Electronic Notes in Theoretical Computer Science 1 (1995) *to appear*

Full Abstraction for Strongly Fair Communicating Processes

Stephen Brookes and Susan Older [1]

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We present a denotational semantics for a language of parallel communicating processes based on Hoare's CSP [10] and Milner's CCS [14], and we prove that the semantics is fully abstract with respect to a deadlock-sensitive notion of fair behavior. The model incorporates the assumption of strong fairness: every process which is enabled infinitely often makes progress infinitely often. The combination of fairness and deadlock causes problems because the "enabledness" of a process may depend on the status of other processes. We formulate a parameterized notion of strong fairness, generalizing the traditional notion of strong fairness [5] in a way that facilitates compositional analysis. We then provide a denotational semantics which uses a form of trace, augmented with information about enabledness, and is related to the failures model for CSP [2] and to Hennessy's acceptance trees [7]. By introducing closure conditions on trace sets, we achieve full abstraction [13]: two processes have the same meaning if and only if they exhibit identical behaviors in all contexts.

1 Introduction

We present a denotational semantics for a language of parallel communicating processes based on Hoare's CSP [10] and Milner's CCS [14]. In this language, processes have disjoint local states and communicate by synchronized message-passing along named channels; unlike the original CSP, we permit nested parallelism. Our model incorporates the assumption of strong fairness: every process which is enabled infinitely often makes progress infinitely often. Fairness assumptions allow us to abstract away from unpredictable details concerning schedulers or the relative speed of parallel processors. The combination of fairness and deadlock (and related forms of blocking like starvation)

[1] This research was sponsored in part by the Office of Naval Research under Grants No. N00014-92-J-1298 and N00014-93-I-0750. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

causes problems because the "enabledness" of a process may depend on the status of other processes: a synchronized communication requires the cooperation of a sender and a receiver. In contrast, enabledness in a shared-variable parallel language depends only on the global state. As a consequence, fair semantic models for shared-variable languages (such as [3]) cannot easily be adapted to yield suitable models for communicating processes.

In this paper we show how to achieve a denotational semantics for communicating processes that handles the combination of fairness and deadlock appropriately. We begin with an operational semantics along traditional lines, and use it to formulate a parameterized notion of strong fairness, generalizing the traditional notion of strong fairness [5] in a way that facilitates compositional analysis. We then provide a denotational semantics using an abstract form of *trace*, augmented with information about enabledness. This semantics is related to the *failures* model for CSP [2] and to Hennessy's *acceptance trees* [7], although neither of these earlier models incorporated fairness. Our semantics is *adequate* with respect to a deadlock-sensitive notion of behavior: whenever two processes have the same set of traces they exhibit the same possible behaviors in all program contexts, assuming fair execution. By introducing closure conditions on trace sets, we achieve a fully abstract semantics [13]: two processes have the same closed set of traces if and only if they exhibit the same behaviors in all program contexts, assuming fair execution. This means that the closed trace semantics is at precisely the correct level of abstraction to support syntax-directed, compositional reasoning about fair behavior. We also discuss how to adapt our semantics to achieve full abstraction with respect to some other natural notions of behavior.

2 Communicating Processes

2.1 Syntax

The abstract syntax of our programming language is defined as follows. Expressions are built from identifiers and boolean and integer constants, using the usual arithmetic and boolean operations. We let I range over the set **Ide** of identifiers, B range over the set **BExp** of boolean expressions, and E range over the set **Exp** of integer expressions. Commands C, guarded commands GC, and guards G are given by the following abstract grammar:

$$\begin{aligned} C ::= {} & \mathsf{skip} \mid I := E \mid C_1; C_2 \mid \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid \mathsf{while}\ B\ \mathsf{do}\ C \mid \\ & GC \mid C_1 \| C_2 \mid C \backslash h \\ GC ::= {} & (G \to C) \mid GC_1 \Box GC_2 \\ G ::= {} & h?I \mid h!E \end{aligned}$$

Here h ranges over a set **Chan** of channel names. In examples, as is conventional, we will use the abbreviation G for a guarded command of form $G \to \mathsf{skip}$. We let **Com** be the set of commands.

We impose the syntactic constraint that in all parallel commands $C_1 \| C_2$ the components C_1 and C_2 must have disjoint sets of free identifiers, corresponding to the requirement that parallel processes have disjoint local states.

$$\langle \bullet, s\rangle term \quad \frac{\langle C, s\rangle term}{\langle C\backslash h, s\rangle term} \quad \frac{\langle C_1, s_1\rangle term \quad \langle C_2, s_2\rangle term}{\langle C_1 \| C_2, s_1 \cup s_2\rangle term} \text{ if } disjoint(s_1, s_2)$$

Fig. 1. Inference rules for the predicate *term*

We write $free[\![B]\!]$ and $free[\![E]\!]$ for the free identifiers of B and E respectively, and for commands C we define $free[\![C]\!]$ as usual, by structural induction:

$$\begin{aligned}
free[\![\textsf{skip}]\!] &= \emptyset \\
free[\![I := E]\!] &= \{I\} \cup free[\![E]\!] \\
free[\![C_1; C_2]\!] &= free[\![C_1]\!] \cup free[\![C_2]\!] = free[\![C_1 \| C_2]\!] \\
free[\![\textsf{if } B \textsf{ then } C_1 \textsf{ else } C_2]\!] &= free[\![B]\!] \cup free[\![C_1]\!] \cup free[\![C_2]\!] \\
free[\![\textsf{while } B \textsf{ do } C]\!] &= free[\![B]\!] \cup free[\![C]\!] \\
free[\![h?I]\!] &= \{I\} \\
free[\![h!E]\!] &= free[\![E]\!] \\
free[\![G \rightarrow C]\!] &= free[\![G]\!] \cup free[\![C]\!] \\
free[\![GC_1 \Box GC_2]\!] &= free[\![GC_1]\!] \cup free[\![GC_2]\!] \\
free[\![C\backslash h]\!] &= free[\![C]\!].
\end{aligned}$$

2.2 Operational semantics

A state is a finite partial function from identifiers to integers. We use N for the set of integers, and we let $S = [\mathbf{Ide} \rightarrow_p N]$ denote the set of states. When s is a state 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 $dom(s)$, is the set of identifiers for which the state has a value. We say that two states s_1 and s_2 are *disjoint*, and write $disjoint(s_1, s_2)$, when their domains are disjoint.

We assume for simplicity that expression evaluation always terminates and causes no side-effects, and we assume that the evaluation semantics for boolean and integer expressions are given. We write $\langle E, s\rangle \longrightarrow^* n$ to indicate that E evaluates to value n in state s, with a similar notation for boolean expressions. We use $V = \{\mathtt{tt}, \mathtt{ff}\}$ for the set of truth values.

For commands, guarded commands and guards we use a *labelled transition system*, much as in [16]. Command configurations have the form $\langle C, s\rangle$, where s is a state defined at least on the free identifiers of C. We use the placeholder $\bullet$ to represent a terminated command, for instance in configurations of the form $\langle \bullet \| C, s\rangle$, $\langle C \| \bullet, s\rangle$ and $\langle \bullet \backslash h, s\rangle$. A configuration $\langle C, s\rangle$ is *terminal* iff $\langle C, s\rangle term$ can be proven from the inference rules in Figure 1. In particular, a parallel command terminates only when each of its component commands has terminated.

