Traces, Pomsets, Fairness and Full Abstraction for Communicating Processes

Stephen Brookes

Department of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA

Abstract. We provide a denotational trace semantics for processes with synchronous communication and a form of weakly fair parallelism. The semantics is fully abstract: processes have the same trace sets if and only if their communication behaviors are identical in all contexts. The model can easily be adapted for asynchronously communicating processes, or for shared-memory parallel programs. We also provide a partial-order semantics, using pomsets adapted for synchronization and our form of fairness. The pomset semantics can also be adjusted to model alternative paradigms. The traces of a process can be recovered from the pomset semantics by taking all fair interleavings consistent with the partial order.

1 Introduction

Traces of various kinds are commonly used in semantic models of languages for parallel programming, with parallel execution usually interpreted as a form of fair interleaving. *Transition traces*, sequences of pairs of states, have been used for shared-memory parallel programs [27, 5], for concurrent logic programs and concurrent constraint programs [4, 34][1], and for networks of asynchronously communicating processes [8], assuming weakly fair execution. Transition traces also provide a semantics for a parallel Algol-like language [6, 7]. *Communication traces*, sequences of input/output events, were the basis for an early model of CSP [19, 20], later augmented with *refusal sets* to permit deadlock analysis in the *failures* model [11], and with *divergence traces* in the *failures-divergences* model [12][2]. *Pomset traces* provide a partial-order based framework based on "true concurrency" rather than interleaving [31, 32].

Fairness assumptions [15], such as *weak process fairness*, the guarantee that each persistently enabled process is eventually scheduled, allow us to abstract

[1] The use of such sequences to model shared-memory programs dates back at least to Park [27]. The term *reactive sequence* is used in [4] for this kind of trace, which is also related to the *ask/tell* sequences of [34].

[2] Roscoe's book [33] gives a detailed account of these and related models of CSP. van Glabbeek's article [16] provides a wide-ranging and detailed survey of process algebras and notions of behavioral equivalence.

away from unknown or unknowable implementation details. A fairness notion is "reasonable" if it provides a good abstraction of realistic schedulers, so that by assuming this form of fairness one is able to deduce program properties which hold under any reasonable implementation. We are typically interested in *safety* and *liveness* properties [26]. A safety property has the general intuitive form that something "bad" never happens. A liveness property asserts that something "good" will eventually happen. Both kinds of property depend on the sequences of states through which a parallel system may pass during execution. Fairness plays a crucial role: it is often impossible to prove liveness properties without fairness assumptions.

CSP [19] is a language of synchronously communicating processes: a process attempting output must wait until another process attempts a matching input, and *vice versa*. The early denotational models of CSP [20, 11, 12, 33], like many operational or denotational accounts of related languages such as CCS [22, 23] and ACP [2, 3], focussed mainly on finite behaviors and consequently did not take fairness into account. It is not easy to extend these models naturally to incorporate fairness. Moreover, there is a plethora of fairness notions, including strong and weak forms of process, channel, and communication fairness [15]. The extent to which these fairness notions provide tractable yet realistic abstractions for CSP-style processes is unclear, although weak process fairness has an intuitively appealing definition and can be ensured by a "reasonable" scheduler using a simple round-robin strategy. Costa and Stirling have shown how to provide an operational semantics for a CCS-like language assuming either weak or strong process fairness [13, 14]. Older shows that one can treat some of these fairness notions denotationally by augmenting failure-style models still further, with detailed book-keeping information concerning processes, communications, and synchronizations which become persistently enabled but not scheduled [24, 10, 25]. Much of the difficulty is caused by the fact that synchronization requires cooperation between processes. Indeed, in Older's formulation even weak process fairness fails to be *equivalence robust* [1], in that there is a pair of computations, one fair and one unfair, which differ only in the interleaving order of independent actions [24]. In contrast, for processes communicating asynchronously one can model weak process fairness using transition traces, and weak process fairness can be given an equivalence-robust formulation [8, 9].

The structural disparity between the simple trace semantics for asynchronous processes and the intricately book-keeping failure semantics for synchronous processes obscures the underlying similarities between the two paradigms. It seems to be widely believed that this disparity is inevitable, that traces are too simple a notion to support the combination of deadlock, fairness, and synchronized communication. This is certainly a valid criticism of the traditional trace-based accounts of CSP, which used prefix-closed sets of finite traces (augmented with refusal sets) and handled infinite traces implicitly based on their finite prefixes. Although these models give an accurate account of deadlock and safety properties, they do not adequately support liveness analysis since they do not admit

fairness: the existence of a fair infinite trace for a process does not follow from the process's ability to perform each of its finite prefixes.

In this paper we show that, despite the above commentary, if we assume a reasonable weak (and robust) notion of fairness, it becomes possible to design a satisfactory trace semantics. Indeed, the *same* notion of trace can be used both for synchronously and asynchronously communicating processes. In each case we model a weak form of fairness consistent with a form of round-robin scheduling, so that we obtain a good abstraction of process behavior independent of implementation details[3]. The trace semantics is compositional, and supports safety and liveness analysis. Indeed our semantics is *fully abstract*, in the sense that two processes have the same trace set if and only if they exhibit identical communication behavior, including any potential for deadlock, in all program contexts. We do not augment traces with extraneous book-keeping information, or impose complex closure conditions. Instead we incorporate the crucial information about blocking directly in the internal structure of traces, in a manner reminiscent of Phillips-style refusal testing [30].

Our achievement is noteworthy, perhaps even surprising, given the history of separate development of semantic frameworks for the two communication paradigms. Traditional denotational models of asynchronous communication and synchronous communication have frustratingly little in common, as shown by the lack of family resemblance between failures and transition traces. In contrast, we treat both kinds of communication as straightforward variations on a trace-theoretic theme, so that we achieve a semantic unification of paradigms. Given prior results concerning the utility of trace semantics for shared-memory parallelism [5, 6], and the availability of trace-based models for concurrent constraint programs [4], the unification goes further still.

We also provide a partial-order model for processes assuming synchronous communication and fair execution [31, 32]. We define a semantics in which a process denotes a set of partially ordered multisets of actions (pomsets). Each pomset determines a set of traces, those obtainable by fair interleaving and synchronizations consistent with the partial order. The trace set of a process can be recovered in this way from its pomset semantics. The pomset semantics supports a style of reasoning which avoids dealing explicitly with interleaving, and this may help to tame the combinatorial explosion inherent in analyzing parallel systems. We can also adapt pomset semantics to model asynchronous communication, with a small number of simple changes to the semantic definitions.

We focus on a CSP-style language with blocking input and output, but our definitions and results can be adapted to handle alternative language design decisions, for example non-blocking guards, mixed boolean and input/output

[3] By this we mean that a family of simple round-robin schedulers can be defined, such that each member of this family ensures weakly fair execution, and every weakly fair execution is allowed by some such scheduler. To handle synchronization we assume that if the process currently scheduled is waiting for communication the scheduler will use a round-robin strategy to see if another process is ready to perform a matching communication.

guards, and general recursive process definitions. We summarize the adjustments required to deal with asynchronous communication.

