Fair Communicating Processes

Stephen Brookes

Published in
A Classical Mind: Essays in Honour of C. A. R. Hoare,

1 Introduction

In an article published in August 1978 Tony Hoare introduced the programming language CSP (Communicating Sequential Processes) [7]. Although Hoare himself stated that the concepts and notations used in this paper “should not be regarded as suitable for use as a programming language, either for abstract or for concrete programming”, these concepts and notations have had significant impact on the design of languages such as Ada [9] and occam [10]. In addition Hoare's ideas have stimulated much research on the development of semantic models and proof methodologies for parallel languages.

The original CSP language is a simple yet powerful and elegant generalization of Dijkstra's guarded commands [4], permitting parallel execution of sequential commands ("processes"). Processes have disjoint "local" states and may communicate by synchronized message-passing: communication occurs when one process names another as destination for output and the second process names the first as source for input, whereupon they perform a synchronized handshake. The syntax of CSP was closely based on Dijkstra's notation for guarded commands, generalized to permit communication guards, and using an \(n\)-ary parallel composition of named sequential processes. Nested parallel compositions were not allowed, and only input commands were allowed as guards. These restrictions were imposed mainly for pragmatic reasons, and have often been relaxed or removed in later developments. For instance, it is common to permit output guards; occam uses channel names rather than process names, has a binary associative form of
parallel composition, and allows nested parallelism; Plotkin [15] discusses a variant of CSP with a more general scoping facility for process names.

Hoare’s work on semantic models for communicating processes has focused mainly on a more abstract process language, which has come to be known as Theoretical CSP (or TCSF) [2, 8]. Like Milner’s Calculus of Communicating Systems (CCS) [11, 12], TCSP provides a collection of primitive processes and operations (like parallel composition) for building complex processes from simpler ones. Atomic actions (like input and output) are treated as events drawn from some given alphabet. Processes may be characterized in terms of the sequences of events (or traces) that they may perform. Thus, in the trace model of [6] the denotation of a process is taken to be a non-empty, prefix-closed set of finite sequences of events.

In many applications it is reasonable to assume that programs executing in parallel are not delayed forever. This is known as a fairness assumption [5]. Hoare remarked in [7] that “an efficient implementation (of CSP) should try to be reasonably fair and should ensure that an output command is not delayed unreasonably often after it first becomes executable.” Hoare also stated that he was “fairly sure”¹ that a programming language definition should not specify that an implementation must be fair, and that the programmer should be responsible for proving that his program terminates correctly without relying on fairness in the implementation.

Hoare’s trace model [6] was not designed to incorporate fairness; indeed, this model ignores the possibility of infinite computation and it is difficult to reconcile fair infinite traces with the prefix-closure assumption. The problem remained of finding a satisfactory semantic account of communicating processes that accurately supports reasoning about programs under fairness assumptions. This is the problem addressed by our paper.

We propose a mathematically straightforward trace semantics for a language of fair communicating processes, and we explore some of its properties. We build on the foundational work of David Park, who gave a semantics for a fair shared variable parallel programming language, based on an elegant characterization of a “fairmerge” operation on finite and infinite sequences [14]. Park’s model is tailored specifically to the purpose of modelling the interactions of parallel programs that share a global state. Since we focus on a CSP-like language, with no sharing of state, a rather different model is appropriate. We adapt and generalize Park’s definitions in a natural way.

¹The pun was (presumably) intended.
The language discussed in this paper is essentially a hybrid derived from
the original CSP and CCS. As in CSP we require that processes have disjoint
local states. As in occam we permit nested parallelism and communication
uses named channels rather than process names. We also prefer an abstract
syntax less closely tied to the guarded command notation, using a binary
form of parallel composition. Thus we obtain a language in which processes
themselves may be parallel combinations of processes, so that it might be
preferable to refer to "communicating parallel processes".

We give an operational semantics, then a denotational semantics, and we
show that the two semantic definitions essentially coincide. We then prove
that the denotational semantics is fully abstract [13] with respect to a natural
notion of program behavior. This means that the semantics distinguishes
between two commands if and only if they induce different behavior in some
program context. We discuss a few well known examples, and we suggest
directions for further research.

2 Syntax

The abstract syntax of our programming language is defined as follows. There
are five syntactic sets: Ide, the set of identifiers, ranged over by I; Exp, the
set of expressions, ranged over by E; BExp, the set of boolean expressions
(or conditions), ranged over by B; Chan, the set of channel names, ranged
over by h; and Com, the set of commands, ranged over by C. The abstract
syntax for identifiers, channel names, expressions and conditions will be taken
for granted; all we assume is that identifiers and expressions denote integer
values, boolean expressions denote truth values, and the language contains
the usual arithmetic and boolean operators and constants. For commands
we specify the following grammar:

\[ C ::= \text{skip} | I:=E | C_1; C_2 | \text{if } B \text{ then } C_1 \text{ else } C_2 | \text{while } B \text{ do } C | \]

\[ h?I | h!E | C_1||C_2 | \sum_{i=1}^{k}(\rho_i \rightarrow C_i) | C\setminus h, \]

where the \( \rho_i \) each have one of the forms \( h?I \) or \( h!E \).