A *label* λ is a member of the set $\Lambda = \{\epsilon\} \cup \{h?n, h!n \mid h \in \mathbf{Chan}, n \in N\}$. The label of a transition indicates the type of atomic action involved: ϵ represents an internal action, $h?n$ represents the receipt of value n on channel h, and $h!n$ represents the transmission of value n along channel h. 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 $match(\lambda_1, \lambda_2)$. For a label λ, $chan(\lambda)$ is the channel associated with λ; by convention, we define $chan(\epsilon) = \epsilon$. We write

$$\langle \mathsf{skip}, s\rangle \xrightarrow{\epsilon} \langle \bullet, s\rangle \qquad \frac{\langle E, s\rangle \longrightarrow^* n}{\langle I := E, s\rangle \xrightarrow{\epsilon} \langle \bullet, [s \mid I = n]\rangle}$$

$$\frac{\langle C_1, s\rangle \xrightarrow{\lambda} \langle C_1', s'\rangle \quad \neg\langle C_1', s'\rangle term}{\langle C_1; C_2, s\rangle \xrightarrow{\lambda} \langle C_1'; C_2, s'\rangle} \qquad \frac{\langle C_1, s\rangle \xrightarrow{\lambda} \langle C_1', s'\rangle term}{\langle C_1; C_2, s\rangle \xrightarrow{\lambda} \langle C_2, s'\rangle}$$

$$\frac{\langle B, s\rangle \longrightarrow^* \mathtt{tt}}{\langle \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s\rangle \xrightarrow{\epsilon} \langle C_1, s\rangle} \qquad \frac{\langle B, s\rangle \longrightarrow^* \mathtt{ff}}{\langle \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s\rangle \xrightarrow{\epsilon} \langle C_2, s\rangle}$$

$$\frac{\langle B, s\rangle \longrightarrow^* \mathtt{tt}}{\langle \mathsf{while}\ B\ \mathsf{do}\ C, s\rangle \xrightarrow{\epsilon} \langle C; \mathsf{while}\ B\ \mathsf{do}\ C, s\rangle} \qquad \frac{\langle B, s\rangle \longrightarrow^* \mathtt{ff}}{\langle \mathsf{while}\ B\ \mathsf{do}\ C, s\rangle \xrightarrow{\epsilon} \langle \bullet, s\rangle}$$

Fig. 2. Transition rules for sequential constructs

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

$$\frac{\langle E, s\rangle \longrightarrow^* n}{\langle h!E, s\rangle \xrightarrow{h!n} \langle \bullet, s\rangle} \qquad \frac{\langle G, s\rangle \xrightarrow{\lambda} \langle \bullet, s'\rangle}{\langle G \to C, s\rangle \xrightarrow{\lambda} \langle C, s'\rangle}$$

$$\frac{\langle GC_1, s\rangle \xrightarrow{\lambda} \langle C, s'\rangle}{\langle GC_1 \Box GC_2, s\rangle \xrightarrow{\lambda} \langle C, s'\rangle} \qquad \frac{\langle GC_2, s\rangle \xrightarrow{\lambda} \langle C, s'\rangle}{\langle GC_1 \Box GC_2, s\rangle \xrightarrow{\lambda} \langle C, s'\rangle}$$

$$\frac{\langle C_1, s_1\rangle \xrightarrow{\lambda} \langle C_1', s_1'\rangle}{\langle C_1 \| C_2, s_1 \cup s_2\rangle \xrightarrow{\lambda} \langle C_1' \| C_2, s_1' \cup s_2\rangle} \text{ if } \mathit{disjoint}(s_1, s_2)$$

$$\frac{\langle C_2, s_2\rangle \xrightarrow{\lambda} \langle C_2', s_2'\rangle}{\langle C_1 \| C_2, s_1 \cup s_2\rangle \xrightarrow{\lambda} \langle C_1 \| C_2', s_1 \cup s_2'\rangle} \text{ if } \mathit{disjoint}(s_1, s_2)$$

$$\frac{\langle C_1, s_1\rangle \xrightarrow{\lambda_1} \langle C_1', s_1'\rangle \quad \langle C_2, s_2\rangle \xrightarrow{\lambda_2} \langle C_2', s_2'\rangle}{\langle C_1 \| C_2, s_1 \cup s_2\rangle \xrightarrow{\epsilon} \langle C_1' \| C_2', s_1' \cup s_2'\rangle} \text{ if } \mathit{disjoint}(s_1, s_2) \ \&\ \mathit{match}(\lambda_1, \lambda_2)$$

$$\frac{\langle C, s\rangle \xrightarrow{\lambda} \langle C', s'\rangle}{\langle C \backslash h, s\rangle \xrightarrow{\lambda} \langle C' \backslash h, s'\rangle} \text{ if } \mathit{chan}(\lambda) \neq h$$

Fig. 3. Transition rules for parallel constructs

$\langle C, s\rangle \xrightarrow{\lambda} \langle C', s'\rangle$ to indicate that command C in state s can perform an action labelled λ, leading to C' in state s'. The transition relations $\xrightarrow{\lambda}$ ($\lambda \in \Lambda$) are characterized by the axioms and inference rules in Figure 2 and Figure 3.

A *direction* d is a member of the set $\Delta = \{h?, h! \mid h \in \mathbf{Chan}\}$. For a label λ, $dir(\lambda)$ is the direction associated with λ, and we define $dir(\epsilon) = \epsilon$. Two directions d_1 and d_2 match iff one has the form $h?$ and the other $h!$ for some channel h, and again we write $match(d_1, d_2)$. For any direction d, $\bar{d}$ is the unique d' such that $match(d, d')$. Two sets X_1 and X_2 of directions match, written $match(X_1, X_2)$, if there exist $d_1 \in X_1$, $d_2 \in X_2$ such that $match(d_1, d_2)$.

A configuration $\langle C, s\rangle$ is *enabled* if $\langle C, s\rangle \xrightarrow{\lambda} \langle C', s'\rangle$ for some C', s' and λ. A configuration $\langle C, s\rangle$ is *blocked* if it is neither enabled nor terminal; in such a case, we write $\langle C, s\rangle dead$. A command C is enabled (respectively, blocked)

in state s iff the configuration $\langle C, s\rangle$ is enabled (respectively, blocked). The set of enabled directions for a configuration $\langle C, s\rangle$ is given by

$$\mathit{inits}(C, s) = \{\mathit{dir}(\lambda) \mid \exists C', s'.\langle C, s\rangle \xrightarrow{\lambda} \langle C', s'\rangle\}.$$

A *computation* is a maximal finite or infinite sequence of transitions; a *partial computation* is a finite sequence of transitions. A finite computation ending in a terminal configuration is said to be *successful*, while a finite computation ending in a blocked configuration is *deadlocked*. We extend the notions of enabling and blocking to a partial computation ρ in the obvious way, taking $\mathit{inits}(\rho)$ to be the set of directions enabled in the final configuration of ρ.

3 Strong Fairness

In this paper, we focus on *strong (process) fairness* which, informally, guarantees that every process which is enabled infinitely often proceeds infinitely often.

Whether a process is enabled depends on the context in which it appears, since communication on a restricted channel is enabled only when synchronization on that channel is possible. For example, consider the program $(C_1 \| (C_2 \| C_3)) \backslash a \backslash b$, where:

$$C_1 \equiv \textsf{while true do } a?x,\ C_2 \equiv \textsf{while true do } a!0,\ C_3 \equiv \textsf{while true do } b!0.$$

Every strongly fair computation of $C_2 \| C_3$ contains infinitely many outputs on both channels a and b, since C_2 can output on a infinitely often and C_3 can output on b infinitely often. When placed in the larger context, however, C_3 is prevented from performing output on b because this channel is now restricted and no matching input on b is ever available. In contrast, even though channel a is also restricted in this context, C_2 is repeatedly enabled for synchronization with C_1. Thus the program has an infinite fair execution in which C_3 is blocked, but none in which C_1 or C_2 ever become blocked.