2 Syntax

Let P range over *processes*, G over *guarded processes*, given by the following abstract grammar, in which e ranges over integer-valued expressions, b over boolean expressions, h over the set **Chan** of channel names, x over the set **Ide** of identifiers. We omit the syntax of expressions, which is conventional.

$$\begin{array}{l} P ::= \mathbf{skip} \mid x{:=}e \mid P_1; P_2 \mid \mathbf{if}\ b\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2 \mid \mathbf{while}\ b\ \mathbf{do}\ P \mid \\ \qquad h?x \mid h!e \mid P_1 \| P_2 \mid P_1 \sqcap P_2 \mid G \mid \mathbf{local}\ h\ \mathbf{in}\ P \\ G ::= (h?x \to P) \mid G_1 \,\Box\, G_2 \end{array}$$

As in CSP, $P_1 \sqcap P_2$ is "internal" choice, and $G_1 \,\Box\, G_2$ is "external" choice[4].

The construct **local** h **in** P introduces a local channel named h with scope P. One can also allow local variable declarations, as in **local** x **in** P, but we omit the semantic details in what follows. We write $chans(P)$ for the set of channel names occurring free in P. In particular, $chans(\mathbf{local}\ h\ \mathbf{in}\ P) = chans(P) - \{h\}$.

3 Actions

Let Z be the set of integers, with typical member v. An *action* has one of the following forms:

- An *evaluation* of form $x{=}v$, where x is an identifier and v is an integer.
- An *assignment* of form $x{:=}v$, where x is an identifier and v is an integer.
- A *communication* $h?v$ or $h!v$, where h is a channel name and v is an integer.
- A *blocking action* of form δ_X, where X is a finite set of directions.

An input action $h?v$ or output action $h!v$ represents the *potential* for a process to perform communication, and can only be completed when another process offers a matching communication on the same channel ($h!v$ or $h?v$, respectively). We write $match(\lambda_1, \lambda_2)$ when λ_1 and λ_2 are matching communication actions, and we let $chan(h?v) = chan(h!v) = h$. Each communication action has a *direction*; let $\mathbf{Dir} = \{h?, h! \mid h \in \mathbf{Chan}\}$ be the set of directions. A blocking action δ_X represents an unrequited attempt to communicate along the directions in X. When X is a singleton we write $\delta_{h?}$ or $\delta_{h!}$. When X is empty we write δ instead of $\delta_{\{\}}$; the action δ is also used to represent a "silent" local action, such as a synchronized handshake or reading or writing a local variable. We let $X \backslash h = X - \{h?, h!\}$. Let Σ be the set of actions, $\Lambda = \{h?v, h!v \mid h \in \mathbf{Chan}\ \&\ v \in Z\}$ be the set of communications, and $\Delta = \{\delta_X \mid X \subseteq_{fin} \mathbf{Dir}\}$ be the set of blocking actions.

[4] Our syntax distinguishes between guarded and general processes merely to enforce the syntactic constraint that the "external choice" construct is only applicable to input-guarded processes, as in Hoare's original CSP language. This allows certain simplifications in the semantic development, but is not crucial.

4 Traces

A trace is a finite or infinite sequence of actions representing a potential behavior of a process. We model persistent waiting for communication and divergence (infinite local activity) as an infinite sequence of blocking actions. We assume that unless and until blocking or divergence occurs we only care about the non-silent actions taken by a process[5]. Accordingly, we assume when concatenating that $\delta\lambda = \lambda\delta = \lambda$ for all actions λ, and we suppress waiting actions which lead to successful communication, so that $\delta_{h?}^* h?v = h?v$ for example. A trace of the form $\alpha{\delta_X}^\omega$ describes an execution in which the process performs α then gets stuck waiting to communicate along the directions in X. For a trace β let $\mathit{blocks}(\beta)$ be the set of all directions which occur infinitely often in blocking steps of β. For example, $\mathit{blocks}(a!0(\delta_{b?}\delta_{c?})^\omega) = \{b?, c?\}$.

Let $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ be the set of traces. We use ϵ for the empty sequence, and α, β, γ as meta-variables ranging over Σ^∞.

We write $\alpha\beta$ for the concatenation of β onto α, which is equal to α if α is infinite. For trace sets T_1 and T_2 we let $T_1T_2 = \{\alpha_1\alpha_2 \mid \alpha_1 \in T_1 \;\&\; \alpha_2 \in T_2\}$. For a trace set T we define $T^0 = \{\delta\}$, $T^{k+1} = TT^k$ for $k \geq 0$, and $T^* = \bigcup_{n=0}^\infty T^n$. We also let T^ω be the set of all traces of form $\alpha_0\alpha_1 \ldots \alpha_n \ldots$ where for each $n \geq 0$, $\alpha_n \in T$. Note that δ^ω is distinct from δ.

Given two traces α_1 and α_2, $\alpha_1 \| \alpha_2$ is the set of all traces formed by merging them fairly, allowing (but not necessarily requiring) synchronization of matching communications. We let $\alpha \| \epsilon = \epsilon \| \alpha = \{\alpha\}$. When α_1 and α_2 are finite and non-empty, say $\alpha_i = \lambda_i\beta_i$, we use the standard inductive definition for $\alpha_1 \| \alpha_2$:

$$\begin{array}{rl}(\lambda_1\beta_1)\|(\lambda_2\beta_2) = & \{\lambda_1\gamma \mid \gamma \in \beta_1\|(\lambda_2\beta_2)\} \;\cup\; \{\lambda_2\gamma \mid \gamma \in (\lambda_1\beta_1)\|\beta_2\} \\ & \cup\; \{\delta\gamma \mid \gamma \in \beta_1\|\beta_2 \;\&\; \mathit{match}(\lambda_1, \lambda_2)\}\end{array}$$

When α_1 and α_2 are infinite, we let $\alpha_1 \| \alpha_2 = \{\}$ if some direction in $\mathit{blocks}(\alpha_1)$ matches a direction in $\mathit{blocks}(\alpha_2)$, since it is unfair to avoid synchronizing two processes which are blocked but trying to synchronize on a common channel. Otherwise, when α_1 and α_2 are infinite and $\neg\mathit{match}(\mathit{blocks}(\alpha_1), \mathit{blocks}(\alpha_2))$, we let $\alpha_1 \| \alpha_2$ consist of all traces of form $\gamma_1\gamma_2 \ldots$ where α_1 can be written as a concatenation of finite traces $\alpha_{1,1}\alpha_{1,2} \ldots$, α_2 can be written as a concatenation of finite traces $\alpha_{2,1}\alpha_{2,2} \ldots$, and for each $i \geq 1$ we have $\gamma_i \in \alpha_{1,i} \| \alpha_{2,i}$.

For example, ${\delta_{h!}}^\omega \| {\delta_{h?}}^\omega = \{\}$ and $(a!0{\delta_{h!}}^\omega) \| (b!1{\delta_{h?}}^\omega) = \{\}$. However, ${\delta_{a!}}^\omega \| {\delta_{b?}}^\omega$ is non-empty and can be written in the form $({\delta_{a!}}^*\delta_{b?}{\delta_{b?}}^*\delta_{a!})^\omega$.

