given. We use a similar convention for $[\![B]\!]_{\text{false}}$.

- The meaning of a while-loop involves iteration:

  $$\begin{array}{rcl} traces[\![\mathbf{while}\ B\ \mathbf{do}\ P]\!] & = & ([\![B]\!]_{\text{true}}; traces[\![P]\!])^*; [\![B]\!]_{\text{false}} \\ & \cup & ([\![B]\!]_{\text{true}}; traces[\![P]\!])^\omega. \end{array}$$

  Equivalently, this trace set is the closure of the least fixed point of the functional

  $$\lambda T.\ [\![B]\!]_{\text{true}}; traces[\![P]\!]; T \ \cup \ [\![B]\!]_{\text{false}}.$$

  For example, the traces of **while true do skip** are infinite stuttering sequences, i.e.

  $$traces[\![\mathbf{while\ true\ do\ skip}]\!] = \{\langle(w, w)\rangle \mid w \in W\}^\omega.$$

  Similarly, the traces of the one-place buffer node *buff* (specified as a while-loop) are as expected.

# 5 Advantages of trace semantics

Our semantics is specifically designed to handle *non-deterministic* networks, is *compositional*, and permits operationally justified *discrimination* between processes. Thus we avoid the problems inherent in Kahn's approach.

## Compositionality

With the trace semantics the Brock-Ackerman anomaly does not cause a problem: although $P_1$ and $P_2$ denoted the same input-output function they do *not* have the same traces, so it is unsurprising that they also induce different trace semantics in context $S[-]$ and also in context $T[-]$. Similarly the problem with sequential composition goes away, since $i?x$ and **skip** do not have the same traces. In fact we are able (as above) to specify the traces of $N_1; N_2$ in terms of the traces of $N_1$ and the traces of $N_2$.

Trace semantics is compositional, so that it can be used to support syntax-directed reasoning about non-deterministic processes. In particular, our semantics supports a hierarchical approach to network analysis and synthesis. When analyzing a network built out of several sub-networks all one needs to know or assume about a sub-network is its trace set. One can replace any node or sub-network by another with the same traces, without affecting the traces of the overall network.

## Discriminative power

The variant forms of one-place buffer discussed earlier have pairwise distinct trace sets. Adapting the notation introduced above, we have (modulo closure):

$$\begin{aligned} \mathit{traces}[\![\mathit{buff}(i,o)]\!] &= \{i?v; o!v \mid v \in V\}^\omega \ \cup\ \{i?v; o!v \mid v \in V\}^*(i?\epsilon)^\omega \\ \mathit{traces}[\![\mathit{buff}'(i,o)]\!] &= \{i?v; o!v \mid v \in V\}^\omega \ \cup\ \{i?v; o!v \mid v \in V\}^* STUT^\omega \\ \mathit{traces}[\![\mathit{buff}_*(i,o)]\!] &= \{i?v; o!v \mid v \in V\}^+ \ \cup\ \{i?v; o!v \mid v \in V\}^*(i?\epsilon)^\omega \end{aligned}$$

where $STUT = \{\langle(\rho,\sigma),(\rho,\sigma)\rangle \mid \rho,\sigma \in V^*\}$.

For *buffs* we need the obvious extension of the above notation to non-empty finite sequences of inputs and outputs, so that when $\rho = [v_1, \ldots, v_n]$ we write $i?\rho$ for $i?v_1; \ldots i?v_n$. The trace set of *buffs* consists of all traces of the form

$$\begin{array}{rl} & i?\rho_1; o!\sigma_1; \dots i?\rho_k; o!\sigma_k \dots \\ \text{or} & i?\rho_1; o!\sigma_1; \dots i?\rho_n; o!\sigma_n; (i?\epsilon)^\omega \end{array}$$

such that

- every input is eventually output, i.e.

$$\forall k. \exists m \geq k.\ \rho_1 \dots \rho_k \leq \sigma_1 \dots \sigma_m.$$

- every output was previously input, i.e.

$$\forall k.\ \sigma_1 \dots \sigma_k \leq \rho_1 \dots \rho_k.$$

Here the $\rho_k$ and $\sigma_k$ range over all non-empty finite sequences, $k$ ranges over the positive integers, and $n$ ranges over the natural numbers.

In particular we are not forced (by any desire to impose continuity) to equate any pair of these processes. For each pair there is a good operational reason to avoid such identification, and our semantics reflects this well. These buffer examples show that the trace semantics permits distinctions to be made between processes based on their stimulus-response behavior.

# 6 Laws of process equivalence

A significant benefit of Kahn's approach is that input-output functions are easy to deal with; for instance, cascading corresponds to composition of input-output functions. Moreover one can appeal to a battery of standard fixed-point theorems (due to Scott, Bekic, Vuillemin and others) for help in proving equivalences between networks and in proving correctness of a network with respect to a specification.

Trace sets are obviously more complex mathematically than input-output functions or input-output relations. Nevertheless a trace set can be regarded as an input-output relation "extended in time", in which state changes are "strung along" in a sequence and the potential for interference is built in. This description, albeit informal, expresses an intuition helpful when trying to understand our semantics. Moreover, trace set specifications such as those given for the buffer processes can be very helpful.

Rather than relying on the semantic definitions themselves directly in reasoning about network behavior we prefer to list a number of useful laws of program equivalence validated by our semantics. Each law, written as an equation of form $P_1 = P_2$, should be interpreted as asserting that in all worlds $W$ and suitable environments $u$, the traces of $P_1$ coincide with the traces of $P_2$.