We refer to \( h?I \) as an input command and \( h!E \) as an output command.
The form \( \sum_{i=1}^{k}(\rho_i \rightarrow C_i) \) corresponds to a guarded command whose guards
involves input or output\(^2\). The command \(C \setminus h\) is \(C\) restricted on channel \(h\): it will behave like \(C\) except that its ability to communicate on channel \(h\) is removed. Note that processes may have \textit{internal} actions (like assignments to local variables) in addition to communication capabilities.

Parallel composition is denoted \(C_1 || C_2\), and we impose the syntactic constraint that in all such commands the components \(C_1\) and \(C_2\) must have disjoint sets of variables. Formally, we make use of the set \(\text{free}[C]\) of identifiers occurring free in \(C\), given as usual by structural induction on \(C\):

\[
\begin{align*}
\text{free}[\text{skip}] & = \{\} \\
\text{free}[I := E] & = \{I\} \cup \text{free}[E] \\
\text{free}[C_1 ; C_2] & = \text{free}[C_1] \cup \text{free}[C_2] \\
\text{free}[	ext{if } B \text{ then } C_1 \text{ else } C_2] & = \text{free}[B] \cup \text{free}[C_1] \cup \text{free}[C_2] \\
\text{free}[	ext{while } B \text{ do } C] & = \text{free}[B] \cup \text{free}[C] \\
\text{free}[h?I] & = \{I\} \\
\text{free}[h!E] & = \text{free}[E] \\
\text{free}[C_1 || C_2] & = \text{free}[C_1] \cup \text{free}[C_2] \\
\text{free}[\sum_{i=1}^{k} (\rho_i \rightarrow C_i)] & = \bigcup_{i=1}^{k} (\text{free}[\rho_i] \cup \text{free}[C_i]) \\
\text{free}[C \setminus h] & = \text{free}[C]
\end{align*}
\]

We say that \(C\) is well-formed iff for every sub-command of \(C\) with form \(C_1 || C_2\) we have \(\text{free}[C_1] \cap \text{free}[C_2] = \{\}\). For example, \((a?x ; x := x+1 ; a!x) || (y := 0 ; a!y ; a?z)\) is well formed, but \(x := 0 ; [a?x ; x := x + 1] \) is not. Throughout the paper we assume that we deal with well-formed commands.

We also define \(\text{chans}[C]\), the finite set of channel names occurring in \(C\), by structural induction on \(C\):

\[
\begin{align*}
\text{chans}[\text{skip}] & = \{\} \\
\text{chans}[I := E] & = \{\} \\
\text{chans}[C_1 ; C_2] & = \text{chans}[C_1] \cup \text{chans}[C_2] \\
\text{chans}[	ext{if } B \text{ then } C_1 \text{ else } C_2] & = \text{chans}[C_1] \cup \text{chans}[C_2] \\
\text{chans}[	ext{while } B \text{ do } C] & = \text{chans}[C] \\
\text{chans}[h?I] & = \{h\} \\
\text{chans}[h!E] & = \{h\} \\
\text{chans}[C_1 || C_2] & = \text{chans}[C_1] \cup \text{chans}[C_2] \\
\text{chans}[\sum_{i=1}^{k} (\rho_i \rightarrow C_i)] & = \bigcup_{i=1}^{k} (\text{chans}[\rho_i] \cup \text{chans}[C_i]) \\
\text{chans}[C \setminus h] & = \text{chans}[C] \setminus \{h\}
\end{align*}
\]

\(^2\)We omit “mixed” guards with an additional boolean component, since this permits a simpler presentation.
3 Operational Semantics

A state is a finite partial function from identifiers to integer values. We use \( N \) for the set of integers, and we let \( S = [\text{Id} \rightarrow_p N] \) denote the set of states. A typical state will be written in form \([I_1 = n_1, \ldots, I_k = n_k]\). We use \( s \) as a meta-variable ranging over \( S \), and we write \([s \mid I = n]\) for the state which agrees with \( s \) except that it gives identifier \( I \) the value \( n \). The domain of a state, denoted \( \text{dom}(s) \), is the set of identifiers for which the state has a value. Two states \( s_1 \) and \( s_2 \) are disjoint, if and only if their domains do not overlap:

\[
\text{disjoint}(s_1, s_2) \iff \text{dom}(s_1) \cap \text{dom}(s_2) = \emptyset.
\]

We assume for simplicity that expression evaluation always terminates and causes no side-effects, and we assume given the evaluation semantics for boolean and integer expressions: we write \( \langle E, s \rangle \rightarrow^* n \) to indicate that \( E \) evaluates to value \( n \) in state \( s \), with a similar notation for boolean expressions.