We write $\mathit{chans}(\alpha)$ for the set of channels occurring in input or output actions along α, and when $h \notin \mathit{chans}(\alpha)$ we let $\alpha\backslash h$ be the trace obtained from α by replacing every δ_X with $\delta_{X\backslash h}$. For instance, the trace $(a!0\,{\delta_{h?}}^\omega)\backslash h$ is $a!0\,\delta^\omega$.

5 Denotational Semantics

We now define a synchronous trace semantics for our programming language. We assume given the semantics of expressions: $\mathcal{T}(e) \subseteq \Sigma^* \times Z$ describes all

[5] Other notions of observable behavior, such as the assumption that we see all actions, including blocking steps, can be incorporated with appropriate modifications.

possible evaluation behaviors of e, and consists of all pairs (ρ, v) where ρ is a sequence of evaluation steps which yield value v for the expression. (We do not need to assume that expression evaluation is an atomic action.) For example, for an identifier y we have $\mathcal{T}(y) = \{(y{=}v, v) \mid v \in Z\}$ and for a numeral $\underline{n}$ we have $\mathcal{T}(\underline{n}) = \{(\delta, n)\}$. For a boolean expression b we assume given $\mathcal{T}(b) \subseteq \Sigma^* \times \{\mathbf{true}, \mathbf{false}\}$, and we let $\mathcal{T}(b)_{\mathbf{true}} = \{\rho \mid (\rho, \mathbf{true}) \in \mathcal{T}(b)\}$ and, similarly, $\mathcal{T}(b)_{\mathbf{false}} = \{(\rho \mid (\rho, \mathbf{false}) \in \mathcal{T}(b)\}$.

For a process P, the trace set $\mathcal{T}(P) \subseteq \Sigma^\infty$ describes all possible executions, assuming fair interaction between the process and its environment.

Definition 1 (Synchronous Trace Semantics)
The synchronous trace semantics of processes is defined compositionally by:

$$
\begin{aligned}
\mathcal{T}(\mathbf{skip}) &= \{\delta\} \\
\mathcal{T}(x{:=}e) &= \{\rho\, x{:=}v \mid (\rho, v) \in \mathcal{T}(e)\} \\
\mathcal{T}(P_1; P_2) &= \mathcal{T}(P_1)\mathcal{T}(P_2) = \{\alpha_1\alpha_2 \mid \alpha_1 \in \mathcal{T}(P_1)\ \&\ \alpha_2 \in \mathcal{T}(P_2)\} \\
\mathcal{T}(\mathbf{if}\ b\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2) &= \mathcal{T}(b)_{\mathbf{true}}\,\mathcal{T}(P_1)\ \cup\ \mathcal{T}(b)_{\mathbf{false}}\,\mathcal{T}(P_2) \\
\mathcal{T}(\mathbf{while}\ b\ \mathbf{do}\ P) &= (\mathcal{T}(b)_{\mathbf{true}}\,\mathcal{T}(P))^*\mathcal{T}(b)_{\mathbf{false}}\ \cup\ (\mathcal{T}(b)_{\mathbf{true}}\,\mathcal{T}(P))^\omega \\
\mathcal{T}(h?x) &= \{h?v\, x{:=}v \mid v \in Z\} \cup \{{\delta_{h?}}^\omega\} \\
\mathcal{T}(h!e) &= \{\rho\, h!v,\ \rho\, {\delta_{h!}}^\omega \mid (\rho, v) \in \mathcal{T}(e)\} \\
\mathcal{T}(P_1 \| P_2) &= \bigcup\{\alpha_1 \| \alpha_2 \mid \alpha_1 \in \mathcal{T}(P_1)\ \&\ \alpha_2 \in \mathcal{T}(P_2)\} \\
\mathcal{T}(P_1 \sqcap P_2) &= \mathcal{T}(P_1) \cup \mathcal{T}(P_2) \\
\mathcal{T}(\mathbf{local}\ h\ \mathbf{in}\ P) &= \{\alpha \backslash h \mid \alpha \in \mathcal{T}(P)\ \&\ h \notin \mathit{chans}(\alpha)\} \\
\mathcal{T}(h?x \to P) &= \{h?v\, x{:=}v\, \alpha \mid v \in Z\ \&\ \alpha \in \mathcal{T}(P)\} \cup \{{\delta_{h?}}^\omega\} \\
\mathcal{T}(G_1 \,\square\, G_2) &= \{\alpha \in \mathcal{T}(G_1) \cup \mathcal{T}(G_2) \mid \alpha \notin \Delta^\omega\} \cup \\
&\quad \{{\delta_{X \cup Y}}^\omega \mid {\delta_X}^\omega \in \mathcal{T}(G_1)\ \&\ {\delta_Y}^\omega \in \mathcal{T}(G_2)\}
\end{aligned}
$$

The traces of a guarded process G have two possible forms: either beginning with an input action $h?v \in \Lambda$, or an infinite sequence of blocking actions δ_X for the set $X = \mathit{inits}(G)$ given inductively in the obvious manner: $\mathit{inits}(h?x \to c) = \{h?\}, \mathit{inits}(G_1 \,\square\, G_2) = \mathit{inits}(G_1) \cup \mathit{inits}(G_2)$.

It is easy to prove that the above semantics satisfies standard algebraic laws, such as associativity of parallel composition and both forms of choice:

Theorem 1 *For all processes P_1, P_2, P_3 and all guarded processes G_1, G_2, G_3,*

$$
\begin{aligned}
&\mathcal{T}(P_1 \| (P_2 \| P_3)) = \mathcal{T}((P_1 \| P_2) \| P_3) \\
&\mathcal{T}(P_1 \sqcap (P_2 \sqcap P_3)) = \mathcal{T}((P_1 \sqcap P_2) \sqcap P_3) \\
&\mathcal{T}(G_1 \,\square\, (G_2 \,\square\, G_3)) = \mathcal{T}((G_1 \,\square\, G_2) \,\square\, G_3)
\end{aligned}
$$

6 Operational Semantics

A state s is a mapping from program variables to values.[6] An action may or may not be enabled in a given state: the action $x{=}v$ is only enabled in a state for

[6] Since channels are only used for synchronized handshaking there is no need to treat channel contents as part of the state.