## Scope contraction

- **local** $h$ **in** $(P \| Q) \ = \ ($**local** $h$ **in** $P) \| Q$
  if $h$ does not occur free in $Q$.

This law is useful in establishing, for instance, that the summation network $sum$ and its three variants $sum_1$, $sum_2$, and $sum_3$ are semantically equivalent.

## Symmetry

- **local** $h_1$ **in local** $h_2$ **in** $P \ = \$ **local** $h_2$ **in local** $h_1$ **in** $P$

This equivalence justifies our use of the abbreviation **local** $h_1, h_2$ **in** $P$.

## Asynchrony

- **local** $h$ **in** $(h!e; P) \ = \ P$
  if $h$ does not occur free in $P$

- **local** $h$ **in** $(h?x; P) =$ **while true do skip**

These two laws emphasize the asynchronous mode of communication: an output just happens, but an input must wait. Note also that local variables and local channels are not recorded in the overall traces. An operation such as outputting to a local channel appears as a stuttering step to the overall program, since it has no effect on the non-local part of the state, and is thus absorbed by closure.

## Local input and output

- **local** $h = v\rho$ **in** $P\|(h?x;Q)$ $=$ **local** $h = \rho$ **in** $P\|(x{:=}v;Q)$
  provided $h?$ does not occur free in $P$.

- **local** $h = \rho$ **in** $P\|(h!v;Q)$ $=$ **local** $h = \rho v$ **in** $P\|Q$
  provided $h!$ does not occur free in $P$.

These laws show that under certain circumstances we lose no generality in assuming that a suitably enabled communication involving a local channel occurs immediately, regardless of the enabledness of other activity.

Note also the following slight generalization of the obvious corollary:

- **local** $h = \epsilon$ **in** $P\|(h?x;Q)\|(h!v;R)$ $=$ **local** $h = \epsilon$ **in** $P\|(x{:=}v;Q)\|R$
  provided $h$ does not occur free in $P$ and $h?$ does not occur free in $R$.

Another special case shows that a local output and a local input done concurrently is equivalent to a "distributed" assignment:

- **local** $h$ **in** $(h!v\|h?x)$ $=$ $x{:=}v$.

## Global promotion

- **local** $h = \epsilon$ **in** $(h?x;P)\|(Q_1;Q_2)$ $=$ $Q_1;$ **local** $h = \epsilon$ **in** $(h?x;P)\|Q_2$
  if $h$ does not occur free in $Q_1$.

The soundness of this law relies crucially on fairness. It can be used to simplify reasoning in cases where local channels are used to enforce synchronization. The significance of this law, and of its obvious generalization to the case when there are several components waiting on local channels, is that it allows one to "move to the front" (or "promote") an initial segment of code performable by a parallel component, provided that code is "global" in that it does not affect any local channel, and provided the "rest" of the parallel composition is waiting on a local channel that is currently empty. This permits reasoning to assume without loss of generality that the "visible" piece of code happens first, even though operationally what really happens is that the two parallel components are interleaved fairly. No generality is lost because the blocked component contributes only stuttering steps to the traces, and these steps are absorbed by the closure rules.

## Cyclic synchronization

A common technique to enforce synchronization involves a cyclic communication pattern of requests and acknowledgements. The simplest case, for two processes, corresponds to the following law:

$$\begin{array}{lcl} \textbf{local}\ h_1, h_2\ \textbf{in} & = & (P_1 \| P_2); \\ \quad (P_1;\ h_1!\star;\ h_2?\star;\ Q_1) & & \textbf{local}\ h_1, h_2\ \textbf{in}\ (Q_1 \| Q_2) \\ \ \| \ (P_2;\ h_2!\star;\ h_1?\star;\ Q_2) & & \end{array}$$

provided $P_1$ and $P_2$ do not use $h_1$ or $h_2$. Here we have assumed that the two channels have type **chan**[**unit**] since the content of the message used for synchronization is immaterial. More generally, when none of the $P_i$ use any of the $h_j$,

$$\begin{array}{l} \textbf{local}\ h_0, \ldots, h_n\ \textbf{in} \\ \qquad \|_{i=0}^{n}(P_i;\ h_i!\star;\ h_{i\ominus 1}?\star;\ Q_i) \end{array}$$

is equivalent to $(\|_{i=1}^{n} P_i)$; **local** $h_0, \ldots, h_n$ **in** $(\|_{i=1}^{n} Q_i)$. Again these laws rely on fairness.

## A buffer property

Recall the process *buff* which behaves like a one-place buffer. It satisfies the following general law, which codifies the sense in which when suitably localized to prevent interaction with extraneous processes its effect is trivial. This is an analogue in the non-deterministic setting of the fact that *buff* computes the identity relation on histories:

$$\textbf{local}\ h, h'\ \textbf{in}\ (P \ \|\ \mathit{buff}(h, h')) \ = \ \textbf{local}\ h'\ \textbf{in}\ P[h'/h],$$

provided $h?$ and $h'!$ do not occur free in $P$. Here $P[h'/h]$ denotes the process obtained from $P$ by replacing every output on $h$ by output on $h'$. Note that the law becomes invalid if we remove the enclosing local variable declaration for $h'$, i.e. under the same assumptions about $h$,$h'$ and $P$ the process **local** $h$ **in** $(P \ \|\ \mathit{buff}(h, h'))$ is not generally equivalent to $P[h'/h]$.