This example motivates the introduction of generalized notions of enabledness and fairness, parameterized by a set of directions representing the context. For a set F of directions we characterize the set of computations that are "fair modulo F". Roughly speaking, a computation ρ of C is fair modulo F if every parallel subcomponent which is enabled infinitely often either makes progress infinitely often or eventually reaches a configuration in which it can only perform actions involving the directions in F and it remains unable to synchronize with any other components. The idea is that even though the directions in F may be enabled infinitely often along ρ, it is possible to construct a program context $P[-]$ which restricts communication on the channels in F and fails to provide those components with sufficient opportunities to synchronize. For instance, in the example above, the infinite computation of $C_2 \| C_3$ which never outputs along channel b is fair modulo $\{b!\}$: the context $(C_1 \| -) \backslash a \backslash b$ restricts communication on channel b and provides no synchronization opportunities for C_3's $b!0$ action. We now show how to capture this notion of parameterized fairness formally.

Definition 3.1 Let F be a finite set of directions. A configuration $\langle C, s\rangle$ is *enabled modulo* F if $\mathit{inits}(C,s) - F$ is non-empty; $\langle C, s\rangle$ is *blocked modulo* F if $\mathit{inits}(C,s) \subseteq F$.

Clearly, any configuration which is blocked mod F is also blocked mod F' for all $F' \supseteq F$. As before, we extend these notions to partial computations in the obvious way: ρ is blocked modulo F if its final configuration is blocked modulo F.

Using these definitions, we can now give an operational characterization of strongly fair computation modulo F. When $F = \emptyset$ this characterization coincides with the traditional notion of strong process fairness (*cf.* [5],[1]).

The fairness of a (possibly partial) computation ρ of a command C is determined by the fairness set F, the syntactic structure of C, and the form of ρ. A finite successful computation is always fair, and a partial computation blocked modulo F is fair modulo F. A computation of a command of form $C\backslash h$ is fair modulo F iff the underlying computation of C is fair modulo $F \cup \{h!, h?\}$. A computation of any other non-parallel command is fair modulo F iff the underlying computations of its component commands are all fair modulo F. Finally, an infinite computation ρ of $C_1 \| C_2$ is fair modulo F if and only if it can be obtained by merging (and synchronizing) a computation ρ_1 of C_1 and a computation ρ_2 of C_2 which satisfy the following conditions: ρ_1 is fair modulo F_1, ρ_2 is fair modulo F_2, $F \supseteq F_1 \cup F_2$, neither component infinitely often enables synchronization with a direction in the other component's fairness set, and neither component infinitely often takes a direction in the other component's fairness set.

Example 3.2 The following examples highlight the compositional aspect of this characterization.

(i) The partial computation $\rho_1 = \langle a!0{\rightarrow}b!1, s\rangle \xrightarrow{a!0} \langle b!1, s\rangle$ is fair modulo $\{b!\}$.

(ii) Let C be the program **while true do** $c!1$. The infinite computation

$$\rho_2 = \langle C, s\rangle \xrightarrow{\epsilon} \langle (c!1; C), s\rangle \xrightarrow{c!1} \langle C, s\rangle \xrightarrow{\epsilon} \langle (c!1; C), s\rangle \xrightarrow{c!1} \langle C, s\rangle \xrightarrow{\epsilon} \cdots$$

is fair modulo $\emptyset$; the only direction enabled infinitely often along ρ_2 is $c!$.

(iii) Let ρ be the infinite computation

$$\begin{aligned}\langle (a!0{\rightarrow}b!1) \| C, s\rangle &\xrightarrow{a!0} \langle b!1 \parallel C, s\rangle \xrightarrow{\epsilon} \langle b!1 \parallel (c!1; C), s\rangle \\ &\xrightarrow{c!1} \langle b!1 \parallel C, s\rangle \xrightarrow{\epsilon} \langle b!1 \parallel (c!1; C), s\rangle \xrightarrow{c!1} \cdots\end{aligned}$$

in which no $b!1$ transition is ever made. This computation can be obtained by merging ρ_1 and ρ_2. Since ρ_2 neither uses nor enables synchronization with $b!$ infinitely often, ρ is fair modulo $\{b!\}$.

(iv) As an immediate consequence, the computation

$$\begin{aligned}\langle ((a!0{\rightarrow}b!1) \| C)\backslash b, s\rangle &\xrightarrow{a!0} \langle (b!1 \parallel C)\backslash b, s\rangle \xrightarrow{\epsilon} \langle (b!1 \parallel c!1; C)\backslash b, s\rangle \\ &\xrightarrow{c!1} \langle (b!1 \parallel C)\backslash b, s\rangle \xrightarrow{\epsilon} \cdots\end{aligned}$$

is fair modulo $\emptyset$.

Example 3.3 The next examples illustrate the rôle of fairness sets in determining those contexts in which a given computation can be considered fair.

(i) Let C be the program **while true do** $(a!1 \,\Box\, b!1)$, and consider the computation

$$\rho_c = \langle C, s\rangle \xrightarrow{\epsilon} \langle (a!1 \,\Box\, b!1); C, s\rangle \xrightarrow{a!1} \langle C, s\rangle \xrightarrow{\epsilon} \langle (a!1 \,\Box\, b!1); C, s\rangle \xrightarrow{a!1} \cdots$$

which never outputs along channel b.

The set of infinitely enabled directions of ρ_c is $\{a!, b!\}$, but ρ_c is fair mod $\emptyset$ because there are no parallel subcomponents of C which become blocked along ρ_c.

(ii) Define $C_1 \equiv$ **while true do** $a!1$ and $C_2 \equiv b!1 \rightarrow$(**while true do** $b!1$), and consider the computation

$$\rho = \langle C_1 \| C_2, s\rangle \xrightarrow{\epsilon} \langle (a!1; C_1) \| C_2, s\rangle \xrightarrow{a!1} \langle C_1 \| C_2, s\rangle \xrightarrow{\epsilon} \cdots$$

which never outputs along channel b.

Again, the set of infinitely enabled directions of ρ is $\{a!, b!\}$. In contrast to the previous example, however, ρ is not fair mod $\emptyset$, because the component C_2 remains blocked mod $\{b!\}$; nevertheless, ρ is fair mod $\{b!\}$.

(iii) Let C_p be the program **while true do** $(a!0 \,\Box\, b?z)$, and let ρ_p be the computation

$$\langle C_p, s\rangle \xrightarrow{\epsilon} \langle (a!0 \,\Box\, b?z); C_p, s\rangle \xrightarrow{a!0} \langle C_p, s\rangle \xrightarrow{\epsilon} \langle (a!0 \,\Box\, b?z); C_p, s\rangle \xrightarrow{a!0} \cdots$$

which never receives input along channel b. ρ_p is fair mod $\emptyset$ and enables both $a!$ and $b?$ infinitely often.

Let $P[-]$ be the context $([-] \| C_p) \backslash b$. There is a fair (mod $\emptyset$) computation of $P[C]$ which never synchronizes on channel b, because no subcomponents of C or the surrounding context become blocked. In contrast, every fair (mod $\emptyset$) computation of $P[C_1 \| C_2]$ must eventually synchronize on channel b, because it is unfair for C_2 to be forced to block on $b!$ when a matching direction is enabled infinitely often. Thus there is no fair execution of $P[C_1 \| C_2]$ in which the C_p component performs ρ_p: such an execution would treat C_2 unfairly.

4 Denotational Semantics

We now construct a denotational semantics which corresponds to the operational characterization of fair execution given in the previous section. The meaning of a program will be a set of traces, each trace representing an abstract view of a fair computation. We show how the traces of a command can be constructed from the traces of its syntactic subcommands, justifying our definitions by appealing to the operational transition rules. The resulting denotational semantics supports compositional reasoning about program behavior, assuming fair execution.