$$\frac{}{h?x, s \xrightarrow{h?v} x{:=}v, s} \qquad \frac{}{h?x, s \xrightarrow{\delta_{h?}} h?x, s}$$

$$\frac{}{h!v, s \xrightarrow{h!v} \mathbf{skip}, s} \qquad \frac{}{h!v, s \xrightarrow{\delta_{h!}} h!v, s}$$

$$\frac{G_1, s \xrightarrow{\lambda} P_1, s' \quad \lambda \notin \Delta}{G_1 \,\square\, G_2, s \xrightarrow{\lambda} P_1, s'} \qquad \frac{G_2, s \xrightarrow{\lambda} P_2, s' \quad \lambda \notin \Delta}{G_1 \,\square\, G_2, s \xrightarrow{\lambda} P_2, s'} \qquad \frac{G_1, s \xrightarrow{\delta_X} G_1, s \quad G_2, s \xrightarrow{\delta_Y} G_2, s}{G_1 \,\square\, G_2, s \xrightarrow{\delta_{X \cup Y}} G_1 \,\square\, G_2, s}$$

$$\frac{}{\mathbf{while}\ b\ \mathbf{do}\ P, s \xrightarrow{\delta} \mathbf{if}\ b\ \mathbf{then}\ P; \mathbf{while}\ b\ \mathbf{do}\ P\ \mathbf{else}\ \mathbf{skip}, s}$$

$$\frac{}{P_1 \sqcap P_2, s \xrightarrow{\delta} P_1, s} \qquad \frac{}{P_1 \sqcap P_2, s \xrightarrow{\delta} P_2, s}$$

$$\frac{P_1, s \xrightarrow{\lambda} P_1', s'}{P_1 \| P_2, s \xrightarrow{\lambda} P_1' \| P_2, s'} \qquad \frac{P_2, s \xrightarrow{\lambda} P_2', s'}{P_1 \| P_2, s \xrightarrow{\lambda} P_1 \| P_2', s'}$$

$$\frac{P_1, s \xrightarrow{\lambda_1} P_1', s \quad P_2, s \xrightarrow{\lambda_2} P_2', s \quad match(\lambda_1, \lambda_2)}{P_1 \| P_2, s \xrightarrow{\delta} P_1' \| P_2', s}$$

$$\frac{P, s \xrightarrow{\lambda} P', s' \quad chan(\lambda) \neq h}{\mathbf{local}\ h\ \mathbf{in}\ P, s \xrightarrow{\lambda} \mathbf{local}\ h\ \mathbf{in}\ P', s'} \qquad \frac{P, s \xrightarrow{\delta_X} P', s}{\mathbf{local}\ h\ \mathbf{in}\ P, s \xrightarrow{\delta_{X \setminus h}} \mathbf{local}\ h\ \mathbf{in}\ P', s'}$$

$$\frac{}{\mathbf{skip}, s\ \mathbf{term}} \qquad \frac{P_1, s\ \mathbf{term} \quad P_2, s\ \mathbf{term}}{P_1 \| P_2, s\ \mathbf{term}} \qquad \frac{P, s\ \mathbf{term}}{\mathbf{local}\ h\ \mathbf{in}\ P, s\ \mathbf{term}}$$

Fig. 1. Operational semantics for processes

which the value of x is v, and in fact a state is uniquely determined by the set of evaluation actions which it enables. We write $[s \mid x : v]$ for the state obtained from s by updating the value of x to v.

The operational semantics for expressions involves non-terminal transitions of form $e, s \xrightarrow{\mu} e', s$ and terminal transitions of form $e, s \xrightarrow{\mu} v$, where μ is an evaluation action and v is an integer. Similarly for boolean expressions. The operational semantics for processes involves transitions of form $P, s \xrightarrow{\lambda} P', s'$ and a termination predicate P, s **term**. Some transition rules are listed in Figure 1. (We omit several rules, including those dealing with sub-expression evaluation and the rules for sequential constructs, which are standard.) Note that an external choice $G_1 \,\square\, G_2$ is "resolved" only when a communication occurs.

A *transition sequence* of process P is a sequence of transitions of form

$$P, s_0 \xrightarrow{\lambda_0} P_1, s_0' \quad P_1, s_1 \xrightarrow{\lambda_1} P_2, s_1' \quad P_2, s_2 \xrightarrow{\lambda_2} P_3, s_2' \quad \ldots$$

either infinite or ending in a terminal configuration. A *computation* is a transition sequence in which the state never changes between steps, so that $s_i' = s_{i+1}$. A transition sequence (or a computation) of P is *fair* if it contains a complete transition sequence for each syntactic sub-process of P, and no pair of sub-processes is permanently blocked yet attempting to synchronize. For example,

the computation

$$a?x \| a!0, s \xrightarrow{\delta_{a?}} a?x \| a!0, s \xrightarrow{\delta_{a!}} a?x \| a!0, s \xrightarrow{\delta_{a?}} a?x \| a!0, s \xrightarrow{\delta_{a!}} \cdots$$

is not fair, because the two processes block on matching directions. However,

$$a?x \| a!0, s \xrightarrow{a?1} x{:=}1 \| a!0, s \xrightarrow{x:=1} \mathbf{skip} \| a!0, s' \xrightarrow{\delta_{a!}} \mathbf{skip} \| a!0, s' \xrightarrow{\delta_{a!}} \cdots$$

where $s' = [s \mid x : 1]$, is fair because only one process is blocked; there is a fair computation of the process $a!1 \| (a?x \| a!0)$ in which the first process performs $a!1$ and the second performs the above transition sequence. Similarly, the sequence $a!0 \| b!1, s \xrightarrow{\delta_{a!}} a!0 \| b!1, s \xrightarrow{\delta_{b!}} a!0 \| b!1, s \xrightarrow{\delta_{a!}} a!0 \| b!1, s \xrightarrow{\delta_{b!}} \cdots$ qualifies as fair, and corresponds to the trace $(\delta_{a!}\delta_{b!})^\omega$ of process $a!0 \| b!1$. We write $P \xrightarrow{\alpha}$ when P has a maximal fair transition sequence on which the actions form the trace α.

7 Semantic Properties

The denotational and operational characterizations of fair traces coincide:

Theorem 2 (Congruence of Operational and Denotational Semantics) *For every process P, $\mathcal{T}(P) = \{\alpha \mid P \xrightarrow{\alpha}\}$.*

Suppose we can observe communication sequences, including persistent blocking and the values of non-local variables, but we cannot backtrack to try alternative runs. This notion of observable behavior suffices to allow safety and liveness analysis and is equivalent to observing traces. It is an obvious consequence of compositionality that our trace semantics is fully abstract for this notion of behavior:

Theorem 3 (Full Abstraction for Synchronous Trace Semantics) *Two processes P_1 and P_2 have the same trace sets iff they have the same observable behavior in all contexts.*

This generalizes the analogous well known full abstraction results for failures semantics, which hold in a much more limited setting, without fairness [33]. The significance is not full abstraction *per se* but the construction of a *simple* trace-based semantics that incorporates a reasonable form of fairness and synchronized communication while supporting safety and liveness analysis.

To demonstrate that trace semantics distinguishes between processes with different deadlock capabilities, note that:

$$\delta_X{}^\omega \in \mathcal{T}((a?x \to P) \,\Box\, (b?x \to Q)) \iff X = \{a?, b?\}$$

$$\delta_X{}^\omega \in \mathcal{T}((a?x \to P) \sqcap (b?x \to Q)) \iff X = \{a?\} \text{ or } X = \{b?\}.$$

If we run these processes in a context which is only capable of communicating on channel b, such as **local** a, b **in** $([-] \| b!0)$, the first process would behave like $x{:=}0$; **local** a, b **in** Q but the second would also have the possibility of behaving like **local** a, b **in** $((a?x \to P) \| b!0)$, which is deadlocked and has trace set $\{\delta^\omega\}$.

The following semantic equivalences, to be interpreted as equality of trace sets, illustrate how our model supports reasoning about process behavior.