# 7 Reasoning about networks

We now return again to the prefix-summation network. The fact that the three different decompositions of the network ($sum_1$, $sum_2$, and $sum_3$) have the same semantics, and that this coincides with the trace set of the "flat" version ($sum$), falls out immediately from the scope contraction law and symmetry, together with the obvious laws of associativity and commutativity for parallel composition. For example,

$$\begin{array}{rcl} sum_1 & = & \mathbf{local}\ out',\ in'\ \mathbf{in}\ (reg \parallel \mathbf{local}\ on\ \mathbf{in}\ (add \parallel dup)) \\ & = & \mathbf{local}\ out',\ in'\ \mathbf{in}\ \mathbf{local}\ on\ \mathbf{in}\ (reg \parallel (add \parallel dup)) \\ & = & \mathbf{local}\ out',\ in',\ on\ \mathbf{in}\ (reg \parallel (add \parallel dup)) \\ & = & \mathbf{local}\ out',\ in',\ on\ \mathbf{in}\ (add \parallel reg \parallel dup) \\ & = & sum \end{array}$$

The first step in the above proof, using scope contraction, relies on the fact that $reg$ does not use the channel $on$.

To prove the correctness of the $sum$ network we first need to specify what correctness should mean. Although the most obvious specification might be that the network should be capable of inputting a sequence of integers and outputting their sum, this description is insufficiently precise to characterize the network's behavior accurately. In fact the network is capable of inputting *two* integers before emitting the first prefix sum, and this pattern recurs persistently. Instead we use the following specification:

$$\begin{array}{rcl} sum & = & \bigcup_{v \in V_{\mathrm{int}}} in?v; SUM(v) \ \cup \ (in?\epsilon)^\omega \\ SUM(v) & = & \bigcup_{v' \in V_{\mathrm{int}}} (in?v' \parallel out!v); SUM(v+v') \ \cup \ out!v; (in?\epsilon)^\omega \end{array}$$

Note that the specification implies that each input triggers the availability of the next output. Moreover if at any stage the input gets blocked any pending output will eventually get emitted.

The proof that $sum$ has the trace set specified here is straightforward, using the laws of the previous section. Of particular relevance are the laws for local input, local output, global promotion, cyclic synchronization, and the buffer property. In fact there are several different proofs possible, essentially corresponding to the different ways in which we might decompose the network into sub-networks, as in $sum_1$, $sum_2$ and $sum_3$. Again this mirrors the situation with Kahn's treatment for the same network.

**process** *filter*$(p,\ a,\ b) =$
  **local** $x$ **in**
    **while true do**
      $(a?x;$ **if** $x$ **mod** $p \neq 0$ **then** $b!x);$

**process** *sift*$(a,\ out) =$
  **local** $b,\ p$ **in**
    **begin**
      $a?p;\ \ out!p;$
      *filter*$(p,\ a,\ b) \parallel$ *sift*$(b,\ out)$
    **end**;

**process** *nats*$(k,\ a) = a!k;$ *nats*$(k+1,\ a);$

**process** *primes*$(out) =$
    **local** $a$ **in** $($*nats*$(2, a) \parallel$ *sift*$(a, out))$

Figure 4: The *primes* network

It is also easy to show (with only a minor alteration to the proof for *sum*) that the non-deterministic network *sum*$'$ has exactly the same traces and therefore satisfies the same specification. The non-determinism here occurs "invisibly", since it only affects local activity. Similarly it makes no difference if we reverse the order in which the *add* node waits for its two input channels, or even if we allow it to wait in parallel. In all of these cases the proof is straightforward.

As an example of dynamic networks, Figure 4 lists the nodes of a network based on the Sieve of Eratosthenes, as discussed in Kahn's paper. Intuitively, *primes*(*out*) is a dynamically evolving network whose structure at any time is a chain of *filter* nodes connecting a *nats* node to a *sift* node. It produces, on channel *out*, the infinite ascending sequence of prime numbers. The correctness of this network can also be proved in a straightforward manner.

# 8 Recovering Kahn

The trace semantics given above makes sense for non-deterministic networks as well as for deterministic ones. We now show that the trace set of a deterministic network is a natural generalization of its Kahn-style history function, in the sense that the history function can be extracted directly from the trace set by focussing on traces of a special format.

Given a network in which all free channels are unidirectional, we can extract an input-output history relation from the traces of the network as follows. Suppose for simplicity that the network has states of the form $(\rho, \sigma)$, where $\rho \in I = I_1^* \times \cdots \times I_k^*$ represents the input channels and $\sigma \in O = O_1^* \times \cdots \times O_n^*$ represents the output channels. Let $T$ be a trace set over the corresponding state set $I \times O$. Let $I^\infty$ stand for $I_1^\infty \times \cdots \times I_k^\infty$ and similarly for $O^\infty$. The relation $rel(T) \subseteq I^\infty \times O^\infty$ is defined to be

$$\begin{array}{rcl} rel(T) & = & \{(\rho, \sigma) \mid \ \rho = \langle \rho_n \rangle,\ \sigma = \langle \sigma_n \rangle\ \& \\ & & \quad \langle (\rho_0, \epsilon), (\delta_0, \sigma_0) \rangle \\ & & \quad\quad \langle (\delta_0 \rho_1, \epsilon), (\delta_1, \sigma_1) \rangle \\ & & \quad\quad\quad \cdots\cdots\cdots\cdots \\ & & \quad\quad\quad\quad \langle (\delta_{n-1} \rho_n, \epsilon), (\delta_n, \sigma_n) \rangle \\ & & \quad\quad\quad\quad\quad \cdots\cdots\cdots\cdots \in T\} \end{array}$$

Note that by convention we write $\epsilon^\omega = \epsilon$, so that the same format can serve both for finite and infinite histories.

