Fair Communicating Processes

Stephen Brookes

Published in
A Classical Mind: Essays in Honour of C. A. R. Hoare,
edited by A. W. Roscoe. Prentice-Hall International, January 1994.

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 focussed mainly on a more abstract process language, which has come to be known as Theoretical CSP (or TCSP) [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.

[1] The 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:

$$\begin{aligned} C \;::=\; & \mathbf{skip} \mid I{:=}E \mid C_1;C_2 \mid \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \mid \mathbf{while}\ B\ \mathbf{do}\ C \mid \\ & h?I \mid h!E \mid C_1\|C_2 \mid \sum_{i=1}^{k}(\rho_i \to C_i) \mid C\backslash h, \end{aligned}$$

where the ρ_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 $\sum_{i=1}^{k}(\rho_i \to C_i)$ corresponds to a guarded command whose guards involve input or output[2]. The command $C\backslash h$ 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 *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 $\mathrm{free}[\![C]\!]$ of identifiers occurring free in C, given as usual by structural induction on C:

$$
\begin{aligned}
&\mathrm{free}[\![\mathbf{skip}]\!] = \{\}\\
&\mathrm{free}[\![I{:=}E]\!] = \{I\} \cup \mathrm{free}[\![E]\!]\\
&\mathrm{free}[\![C_1; C_2]\!] = \mathrm{free}[\![C_1]\!] \cup \mathrm{free}[\![C_2]\!]\\
&\mathrm{free}[\![\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2]\!] = \mathrm{free}[\![B]\!] \cup \mathrm{free}[\![C_1]\!] \cup \mathrm{free}[\![C_2]\!]\\
&\mathrm{free}[\![\mathbf{while}\ B\ \mathbf{do}\ C]\!] = \mathrm{free}[\![B]\!] \cup \mathrm{free}[\![C]\!]\\
&\mathrm{free}[\![h?I]\!] = \{I\}\\
&\mathrm{free}[\![h!E]\!] = \mathrm{free}[\![E]\!]\\
&\mathrm{free}[\![C_1 \| C_2]\!] = \mathrm{free}[\![C_1]\!] \cup \mathrm{free}[\![C_2]\!]\\
&\mathrm{free}[\![\textstyle\sum_{i=1}^{k}(\rho_i \to C_i)]\!] = \textstyle\bigcup_{i=1}^{k}(\mathrm{free}[\![\rho_i]\!] \cup \mathrm{free}[\![C_i]\!])\\
&\mathrm{free}[\![C\backslash h]\!] = \mathrm{free}[\![C]\!]
\end{aligned}
$$

We say that C is well-formed iff for every sub-command of C with form $C_1 \| C_2$ we have $\mathrm{free}[\![C_1]\!] \cap \mathrm{free}[\![C_2]\!] = \{\}$. 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 $\mathrm{chans}[\![C]\!]$, the finite set of channel names occurring in C, by structural induction on C:

$$
\begin{aligned}
&\mathrm{chans}[\![\mathbf{skip}]\!] = \{\}\\
&\mathrm{chans}[\![I{:=}E]\!] = \{\}\\
&\mathrm{chans}[\![C_1; C_2]\!] = \mathrm{chans}[\![C_1]\!] \cup \mathrm{chans}[\![C_2]\!]\\
&\mathrm{chans}[\![\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2]\!] = \mathrm{chans}[\![C_1]\!] \cup \mathrm{chans}[\![C_2]\!]\\
&\mathrm{chans}[\![\mathbf{while}\ B\ \mathbf{do}\ C]\!] = \mathrm{chans}[\![C]\!]\\
&\mathrm{chans}[\![h?I]\!] = \{h\}\\
&\mathrm{chans}[\![h!E]\!] = \{h\}\\
&\mathrm{chans}[\![C_1 \| C_2]\!] = \mathrm{chans}[\![C_1]\!] \cup \mathrm{chans}[\![C_2]\!]\\
&\mathrm{chans}[\![\textstyle\sum_{i=1}^{k}(\rho_i \to C_i)]\!] = \textstyle\bigcup_{i=1}^{k}(\mathrm{chans}[\![\rho_i]\!] \cup \mathrm{chans}[\![C_i]\!])\\
&\mathrm{chans}[\![C\backslash h]\!] = \mathrm{chans}[\![C]\!] - \{h\}
\end{aligned}
$$