For commands, in order to model communication properly we use a labelled transition system, much as in [15]. Configurations have the form \( \langle C, s \rangle \), where \( s \) is a state defined at least on the free identifiers of \( C^3 \):

\[
\text{Conf} = \{ \langle C, s \rangle \mid \text{free}[C] \subseteq \text{dom}(s) \}.
\]

We decorate transitions with a label indicating the type of atomic action involved: \( \epsilon \) represents an internal action, \( h?n \) represents receiving value \( n \) on channel \( h \), and \( h!n \) represents sending value \( n \) along channel \( h \). We let \( \Lambda \) be the set of all labels: \( \Lambda = \{ \epsilon \} \cup \{ h?n, h!n \mid n \in N, h \in \text{Chan} \} \). We use \( \lambda \) as a meta-variable ranging over action labels, and we write

\[
\langle C, s \rangle \xrightarrow{\lambda} \langle C', s' \rangle
\]

to indicate that command \( C \) in state \( s \) can perform an action labelled \( \lambda \), leading to \( C' \) in state \( s' \). Two labels \( \lambda_1 \) and \( \lambda_2 \) match iff one has form \( h?n \) and the other \( h!n \) for some channel name \( h \) and value \( n \); when this holds we write \( \text{match}(\lambda_1, \lambda_2) \).

We identify the successfully terminated (or terminal) configurations by means of a predicate \( \text{term} \). The termination predicate and the transition relations \( \xrightarrow{\lambda}, (\lambda \in \Lambda) \) are defined to be the least relations on configurations

\footnote{This means that we need not be concerned with the possibility of uninitialized identifiers in our semantics.}
\[
\langle \text{skip}, s \rangle \text{term} \\
\langle E, s \rangle \rightarrow^* n \\
\langle I := E, s \rangle \xrightarrow{\epsilon} \langle \text{skip}, [s \mid I = n] \rangle \\
\langle C_1, s \rangle \xrightarrow{\lambda} \langle C_1', s' \rangle \\
\langle C_1; C_2, s \rangle \xrightarrow{\lambda} \langle C_1'; C_2, s' \rangle \\
\langle C_1, s \rangle \text{term} \\
\langle C_1; C_2, s \rangle \xrightarrow{\epsilon} \langle C_2, s \rangle \\
\langle B, s \rangle \rightarrow^* \text{tt} \\
\langle \text{if } B \text{ then } C_1 \text{ else } C_2, s \rangle \xrightarrow{\epsilon} \langle C_1', s \rangle \\
\langle B, s \rangle \rightarrow^* \text{ff} \\
\langle \text{if } B \text{ then } C_1 \text{ else } C_2, s \rangle \xrightarrow{\epsilon} \langle C_2, s \rangle \\
\langle \text{while } B \text{ do } C, s \rangle \xrightarrow{\epsilon} \langle \text{if } B \text{ then } C; \text{ while } B \text{ do } C \text{ else } \text{skip}, s \rangle
\]

Figure 1: Transition rules for sequential constructs

satisfying the axioms and rules of Figures 1 and 2. The rules specify that a parallel composition terminates when all of its components have terminated\(^4\).

Parallel execution is modelled by interleaving, but with the extra possibility of communication. The transition rule for communication between parallel processes is carefully constructed so as to make precise the intuitive description given earlier of the synchronized handshake mechanism. The disjointness assumption on states \(s_1\) and \(s_2\), together with the implicit requirement that \(\text{free}[C_1] \subseteq \text{dom}(s_1)\) and \(\text{free}[C_2] \subseteq \text{dom}(s_2)\), are enough to make the communication rule unambiguous. (To make this precise, we should first note that commands can only affect and be affected by the values of their

\(^4\)We do not model the “distributed termination convention” used in the original paper on CSP.
\[
\begin{array}{c}
\langle h?I, s \rangle \xrightarrow{\text{hln}} \langle \text{skip}, [s \mid I = n] \rangle \\
\langle E, s \rangle \xrightarrow{\ast} n \\
\langle h!E, s \rangle \xrightarrow{\text{hln}} \langle \text{skip}, s \rangle \\
\langle \rho_i, s \rangle \xrightarrow{\lambda} \langle \text{skip}, s' \rangle \\
\sum_{j=1}^{k} (\rho_j \rightarrow C_j), s) \xrightarrow{\lambda} (C_i, s') \\
\langle C, s \rangle \xrightarrow{\lambda} \langle C', s' \rangle \\
\langle C \setminus h, s \rangle \xrightarrow{\lambda} \langle C' \setminus h, s' \rangle \\
\text{if } \lambda \notin \{h?n, h!n \mid n \in N\} \\
\langle C, s \rangle \text{term} \\
\langle C \setminus h, s \rangle \text{term} \\
\langle C_1, s \rangle \xrightarrow{\lambda} \langle C'_1, s' \rangle \\
\langle C_1 || C_2, s \rangle \xrightarrow{\lambda} \langle C'_1 || C_2, s' \rangle \\
\langle C_2, s \rangle \xrightarrow{\lambda} \langle C'_2, s' \rangle \\
\langle C_1 || C_2, s \rangle \xrightarrow{\lambda} \langle C_1 || C'_2, s' \rangle \\
\langle C_1, s_1 \rangle \xrightarrow{\lambda_1} \langle C'_1, s'_1 \rangle \\
\langle C_2, s_2 \rangle \xrightarrow{\lambda_2} \langle C'_2, s'_2 \rangle \\
\langle C_1 || C_2, s_1 \cup s_2 \rangle \xrightarrow{\epsilon} \langle C'_1 || C'_2, s'_1 \cup s'_2 \rangle \\
\text{provided match}(\lambda_1, \lambda_2) \text{ and disjoint}(s_1, s_2) \\
\langle C_1, s \rangle \text{term} \\
\langle C_2, s \rangle \text{term} \\
\langle C_1 || C_2, s \rangle \text{term}
\end{array}
\]

Figure 2: Transition rules for parallel constructs
free identifiers.)