Intuitively, input history $\rho$ is related by $rel(T)$ to output history $\sigma$ when there is a "decomposition" $\langle \sigma_n \rangle$, expressing $\sigma$ as a sequence of finite chunks, and a corresponding decomposition of $\rho$ into $\langle \rho_n \rangle$, such that when the input chunks are supplied "successively" the corresponding output chunks are produced. We may refer to a trace of the above format as a *justifying trace* for $(\rho, \sigma)$. Given the above assumptions on channel usage, the trace structure implies that $\delta_0$ is a suffix of $\rho_0$, $\delta_1$ is a suffix of $\rho_0 \rho_1$, and so on.

In fact it is easy to see that input and output are "oblivious" in the sense that the potential for input or output to occur does not go away if channels are primed with "extra" data in the following sense:

- A process has a trace of form $\alpha \langle (v\rho, \sigma), (\rho, \sigma') \rangle \beta$ if and only if it has the trace $\alpha \langle (v\rho\delta, \sigma), (\rho\delta, \sigma') \rangle \beta$, for all $\delta \in I$.

- A process has a trace of form $\alpha\langle(\rho,\sigma),(\rho',\sigma v)\rangle\beta$ if and only if it has the trace $\alpha\langle(\rho,\epsilon),(\rho',v)\rangle\beta$.

As a consequence, we can give the following alternative (but equivalent) formulation of $rel(T)$, which is sometimes easier to work with:

$$\begin{array}{rcl} rel(T) & = & \{(\rho,\sigma) \mid \ \rho = \langle\rho_n\rangle,\ \ \sigma = \langle\sigma_n\rangle\ \& \\ & & \quad \langle(\rho_0,\epsilon),(\delta_0,\sigma_0)\rangle \\ & & \qquad \langle(\delta_0\rho_1,\sigma_0),(\delta_1,\sigma_0\sigma_1)\rangle \\ & & \qquad\quad \ldots\ldots\ldots\ldots \\ & & \qquad\qquad \langle(\delta_{n-1}\rho_n,\sigma_0\ldots\sigma_{n-1}),(\delta_n,\sigma_0\ldots\sigma_n)\rangle \\ & & \qquad\qquad\quad \ldots\ldots\ldots\ldots \in T\}. \end{array}$$

Another important feature of $rel(T)$ is the following *Decomposition Property*: the presence of a particular input-output history pair $(\rho,\sigma)$ in $rel(T)$ can be shown by choosing *any* decomposition of $\sigma$ and finding a corresponding decomposition for $\rho$; the trace set is guaranteed to contain a suitably structured trace of the format required to establish that $(\rho,\sigma)\in rel(T)$.

The definition of *trace-based history relation* makes sense for any network, deterministic or non-deterministic, provided the network uses each channel unequivocally either for input or for output. To illustrate, we return again to the buffer processes. According to the above definition, the (deterministic) one-place and unbounded buffer processes each determine the identity function on histories:

$$\begin{array}{rcl} rel(traces[\![buff]\!]) & = & \{(\rho,\rho) \mid \rho\in V^\infty\} \\ rel(traces[\![buffs]\!]) & = & \{(\rho,\rho) \mid \rho\in V^\infty\} \end{array}$$

Similarly, each (non-deterministic) variant also determines the intended history relation:

$$\begin{array}{rcl} rel(traces[\![buff']\!]) & = & \{(\rho,\sigma) \mid \sigma\le\rho\ \&\ \rho,\sigma\in V^\infty\} \\ rel(traces[\![buff_*]\!]) & = & \{(\rho,\sigma) \mid \sigma\le\rho\ \&\ \rho\in V^\infty\ \&\ \sigma\in V^*\}. \end{array}$$

Now recalling some of our other non-deterministic examples, the relations obtained from the *merge* and *spray* processes are also as expected:

$$\begin{array}{rcl} rel(traces[\![merge]\!]) & = & fairmerge_V \\ rel(traces[\![spray]\!]) & = & \{(\rho,\sigma_1,\sigma_2) \mid (\sigma_1,\sigma_2,\rho)\in fairmerge_V\}. \end{array}$$

Moreover, it is easy to see from the characterization given earlier for the traces of *sum* that $\mathit{rel}(\mathit{traces}[\![\mathit{sum}]\!])$ is indeed the (graph of the) prefix-sum function. It follows easily that the deterministic networks $\mathit{sum}_1$, $\mathit{sum}_2$, $\mathit{sum}_3$ and the non-deterministic network $\mathit{sum}'$ also determine the same relation, since they all have the same trace set as *sum*.

The prescription given above for $\mathit{rel}(T)$ is rather intuitive, but differs slightly from Kahn's approach in that we did not build in continuity. For example, we have

$$\begin{array}{rcl} \mathit{rel}(\mathit{traces}[\![\mathbf{skip}]\!]) & = & \{(\rho, \epsilon) \mid \rho \in V^*\} \\ \mathit{rel}(\mathit{traces}[\![\mathbf{while\ true\ do\ skip}]\!]) & = & \{(\rho, \epsilon) \mid \rho \in V^\infty\}, \end{array}$$

since **skip** has only finite traces. In Kahn's setting both of these processes denote the same function, i.e. $\lambda\rho \in V^\infty.\epsilon$. This inability of Kahn's model to distinguish between termination and divergence is insignificant in Kahn's setting, primarily since sequential composition is not allowed at the network level. In our setting it makes sense to make the distinction. For comparison to Kahn's model we must therefore focus on the *limit-closure* of $\mathit{rel}(T)$, which we define to be the smallest relation $R$ containing $\mathit{rel}(T)$ such that whenever $\rho_0 \leq \rho_1 \leq \ldots$ and $\sigma_0 \leq \sigma_1 \leq \ldots$ are increasing sequences of finite histories with limits $\rho$ and $\sigma$, the presence of $(\rho_n, \sigma_n)$ in $R$ for all $n$ implies that $(\rho, \sigma)$ belongs to $R$. For example it is easy to see that the limit-closure of $\{(\rho, \epsilon) \mid \rho \in V^*\}$ is $\{(\rho, \epsilon) \mid \rho \in V^\infty\}$, as desired to make the above identification.