4.1 Traces

A *step* is a member of $\Sigma = S \times \Lambda \times S$; intuitively, (s, λ, s') represents a transition of form $\langle C, s\rangle \xrightarrow{\lambda} \langle C', s'\rangle$. A *(simple) trace* is a finite or infinite sequence of steps representing a sequence of uninterrupted transitions. It is convenient to introduce a family ϵ_s ($s \in S$) of "local units" for concatenation, so that $\alpha\varepsilon_s = \alpha$ and $\varepsilon_s\beta = \beta$ whenever s is the final state of α and the initial state of β. Thus we define the set of traces to be $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$, where $\Sigma^0 = \{\varepsilon_s \mid s \in S\}$, $\Sigma^* = \Sigma^+ \cup \Sigma^0$, and

$$\Sigma^+ = \{(s_0, \lambda_0, s_1)(s_1, \lambda_1, s_2)\ldots(s_k, \lambda_k, s_{k+1}) \mid k \geq 0 \;\&\; (\forall i \leq k+1.s_i \in S) \;\&\; (\forall i \leq k.\lambda_i \in \Lambda)\},$$
$$\Sigma^\omega = \{(s_0, \lambda_0, s_1)(s_1, \lambda_1, s_2)\ldots(s_k, \lambda_k, s_{k+1})\ldots \mid \forall i \geq 0.\, s_i \in S \,\&\, \lambda_i \in \Lambda\}.$$

Given a (possibly partial) computation ρ, $trace(\rho)$ records the state transitions and actions occurring along ρ. For example, if ρ is the computation

$$\langle C, s_0\rangle \xrightarrow{\lambda_0} \langle C_1, s_1\rangle \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_k} \langle C_{k+1}, s_{k+1}\rangle term,$$

then $trace(\rho) = (s_0, \lambda_0, s_1)(s_1, \lambda_1, s_2)\ldots(s_k, \lambda_k, s_{k+1})$. For a trace α, we write $dirs(\alpha)$ and $chans(\alpha)$ for the set of directions and the set of channels appearing along α.

Simple traces are insufficient for reasoning about fairness, because they fail to provide information about enabledness; knowing which actions could have been taken is as important as knowing which actions were taken. Obviously we need to keep track of the set of directions which are enabled infinitely often along an infinite computation. In addition, since a finite computation of a loop body may be used to generate an infinite computation of the corresponding loop, we also need to record the set of directions which are enabled along a finite computation. Thus, we let $en(\rho)$ represent the "relevant" enabling information about the computation ρ: if ρ is an infinite computation, $en(\rho)$ is the set of directions enabled infinitely often along ρ; if ρ is a finite computation, $en(\rho)$ is the set of directions enabled in some configuration of ρ.

We will represent a successful computation ρ by the pair $\langle en(\rho), trace(\rho)\rangle$ in $\mathcal{P}_{\text{fin}}(\Delta) \times \Sigma^*$; a partial computation ρ by the *acceptance* $\langle trace(\rho), inits(\rho)\rangle$ in $\Sigma^* \times \mathcal{P}_{\text{fin}}(\Delta \cup \{\epsilon\})$; and an infinite, fair mod F computation ρ by the triple $\langle F, en(\rho), trace(\rho)\rangle$. We therefore define the set Ξ of *(augmented) traces* to be:

$$\Xi = \mathcal{P}_{\text{fin}}(\Delta) \times \Sigma^* \;\cup\; \Sigma^* \times \mathcal{P}_{\text{fin}}(\Delta \cup \{\epsilon\}) \;\cup\; \mathcal{P}_{\text{fin}}(\Delta) \times \mathcal{P}_{\text{fin}}(\Delta) \times \Sigma^\omega.$$

The fair trace semantics $\mathcal{T} : \mathbf{Com} \to \mathcal{P}\Xi$ is then characterized operationally by:

$$\begin{aligned}
\mathcal{T}[\![C]\!] = \{&\langle en(\rho), trace(\rho)\rangle \mid \\
&\rho = \langle C, s_0\rangle \xrightarrow{\lambda_0} \langle C_1, s_1\rangle \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_k} \langle C_{k+1}, s_{k+1}\rangle term\} \\
\cup \{&\langle trace(\rho),\ inits(\rho)\rangle \mid \neg\langle C_{k+1}, s_{k+1}\rangle term \;\& \\
&\rho = \langle C, s_0\rangle \xrightarrow{\lambda_0} \langle C_1, s_1\rangle \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_k} \langle C_{k+1}, s_{k+1}\rangle\} \\
\cup \{&\langle F, en(\rho), trace(\rho)\rangle \mid F \in \mathcal{P}_{\text{fin}}(\Delta), \\
&\rho = \langle C, s_0\rangle \xrightarrow{\lambda_0} \cdots \xrightarrow{\lambda_k} \langle C_{k+1}, s_{k+1}\rangle \xrightarrow{\lambda_{k+1}} \cdots \text{ is fair mod } F\}.
\end{aligned}$$

4.2 Operations on trace sets

We can now give a denotational characterization of $\mathcal{T}$ by defining, for each construct in the language, a corresponding operation on trace sets.

We assume a semantic function $\mathcal{E} : \mathbf{Exp} \to \mathcal{P}(S \times N)$ characterized operationally by $\mathcal{E}[\![E]\!] = \{(s,n) \mid \langle E, s\rangle \longrightarrow^* n\}$. Using the operational characterization of $\mathcal{T}$ it is easy to see that:

$$\mathcal{T}[\![\mathsf{skip}]\!] = \{\langle \emptyset, (s, \epsilon, s)\rangle,\ \langle \varepsilon_s, \{\epsilon\}\rangle \mid s \in S\},$$
$$\mathcal{T}[\![I := E]\!] = \{\langle \emptyset, (s, \epsilon, [s|I = n])\rangle,\ \langle \varepsilon_s, \{\epsilon\}\rangle \mid I \in dom(s)\ \&\ (s,n) \in \mathcal{E}[\![E]\!]\}.$$

In each case, the acceptance $\langle \varepsilon_s, \{\epsilon\}\rangle$ reflects the fact that only an ϵ-transition is possible from the initial state. Similarly, for guards we obtain:

$$\begin{aligned}\mathcal{T}[\![h?I]\!] &= \{\langle \{h?\}, (s, h?n, [s|I = n])\rangle \mid s \in S\ \&\ I \in dom(s)\ \&\ n \in N\}\\ &\quad \cup \{\langle \varepsilon_s, \{h?\}\rangle \mid s \in S\ \&\ I \in dom(s)\},\\ \mathcal{T}[\![h!E]\!] &= \{\langle \{h!\}, (s, h!n, s)\rangle \mid (s,n) \in \mathcal{E}[\![E]\!]\}\\ &\quad \cup \{\langle \varepsilon_s, \{h!\}\rangle \mid s \in S\ \&\ free[\![E]\!] \subseteq dom(s)\}.\end{aligned}$$

Here, for example, the acceptance $\langle \varepsilon_s, \{h!\}\rangle$ for $h!E$ indicates that only output on channel h is initially possible.

For a sequential composition $C_1; C_2$ we need a notion of concatenation on trace sets, adapted to combine and propagate enabling information appropriately. We therefore define the sequential composition $T_1; T_2$ of trace sets T_1 and T_2 by:

$$\begin{aligned}T_1; T_2 &= \{\langle D_1 \cup D_2, \alpha\beta\rangle \mid \langle D_1, \alpha\varepsilon_s\rangle \in T_1\ \&\ \langle D_2, \varepsilon_s\beta\rangle \in T_2\}\\ &\quad \cup \{\langle \alpha, X\rangle \mid \langle \alpha, X\rangle \in T_1\}\\ &\quad \cup \{\langle \alpha\beta, X\rangle \mid \langle D, \alpha\varepsilon_s\rangle \in T_1\ \&\ \langle \varepsilon_s\beta, X\rangle \in T_2\}\\ &\quad \cup \{\langle F, D, \alpha\rangle \mid \langle F, D, \alpha\rangle \in T_1\}\\ &\quad \cup \{\langle F, D_2, \alpha\beta\rangle \mid \langle D_1, \alpha\varepsilon_s\rangle \in T_1\ \&\ \langle F, D_2, \varepsilon_s\beta\rangle \in T_2\}.\end{aligned}$$

This definition reflects the fact that when concatenating a finite trace α with an infinite trace β, only the enabling information about β remains relevant: a direction is enabled infinitely often along $\alpha\beta$ if and only if it is enabled infinitely often along β. We thus define $\mathcal{T}[\![C_1; C_2]\!] = \mathcal{T}[\![C_1]\!]; \mathcal{T}[\![C_2]\!]$ and $\mathcal{T}[\![G \to C]\!] = \mathcal{T}[\![G]\!]; \mathcal{T}[\![C]\!]$.