[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 $S = [\mathbf{Ide} \to_p N]$ 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 $\mathrm{dom}(s)$, 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:

$$\mathrm{disjoint}(s_1, s_2) \iff \mathrm{dom}(s_1) \cap \mathrm{dom}(s_2) = \{\}.$$

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 $\langle E, s\rangle \to^* n$ 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]:

$$\mathbf{Conf} = \{\langle C, s\rangle \mid \mathrm{free}[\![C]\!] \subseteq \mathrm{dom}(s)\}.$$

We decorate transitions with a label indicating the type of atomic action involved: ϵ 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 Λ be the set of all labels: $\Lambda = \{\epsilon\} \cup \{\, h?n, h!n \mid n \in N, h \in \mathbf{Chan}\}$. We use λ 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 λ, leading to C' in state s'. Two labels λ_1 and λ_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 $\mathrm{match}(\lambda_1, \lambda_2)$.

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

[3]This means that we need not be concerned with the possibility of uninitialized identifiers in our semantics.

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

$$\frac{\langle E, s\rangle \rightarrow^* n}{\langle I{:=}E, s\rangle \xrightarrow{\epsilon} \langle \mathbf{skip}, [s \mid I = n]\rangle}$$

$$\frac{\langle C_1, s\rangle \xrightarrow{\lambda} \langle C_1', s'\rangle}{\langle C_1; C_2, s\rangle \xrightarrow{\lambda} \langle C_1'; C_2, s'\rangle}$$

$$\frac{\langle C_1, s\rangle \mathrm{term}}{\langle C_1; C_2, s\rangle \xrightarrow{\epsilon} \langle C_2, s\rangle}$$

$$\frac{\langle B, s\rangle \rightarrow^* \mathtt{tt}}{\langle \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2, s\rangle \xrightarrow{\epsilon} \langle C_1, s\rangle}$$

$$\frac{\langle B, s\rangle \rightarrow^* \mathtt{ff}}{\langle \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2, s\rangle \xrightarrow{\epsilon} \langle C_2, s\rangle}$$

$$\langle \mathbf{while}\ B\ \mathbf{do}\ C, s\rangle \xrightarrow{\epsilon} \langle \mathbf{if}\ B\ \mathbf{then}\ C; \mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{else}\ \mathbf{skip}, s\rangle$$

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 $\mathrm{free}[\![C_1]\!] \subseteq \mathrm{dom}(s_1)$ and $\mathrm{free}[\![C_2]\!] \subseteq \mathrm{dom}(s_2)$, 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.

$$\frac{}{\langle h?I, s\rangle \xrightarrow{h?n} \langle \mathbf{skip}, [s \mid I = n]\rangle} \quad \text{for each } n \in N$$

$$\frac{\langle E, s\rangle \rightarrow^* n}{\langle h!E, s\rangle \xrightarrow{h!n} \langle \mathbf{skip}, s\rangle}$$

$$\frac{\langle \rho_i, s\rangle \xrightarrow{\lambda} \langle \mathbf{skip}, s'\rangle}{\langle \sum_{j=1}^{k} (\rho_j \rightarrow C_j), s\rangle \xrightarrow{\lambda} \langle C_i, s'\rangle} \quad \text{for each } i \in 1 \ldots k$$

$$\frac{\langle C, s\rangle \xrightarrow{\lambda} \langle C', s'\rangle}{\langle C\backslash h, s\rangle \xrightarrow{\lambda} \langle C'\backslash h, s'\rangle} \quad \text{if } \lambda \notin \{h?n, h!n \mid n \in N\}$$

$$\frac{\langle C, s\rangle \text{term}}{\langle C\backslash h, s\rangle \text{term}}$$