We are now in a position to state formally the sense in which our trace semantics is a natural generalization of Kahn's model. When $T$ is the trace set of a uni-directional deterministic network the limit-closure of $\mathit{rel}(T)$ coincides with the graph of the network's input-output function as predicted by Kahn's semantics. Using the terminology introduced earlier, we have

$$\mathit{limit\text{-}closure}(\mathit{rel}(\mathit{traces}[\![P]\!])) = \mathit{str}[\![P]\!]$$

for all Kahn-style deterministic networks $P$. We defer the details of this proof to Appendix B.

As an important consequence of this result, whenever two networks have the same traces they induce the same history relation *in all contexts*. This follows by compositionality of the traces semantics.

# 9 Conclusions

We have given a semantics for fair networks of non-deterministic asynchronous communicating processes. We have shown that our model is a natural generalization of Kahn-style input-output functions, extended to take into account the potential for interference between processes. Fairness plays a vital role in our semantics, a natural outgrowth of its understated supportive role in Kahn's original semantics. Despite its historical reputation, fairness is not especially problematic from the semantic point of view, and can be incorporated without difficulty.

We have shown that our semantics supports a number of useful laws of program equivalence that may be used to simplify reasoning about network behavior. Several of these laws rely crucially on fairness for their soundness, and this can be an advantage when reasoning about liveness properties. Local declarations can be used to great effect to build in non-interference assumptions, such as the inability of one parallel component to modify private data used by other components.

We have shown that our trace semantics is *adequate* for reasoning about history relations, in the sense that processes (either nodes or entire subnetworks) with the same traces can be interchanged in any network context without affecting the history relation computed by the resulting network. It would be interesting to see what additional programming language constructs need to be added in order to guarantee the converse of this property, i.e. *full abstraction* [Mil77, Sto88]. We conjecture that it suffices to add a simple form of *conditional critical region* construct, usually written as **await** $B$ **then** $C$, by analogy with the full abstraction result proven in [Bro93].

The idea of using traces of some kind to model concurrent processes is widespread. Unlike many traditional models for communicating processes, such as [Bro94, Hoa78], we work entirely with "complete" traces and we build in fairness so that the semantics of a process provides a full and precise description of its potential behaviors under any reasonable scheduling strategy. By blending channels into the state structure so that communication becomes a state change we are able to avoid using "process labels" or "channel labels" to decorate the steps of a trace, and we can avoid the corresponding book-keeping details that tend to clutter up labelled trace models. For instance, we have avoided the need for "refusal sets" as a means of modelling deadlock [Hoa78]. Instead, a deadlocked process manifests itself in our model as infinite stuttering, which after all is how it will appear to any process attempting to interact with it: a deadlocked process will never change the state, and never terminates.

In contrast to several earlier trace-theoretic models [Jon89, Kok87, Rus90, KP84] we take a different view of state, and of the nature of a step in a trace, and we build in a different combination of closure conditions on trace sets. Typically these earlier models were concerned with the search for fully abstract models of Kahn networks, with respect to a kind of observable behavior based on Kahn-style input-output functions. Our model is designed to provide more discriminatory power than Kahn's semantics, so that our semantics solves a different problem and fits in a niche at a different level of abstraction than these models.

It would be interesting to investigate if our semantics can be used to analyze the relative expressive power of communication primitives, perhaps along lines suggested by [PS88]. In particular it seems obvious that the expressive power of our language would be improved if we add a form of *channel probe*, permitting a process to test for availability of data without necessarily inputting it.

# References


[Abr90] S. Abramsky. A Generalized Kahn Principle for Abstract Asynchronous Networks. In *Mathematical Foundations of Programming Semantics, 5th International Conference, March/April 1989*, Lecture Notes in Computer Science, pages 1–21. Springer-Verlag, 1990.

[BA81] J. Brock and W. Ackerman. Scenarios: a model of non-determinate computation. In *Formalization of Programming Concepts*, volume 107 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[Bro93] S. Brookes. Full abstraction for a shared variable parallel language. In *8th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, June 1993.

[Bro94] S. Brookes. Fair communicating processes. In A. W. Roscoe, editor, *A Classical Mind: Essays in Honour of C. A. R. Hoare*. Prentice-Hall, 1994.

[Bro96] S. Brookes. The essence of Parallel Algol. In *11th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, June 1996.

[Bro97] S. Brookes. Idealized CSP: Combining procedures with communicating processes. In *13th Annual Conference on Mathematical Foundations of Programming Semantics (MFPS'97)*, Electronic Notes in Theoretical Computer Science. Elsevier, 1997.

[Fau82] A. Faustini. An operational semantics for pure dataflow. In *Automata, Languages, and Programming, 9th Colloquium*, volume 140 of *Lecture Notes in Computer Science*. Springer-Verlag, 1982.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Jon89] B. Jonsson. A fully abstract trace model for dataflow networks. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 143–154. ACM Press, 1989.