We will write $\Lambda^*$ for the set of finite sequences of communications:

$$
\Lambda^* = \{\epsilon\} \cup \{ h?n, h!n \mid n \in N, h \in \text{Chan} \}^+,
$$

where $\Lambda^+$ is the set of non-empty sequences over $\Lambda$. This definition of $\Lambda^*$ is a slight abuse of notation, since the usual form of Kleene star operation would include "mixed" sequences containing communications and occurrences of $\epsilon$; our definition absorbs such occurrences of $\epsilon$, and this corresponds to the fact that $\epsilon$ represents the empty sequence, which is a unit for concatenation. We also define (with a similar abuse of notation)

$$
\Lambda^\omega = \{ \alpha\epsilon^\omega \mid \alpha \in \Lambda^* \} \cup \{ h?n, h!n \mid n \in N, h \in \text{Chan} \}^\omega.
$$

Again this definition builds in the property that $\epsilon$ is a unit for concatenation (even for infinite sequences). However, it is important to note that $\epsilon^\omega$ is not the same as $\epsilon$; the former represents divergence, the latter represents termination. Finally, we let $\Lambda^\omega = \Lambda^* \cup \Lambda^\omega$. For $\alpha \in \Lambda^\omega$ we denote by $\text{chans}(\alpha)$ the set of channel names occurring in $\alpha$.

We now define generalized transition relations $\overset{\alpha}{\Rightarrow}$, where $\alpha \in \Lambda^\omega$:

- For finite $\alpha$, $\langle C, s \rangle \overset{\alpha}{\Rightarrow} \langle C', s' \rangle$ means that $C$ from state $s$ may perform the sequence of communications $\alpha$, leading to the configuration $C'$ in state $s'$; finitely many $\epsilon$-transitions are permitted between communications. Note the special case when $\alpha$ is $\epsilon$, representing a finite (possibly empty) sequence of $\epsilon$-transitions.

- For infinite $\alpha$, $\langle C, s \rangle \overset{\alpha}{\Rightarrow}$ means that there is a fair infinite computation of $C$ from initial state $s$ in which the action labels form the sequence $\alpha$. The special case $\alpha = \epsilon^\omega$ indicates that $C$ has a fair infinite internal computation starting from $s$.

To be precise about fairness we should tag each transition with an indication of which sub-commands are responsible for the atomic action that causes it, and ensure that the interleaving operation takes proper account of tags. For instance, see [5, 1].

8
4 Examples

1. Let $a$ be a channel name. Then the possible transition sequences of $a?x||a!0$ from an initial state in which the value of $x$ is 1 are:

\[
\begin{align*}
\langle a?x||a!0, [x = 1] \rangle & \xrightarrow{a?n} \langle \text{skip}||a!0, [x = n] \rangle & \xrightarrow{a!0} \langle \text{skip}||\text{skip}, [x = n] \rangle \\
\langle a?x||a!0, [x = 1] \rangle & \xrightarrow{a!0} \langle a?x||\text{skip}, [x = 1] \rangle & \xrightarrow{a?n} \langle \text{skip}||\text{skip}, [x = n] \rangle \\
\langle a?x||a!0, [x = 1] \rangle & \xrightarrow{c} \langle \text{skip}||\text{skip}, [x = 0] \rangle
\end{align*}
\]

In each case the final configuration is terminal.

2. In contrast, restricting on $a$ in the previous example forces the communication to take place:

\[
\langle (a?x||a!0)\backslash a, [x = 1] \rangle \xrightarrow{c} \langle (\text{skip}||\text{skip})\backslash a, [x = 0] \rangle.
\]

Again the final configuration is terminal.

3. Let $B_1, B_2, B_{12}$ be the processes defined by:

\[
\begin{align*}
B_1 & = \text{while true do (in?x; link!x)} \\
B_2 & = \text{while true do (link?y; out!y)} \\
B_{12} & = [B_1||B_2]\backslash link
\end{align*}
\]

Intuitively, $B_1$ behaves like a buffer of capacity 1, repeatedly inputting a value on channel $in$ and outputting it on channel $link$. Similarly, $B_2$ is a buffer with input $link$ and output $out$. $B_{12}$ behaves like a buffer from input $in$ to output $out$, with capacity 2. A discussion of similar processes (in a non-imperative setting) occurs in [8].

4. The program $[C_1||C_2||C_3]\backslash left\backslash right$, where

\[
\begin{align*}
C_1 & = \text{while true do (left?x → out!x) + (right?x → out!x)} \\
C_2 & = \text{while true do left!0} \\
C_3 & = \text{while true do right!1},
\end{align*}
\]

performs a "merge" of a sequence of 0's with a sequence of 1's. According to the transition rules, this program has an infinite transition sequence corresponding to the sequence of communications $out!0$. This is an unfair computation sequence for this program, because it cannot be obtained by fairly interleaving communication traces for each
of the constituent processes: the only way for this sequence to arise is by ignoring the communication capability for \( C_3 \). The fair communication traces of this program have form \( \text{out}_0v_0.\text{out}_1v_1.\text{out}_2v_2.\ldots.\text{out}_nv_n\ldots \), where \( v_0v_1\ldots v_n\ldots \) is a fair merge of \( 0^\omega \) and \( 1^\omega \), so that it contains infinitely many 0's and infinitely many 1's.