$$\frac{\langle C_1, s\rangle \xrightarrow{\lambda} \langle C_1', s'\rangle}{\langle C_1 \| C_2, s\rangle \xrightarrow{\lambda} \langle C_1' \| C_2, s'\rangle}$$

$$\frac{\langle C_2, s\rangle \xrightarrow{\lambda} \langle C_2', s'\rangle}{\langle C_1 \| C_2, s\rangle \xrightarrow{\lambda} \langle C_1 \| C_2', s'\rangle}$$

$$\frac{\langle C_1, s_1\rangle \xrightarrow{\lambda_1} \langle C_1', s_1'\rangle \quad \langle C_2, s_2\rangle \xrightarrow{\lambda_2} \langle C_2', s_2'\rangle}{\langle C_1 \| C_2, s_1 \cup s_2\rangle \xrightarrow{\epsilon} \langle C_1' \| C_2', s_1' \cup s_2'\rangle}$$

provided $\text{match}(\lambda_1, \lambda_2)$ and $\text{disjoint}(s_1, s_2)$

$$\frac{\langle C_1, s\rangle \text{term} \quad \langle C_2, s\rangle \text{term}}{\langle C_1 \| C_2, s\rangle \text{term}}$$

Figure 2: Transition rules for parallel constructs

free identifiers.)

We will write Λ^* for the set of finite sequences of communications:

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

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

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

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

We now define generalized transition relations $\stackrel{\alpha}{\Longrightarrow}$, where $\alpha \in \Lambda^\infty$:

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

- For infinite α, $\langle C, s\rangle \stackrel{\alpha}{\Longrightarrow}$ means that there is a *fair* infinite computation of C from initial state s in which the action labels form the sequence α. 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].

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:

$$
\begin{array}{l}
\langle a?x\|a!0, [x=1]\rangle \xrightarrow{a?n} \langle \mathbf{skip}\|a!0, [x=n]\rangle \xrightarrow{a!0} \langle \mathbf{skip}\|\mathbf{skip}, [x=n]\rangle \\
\langle a?x\|a!0, [x=1]\rangle \xrightarrow{a!0} \langle a?x\|\mathbf{skip}, [x=1]\rangle \xrightarrow{a?n} \langle \mathbf{skip}\|\mathbf{skip}, [x=n]\rangle \\
\langle a?x\|a!0, [x=1]\rangle \xrightarrow{\epsilon} \langle \mathbf{skip}\|\mathbf{skip}, [x=0]\rangle
\end{array}
$$

   In each case the final configuration is terminal.

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

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

   Again the final configuration is terminal.

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

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

   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

$$
\begin{array}{rcl}
C_1 & = & \mathbf{while\ true\ do}\ (left?x \rightarrow out!x) + (right?x \rightarrow out!x) \\
C_2 & = & \mathbf{while\ true\ do}\ left!0 \\
C_3 & = & \mathbf{while\ true\ do}\ right!1,
\end{array}
$$

   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 $out!v_0.out!v_1.out!v_2.\ldots out!v_n\ldots$, where $v_0v_1\ldots v_n\ldots$ is a fair merge of 0^ω and 1^ω, so that it contains infinitely many 0's and infinitely many 1's.

5. Consider the processes C_1 and C_2 given by

$$C_1 = (a?x \rightarrow ((b!x \rightarrow \mathbf{skip}) + (c!x \rightarrow \mathbf{skip})))$$

$$C_2 = (a?x \rightarrow b!x) + (a?x \rightarrow c!x)$$

Each can perform the sequence of communications $a?nb!n$ and can perform $a?nc!n$, 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 $\mathcal{E} : \mathbf{Exp} \rightarrow \mathcal{P}(S \times N)$ and $\mathcal{B} : \mathbf{BExp} \rightarrow \mathcal{P}(S \times V)$, where $V = \{\mathtt{tt}, \mathtt{ff}\}$ is the set of truth values:

$$\begin{aligned} \mathcal{E}[\![E]\!] &= \{(s,n) \mid \langle E, s\rangle \rightarrow^* n\} \\ \mathcal{B}[\![B]\!] &= \{(s,v) \mid \langle B, s\rangle \rightarrow^* v\}. \end{aligned}$$