[Kah77] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74*, pages 993–998. North Holland, 1977.

[KM77] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *Information Processing 1977*, pages 993–998. North Holland, 1977.

[Kok87] J. Kok. A fully abstract semantics for dataflow nets. In *Proceedings of PARLE*, pages 351–368. Springer-Verlag, 1987.

[KP84] R. Keller and P. Panangaden. Semantics of networks containing indeterminate operators. In *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*. Springer-Verlag, 1984.

[LS89] N. Lynch and E. Stark. A proof of the Kahn principle for input/output automata. *Information and Computation*, 82(1):81–92, July 1989.

[Mil77] R. Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[Ole82] F. J. Oles. *A Category-Theoretic Approach to the Semantics of Programming Languages*. PhD thesis, Syracuse University, 1982.

[Par79] D. Park. On the semantics of fair parallelism. In *Abstract Software Specifications*, volume 86 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[PS88] P. Panangaden and E. Stark. Computations, residuals and the power of indeterminacy. In *Proceedings of the 15th ICALP*, pages 439–454. Springer-Verlag, 1988.

[Rey81] J. C. Reynolds. The essence of Algol. In *Algorithmic Languages*, pages 345–372. North-Holland, Amsterdam, 1981.

[Ros98] A. W. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, 1998.

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

[Rus90] J. Russell. *Full Abstraction and Fixed-Point Principles for Indeterminate Computation.* PhD thesis, Cornell University, April 1990.

[Sco82] D.S. Scott. Domains for denotational semantics. In *Automata, Languages, and Programming, 9th Colloquium*, volume 140 of *Lecture Notes in Computer Science.* Springer-Verlag, 1982.

[Smy78] M. B. Smyth. Power domains. *Journal of Computer and System Sciences*, 16(1):23–36, February 1978.

[Sta89] E. Stark. Concurrent transition systems. *Theoretical Computer Science*, 64:221–269, 1989.

[Sto88] A. Stoughton. *Fully Abstract Models of Programming Languages.* Research Notes in Theoretical Computer Science. Pitman, London, 1988.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5, 1955.

# 10 Appendix A: Operational semantics

We present a structured operational semantics for processes. Configurations have the form $\langle P, s\rangle$, where $P$ is a process and $s$ is a state in some state set $S$; there is also assumed to be an environment $u$ mapping all free identifiers of $P$ to appropriate bindings over $S$. A configuration is either *terminal*, or has one or more enabled transitions. We write $\langle P, s\rangle \rightarrow \langle P', s'\rangle$ to indicate an enabled transition. We write $\langle P, s\rangle term$ to indicate a terminal configuration.

We assume given the transition rules for expressions. We write, for instance, $\langle e, s\rangle \rightarrow^{*} v$ to indicate that $e$ evaluates to value $v$ in state $w$ (and the given environment). Expression evaluation is assumed to be free of side-effects.

For simplicity in presenting the transition rules let $s = (w, v, \rho)$ be a state of shape $W \times V \times V^{*}$, and let $x$ and $h$ be bound to the variable and the channel represented by the final two components of the state, respectively. We then write $[s \mid x : v']$ for the state $(w, v', \rho)$ obtained by updating the $x$-component of $s$, and $[s \mid h :!v']$ for the state $(w, v, \rho v')$ obtained by sending $v'$ to the $h$-component of $s$. We also write $s(x)$ and $s(h)$ for the value of the respective component of $s$. We also assume that $t$ is a state over $W \times V$ and write $(t, h : \rho)$ for the obvious corresponding state over $W \times V \times V^{*}$.

The termination predicate *term* and the one-step transition relation are defined to be the smallest relations satisfying the following rules.

$$\frac{}{\langle \mathbf{skip}, s\rangle \mathrm{term}}$$

$$\frac{\langle e, s\rangle \rightarrow^{*} v}{\langle x{:=}e, s\rangle \rightarrow \langle \mathbf{skip}, [s \mid x : v]\rangle}$$

$$\frac{\langle e, s\rangle \rightarrow^{*} v}{\langle h!e, s\rangle \rightarrow [s \mid h :!v]}$$

$$\frac{s(h) = v\rho}{\langle h?x, s\rangle \rightarrow \langle \mathbf{skip}, [s \mid x : v, h : \rho]}$$

$$\frac{s(h) = \epsilon}{\langle h?x, s\rangle \rightarrow \langle h?x, s\rangle}$$

$$\frac{\langle P_1, s\rangle \to \langle P_1', s'\rangle}{\langle P_1; P_2, s\rangle \to \langle P_1'; P_2, s'\rangle}$$

$$\frac{\langle P_1, s\rangle \mathrm{term}}{\langle P_1; P_2, s\rangle \to \langle P_2, s\rangle}$$

$$\frac{\langle B, s\rangle \to^* \mathtt{tt}}{\langle \mathbf{if}\ B\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2, s\rangle \to \langle P_1, s\rangle}$$

$$\frac{\langle B, s\rangle \to^* \mathtt{ff}}{\langle \mathbf{if}\ B\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2, s\rangle \to \langle P_2, s\rangle}$$

$$\frac{}{\langle \mathbf{while}\ B\ \mathbf{do}\ P, s\rangle \to \langle \mathbf{if}\ B\ \mathbf{then}\ P;\ \mathbf{while}\ B\ \mathbf{do}\ P\ \mathbf{else}\ \mathbf{skip}, s\rangle}$$