5. Consider the processes \( C_1 \) and \( C_2 \) given by
\[
C_1 = (a?x \rightarrow ((b!x \rightarrow \text{skip}) + (c!x \rightarrow \text{skip})))
\]
\[
C_2 = (a?x \rightarrow b!x) + (a?x \rightarrow c!x)
\]
Each can perform the sequence of communications \( a?nbhn \) and can perform \( a?ncnhn \), for each \( n \in N \). But the second process has two essentially different \( a?n \) transitions, leading to configurations where either the only possible next step involves channel \( b \) or the only possible next step involves \( c \). In the first process, after doing input on channel \( a \) it will be possible to do output on \( b \) or on \( c \).

5 Program behavior

The only important attribute of an expression in the transition system for commands (Figures 1 and 2) is its value. We therefore define evaluation functions \( \mathcal{E} : \text{Exp} \rightarrow \mathcal{P}(S \times N) \) and \( \mathcal{B} : \text{BExp} \rightarrow \mathcal{P}(S \times V) \), where \( V = \{tt, ff\} \) is the set of truth values:

\[
\mathcal{E}[[E]] = \{(s, n) \mid \langle E, s \rangle \rightarrow^* n\} \quad \mathcal{B}[[B]] = \{(s, v) \mid \langle B, s \rangle \rightarrow^* v\}.
\]

We want to be able to reason about the effect of command execution, including whether or not it terminates successfully, assuming fair execution. We therefore define the "state transformation" behavior of a command \( C \), denoted \( \mathcal{M}[[C]] \), as follows:

**Definition 5.1** The behavior function \( \mathcal{M} : \text{Com} \rightarrow \mathcal{P}(S \times S_\bot) \) is defined by:

\[
\mathcal{M}[[C]] = \{(s, s') \mid \langle C, s \rangle \xrightarrow{\text{\_}} \langle C', s'\rangle \text{term}\} \cup \{(s, \bot) \mid \langle C, s \rangle \xrightarrow{\text{\_}}\}.
\]
We use $\perp$ to represent non-termination, and $S_\perp = S \cup \{\perp\}$. A command $C$ has a fair infinite computation (involving only internal actions) from state $s$ if and only if $(s, \perp) \in \mathcal{M}[C]$.

We have defined this behavioral notion by reference to the transition system given above: this is an operational characterization. It is obvious that $\mathcal{M}$ cannot be defined compositionally, since (for instance) $\mathcal{M}[C_1 \parallel C_2]$ cannot be determined from $\mathcal{M}[C_1]$ and $\mathcal{M}[C_2]$. We now give a compositional notion of behavior generalizing $\mathcal{M}$ in a natural way.

**Definition 5.2** The trace semantic function $T : \textbf{Com} \to \mathcal{P}(S \times \Lambda^\infty \times S_\perp)$ is characterized operationally by:

$$ T[C] = \{(s, \alpha, s') \mid \langle C, s \rangle \xrightarrow{\alpha} \langle C', s' \rangle \text{term} \} \cup \{(s, \alpha, \perp) \mid \alpha \in \Lambda^\omega \text{ & } \langle C, s \rangle \xrightarrow{\alpha} \}. $$

In contrast to [6], our traces are adapted to the imperative setting: we model state changes explicitly. Moreover, since we focus only on the terminal finite traces we do not impose the prefix-closure condition on trace sets. Nor do we require that an infinite trace be included in a trace set if each of its prefixes is present in the set: this would be incompatible with our desire to model fairness properly. From now on, we use the term trace for a triple of form $(s, \alpha, s')$ (where $s' \in S_\perp$), and we will refer to the $\alpha$ component as a communication trace.

The state transformation behavior of a command is derivable from its traces:

$$ \mathcal{M}[C] = \{(s, s') \mid (s, \epsilon, s') \in T[C] \} \cup \{(s, \perp) \mid (s, \epsilon^\omega, \perp) \in T[C] \}. $$

This obvious property will be useful later.

### 6 Denotational Semantics

We now show that $T$ can be defined compositionally. This gives a denotational characterization to complement the operational characterization just given.

To start, notice that we can regard the semantic domain $\mathcal{P}(S \times \Lambda^\infty \times S_\perp)$ as a complete partial order (in fact, a complete lattice), with set inclusion as the underlying order.
We begin by defining a semantic analogue to the syntactic operation of sequential composition. For trace sets $T_1$ and $T_2$ we define

$$T_1;T_2 = \{(s, \alpha \beta, s') \mid \exists s'. (s, \alpha, s') \in T_1 \& (s', \beta, s'') \in T_2\}$$

$$\cup \{(s, \alpha, \perp) \mid (s, \alpha, \perp) \in T_1\},$$

where concatenation of communication sequences is defined as usual, so that $\alpha \beta = \alpha$ when $\alpha$ is infinite.