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 $\mathcal{M} : \mathbf{Com} \rightarrow \mathcal{P}(S \times S_\perp)$ is defined by:

$$\mathcal{M}[\![C]\!] = \{(s,s') \mid \langle C, s\rangle \stackrel{\epsilon}{\Longrightarrow} \langle C', s'\rangle\mathrm{term}\} \cup \{(s,\perp) \mid \langle C, s\rangle \stackrel{\epsilon^\omega}{\Longrightarrow}\}.$$

•

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) $\mathcal{M}[\![C_1 \| C_2]\!]$ 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 $\mathcal{T} : \mathbf{Com} \to \mathcal{P}(S \times \Lambda^\infty \times S_\perp)$ is characterized operationally by:

$$\begin{aligned}\mathcal{T}[\![C]\!] = \{(s, \alpha, s') \mid \langle C, s\rangle \overset{\alpha}{\Longrightarrow} \langle C', s'\rangle \mathrm{term}\} \\ \cup\{(s, \alpha, \perp) \mid \alpha \in \Lambda^\omega \ \& \ \langle C, s\rangle \overset{\alpha}{\Longrightarrow}\}.\end{aligned}$$

•

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, α, s') (where $s' \in S_\perp$), and we will refer to the α component as a *communication trace.*

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

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

This obvious property will be useful later.

6 Denotational Semantics

We now show that $\mathcal{T}$ 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

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

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

$$\begin{aligned} T^0 &= \{(s,\epsilon,s) \mid s \in S\} \\ T^{k+1} &= T;T^k \qquad (k \geq 0). \end{aligned}$$

We then define T^* and T^ω by:

$$\begin{aligned} T^* &= \textstyle\bigcup_{n=0}^{\infty} T^n \\ T^\omega &= \{(s_0,\alpha_0\alpha_1\ldots\alpha_n\ldots,\perp) \mid \forall n.(s_n,\alpha_n,s_{n+1}) \in T\}. \end{aligned}$$

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:

$$\begin{aligned} T_1\|T_2 \;=\; \{(s_1 \cup s_2,\gamma,s_1' \cup s_2') \mid \exists (s_1,\alpha,s_1') \in T_1, (s_2,\beta,s_2') \in T_2. \\ disjoint(s_1,s_2) \ \&\ (\alpha,\beta,\gamma) \in fairmerge\}, \end{aligned}$$

where

$$\begin{aligned} fairmerge &= (L^*RR^*L)^\omega \ \cup\ (L \cup R)^*A, \\ L &= \{(\lambda,\epsilon,\lambda) \mid \lambda \in \Lambda\} \cup M, \\ R &= \{(\epsilon,\lambda,\lambda) \mid \lambda \in \Lambda\} \cup M, \\ M &= \{(\lambda_1,\lambda_2,\epsilon) \mid \mathrm{match}(\lambda_1,\lambda_2)\}, \\ A &= \{(\alpha,\epsilon,\alpha),(\epsilon,\alpha,\alpha) \mid \alpha \in \Lambda^\infty\}. \end{aligned}$$

In this definition we extend the set-theoretic union operator to $S_\perp \times S_\perp$ 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 $(\alpha, \beta, \gamma) \in \mathit{fairmerge}$ we say that γ is a fair synchronizing merge of α and β. Intuitively, the definition is intended to specify that γ is constructed from α and β by a combination of interleaving and synchronization of matching input and output, and in the construction all actions from α and β are used up. If α is finite then as soon as all of α has been used up there is no further fairness requirement to fulfill, and similarly if β is finite; all such cases give rise to triples (α, β, γ) expressible in the form $(L \cup R)^* A$. The term $(L^* R R^* L)^\omega$ deals with the cases where α and β 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 ϵ-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 ϵ. 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 ϵ^ω with β is β itself if β is infinite, and $\beta\epsilon^\omega$ if β 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 $\mathcal{T}$.