$$\frac{\langle P_1, s\rangle \to \langle P_1', s'\rangle}{\langle P_1 \| P_2, s\rangle \to \langle P_1' \| P_2, s'\rangle}$$

$$\frac{\langle P_2, s\rangle \to \langle P_2', s'\rangle}{\langle P_1 \| P_2, s\rangle \to \langle P_1 \| P_2', s'\rangle}$$

$$\frac{\langle P_1, s\rangle \mathrm{term} \quad \langle P_2, s\rangle \mathrm{term}}{\langle P_1 \| P_2, s\rangle \mathrm{term}}$$

$$\frac{\langle P, (t, h : \rho)\rangle \to \langle P', (t', h : \rho')\rangle}{\langle \mathbf{local}\ h = \rho\ \mathbf{in}\ P, t\rangle \to \langle \mathbf{local}\ h = \rho'\ \mathbf{in}\ P', t'\rangle}$$

$$\frac{\langle P, (t, h : \rho)\rangle \mathrm{term}}{\langle \mathbf{local}\ h = \rho\ \mathbf{in}\ P, t\rangle \to \langle P, t\rangle}$$

# 11 Appendix B: Recovering Kahn

We sketch the main ideas behind the key result that connects our semantics and Kahn's, i.e.

> If each node $P$ in a uni-directional deterministic network $N$ satisfies
> $$\mathit{limit\text{-}closure}(\mathit{rel}(\mathit{traces}[\![P]\!])) = \mathit{str}[\![P]\!],$$
> then the network as a whole also has this property, i.e.
> $$\mathit{limit\text{-}closure}(\mathit{rel}(\mathit{traces}[\![N]\!])) = \mathit{str}[\![N]\!].$$

The proof is by structural induction on the way the network $N$ is built up using Kahn-style constructs. There are three cases: *juxtaposition*, *cascading*, and *feedback*.

## Juxtaposition

Let $P_1$ and $P_2$ be disjoint networks. Assume without loss of generality that the state set has shape $W = (I_1 \times I_2) \times (O_1 \times O_2)$, and that $P_1$ has inputs over $I_1$ and outputs over $O_1$, and similarly for $P_2$. The network obtained by juxtaposition of $P_1$ and $P_2$ is

$$\mathbf{juxtapose}(P_1, P_2) =_{\mathrm{def}} P_1 \| P_2.$$

Each of its traces is therefore a fair merge of a trace of $P_1$ with a trace of $P_2$. Since $P_1$ does not use any of the channels of $P_2$, these channels are left unchanged in every step of every trace of $P_1$; likewise for $P_2$ and the channels of $P_1$. It is thus easy to see that whenever $((\rho_1, \rho_2), (\sigma_1, \sigma_2))$ belongs to $\mathit{rel}(\mathit{traces}[\![P_1 \| P_2]\!]$, a justifying trace of $P_1 \| P_2$ is built from a justifying trace of $P_1$ for $(\rho_1, \sigma_1)$ and a justifying trace of $P_2$ for $(\rho_2, \sigma_2)$. The converse is also true: merging a justifying trace for $(\rho_1, \sigma_1)$ with a justifying trace for $(\rho_2, \sigma_2)$ yields a justifying trace for $((\rho_1, \rho_2), (\sigma_1, \sigma_2))$. Thus

$$\begin{array}{rcl} \mathit{rel}(\mathit{traces}[\![P_1 \| P_2]\!]) & = & \{((\rho_1, \rho_2), (\sigma_1, \sigma_2)) \mid \\ & & \quad (\rho_1, \sigma_1) \in \mathit{rel}(\mathit{traces}[\![P_1]\!]) \ \& \ (\rho_2, \sigma_2) \in \mathit{rel}(\mathit{traces}[\![P_2]\!])\}. \end{array}$$

The desired result for $P_1 \| P_2$ then follows from the induction hypothesis for $P_1$ and $P_2$, since

$$\begin{array}{rcl} \mathit{str}[\![\mathbf{juxtapose}(P_1, P_2)]\!] & = & \{((\rho_1, \rho_2), (\sigma_1, \sigma_2)) \mid \\ & & \quad (\rho_1, \sigma_1) \in \mathit{str}[\![P_1]\!] \ \& \ (\rho_2, \sigma_2) \in \mathit{str}[\![P_2]\!]\}. \end{array}$$

## Cascading

For ease of presentation we consider only the case involving a single linking channel. The fully general case can be treated analogously.

Let $W = I \times O$, and let $P_1$ be a process with input channels corresponding to components of $I$ and a single output channel named $h$ of type $\mathbf{chan}[\tau]$, and let $P_2$ have output channels in $O$ and a single input channel $h$. The network formed by cascading $P_1$ onto $P_2$ is

$$\mathbf{cascade}(P_1, h, P_2) =_{\mathrm{def}} \mathbf{local}\ h\ \mathbf{in}\ (P_1 \| P_2).$$

We claim that

$$\mathit{rel}(\mathit{traces}[\![\mathbf{cascade}(P_1, h, P_2)]\!]) \ = \ (\mathit{rel}(\mathit{traces}[\![P_2]\!])) \circ (\mathit{rel}(\mathit{traces}[\![P_1]\!])).$$

- To show the inclusion from left to right, suppose

  $$(\rho, \sigma) \in \mathit{rel}(\mathit{traces}[\![\mathbf{cascade}(P_1, h, P_2)]\!]).$$