Next we generalize from concatenation to iteration. For a trace set $T$ we define $T^n$, the $n$-fold iteration of $T$, by induction on $n$:

$$T^0 = \{(s, \epsilon, s) \mid s \in S\}$$

$$T^{k+1} = T; T^k \quad (k \geq 0).$$

We then define $T^*$ and $T^\omega$ by:

$$T^* = \bigcup_{n=0}^{\infty} T^n$$

$$T^\omega = \{(s_0, \alpha_0 \alpha_1 \ldots \alpha_n \ldots, \perp) \mid \forall n.(s_n, \alpha_n, s_{n+1}) \in T\}.$$ 

Note that $\{(s, \epsilon, s) \mid s \in S\}$ is a unit for sequential composition of trace sets, and $T^1 = T$ for all trace sets $T$.

Parallel composition is modelled by a form of interleaving of traces, allowing for synchronized communication. We need to define a fairmerge operator on traces, so that we only include interleavings corresponding to fair behaviors. The following definitions are based on [14], adapted to deal with communicating processes and synchronization. Let $T_1$ and $T_2$ represent the trace sets of disjoint processes. Then we define $T_1 || T_2$, the set of all fair synchronizing merges of a trace from $T_1$ and a trace in $T_2$, as follows:

$$T_1 || T_2 = \{(s_1 \cup s_2, \gamma, s'_1 \cup s'_2) \mid \exists (s_1, \alpha, s'_2) \in T_1, (s_2, \beta, s'_1) \in T_2, \text{disjoint}(s_1, s_2) \& (\alpha, \beta, \gamma) \in \text{fairmerge}\},$$

where

$$\text{fairmerge} = (L^* R R^* L)^\omega \cup (L \cup R)^* A,$$

$L = \{((\lambda, \epsilon, \lambda) \mid \lambda \in \Lambda\} \cup M,$

$R = \{((\epsilon, \lambda, \lambda) \mid \lambda \in \Lambda\} \cup M,$

$M = \{(\lambda_1, \lambda_2, \epsilon) \mid \text{match}(\lambda_1, \lambda_2)\},$

$A = \{((\alpha, \epsilon, \alpha), (\epsilon, \alpha, \alpha) \mid \alpha \in \Lambda^\infty\}.$

In this definition we extend the set-theoretic union operator to $S \times S \perp$ in the obvious way, defining $\perp \cup s = s \cup \perp = \perp$. We also extend the concatenation
operation to triples of traces in the obvious componentwise way and we use the pointwise extension to sets of triples.

When $(\alpha, \beta, \gamma) \in \text{fairmerge}$ we say that $\gamma$ is a fair synchronizing merge of $\alpha$ and $\beta$. Intuitively, the definition is intended to specify that $\gamma$ is constructed from $\alpha$ and $\beta$ by a combination of interleaving and synchronization of matching input and output, and in the construction all actions from $\alpha$ and $\beta$ are used up. If $\alpha$ is finite then as soon as all of $\alpha$ has been used up there is no further fairness requirement to fulfill, and similarly if $\beta$ is finite; all such cases give rise to triples $(\alpha, \beta, \gamma)$ expressible in the form $(L \cup R)^* A$. The term $(L^*RR^*L)^\omega$ deals with the cases where $\alpha$ and $\beta$ are both infinite. Apart from the difference in the underlying notion of atomic action, this fairmerge definition is obtained from Park’s by adding states, taking advantage of the disjointness assumption (so that states may be combined using union), and by including a component $M$ dealing with synchronization. Note that a synchronized pair of communications produces an $\epsilon$-step and counts as an atomic action by both of the participating processes; this is important in ensuring a proper account of fair execution.

For example, the possible fair merges of $a?n$ and $a!0$ are $a?n.a!0$, $a!0.a?n$ and $\epsilon$. The fair merges of $a?0.b?0.a?0$ and $a!0$ include $b?0.a?0$, $a?0.b?0$, $a?0.b?0.a?0.a!0$, but not $a!0.b?0.a?0$ and not $a!0.a?0.b?0$. The only fair merge of $\epsilon$ with $\beta$ is $\beta$ itself if $\beta$ is infinite, and $\beta \epsilon$ if $\beta$ is finite. The fair merges of $(a?0)^\omega$ and $(a!0)^\omega$ include $(a?0)^n \epsilon^\omega$ and $(a!0)^n \epsilon^\omega$ (for all $n \geq 0$), but not $(a?0)^\omega$ or $(a!0)^\omega$.

With these definitions in hand, it is now easy to give a denotational description of $T$. 
Proposition 6.1 The trace semantics $T : \text{Com} \rightarrow \mathcal{P}(S \times \Lambda^\infty \times S)$ is characterized by the following clauses:

\[
\begin{align*}
T[\text{skip}] & = \{ (s, \epsilon, s) \mid s \in S \} \\
T[I := E] & = \{ (s, \epsilon, [s \mid I = n]) \mid (s, n) \in E \} \\
T[C_1; C_2] & = T[C_1]; T[C_2] \\
T[\text{if } B \text{ then } C_1 \text{ else } C_2] & = T[B]; T[C_1] \cup T[\neg B]; T[C_2] \\
& \quad \text{where } T[B] = \{ (s, \epsilon, s) \mid (s, tt) \in B \} \\
T[\text{while } B \text{ do } C] & = (T[B]; T[C])^*; T[\neg B] \cup (T[B]; T[C])^\omega \\
T[h?I] & = \{ (s, h?n, [s \mid I = n]) \mid s \in S, n \in N \} \\
T[h!E] & = \{ (s, h!n, s) \mid (s, n) \in E \} \\
T[\Sigma_{i=1}^n (\rho_i \rightarrow C_i)] & = \bigcup_{i=1}^n (T[\rho_i]; T[C_i]) \\
T[C/h] & = \{ (s, \alpha, s') \in T[C] \mid h \notin \text{chans(\alpha)} \} \\
T[C_1 || C_2] & = T[C_1] || T[C_2]
\end{align*}
\]