Theorem 4 (Fair Synchronous Laws)
The following laws of equivalence hold in synchronous trace semantics:

1. **local** h **in** $(h?x;P)\|(h!v;Q) \;=\; $ **local** h **in** $(x{:=}v;\ (P\|Q))$
2. **local** h **in** $(h?x;P)\|(Q_1;Q_2) \;=\; Q_1;$ **local** h **in** $(h?x;P)\|Q_2$
   provided $h \notin \mathtt{chans}(Q_1)$
3. **local** h **in** $(h!v;P)\|(Q_1;Q_2) \;=\; Q_1;$ **local** h **in** $(h!v;P)\|Q_2$
   provided $h \notin \mathtt{chans}(Q_1)$.

These properties reflect our assumption of fairness, and are particularly helpful in proving liveness properties. They are not valid in an unfair semantics: if execution is unfair there is no guarantee in the first law that the synchronization will eventually occur, and there is no guarantee in the second or third laws that the right-hand process will ever execute its initial (non-local) code.

8 Pomset Semantics

We now introduce a "truly concurrent" interpretation for our process language and show that it is a natural generalization of the trace semantics. We adapt Pratt-style "pomsets" to handle synchronization and our notion of fairness.

A *pomset* $(T, <)$ is a partially ordered countable multiset of actions: T is a multiset whose elements are drawn from the set Σ of actions, and $<$ is a partial order on T, representing a "precedence" relation on action occurrences in T. Actually we allow the precedence relation to be a pre-order: when T contains a pair of matching communication occurrences which precede each other this will force a synchronization. We also assume that every action dominates finitely many actions, so the precedence relation is well founded. We write $|T|$ for the cardinality of T, which is either finite or ω.

The *kernel* of an ordering relation $<$ is the subset consisting of the pairs (μ, μ') such that $\mu < \mu'$ and there is no μ'' such that $\mu < \mu'' < \mu'$. The full ordering relation can be recovered by taking the transitive closure of the kernel. We also elide non-final occurrences of δ, for example replacing $\mu < \delta < \mu'$ by $\mu < \mu'$. (This is analogous to our earlier convention for concatenating δ.)

Each pomset $(T, <)$ determines a set of traces, those traces containing all of the action occurrences from T in a linear order consistent with the precedence relation, *modulo* synchronization. We will refer to these as the traces *consistent* with the pomset. A single trace can be viewed as a pomset with a linear precedence order. A pomset consists of a number of connected components, or *threads*.

We define $T_1;T_2 = T_1$ if $|T_1| = \omega$, otherwise $T_1;T_2$ is the ordering on $T_1 \cup T_2$ obtained by putting T_2 after T_1. Likewise we define $T^0 = \{\delta\}$, $T^{k+1} = T;T^k$ for $k \geq 0$, and T^ω. $T_1\|T_2$ is the disjoint union of T_1 and T_2 ordered with the disjoint union of the orderings from T_1 and T_2. We say that a pomset is *fair* iff it does not contain a pair of concurrent threads which eventually block on a pair of matching directions. For example, the pomset $\{a!0{\delta_{b!}}^\omega, a?0{\delta_{b?}}^\omega\}$ is unfair. We write T *fair* to indicate that T is a fair pomset.

We define $T \preceq_h T'$ to mean that T' arises by choosing for each occurrence of $h?v$ (or $h!v$) in T a unique concurrent matching action occurrence $h!v$ (respectively, $h?v$) in T, and augmenting the ordering accordingly, with an arrow each way between the matched pairs. This can be formalized as a *synchronizing schedule* for channel h. For a given T and h there may be no such T', or there may be multiple such T', each corresponding to a sequence of synchronization choices. Given a pomset T' in which all visible actions on h are matched, we define $T'\backslash h$ to be the result of replacing all matching pairs by δ (i.e. enforcing synchronization), replacing every δ_X by $\delta_{X\backslash h}$, and eliding non-final δ actions.

We assume that the pomset semantics of expressions is given, so that for an expression e, $\mathcal{P}(e)$ is a set of pairs of the form (T, v), where $v \in Z$ and T is a pomset of evaluation actions. Intuitively, $(T, v) \in \mathcal{P}(e)$ means that if the evaluation trace of e is consistent with T then v is a possible final value. We also assume given the pomset semantics of boolean expressions, and we let $\mathcal{P}(b)_{\mathbf{true}} = \{T \mid (T, \mathbf{true}) \in \mathcal{P}(b)\}$ and similarly for $\mathcal{P}(b)_{\mathbf{false}}$. A process P denotes a set (or "family") $\mathcal{P}(P)$ of pomsets.

Definition 2 (Synchronous Pomset Semantics)
The pomset semantics $\mathcal{P}(P)$ is given compositionally by:

$$
\begin{aligned}
\mathcal{P}(\mathbf{skip}) &= \{\{\delta\}\} \\
\mathcal{P}(x{:=}e) &= \{T; \{x{:=}v\} \mid (T, v) \in \mathcal{P}(e)\} \\
\mathcal{P}(P_1; P_2) &= \{T_1; T_2 \mid T_1 \in \mathcal{P}(P_1) \;\&\; T_2 \in \mathcal{P}(P_2)\} \\
\mathcal{P}(\mathbf{if}\ b\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2) &= \mathcal{P}(b)_{\mathbf{true}}; \mathcal{P}(P_1) \cup \mathcal{P}(b)_{\mathbf{false}}; \mathcal{P}(P_2) \\
\mathcal{P}(\mathbf{while}\ b\ \mathbf{do}\ P) &= (\mathcal{P}(b)_{\mathbf{true}}; \mathcal{P}(P))^*; \mathcal{P}(b)_{\mathbf{false}} \;\cup\; (\mathcal{P}(b)_{\mathbf{true}}; \mathcal{P}(P))^\omega \\
\mathcal{P}(h?x) &= \{\{h?v\} \mid v \in Z\} \cup \{\{{\delta_{h?}}^\omega\}\} \\
\mathcal{P}(h!e) &= \{T; \{h!v\} \mid (T, v) \in \mathcal{P}(e)\} \cup \{\{{\delta_{h!}}^\omega\}\} \\
\mathcal{P}(P_1 \| P_2) &= \{T_1 \| T_2 \mid\; T_1 \in \mathcal{P}(P_1) \;\&\; T_2 \in \mathcal{P}(T_2) \;\&\; (T_1 \| T_2)\ \mathit{fair}\} \\
\mathcal{P}(P_1 \sqcap P_2) &= \mathcal{P}(P_1) \cup \mathcal{P}(P_2) \\
\mathcal{P}(\mathbf{local}\ h\ \mathbf{in}\ P) &= \{T'\backslash h \mid T \in \mathcal{P}(P) \;\&\; T \preceq_h T'\} \\
\mathcal{P}(h?x \to P) &= \{\{h?v\}; T \mid v \in Z \;\&\; T \in \mathcal{P}(P)\} \;\cup\; \{\{{\delta_{h?}}^\omega\}\} \\
\mathcal{P}(G_1 \,\square\, G_2) &= \{T \in \mathcal{P}(G_1) \cup \mathcal{P}(G_2) \mid T \cap \Delta^\omega = \{\}\} \cup \\
&\quad \{\{{\delta_{X \cup Y}}^\omega\} \mid \{{\delta_X}^\omega\} \in \mathcal{P}(G_1) \;\&\; \{{\delta_Y}^\omega\} \in \mathcal{P}(G_2)\}
\end{aligned}
$$

It can be proven by structural induction that every pomset $T \in \mathcal{P}(P)$ is fair.