  Then there are decompositions $\rho = \langle \rho_n \rangle$ and $\sigma = \langle \sigma_n \rangle$ and a trace of $P_1 \| P_2$ over $I \times O \times V_\tau^*$ of form

  $$\begin{array}{l} \langle(\rho_0, \epsilon, \epsilon), (\rho_0', \epsilon, \nu_0)\rangle \\ \quad \langle(\rho_0', \epsilon, \nu_0), (\rho_0', \sigma_0, \nu_0')\rangle \\ \quad\quad \langle(\rho_0'\rho_1, \sigma_0, \nu_0'), (\rho_1', \sigma_0, \nu_0'\nu_1)\rangle \\ \quad\quad\quad \langle(\rho_1', \sigma_0, \nu_0'\nu_1), (\rho_1', \sigma_0\sigma_1, \nu_1')\rangle \\ \quad\quad\quad\quad \langle(\rho_1'\rho_2, \sigma_0\sigma_1, \nu_1'), (\rho_2', \sigma_0\sigma_1, \nu_1'\nu_2)\rangle \\ \quad\quad\quad\quad\quad \cdots\cdots \end{array}$$

  in which (without loss of generality) $P_1$ and $P_2$ contribute alternate steps[8]. This trace arises as a fair merge of the traces

  $$\begin{array}{l} \langle(\rho_0, \epsilon), (\rho_0', \nu_0\rangle \\ \quad \langle(\rho_0'\rho_1, \nu_0'), (\rho_1', \nu_0'\nu_1)\rangle \\ \quad\quad \langle(\rho_1'\rho_2, \nu_1'), (\rho_2', \nu_1'\nu_2)\rangle \\ \quad\quad\quad \cdots\cdots \end{array}$$

  of $P_1$, and

  $$\begin{array}{l} \langle(\nu_0, \epsilon), (\nu_0', \sigma_0)\rangle \\ \quad \langle(\nu_0'\nu_1, \sigma_0), (\nu_1', \sigma_0\sigma_1)\rangle \\ \quad\quad \cdots\cdots \end{array}$$

[8] Any other trace of this process can be put into this form by inserting stuttering steps.

of $P_2$. Let $\nu = \nu_0\nu_1 \ldots$. It follows that $(\rho, \nu)$ belongs to $rel(traces[\![P_1]\!])$ and $(\nu, \sigma)$ belongs to $rel(traces[\![P_2]\!])$. Hence $(\rho, \sigma)$ belongs to the composition of these two relations, as required.

- For the reverse direction, suppose $(\rho, \nu) \in rel(traces[\![P_1]\!])$ and $(\nu, \sigma) \in rel(traces[\![P_2]\!])$. Choose the "justifying" trace for $(\rho, \nu)$ of $P_1$ to match the decomposition of $\nu$ used in the justifying trace for $(\nu, \sigma)$ of $P_2$. (The ability to make this choice relies on the *Decomposition Property* mentioned earlier.) Then interleave these traces in the obvious manner, leaving the local channel unchanged across step boundaries, to obtain a justifying trace for $(\rho, \sigma)$ of **local** $h$ **in** $(P_1 \| P_2)$, as required.

It then follows that if $P_1$ and $P_2$ satisfy the induction hypothesis, so does **cascade**$(P_1, h, P_2)$, since

$$str[\![\mathbf{cascade}(P_1, h, P_2)]\!] = str[\![P_2]\!] \circ str[\![P_2]\!].$$

## Feedback

Let $P$ be a network using channel $h$ for input and $h'$ for output, and assume that the state set has shape $W \times V^* \times V^*$, the last two components representing these two channels respectively. The network obtained by feeding $h'$ back as input to $h$ is:

$$\mathbf{feedback}(P, h, h') \ =_{\mathrm{def}} \ \mathbf{local}\ h\ \mathbf{in}\ [h'/h]P.$$

This feedback network has traces of the form

$$\langle w_0, w_0' \rangle \langle w_1, w_1' \rangle \langle w_2, w_2' \rangle \ldots$$

such that $P$ has a trace of the form

$$\langle (w_0, \epsilon, \epsilon), (w_0', \epsilon, \nu_0) \rangle \langle (w_1, \nu_0, \epsilon), (w_1', \nu_0', \nu_1) \rangle \langle (w_2, \nu_0'\nu_1, \epsilon), (w_2', \nu_1', \nu_2) \rangle \ldots$$

(Feedback is effected here, intuitively, by sliding messages from the $h'$-component to the $h$-component.) According to Kahn's least-fixed-point characterization a pair $(\rho, \sigma)$ belongs to the limit-closure of the input-output function of the feedback network if and only if there are decompositions $\rho = \langle \rho_n \rangle$ and $\sigma = \langle \sigma_n \rangle$, and a sequence $\nu = \nu_0\nu_1 \ldots$ such that

- $((\rho_0, \epsilon), (\sigma_0, \nu_0)) \in str[\![P]\!]$;
- $((\rho_0\rho_1, \nu_0), (\sigma_0\sigma_1, \nu_0\nu_1)) \in str[\![P]\!]$;
- $((\rho_0\rho_1\rho_2, \nu_0\nu_1), (\sigma_0\sigma_1\sigma_2, \nu_0\nu_1\nu_2)) \in str[\![P]\!]$

and so on. Using this formulation, which echoes the way in which data is transferred on the internal channel, it is straightforward to establish the connection: the sequence $\epsilon$, $\nu_0$, $\nu_0\nu_1$, and so on converges to the history (for the feedback channel) corresponding to the least fixed point, as prescribed by Kahn's semantics. Hence, if $P$ satisfies the induction hypothesis, so does $\mathbf{feedback}(P, h, h')$.