For loops we need a form of iteration on trace sets, again propagating information about enabling. We define the finite iteration of the trace set T to be $T^* = \bigcup_{i=0}^{\infty} T^i$, where $T^0 = \{\langle \emptyset, \varepsilon_s\rangle \mid s \in S\}$ and $T^{n+1} = T^n; T$. We then define the infinite iteration T^ω by:

$$\begin{aligned}T^\omega = &\ \{\langle \alpha, X\rangle \mid \langle \alpha, X\rangle \in T^*\}\ \cup\ \{\langle F, D, \alpha\rangle \mid \langle F, D, \alpha\rangle \in T^*\}\\ &\cup \{\langle F, D, \alpha_0\alpha_1 \ldots \alpha_k \ldots\rangle \mid (\forall i \geq 0.\, \langle D_i, \varepsilon_{s_i}\alpha_i\varepsilon_{s_{i+1}}\rangle \in T)\ \&\\ &\qquad F \in \mathcal{P}_{\mathsf{fin}}(\Delta)\ \&\ D = \{d \mid \forall i \geq 0. \exists j > i.\, d \in D_j\}\}.\end{aligned}$$

This definition may be justified intuitively as follows. Clearly T^ω contains no finite successful traces. Every acceptance of T^* is an acceptance of T^ω, and every infinite trace of T^* is also an infinite trace of T^ω. In addition, T^ω contains those infinite traces which arise by concatenating infinitely many

finite traces α_k $(k \geq 0)$ from T; in such a case, a direction is enabled infinitely often along the resulting trace iff it is enabled along infinitely many of the α_k. Letting $\mathcal{T}[\![B]\!] = \{\langle \emptyset, (s, \epsilon, s)\rangle,\ \langle \varepsilon_s, \{\epsilon\}\rangle \mid \langle B, s\rangle \longrightarrow^* \mathtt{tt}\}$, we then have $\mathcal{T}[\![\textsf{while } B \textsf{ do } C]\!] = (\mathcal{T}[\![B]\!]; \mathcal{T}[\![C]\!])^\omega \cup (\mathcal{T}[\![B]\!]; \mathcal{T}[\![C]\!])^*; \mathcal{T}[\![\neg B]\!]$.

The command $GC_1 \Box GC_2$ represents a choice between GC_1 and GC_2 to be made on the first step. Each computation of GC_i $(i = 1, 2)$ therefore gives rise to a corresponding computation of $GC_1 \Box GC_2$, in which initially any action enabled by either component was enabled. Thus, we define the guarded choice operator on trace sets as follows:

$$\begin{aligned} T_1 \Box T_2 = {} & \{\langle D_1 \cup X, \alpha_1\rangle \mid \langle D_1, \varepsilon_s \alpha_1\rangle \in T_1 \ \&\ \langle \varepsilon_s, X\rangle \in T_2\} \\ & \cup \{\langle D_2 \cup X, \alpha_2\rangle \mid \langle D_2, \varepsilon_s \alpha_2\rangle \in T_2 \ \&\ \langle \varepsilon_s, X\rangle \in T_1\} \\ & \cup \{\langle \varepsilon_s, X_1 \cup X_2\rangle \mid \langle \varepsilon_s, X_1\rangle \in T_1 \ \&\ \langle \varepsilon_s, X_2\rangle \in T_2\} \\ & \cup \{\langle \alpha, X\rangle \in T_1 \cup T_2 \mid \alpha \neq \varepsilon_s\} \\ & \cup \{\langle F, D, \alpha\rangle \mid \langle F, D, \alpha\rangle \in T_1 \cup T_2\}. \end{aligned}$$

We thus obtain the equation $\mathcal{T}[\![GC_1 \Box GC_2]\!] = \mathcal{T}[\![GC_1]\!] \Box \mathcal{T}[\![GC_2]\!]$.

The computations of $C\backslash h$ are the computations of C which do not visibly use channel h. Correspondingly, $T\backslash h$ can be obtained from T by removing traces that use h and deleting $h?$ and $h!$ from the enabling sets in the remaining traces. Given a set of directions X, we let $X\backslash h = X - \{h!, h?\}$. We then define $T\backslash h$ by:

$$\begin{aligned} T\backslash h = {} & \{\langle D\backslash h, \alpha\rangle \mid \langle D, \alpha\rangle \in T \ \&\ h \notin \mathit{chans}(\alpha)\} \\ & \cup \{\langle \alpha, X\backslash h\rangle \mid \langle \alpha, X\rangle \in T \ \&\ h \notin \mathit{chans}(\alpha)\} \\ & \cup \{\langle F', D\backslash h, \alpha\rangle \mid \langle F, D, \alpha\rangle \in T \ \&\ F' \supseteq F\backslash h \ \&\ h \notin \mathit{chans}(\alpha)\}, \end{aligned}$$

so that $\mathcal{T}[\![C\backslash h]\!] = \mathcal{T}[\![C]\!]\backslash h$.

For a parallel command $C_1 \| C_2$, we begin by formulating a "fairmerge" relation for simple traces. This adapts Park's fairmerge definition [15,3], originally given for shared-variable parallel programs, to incorporate local states and synchronized communication. We define a relation $\mathit{fairmerge} \subseteq \Sigma^\infty \times \Sigma^\infty \times \Sigma^\infty$ such that $(\alpha, \beta, \gamma) \in \mathit{fairmerge}$ iff γ can be obtained by merging (and possibly synchronizing) α with β.

Consider first the steps of $C_1 \| C_2$. If $\sigma_1 = (s_1, \lambda, s_1')$ is a step of C_1 and s is a local state of C_2, then $\sigma_1 \| \varepsilon_s = (s_1 \cup s, \lambda, s_1' \cup s)$ represents a step of $C_1 \| C_2$ in which C_2 idles in its local state. If $\sigma_1 = (s_1, \lambda, s_1')$ and $\sigma_2 = (s_2, \bar{\lambda}, s_2')$ are matching steps of C_1 and C_2 then $\sigma_1 \| \sigma_2 = (s_1 \cup s_2, \epsilon, s_1' \cup s_2')$ represents a synchronizing step of $C_1 \| C_2$. Hence the steps of $C_1 \| C_2$ can be characterized in terms of the following set of triples:

$$\begin{aligned} \mathcal{A} = {} & \{(\sigma, \varepsilon_s, \sigma \| \varepsilon_s), (\varepsilon_s, \sigma, \sigma \| \varepsilon_s) \mid \sigma \in \Sigma \ \&\ s \in S \ \&\ \mathit{disjoint}(s, \sigma)\} \\ & \cup \{(\sigma_1, \sigma_2, \sigma_1 \| \sigma_2) \mid \sigma_1 \in \Sigma \,\&\, \sigma_2 \in \Sigma \,\&\, \mathit{disjoint}(\sigma_1, \sigma_2) \,\&\, \mathit{match}(\sigma_1, \sigma_2)\}. \end{aligned}$$

For a trace α of C_1 and a local state s of C_2, we let $\alpha \| \varepsilon_s$ be the trace obtained by combining s with all states in α. If C_2 has terminated in state s, this again represents a fair execution of $C_1 \| C_2$. We therefore define

$$\mathcal{B} = \{(\alpha, \varepsilon_s, \alpha \| \varepsilon_s),\ (\varepsilon_s, \alpha, \alpha \| \varepsilon_s) \mid \alpha \in \Sigma^\infty \ \&\ s \in S \ \&\ \mathit{disjoint}(s, \alpha)\}.$$

We extend the operations of concatenation and iteration to triples of traces

(componentwise), and to sets of triples (elementwise), in the obvious way. It then follows that the desired fairmerge relation can be characterized as $fairmerge = \mathcal{A}^*\mathcal{B} \cup \mathcal{A}^\omega$.