Proposition 6.1 *The trace semantics $\mathcal{T} : \mathbf{Com} \to \mathcal{P}(S \times \Lambda^\infty \times S_\perp)$ is characterized by the following clauses:*

$$
\begin{aligned}
&\mathcal{T}[\![\mathbf{skip}]\!] \;=\; \{(s, \epsilon, s) \mid s \in S\} \\
&\mathcal{T}[\![I{:=}E]\!] \;=\; \{(s, \epsilon, [s \mid I = n]) \mid (s, n) \in \mathcal{E}[\![E]\!]\} \\
&\mathcal{T}[\![C_1; C_2]\!] \;=\; \mathcal{T}[\![C_1]\!]; \mathcal{T}[\![C_2]\!] \\
&\mathcal{T}[\![\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2]\!] \;=\; \mathcal{T}[\![B]\!]; \mathcal{T}[\![C_1]\!] \cup \mathcal{T}[\![\neg B]\!]; \mathcal{T}[\![C_2]\!] \\
&\qquad\qquad\qquad\qquad \textit{where } \mathcal{T}[\![B]\!] = \{(s, \epsilon, s) \mid (s, \mathtt{tt}) \in \mathcal{B}[\![B]\!]\} \\
&\mathcal{T}[\![\mathbf{while}\ B\ \mathbf{do}\ C]\!] \;=\; (\mathcal{T}[\![B]\!]; \mathcal{T}[\![C]\!])^*; \mathcal{T}[\![\neg B]\!] \;\cup\; (\mathcal{T}[\![B]\!]; \mathcal{T}[\![C]\!])^\omega \\
&\mathcal{T}[\![h?I]\!] \;=\; \{(s, h?n, [s \mid I = n]) \mid s \in S, n \in N\} \\
&\mathcal{T}[\![h!E]\!] \;=\; \{(s, h!n, s) \mid (s, n) \in \mathcal{E}[\![E]\!]\} \\
&\mathcal{T}[\![\textstyle\sum_{i=1}^k (\rho_i \to C_i)]\!] \;=\; \textstyle\bigcup_{i=1}^k (\mathcal{T}[\![\rho_i]\!]; \mathcal{T}[\![C_i]\!]) \\
&\mathcal{T}[\![C \backslash h]\!] \;=\; \{(s, \alpha, s') \in \mathcal{T}[\![C]\!] \mid h \notin \mathit{chans}(\alpha)\} \\
&\mathcal{T}[\![C_1 \| C_2]\!] \;=\; \mathcal{T}[\![C_1]\!] \| \mathcal{T}[\![C_2]\!]
\end{aligned}
$$

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

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

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:*

$$\mathcal{T}[\![C]\!] \subseteq \mathcal{T}[\![C']\!] \ \Rightarrow\ \mathcal{T}[\![P[C]]\!] \subseteq \mathcal{T}[\![P[C']]\!].$$

7 Examples

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

$$\begin{array}{l}
\mathcal{T}[\![a?x]\!] = \{(s, a?n, [s \mid x = n]) \mid s \in S \ \&\ n \in N\} \\
\mathcal{T}[\![a!0]\!] = \{(s, a!0, s) \mid s \in S\} \\
\mathcal{T}[\![a?x \| a!0]\!] = \{(s, \epsilon, [s \mid x = 0]) \mid s \in S\} \\
\qquad\qquad\qquad \cup\{(s, a?n.a!0, [s \mid x = n]) \mid s \in S \ \&\ n \in N\} \\
\qquad\qquad\qquad \cup\{(s, a!0.a?n, [s \mid x = n]) \mid s \in S \ \&\ n \in N\} \\
\mathcal{T}[\![(a?x \| a!0)\backslash a]\!] \ = \ \{(s, \epsilon, [s \mid x = 0]) \mid s \in S\}.
\end{array}$$

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

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

   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 $[C_1 \| C_2 \| C_3]\backslash left\backslash right$, where

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

   The traces of $C_2 \| C_3$ have form

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

   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 $h_n \in \{left, right\}$

($n \geq 0$). 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 *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:

$$\mathcal{T}[\![\textbf{while true do skip}]\!] = \{(s, \epsilon^\omega, \perp) \mid s \in S\}.$$