An Example

Let $\mathit{buff}_1(\mathit{in}, \mathit{mid})$ be **local** x **in while true do** $(\mathit{in}?x; \mathit{mid}!x)$, which behaves like a 1-place buffer. Let $\mathit{buff}_1(\mathit{mid}, \mathit{out})$ be similarly defined. It is easy to prove using pomsets that with synchronized communication

$$\mathit{buff}_2(\mathit{in}, \mathit{out}) =_{\mathrm{def}} \mathbf{local}\ \mathit{mid}\ \mathbf{in}\ \mathit{buff}_1(\mathit{in}, \mathit{mid}) \| \mathit{buff}_1(\mathit{mid}, \mathit{out})$$

behaves like a 2-place buffer. One can also use the semantics to analyze a variety of alternative buffer-like constructs, such as

$$\mathbf{local}\ \mathit{mid}\ \mathbf{in}\ \mathit{buff}_2(\mathit{in}, \mathit{mid}) \| \mathit{buff}_2(\mathit{mid}, \mathit{out})$$

and one can validate a number of general buffer laws as in [33].

9 Recovering Traces

The pomset semantics determines the trace semantics in a natural manner.

Definition 3 *The set of synchronous traces consistent with a pomset T, written $\mathcal{L}(T)$, consists of all traces which arise by fair interleaving the threads of T, possibly allowing synchronization.*

Equivalently, $\mathcal{L}(T)$ is the set of all linear orders on the multi-set T which extend the order of T, allowing for the possibility of synchronization.

Note that distinct pomset families may determine the same trace sets. For example the pomset families $\{\{a!0, b!1\}, \{a!0, {\delta_{b!}}^{\omega}\}, \{b!1, {\delta_{a!}}^{\omega}\}, \{{\delta_{a!}}^{\omega}, {\delta_{b!}}^{\omega}\}\}$ and $\{\{a!0\, b!1\}, \{b!1\, a!0\}, \{a!0, {\delta_{b!}}^{\omega}\}, \{b!1, {\delta_{a!}}^{\omega}\}, \{{\delta_{a!}}^{\omega}, {\delta_{b!}}^{\omega}\}\}$ both determine the trace set of $a!0 \| b!1$.

Assuming that the pomset semantics of expressions is consistent with the trace semantics, i.e. that for all e, $\mathcal{T}(e) = \{(\rho, v) \mid \rho \in \mathcal{L}(T)\ \&\ (T, v) \in \mathcal{P}(e)\}$, with a similar assumption for boolean expressions, we can prove the analogous property for processes by structural induction:

Theorem 5 *For all processes P, $\mathcal{T}(P) = \bigcup \{\mathcal{L}(T) \mid T \in \mathcal{P}(P)\}$.*

Note the obvious but useful corollary:

Corollary 6 *For all P_1 and P_2, if $\mathcal{P}(P_1) = \mathcal{P}(P_2)$ then $\mathcal{T}(P_1) = \mathcal{T}(P_2)$.*

Many laws of process equivalence hold for pomset semantics, and can be proven without dealing with fully expanded trace sets. For example, the pomset semantics also satisfies standard algebraic laws, such as associativity of parallel composition and both forms of choice:

Theorem 7 *For all processes P_1, P_2, P_3 and all guarded processes G_1, G_2, G_3,*

$$\mathcal{P}(P_1 \| (P_2 \| P_3)) = \mathcal{P}((P_1 \| P_2) \| P_3)$$
$$\mathcal{P}(P_1 \sqcap (P_2 \sqcap P_3)) = \mathcal{P}((P_1 \sqcap P_2) \sqcap P_3)$$
$$\mathcal{P}(G_1 \,\square\, (G_2 \,\square\, G_3)) = \mathcal{P}((G_1 \,\square\, G_2) \,\square\, G_3)$$

Using the above Corollary such laws transfer immediately to trace semantics, giving an alternative proof of the validity of these laws in the trace semantics. Pomset semantics thus provides a potentially more succinct model of process behavior and an alternative compositional approach to parallel program analysis.

Our pomset semantics and even our trace semantics make certain behavioral distinctions which might seem more consistent with the philosophy of "true concurrency" than with the trace-theoretic rationale for our models. In particular the so-called "interleaving law" [23, 18, 21] does not hold, and parallel composition cannot be "expanded away" and expressed equivalently as a guarded choice of interleavings. For example, $\mathcal{P}(a!0 \| b!1)$ is given above, whereas $\mathcal{P}(a!0; b!1 \sqcap b!1; a!0)$ is the family

$$\{\{a!0\, b!1\}, \{b!1\, a!0\}, \{a!0\, {\delta_{b!}}^{\omega}\}, \{b!1\, {\delta_{a!}}^{\omega}\}, \{{\delta_{a!}}^{\omega}\}, \{{\delta_{b!}}^{\omega}\}\}$$

so that $a!0 \| b!1$ is not pomset-equivalent to $(a!0; b!1) \sqcap (b!1; a!0)$. Indeed this distinction also holds in the trace semantics, since

$$\mathcal{T}(a!0 \| b!1) \cap \Delta^\omega = {\delta_{a!}}^\omega \| {\delta_{b!}}^\omega = ({\delta_{a!}}^* \delta_{b!} {\delta_{b!}}^* \delta_{a!})^\omega$$

$$\mathcal{T}(a!0; b!1 \sqcap b!1; a!0) \cap \Delta^\omega = \{{\delta_{a!}}^\omega, {\delta_{b!}}^\omega\}.$$

This difference in trace sets can be explained intuitively, without appealing to considerations of true concurrency, since $a!0 \| b!1$ can be observed (if placed in a suitable environment) waiting repeatedly for action on one of the two channels, whereas the other process makes a non-deterministic choice and thereafter fixates on one particular channel. Note that $a!0 \| b!1$ also fails to be trace- or pomset-equivalent to $(a!0 \to b!1) \,\Box\, (b!1 \to a!0)$, since

$$\mathcal{T}((a!0 \to b!1) \,\Box\, (b!1 \to a!0)) \cap \Delta^\omega = \{{\delta_{\{a!,b!\}}}^\omega\},$$

so neither form of non-deterministic choice can be used to expand away a parallel composition.

10 Asynchronous Communication

Now suppose that communication is asynchronous, as in non-deterministic Kahn-style networks, so that output actions are always enabled, and channels behave like unbounded queues; a process wishing to perform input from a channel must wait (only) if the queue is empty. We can easily adapt the trace semantics to model asynchronous communication. We only need to include blocking actions δ_X where X is a set of *input* directions[7]. We redefine $\alpha_1 \| \alpha_2$ to be the set of fair interleavings of α_1 with α_2, without allowing any synchronization. We say that α is *local for* h if the communications on h along α obey the queue discipline, and we redefine $\alpha \backslash h$ to replace all communications on h by δ and replace δ_X by $\delta_{X \setminus h}$.