Next we define the parallel composition $T_1 \| T_2$ of trace sets T_1 and T_2. This corresponds directly to the operational characterization of fair computation for $C_1 \| C_2$ given in Section 3, although the presentation here is more explicit (formalizing the notion of fair merging) and more detailed (propagating information about enabledness). For acceptance sets X_1 and X_2, we define $X_1 \| X_2 = X_1 \cup X_2 \cup \{\epsilon \mid match(X_1, X_2)\}$, and we write $bounded(F, \alpha)$ when each direction in F occurs only finitely often along α. Where the subscripts i and j appear below, we assume $i, j \in \{1, 2\}$ and $i \neq j$.

$$
\begin{aligned}
T_1 \| T_2 = \{&\langle D_1 \cup D_2, \alpha\rangle \mid \\
&\quad \langle D_1, \alpha_1\rangle \in T_1 \;\&\; \langle D_2, \alpha_2\rangle \in T_2 \;\&\; (\alpha_1, \alpha_2, \alpha) \in fairmerge\} \\
&\cup \{\langle \alpha, X_1 \| X_2\rangle \mid \\
&\quad \langle \alpha_1, X_1\rangle \in T_1 \;\&\; \langle \alpha_2, X_2\rangle \in T_2 \;\&\; (\alpha_1, \alpha_2, \alpha) \in fairmerge\} \\
&\cup \{\langle \gamma, X\rangle \mid \langle \alpha, X\rangle \in T_i \;\&\; \langle D, \beta\rangle \in T_j \;\&\; (\alpha, \beta, \gamma) \in fairmerge\} \\
&\cup \{\langle F, D_j, \gamma\rangle \mid \\
&\quad \langle D_i, \alpha\rangle \notin T_i \;\&\; \langle F, D_j, \beta\rangle \in T_j \;\&\; (\alpha, \beta, \gamma) \in fairmerge\} \\
&\cup \{\langle F', D \cup X, \gamma\rangle \mid \\
&\quad F' \supseteq F \cup X \;\&\; \langle F, D, \alpha\rangle \in T_i \;\&\; \langle \beta, X\rangle \in T_j \;\&\; \epsilon \notin X \;\& \\
&\quad bounded(X, \alpha) \;\&\; \neg match(D, X) \;\&\; (\alpha, \beta, \gamma) \in fairmerge\} \\
&\cup \{\langle F', D_1 \cup D_2, \gamma\rangle \mid \\
&\quad F' \supseteq F_1 \cup F_2 \;\&\; \langle F_1, D_1, \alpha\rangle \in T_1 \;\&\; \langle F_2, D_2, \beta\rangle \in T_2 \;\& \\
&\quad bounded(F_1, \beta) \;\&\; bounded(F_2, \alpha) \;\&\; \neg match(F_1, D_2) \;\& \\
&\quad \neg match(F_2, D_1) \;\&\; (\alpha, \beta, \gamma) \in fairmerge\}.
\end{aligned}
$$

We then have $\mathcal{T}[\![C_1 \| C_2]\!] = \mathcal{T}[\![C_1]\!] \| \mathcal{T}[\![C_2]\!]$.

4.3 Denotational semantics

Proposition 4.1 *The trace semantics $\mathcal{T} : \mathbf{Com} \to \mathcal{P}\Xi$ is characterized denotationally by:*

$$
\begin{aligned}
\mathcal{T}[\![\mathsf{skip}]\!] &= \{\langle \emptyset, (s, \epsilon, s)\rangle,\ \langle \varepsilon_s, \{\epsilon\}\rangle \mid s \in S\} \\
\mathcal{T}[\![I := E]\!] &= \{\langle \emptyset, (s, \epsilon, [s|I = n])\rangle,\ \langle \varepsilon_s, \{\epsilon\}\rangle \mid (s, n) \in \mathcal{E}[\![E]\!]\} \\
\mathcal{T}[\![C_1; C_2]\!] &= \mathcal{T}[\![C_1]\!]; \mathcal{T}[\![C_2]\!] \\
\mathcal{T}[\![\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2]\!] &= \mathcal{T}[\![B]\!]; \mathcal{T}[\![C_1]\!] \cup \mathcal{T}[\![\neg B]\!]; \mathcal{T}[\![C_2]\!] \\
\mathcal{T}[\![\mathsf{while}\ B\ \mathsf{do}\ C]\!] &= (\mathcal{T}[\![B]\!]; \mathcal{T}[\![C]\!])^\omega \cup (\mathcal{T}[\![B]\!]; \mathcal{T}[\![C]\!])^*; \mathcal{T}[\![\neg B]\!] \\
\mathcal{T}[\![h?I]\!] &= \{\langle \{h?\}, (s, h?n, [s|I = n])\rangle \mid s \in S \;\&\; n \in N\} \\
&\quad \cup \{\langle \varepsilon_s, \{h?\}\rangle \mid s \in S \;\&\; I \in dom(s)\} \\
\mathcal{T}[\![h!E]\!] &= \{\langle \{h!\}, (s, h!n, s)\rangle \mid (s, n) \in \mathcal{E}[\![E]\!]\} \\
&\quad \cup \{\langle \varepsilon_s, \{h!\}\rangle \mid s \in S \;\&\; free[\![E]\!] \subseteq dom(s)\} \\
\mathcal{T}[\![G \to C]\!] &= \mathcal{T}[\![G]\!]; \mathcal{T}[\![C]\!] \\
\mathcal{T}[\![GC_1 \,\square\, GC_2]\!] &= \mathcal{T}[\![GC_1]\!] \,\square\, \mathcal{T}[\![GC_2]\!] \\
\mathcal{T}[\![C_1 \| C_2]\!] &= \mathcal{T}[\![C_1]\!] \| \mathcal{T}[\![C_2]\!]
\end{aligned}
$$

$$\mathcal{T}[\![C\backslash h]\!] = \mathcal{T}[\![C]\!]\backslash h.$$

□

This shows that our denotational semantics accurately reflects the operational behavior of programs executing under the assumption of strong fairness.

Example 4.2 Consider the program $(a!0 \parallel a?x)\backslash a$. By definition,

$$\mathcal{T}[\![a!0]\!] = \{\langle \varepsilon_s, \{a!\}\rangle, \langle \{a!\}, (s, a!0, s)\rangle \mid s \in S\},$$
$$\mathcal{T}[\![a?x]\!] = \{\langle \varepsilon_s, \{a?\}\rangle, \langle \{a?\}, (s, a?n, [s|x = n])\rangle \mid s \in S \ \&\ x \in dom(s)\}.$$

Letting s range over S, and letting s_n abbreviate the state $[s|x = n]$, it follows that

$$\begin{aligned}\mathcal{T}[\![a!0 \parallel a?x]\!] = \{&\langle \varepsilon_s, \{a!, a?, \epsilon\}\rangle,\ \langle (s, a!0, s), \{a?\}\rangle,\ \langle (s, a?n, s_n), \{a!\}\rangle,\\ &\langle \{a!, a?\}, (s, \epsilon, s_0)\rangle,\ \langle \{a!, a?\}, (s, a?n, s_n)(s_n, a!0, s_n)\rangle,\\ &\langle \{a!, a?\}, (s, a!0, s)(s, a?n, s_n)\rangle \mid n \in N \ \&\ x \in dom(s)\}.\end{aligned}$$

Consequently,

$$\begin{aligned}\mathcal{T}[\![(a!0 \| a?x)\backslash a]\!] &= \{\langle \varepsilon_s, \{\epsilon\}\rangle, \langle \emptyset, (s, \epsilon, [s|x = 0])\rangle \mid x \in dom(s)\}\\ &= \mathcal{T}[\![x{:=}0]\!].\end{aligned}$$

Example 4.3 Consider the program $C \equiv ((C_1 \| C_2) \| C_3))\backslash left\backslash right$, where the processes C_i are defined as follows:

$C_1 \equiv$ **while true do** $left!0$,
$C_2 \equiv$ **while true do** $right!1$,
$C_3 \equiv$ **while true do** $((left?x \rightarrow out!x) \Box (right?x \rightarrow out!x))$.