Proof: It is straightforward to show, for each command $C$, that the operational description of $T[C]$ coincides with the set $T[C]$ prescribed by this denotational definition. The details for parallel composition rely on the operational characterization of fair infinite computation. \text{(End of Proof)}

This semantic description makes certain equivalences obvious. For instance, writing $C_1 \equiv C_2$ to mean that $C_1$ and $C_2$ have the same trace semantics, it is easy to verify the following laws:

\[
\begin{align*}
C; \text{skip} & \equiv C \\
\text{skip}; C & \equiv C \\
\text{while } B \text{ do } C & \equiv \text{if } B \text{ then } (C; \text{while } B \text{ do } C) \text{ else } \text{skip} \\
(h?I || h!E)\backslash h & \equiv I := E \\
C || \text{skip} & \equiv C \\
C_1 || C_2 & \equiv C_2 || C_1 \\
(C_1 || C_2) || C_3 & \equiv C_1 || (C_2 || C_3) \\
(C \backslash h_1) \backslash h_2 & \equiv (C \backslash h_2) \backslash h_1 \\
C \backslash h & \equiv C \backslash h
\end{align*}
\]

The last two laws allow us to write $C \backslash \{ h_1, \ldots, h_k \}$ for $(C \backslash h_1) \ldots \backslash h_k$, the result of restricting $C$ on a finite set of channels.

The following result is an easy consequence of the fact that all operations on trace sets used in these semantic clauses are monotone with respect to set inclusion. A program context $P[-]$ is a program containing a hole (denoted $[-]$) into which a command may be inserted; $P[C]$ denotes the program
obtained by inserting $C$ into the hole. We restrict attention to contexts $P[-]$ and commands $C$ such that $P[C]$ is well-formed.

**Proposition 6.2** For every program context $P[-]$ and all commands $C$ and $C'$, we have the following “contextual monotonicity” property:

$$T[C] \subseteq T[C'] \Rightarrow T[P[C]] \subseteq T[P[C']]$$.

7 Examples

1. It is easy to check the following details, illustrating the correspondence between the denotational and operational definitions of $T$:

   $$T[a?x] = \{(s,a?n,[s | x = n]) | s \in S & n \in N\}$$
   $$T[a!0] = \{(s,a!0,s) | s \in S\}$$
   $$T[a?x|a!0] = \{(s,\epsilon,[s | x = 0]) | s \in S\}$$
   $$\cup\{(s,a?n.a!0,[s | x = n]) | s \in S & n \in N\}$$
   $$\cup\{(s,a!0.a?n,[s | x = n]) | s \in S & n \in N\}$$
   $$T[(a?x|a!0)\backslash a] = \{(s,\epsilon,[s | x = 0]) | s \in S\}.$$.

2. Recall the processes $B_1, B_2, B_{12}$ discussed earlier:

   $$B_1 = \textbf{while true do (in?x; link!x)}$$
   $$B_2 = \textbf{while true do (link?y; out!y)}$$
   $$B_{12} = [B_1 || B_2] \backslash \text{link}$$

   One can use the denotational semantics to show that $B_1$ and $B_2$ behave like 1-place buffers and $B_{12}$ behaves like a 2-place buffer.

3. Consider again the program $[C_1 || C_2 || C_3] \backslash \text{left \backslash right}$, where

   $$C_1 = \textbf{while true do (left?x \rightarrow out!x) + (right?x \rightarrow out!x)}$$
   $$C_2 = \textbf{while true do left!0}$$
   $$C_3 = \textbf{while true do right!1}.$$.

   The traces of $C_2 || C_3$ have form

   $$(\text{left!0})^\omega ||(\text{right!1})^\omega = ((\text{left!0})^* \text{right!1}(\text{right!1})^* \text{left!0})^\omega$$,

   each containing infinitely many left and infinitely many right steps. The traces of $C_1$ have form $(h_n.v_n.out!v_n)_{n=0}^\infty$, where each $h_n \in \{\text{left, right}\}$
\( (n \geq 0) \). The only fair merges of a trace of \( C_1 \) with a trace of \( C_2 \| C_3 \), restricted so as to contain no left and right steps, must therefore involve all synchronization steps. The possible sequences of values output on channel \( \text{out} \) will therefore correspond to the extended regular expression \( (0^*11^*0)^\omega \). As required, this is the set of sequences of 0’s and 1’s that contain infinitely many of each.