8 Full Abstraction

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

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

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

$$\mathcal{T}[\![C]\!] \subseteq \mathcal{T}[\![C']\!] \iff \forall P[-].(\mathcal{M}[\![P[C]]\!] \subseteq \mathcal{M}[\![P[C']]\!]).$$

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 α containing k output actions, and k distinct identifiers $z_1, \ldots, z_k$, there is a command $D0_\alpha(z_1, \ldots, z_k)$ that performs a sequence of communications matching α and uses the z_i to store the values output in α.

2. For an infinite sequence α, $\langle C', s\rangle$ cannot perform α if and only if there is some finite prefix β of α such that either $\alpha = \beta\epsilon^\omega$ and no β-derivative of $\langle C', s\rangle$ can do ϵ^ω; or α has the form $\beta\lambda\gamma$ where λ is a communication (not ϵ), and no β-derivative of $\langle C', s\rangle$ can do λ.

3. For any configuration $\langle C, s\rangle$, and any finite communication trace α the set $\{C'' \mid \exists s''.\langle C, s\rangle \stackrel{\alpha}{\Longrightarrow} \langle C'', s''\rangle\}$ is finite.

If (s, α, s') is a trace of C but not of C' there is a finite prefix β of α after which a behavioral difference is detectable, and we may use a parallel context containing a command of form $DO_\beta(z_1, \ldots, z_k)$ to distinguish between C and C'.

(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 traces representing communication sequences that lead to deadlock. This is easily seen in one of the examples discussed earlier: the two commands

$$\begin{aligned} C_1 &= (a?x \to ((b!x \to \textbf{skip}) + (c!x \to \textbf{skip}))) \\ C_2 &= (a?x \to b!x) + (a?x \to c!x). \end{aligned}$$

have the same successful traces and no deadlock traces, but they induce different deadlock traces in the context $[-\|b?y]\backslash b\backslash c$: 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 communicating 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 Concurrent Programs**, Springer-Verlag, 1991.

[2] Brookes, S.D., Hoare, C.A.R., and Roscoe, A.W., *A theory of communicating sequential processes*, JACM 31(3):560–599 (1984).

[3] Brookes, S., Full Abstraction for a shared variable parallel language, Proc. 8^{th} Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, June 1993.

[4] Dijkstra, E. W., *Cooperating sequential processes*, in: **Programming Languages**, NATO Advanced Study Institute, pp. 43-112, Academic Press, 1968.

[5] Francez, N., **Fairness**, Springer-Verlag (1986).

[6] Hoare, C. A. R., A model for communicating sequential processes, Technical Report PRG-22, Oxford University, Programming Research Group, 1981.

[7] Hoare, C. A. R., *Communicating Sequential Processes*, Comm. ACM, 21(8):666–677 (1978).

[8] Communicating Sequential Processes, Hoare, C. A. R., Prentice-Hall International, 1985.

[9] Ichbiah, J. D., Reference Manual for the Ada Programming Language, ANSI MIL-STD-1815A-1983, 1983.

[10] The occam programming manual, INMOS Ltd, Prentice-Hall, 1984.

[11] R. Milner, *A Calculus of Communicating Systems*, Springer LNCS 92, 1980.

[12] R. Milner, **Communication and Concurrency**, Prentice-Hall, London, 1989.

[13] Milner, R., *Fully Abstract Models of Typed Lambda-Calculi*, Theoretical Computer Science, vol. 4, pp. 1-22, 1977.

[14] Park, D., *On the semantics of fair parallelism*, in: **Abstract Software Specifications**, pp. 504-526, Springer LNCS 86, 1979.

[15] Plotkin, G. D., *An operational semantics for CSP*, In D. Bjørner, editor, **Formal Description of Programming Concepts II**, Proc. IFIP Working Conference, North-Holland (1983), 199-225.

[16] Roscoe, A. W., *Denotational semantics for occam*, Seminar on Concurrency, Springer-Verlag, LNCS 197, 1984.