Definition 4 *The set $\mathcal{AT}(P)$ of asynchronous traces of P is defined compositionally, exactly as for the synchronous traces but with modifications in the clauses for output, parallel composition, and local channels, which become:*

$$\begin{aligned} \mathcal{AT}(h!e) &= \{\rho\, h!v \mid (\rho, v) \in \mathcal{T}(e)\} \\ \mathcal{AT}(P_1 \| P_2) &= \bigcup\{\alpha_1 \| \alpha_2 \mid \alpha_1 \in \mathcal{AT}(P_1) \;\&\; \alpha_2 \in \mathcal{AT}(P_2)\} \\ \mathcal{AT}(\textbf{local } h \textbf{ in } P) &= \{\alpha \backslash h \mid \alpha \in \mathcal{AT}(P) \;\&\; \alpha \textit{ local for } h\} \end{aligned}$$

The asynchronous trace semantics also validates the associativity laws for parallel composition and both forms of choice.

The asynchronous operational semantics is obtained by making similar adjustments to the rules for output, parallel composition, and local channels, and

[7] It would obviously suffice to work with blocking actions decorated with the set of channel names h rather than the corresponding input directions $h?$, but we resist this temptation to emphasize the similarity with the synchronous case.

including channel contents as part of the state. The operational notion of fair transition sequence is as before, except that the transition relation no longer includes synchronizing steps. Again the denotationally characterized trace set coincides with the operationally characterized trace set, and we have full abstraction with respect to communication behavior.

Theorem 8 (Congruence of Denotational and Operational Semantics) *For every process P, $\mathcal{AT}(P)$ consists of the traces generated by the fair asynchronous transition sequences of P.*

Theorem 9 (Full Abstraction for Asynchronous Trace Semantics) *Two processes P_1 and P_2 have the same asynchronous trace sets iff they have the same asynchronous communication behavior in all contexts.*

We can define an asynchronous pomset semantics $\mathcal{AP}(P)$, again adjusting the clauses for output, parallel composition and local channels. We no longer need a side condition in the clause for $P_1 \| P_2$: every trace consistent with the disjoint union $T_1 \| T_2$ will represent a fair asynchronous behavior of the parallel process. We redefine $T \preceq_h T'$ to mean that T' arises by choosing, for each *input* occurrence $h?v$ in T an output occurrence $h!v$ in T which *justifies* it, all choices respecting the precedence order of T and the queue discipline of the channel, and augmenting the ordering so that each input is preceded by its justifying output. This can be viewed as imposing an *asynchronous schedule*. For a given T and h, there may be no such T', in which case T does not describe any traces which are local for h, or there may be multiple such T', each corresponding to a different sequence of scheduling choices. Given a pomset T' in which all inputs on h are justified in this manner, we define $T' \backslash h$ to replace each communication on h by δ, replace δ_X by $\delta_{X \setminus h}$, and elide non-final δ actions.

Definition 5 (Asynchronous Pomset Semantics) *The asynchronous pomset semantics is given compositionally as for the synchronous pomset semantics, with the following modifications:*

$$\begin{aligned} \mathcal{AP}(h!e) &= \{T; \{h!v\} \mid (T, v) \in \mathcal{P}(e)\} \\ \mathcal{AP}(P_1 \| P_2) &= \{T_1 \| T_2 \mid \ T_1 \in \mathcal{AP}(P_1) \ \&\ T_2 \in \mathcal{AP}(P_2)\} \\ \mathcal{AP}(\textbf{local } h \textbf{ in } P) &= \{T' \backslash h \mid T \in \mathcal{AP}(P) \ \&\ T \preceq_h T'\} \end{aligned}$$

Definition 6 *The asynchronous trace set denoted by a pomset T, written $\mathcal{AL}(T)$, is the set of all traces which arise by fair interleaving the threads of T.*

Again the asynchronous traces of a process can be recovered naturally:

Theorem 10 *For all processes P, $\mathcal{AT}(P) = \bigcup \{\mathcal{AL}(T) \mid T \in \mathcal{AP}(P)\}$.*

Corollary 11 *For all P_1 and P_2, if $\mathcal{AP}(P_1) = \mathcal{AP}(P_2)$ then $\mathcal{AT}(P_1) = \mathcal{AT}(P_2)$.*

The process **local** *mid* **in** $\mathit{buff}_1(\mathit{in}, \mathit{mid}) \| \mathit{buff}_1(\mathit{mid}, \mathit{out})$ behaves like a finite *unbounded* buffer if we assume asynchronous commuication, in contrast to the 2-place buffer which described its behavior under synchronous communication.

The following laws hold for *asynchronous* trace semantics, and are analogues of the first two laws given earlier for synchronous communication. The third law does not hold, because of the assumption that output is always enabled.

Theorem 12 (Fair Asynchronous Laws)
The following laws of equivalence hold in asynchronous trace semantics:

1. $\textbf{local}\ h\ \textbf{in}\ (h?x;P)\|(h!v;Q) \ = \ \textbf{local}\ h\ \textbf{in}\ (x{:=}v;P)\|Q$
2. $\textbf{local}\ h\ \textbf{in}\ (h?x;P)\|(Q_1;Q_2) \ = \ Q_1;\textbf{local}\ h\ \textbf{in}\ (h?x;P)\|Q_2$
   provided $h \notin \mathtt{chans}(Q_1)$.

These laws also hold for asynchronous pomset semantics.

11 Related Work

Hoare's "trace model" of CSP [20] interpreted a process as a prefix-closed set of finite communication traces, recording only visible actions such as $h!v$ and $h?v$. Each such trace represents a *partial* behavior. Hoare's model did not treat infinite behaviors, and is mainly suitable for proving safety properties. The failures semantics of CSP [11] modelled a process as a set of failures (α, X) consisting of a finite sequence α of communications and a "refusal set" X. Our notion of trace subsumes failures: a process with failure (α, X) would have a trace $\alpha{\delta_Y}^\omega$, for some set Y disjoint from X. Our notion of trace is more general, allowing for instance traces of form $\alpha(\delta_A\delta_B)^\omega$ which cannot be represented in failure format. The extra generality is needed in order to cope properly with fair parallel composition. Our traces represent *entire* computations, so our trace sets are not prefix-closed. Again this is more than a philosophical difference: one cannot deduce the fair traces of a parallel process by looking at the prefixes of the traces of its constituent processes. The failures semantics and its later more refined extensions, all building on a prefix-closed trace set, were not designed with fairness in mind [33].

Our use of traces in which blocking actions play a crucial role is reminiscent of Phillips-style refusal testing, although Phillips did not incorporate fairness [30]. Refusal testing assumes that under certain circumstances one may detect that a process is *unable* to perform a communication; one may obviously view a process which is capable of performing δ_X as being (potentially) unable to perform $h?v$ for any $h? \in X$. It is not clear how to generalize Phillips's testing equivalence to deal properly with fairness.