The infinite traces of C_1 all have form $\langle F, \{left!\}, ((s, \epsilon, s)(s, left!0, s))^\omega\rangle$, and C_2's infinite traces all have form $\langle F, \{right!\}, ((s, \epsilon, s)(s, right!1, s))^\omega\rangle$. Because every (non-acceptance) trace of C_3 has form $\langle F, \{left?, right?, out!\}, \alpha\rangle$, the only traces of $C_1 \| C_2$ which can be successfully merged with C_3's traces must have the form $\langle \emptyset, \{left!, right!\}, \beta\rangle$. Therefore, each such β must contain infinitely many $left!0$ actions and infinitely many $right!1$ actions. As a result, every fair computation of C contains infinitely many $out!0$ actions as well as infinitely many $out!1$ actions.

The trace semantics can be used directly to prove that certain equivalences of programs hold under strong fairness. The following proposition states some of these program equivalences.

Proposition 4.4 *Each of the following laws holds under strong fairness, where we write $C_1 \equiv C_2$ to indicate that $\mathcal{T}[\![C_1]\!] = \mathcal{T}[\![C_2]\!]$:*

$$\begin{aligned}C_1 \parallel C_2 &\equiv C_2 \parallel C_1\\ (C_1 \parallel C_2) \parallel C_3 &\equiv C_1 \parallel (C_2 \parallel C_3)\\ (C_1 \parallel C_2)\backslash h &\equiv C_1 \parallel (C_2\backslash h),\ \textit{provided } h \notin \textit{chans}(C_1)\\ C\backslash h &\equiv C,\ \textit{provided } h \notin \textit{chans}(C)\\ (C\backslash h_1)\backslash h_2 &\equiv (C\backslash h_2)\backslash h_1.\end{aligned}$$

□

5 Full Abstraction

A semantics is *adequate* with respect to a given notion of behavior if whenever two terms have the same meaning, they induce the same behaviors in all program contexts. A semantics is *fully abstract* [13] with respect to a given behavioral notion if it gives two terms the same meaning if and only if they induce the same behaviors in all contexts. A fully abstract semantics makes precisely the right distinctions and retains just enough detail to support compositional reasoning about behavior.

For communicating processes there are several different natural notions of behavior. We will focus first on a form of *state trace* behavior; this corresponds to the assumption that a program is a closed system (no external communication) and that one can observe and detect each state change. We also suppose that it is possible to distinguish between deadlock and successful termination. We therefore introduce the notation $S^*\delta = \{s_0 s_1 \ldots s_k\delta \mid \forall i \in 0..k.\ s_i \in S\}$, letting the tag δ indicate deadlock.

Definition 5.1 The state trace behavior $\mathcal{M} : \mathbf{Com} \to \mathcal{P}(S^\infty \cup S^*\delta)$ is defined by:

$$\begin{aligned}\mathcal{M}[\![C]\!] = \{&s_0 s_1 \ldots s_k \mid \langle C, s_0\rangle \xrightarrow{\epsilon} \langle C_1, s_1\rangle \xrightarrow{\epsilon} \cdots \xrightarrow{\epsilon} \langle C_k, s_k\rangle term\}\\ \cup\ \{&s_0 s_1 \ldots s_k\delta \mid \langle C_0, s_0\rangle \xrightarrow{\epsilon} \langle C_1, s_1\rangle \xrightarrow{\epsilon} \cdots \xrightarrow{\epsilon} \langle C_k, s_k\rangle dead\}\\ \cup\ \{&s_0 s_1 \ldots s_k \ldots \mid\\ &\langle C_0, s_0\rangle \xrightarrow{\epsilon} \langle C_1, s_1\rangle \xrightarrow{\epsilon} \cdots \xrightarrow{\epsilon} \langle C_k, s_k\rangle \xrightarrow{\epsilon} \cdots \text{ is fair}\}.\end{aligned}$$

The semantics $\mathcal{T}$ is *adequate* with respect to $\mathcal{M}$ but makes more distinctions than it should for full abstraction. For example, consider the following commands:

$C_1 \equiv (a!0 \to b!0)\ \Box\ (a!0 \to c!0)$,
$C_2 \equiv (a!0 \to b!0)\ \Box\ (a!0 \to c!0)\ \Box\ (a!0 \to (b!0\ \Box\ c!0))$.

The trace $\langle\{b!, c!\}, (s, a!0, s)(s, b!0, s)\rangle$ and the acceptance $\langle(s, a!0, s), \{b!, c!\}\rangle$ are both possible for C_2 but not for C_1. However, after performing an $a!0$, each may perform $b!0$ or $c!0$ and each may refuse one of these actions (but not both). From this it follows that C_1 and C_2 can respond identically to all communication attempts from their environment, and as a result behave identically in all contexts.

Next consider the following guarded commands:

$GC_1 \equiv (a!0 \to b!0) \Box (a!0 \to (b!0 \Box c!0 \Box d!0))$
$GC_2 \equiv (a!0 \to b!0) \Box (a!0 \to (b!0 \Box c!0 \Box d!0)) \Box (a!0 \to (b!0 \Box c!0))$.

For similar reasons, it is impossible to find a context distinguishing between GC_1 and GC_2, even though only GC_2 has the acceptance $\langle(s, a!0, s), \{b!, c!\}\rangle$.

These two examples motivate the imposition of closure conditions on trace sets similar to those used in [2,7]. However, this still does not suffice to achieve full abstraction. To see why, recall that in a trace $\langle F, D, \alpha\rangle$ of C, F represents a set of constraints on the type of context in which α will represent a fair computation of C, and D provides information about the directions

enabled infinitely often along α. The difficulty is in determining under what circumstances one can distinguish between two commands whose traces differ only in their accompanying fairness sets or enabled sets.

Distinguishing between a process with the trace $\langle F, D_1, \alpha\rangle$ and one with the trace $\langle F, D_2, \alpha\rangle$ requires a context with a subcomponent whose own fairness constraints are satisfied by D_1 and not by D_2 (or vice versa). When placed in such a context, one process will be able to perform α without ever synchronizing with that subcomponent, whereas the other process cannot perform α without enabling that subcomponent infinitely often and therefore being forced by fairness to synchronize with it eventually.

In contrast, distinguishing a process with the trace $\langle F_1, D, \alpha\rangle$ from one with the trace $\langle F_2, D, \alpha\rangle$ requires a context which enables some direction in F_1 or F_2 (but not both) infinitely often as part of a guarded choice. Again, one process will be forced eventually to synchronize with that choice but the other process can fairly ignore it forever. For example, the context $P[-]$ in Example 3.3 was able to distinguish the traces $\langle \emptyset, \{a!, b!\}, \mathit{trace}(\rho_c)\rangle$ and $\langle \{b!\}, \{a!, b!\}, \mathit{trace}(\rho)\rangle$ by placing the $b?z$ command within a guarded choice; this was sufficient to force $C_1 \| C_2$ to synchronize on channel b but permitted C to refrain from using b at all.