4. The loop **while true do skip** diverges:

\[
T[\text{while true do skip}] = \{(s, e^\omega, \bot) \mid s \in S\}.
\]

8 Full Abstraction

Having presented a denotational description of \( T \) it is clear that we can use traces to reason compositionally about the communication sequences of fair parallel programs: \( T \) distinguishes between a pair of commands \( C_1 \) and \( C_2 \) if and only if there is a context \( P[-] \) such that \( T[P[C_1]] \) and \( T[P[C_2]] \) differ. The proof of this is almost trivial, using the contextual monotonicity property mentioned above.

Since the behavior \( M[C] \) can be extracted from \( T[C] \) the trace semantics also supports compositional reasoning about behavior. In fact, we obtain full abstraction: \( T \) distinguishes between \( C_1 \) and \( C_2 \) if and only if there is a context \( P[-] \) such that \( M[P[C_1]] \) and \( M[P[C_2]] \) differ.

**Proposition 8.1** The trace semantics \( T \) is (inequationally) fully abstract with respect to \( M \):

\[
T[C] \subseteq T[C'] \iff \forall P[-].(M[P[C]] \subseteq M[P[C']]).
\]

**Proof**: The proof of the forward implication follows easily by contextual monotonicity and the fact that the behavior of a program is extractable from its trace set.

For the reverse implication we rely on the following key facts:

1. For a finite communication sequence \( \alpha \) containing \( k \) output actions, and \( k \) distinct identifiers \( z_1, \ldots, z_k \), there is a command \( D_0\alpha(z_1, \ldots, z_k) \) that performs a sequence of communications matching \( \alpha \) and uses the \( z_i \) to store the values output in \( \alpha \).
2. For an infinite sequence \( \alpha \), \( \langle C', s \rangle \) cannot perform \( \alpha \) if and only if there is some finite prefix \( \beta \) of \( \alpha \) such that either \( \alpha = \beta \varepsilon \gamma \) and no \( \beta \)-derivative of \( \langle C', s \rangle \) can do \( \varepsilon \); or \( \alpha \) has the form \( \beta \lambda \gamma \) where \( \lambda \) is a communication (not \( \varepsilon \)), and no \( \beta \)-derivative of \( \langle C', s \rangle \) can do \( \lambda \).

3. For any configuration \( \langle C, s \rangle \), and any finite communication trace \( \alpha \) the set \( \{ C'' \mid \exists s''. \langle C, s \rangle \Rightarrow (C'', s'') \} \) is finite.

If \((s, \alpha, s')\) is a trace of \( C \) but not of \( C' \) there is a finite prefix \( \beta \) of \( \alpha \) after which a behavioral difference is detectable, and we may use a parallel context containing a command of form \( \text{DO}_{\beta}(z_1, \ldots, z_k) \) to distinguish between \( C \) and \( C' \).

\[(\text{End of Proof})\]

As an immediate corollary, we obtain (equational) full abstraction: two commands have the same trace sets if and only if they may be interchanged in all program contexts without altering the behavior of the overall program. Thus all of the semantic equivalences validated by this model can be used in any program context with the guarantee that replacing any command by an equivalent one has no effect on program behavior.

9 Conclusions

We have presented a semantic model, based on fair traces, for a CSP-like language of communicating processes. We have shown that this semantics is fully abstract with respect to a natural notion of program behavior, so that the semantics exactly supports compositional reasoning about behavior.

A configuration is **deadlocked** iff it is not terminal but has no transitions. For a trivial example, the command \( h!0 \backslash h \) is deadlocked in any state.

Trace models like this are well suited to reasoning about safety properties but inadequate for reasoning about the possibility of deadlock. A non-terminal configuration is deadlocked if it has no transitions. Traces do not provide enough information to distinguish between a process that may either deadlock or perform a communication and the corresponding deadlock-free process. It is not even enough to augment the trace model with extra features representing communication sequences that lead to deadlock. This is easily seen in one of the examples discussed earlier: the two commands

\[
C_1 = (a?x \rightarrow ((b!x \rightarrow \text{skip}) + (c!x \rightarrow \text{skip})))
\]

\[
C_2 = (a?x \rightarrow b!x) + (a?x \rightarrow c!x).
\]
have the same successful traces and no deadlock traces, but they induce
different deadlock traces in the context $[-|| b?y] \text{b}\text{c}$: only the second command
may deadlock after doing $a?0$.

One way to add appropriate extra structure to the semantic model is to
work with failure sets [2]: a failure of a process is a (finite) trace together
with a set of events that the process may be able to refuse after having
performed the trace. The possibility of deadlock is represented by the ability
to refuse all events. To extend this idea to the imperative setting we need to
incorporate a suitable treatment of program states, perhaps along the lines
discussed by Roscoe in [16]. The two commands $C_1$ and $C_2$ have different
failure sets: the failure $(s, a?0, [s \mid x = 0], \{b!0\})$ is only possible for $C_2$,
corresponding precisely to the behavioral difference noted above.

We plan to investigate further the full abstraction problem for communicat-
ing processes and various natural notions of program behavior, including
partial and total correctness, and deadlock-freedom. The analogous problems
for a shared variable parallel language were discussed in [3].

References

[1] Apt, K. R. and Olderog, E.-R., Verification of Sequential and Con-


[3] Brookes, S., Full Abstraction for a shared variable parallel language,
Proc. 8th Annual IEEE Symposium on Logic in Computer Science, IEEE

Languages, NATO Advanced Study Institute, pp. 43-112, Academic


1981.