Older's Ph.D. thesis [24] provides a framework capable of modelling several forms of fairness in the synchronous setting. She introduced generalized notions of fairness and blocking *modulo* a set of directions, and her models incorporated extensive book-keeping information concerning enabling, together with cleverly devised but complex closure conditions designed to achieve full abstraction [25, 10]. As Older comments, it is questionable if these fairness notions are useful abstractions of realistic schedulers, since their implementation requires meticulous attention to so much enabling information. Moreover, these forms of fairness

tend to lack equivalence robustness, being sensitive to subtle nuances in the formulation of the operational semantics [1].

In contrast we assume a robust yet simple form of weakly fair execution, suitably adapted to deal reasonably with synchronization to ensure that two processes waiting to perform matching communications will not be ignored forever. This property would be guaranteed for instance by any round-robin scheduler which runs each process for an randomly chosen number of steps, and also uses a round-robin strategy to look for matching communications if the chosen process blocks while attempting input or output. This form of fairness is a simple variant of weak process fairness and we believe this is a reasonable abstraction from the behavior of realistic implementations. Furthermore, in adopting this notion of fairness we avoid the need for separate book-keeping: the traces themselves can be designed to carry the relevant information in their δ actions. Older's notion of being blocked but fair *modulo* a set X of directions corresponds to a trace β such that $\mathit{blocks}(\beta) \subseteq X$. It would be interesting to see if the other forms of fairness discussed by Older can be treated within our framework.

Hennessy [17] gave an earlier treatment of a CCS-like language with weakly fair execution. Parrow [29] discusses various fairness notions for CCS-like processes. Costa and Stirling gave an operational semantics for a CCS-like language assuming either weak or strong process fairness [13, 14].

Partial-order semantics of various kinds, such as Pratt-style pomsets [31, 32], Winskel's event structures [35][8], and Petri nets [28] have been widely used, with parallel composition interpreted as so-called "true concurrency" rather than interleaving. Avoiding explicit interleaving can be advantageous, both in avoiding a combinatorial explosion and in side-stepping the need to define fairmerge operations. Indeed our motivation in developing a pomset formulation was to obtain an alternative and more tractable methodology for dealing with trace sets. The pomset union operation nicely handles weakly fair parallel composition in the absence of blocking. We show how to extend these ideas to incorporate blocking.

References

1. K. R. Apt, N. Francez, and S. Katz, *Appraising fairness in languages for distributed programming*, Distributed Computing, 2(4):226-241, 1988.
2. J. A. Bergstra and J.-W. Klop, *Process Algebra for Synchronous Communication*, Information and Computation, 60(1/3):109–137, 1984.
3. J. A. Bergstra, J.-W. Klop, and J. Tucker, *Process algebra with asynchronous communication mechanisms*, Seminar on Concurrency, Springer LNCS 197, pp. 76–95, 1985.
4. F. de Boer, J. Kok, C. Palamidessi, and J. Rutten, *The failure of failures in a paradigm for asynchronous concurrency*, CONCUR'91, Springer LNCS 527, pp. 111-126, 1991.

[8] Event structures can be seen as pomsets equipped with a "conflict relation", although this characterization does not reflect their original, independent, development and subsequent usage.

5. S. Brookes, *Full abstraction for a shared-variable parallel language*, Information and Computation, 127(2):145-163, June 1996.
6. S. Brookes, *The Essence of Parallel Algol*, 11^{th} LICS, pp. 164-173, July 1996.
7. S. Brookes, *Idealized CSP: Combining Procedures with Communicating Processes*, Proc. MFPS XIII, ENTCS 6, Elsevier Science, 1997.
8. S. Brookes, *On the Kahn Principle and Fair Networks*, MFPS XIV, Queen Mary Westfield College, University of London, May 1998.
9. S. Brookes, *Deconstructing CCS and CSP: Asynchronous Communication, Fairness and Full Abstraction*, MFPS XVI, 2000.
10. S. Brookes and S. Older, *Full abstraction for strongly fair communicating processes*, MFPS XI, ENTCS 1, Elsevier Science, 1995.
11. S. Brookes, C. A. R. Hoare, and A. W. Roscoe, *A Theory of Communicating Sequential Processes*, JACM 31(3):560-599, July 1984.
12. S. Brookes, and A. W. Roscoe, *An improved failures model for CSP*, Seminar on concurrency, Springer-Verlag, LNCS 197, 1984.
13. G. Costa and C. Stirling, *A fair calculus of communicating systems*, ACTA Informatica 21:417-441, 1984.
14. G. Costa and C. Stirling, *Weak and strong fairness in CCS*, Technical Report CSR-16-85, University of Edinburgh, January 1985.
15. N. Francez, **Fairness**, Springer-Verlag, 1986.
16. R. van Glabbeek, *The Linear Time–Branching Time Spectrum*, **Handbook of Process Algebra**, Elsevier, 2001.
17. M. Hennessy, *An algebraic theory of fair asynchronous communicating processes*, Theoretical Computer Science, 49:121-143, 1987.
18. M. Hennessy and R. Milner, *Algebraic laws for nondeterminism and concurrency*, JACM 32(1):137–161, 1985.
19. C. A. R. Hoare, *Communicating Sequential Processes*, CACM 21(8):666–677, 1978.
20. C. A. R. Hoare, *A Model for Communicating Sequential Processes*, Technical Monograph PRG-22, Oxford University, June 1981.
21. C. A. R. Hoare, **Communicating Sequential Processes**, Prentice-Hall, 1985.
22. R. Milner, *A Calculus of Communicating Systems*, Springer LNCS 92, 1980.
23. R. Milner, **Communication and Concurrency**, Prentice-Hall, London, 1989.
24. S. Older, *A Denotational Framework for Fair Communicating Processes*, Ph.D. thesis, Carnegie Mellon University, December 1996.
25. S. Older, *A Framework for Fair Communicating Processes*, Proc. MFPS XIII, ENTCS 6, Elsevier Science, 1997.
26. S. Owicki and L. Lamport, *Proving liveness properties of concurrent programs*, ACM TOPLAS, 4(3): 455-495, July 1982.
27. D. Park, *On the semantics of fair parallelism.* **Abstract Software Specifications**, Springer-Verlag LNCS vol. 86, 504–526, 1979.
28. C. A. Petri, *Concepts of Net Theory*, Symposium on Mathematical Foundations of Computer Science, September 1973.
29. J. Parrow, *Fairness Properties in Process Algebras*, Ph. D. thesis, University of Uppsala, 1985.
30. I. Phillips, *Refusal testing*, Theoretical Computer Science, 50(2):241–284, 1987.
31. V. Pratt, *On the Composition of Processes*, Proc. 9^{th} ACM POPL Symp., 1982.
32. V. Pratt, *Modeling concurrency with partial orders*, International Journal on Parallel Processing, 15(1): 33–71, 1986.
33. A. W. Roscoe, **The Theory and Practice of Concurrency**, Prentice-Hall, 1998.
34. V. Saraswat, M. Rinard, and P. Panangaden, *Semantic foundations of concurrent constraint programming*, Proc. 18^{th} ACM POPL Symposium, 1991.
35. G. Winskel, *Events in Computation*, Ph. D. thesis, Edinburgh University, 1980.