Bearing these considerations in mind, consider the following commands:

$$C_1 \equiv (a!0 \rightarrow b!0 \rightarrow c!0) \,\Box\, (a!0 \rightarrow b?x) \,\Box\, (a!0 \rightarrow (b!0 \,\Box\, b?x)),$$
$$C_2 \equiv (a!0 \rightarrow b!0 \rightarrow c!0) \,\Box\, (a!0 \rightarrow b?x) \,\Box\, (a!0 \rightarrow (b!0 \,\Box\, b?x)) \,\Box\, (a!0 \rightarrow b!0).$$

The trace $\langle \{a!, b!, b?\}, (s, a!0, s)(s, b!0, s)\rangle$ is possible for C_1 and for C_2, but only $\mathcal{T}[\![C_2]\!]$ contains the trace $\langle \{a!, b!\}, (s, a!0, s)(s, b!0, s)\rangle$. To distinguish between C_1 and C_2 we would therefore need a context which allows each C_i to repeatedly perform $a!0$ then $b!0$. In order to allow an observer to determine whether $b?$ is enabled infinitely often, the context would have to enable $b!$ infinitely often and restrict communication on channel b. Since the context also needs to allow each C_i to keep performing $b!0$, it must also enable $b?$ repeatedly. As discussed above, the context must therefore contain parallel components, one which may block while continuously attempting to output on b and one which repeats $b?$ actions. However, the component which is meant to block will be enabled infinitely often and hence make progress, regardless of whether C_1 or C_2 is inserted. Essentially, the fact that $b?$ is enabled infinitely often by C_1 during this execution is masked by the context's own behavior.

Similarly, consider the guarded commands $C \equiv GC_1 \,\Box\, GC_2$ and $C' \equiv GC_1 \,\Box\, GC_2 \,\Box\, GC_3$, where:

$$GC_1 \equiv b!0 \rightarrow (\textsf{while true do } (b!0 \,\Box\, a?x \,\Box\, a!0))$$
$$GC_2 \equiv b!0 \rightarrow ((\textsf{while true do } b!0) \,\|\, (a?x \rightarrow \textsf{while true do } a?x))$$
$$GC_3 \equiv b!0 \rightarrow (\textsf{while true do } (b!0 \,\Box\, a?x)).$$

Letting α represent the simple trace $[(s, b!0, s)(s, \epsilon, s)]^\omega$, the trace sets of C and C' both contain the traces $\langle \emptyset, \{b!, a?, a!\}, \alpha\rangle$ and $\langle \{a?\}, \{a?, b!\}, \alpha\rangle$. However, the trace $\langle \emptyset, \{a?, b!\}, \alpha\rangle$ is possible for C' but not for C. To distinguish between C and C' requires a context in which GC_3 can be distinguished from both GC_1 and GC_2 *at the same time.* To distinguish GC_3 from GC_1, the context would

need to place the relevant GC_i in parallel with a process which blocks while trying to perform input on channel a. To distinguish GC_3 from GC_2 the context would need to place the relevant GC_i in parallel with a process which has infinitely many opportunities to perform output on channel a but also has other options available. The context would therefore need to contain two parallel components, one continuously attempting to perform input, the other repeatedly offering matching output. Again the "blocking" component will be enabled infinitely often by the other component, regardless of whether C or C' is inserted. Hence, no context can distinguish between them.

Therefore, in order to achieve full abstraction, we introduce the following closure conditions on trace sets. The first three conditions are analogous to conditions used in [2] and [7]; the last two conditions are motivated by the preceding examples.

Definition 5.2 Given a trace set T, the *closure* of T (written $T^\dagger$) is the smallest set containing T and satisfying the following conditions:

- Superset:
  If $\langle D, \alpha\rangle$ is in $T^\dagger$ and $D \subseteq D'$, then $\langle D', \alpha\rangle$ is in $T^\dagger$.

  If $\langle F, D, \alpha\rangle$ is in $T^\dagger$, $F \subseteq F'$ and $D \subseteq D'$, then $\langle F', D', \alpha\rangle$ is in $T^\dagger$.
- Union:
  If $\langle \alpha, X\rangle$ and $\langle \alpha, Y\rangle$ are in $T^\dagger$, then $\langle \alpha, X \cup Y\rangle$ is in $T^\dagger$.
- Convexity:
  If $\langle \alpha, X\rangle$ and $\langle \alpha, Z\rangle$ are in $T^\dagger$ and $X \subseteq Y \subseteq Z$, then $\langle \alpha, Y\rangle$ is in $T^\dagger$.
- Displacement:
  If $\langle D \cup \{d\}, \alpha\rangle$ is in $T^\dagger$, $d \notin \mathit{dirs}(\alpha)$ and $\bar{d} \in \mathit{dirs}(\alpha)$, then $\langle D, \alpha\rangle$ is in $T^\dagger$.

  If $\langle F, D \cup \{d\}, \alpha\rangle$ is in $T^\dagger$, d appears only finitely often along α, and $\bar{d}$ appears infinitely often along α, then $\langle F, D, \alpha\rangle$ is in $T^\dagger$.
- Contention:
  If $\langle F_1, D_1, \alpha\rangle$ and $\langle F_2, D_2, \alpha\rangle$ are in $T^\dagger$, $d \in (F_1 - F_2)$ and $\bar{d} \in (D_2 - D_1) - F_2$, then $\langle (F_1 \cup F_2) - \{d\}, (D_1 \cup D_2) - \{\bar{d}\}, \alpha\rangle$ is also in $T^\dagger$.

These closure conditions are precisely what is needed to obtain full abstraction. Let $\mathcal{P}^\dagger\Xi$ be the set of closed sets of traces. We define a *closed trace* semantic function $\mathcal{T}^\dagger : \mathbf{Com} \to \mathcal{P}^\dagger\Xi$ denotationally, modifying the semantic equations given for $\mathcal{T}$ earlier by building the closure property into each clause.

Definition 5.3 The closed trace semantic function $\mathcal{T}^\dagger : \mathbf{Com} \to \mathcal{P}^\dagger\Xi$ is given by:

$$\mathcal{T}^\dagger[\![\mathsf{skip}]\!] = \{\langle \emptyset, (s, \epsilon, s)\rangle,\ \langle \varepsilon_s, \{\epsilon\}\rangle \mid s \in S\}^\dagger$$
$$\mathcal{T}^\dagger[\![I := E]\!] = \{\langle \emptyset, (s, \epsilon, [s|I = n])\rangle,\ \langle \varepsilon_s, \{\epsilon\}\rangle \mid (s, n) \in \mathcal{E}[\![E]\!]\}^\dagger$$
$$\mathcal{T}^\dagger[\![C_1; C_2]\!] = (\mathcal{T}^\dagger[\![C_1]\!]; \mathcal{T}^\dagger[\![C_2]\!])^\dagger$$
$$\mathcal{T}^\dagger[\![\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2]\!] = (\mathcal{T}^\dagger[\![B]\!]; \mathcal{T}^\dagger[\![C_1]\!] \cup \mathcal{T}^\dagger[\![\neg B]\!]; \mathcal{T}^\dagger[\![C_2]\!])^\dagger$$

$$\begin{aligned}
\mathcal{T}^\dagger[\![\textsf{while } B \textsf{ do } C]\!] &= ((\mathcal{T}^\dagger[\![B]\!];\mathcal{T}^\dagger[\![C]\!])^\omega \cup (\mathcal{T}^\dagger[\![B]\!];\mathcal{T}^\dagger[\![C]\!])^*;\mathcal{T}^\dagger[\![\neg B]\!])^\dagger \\
\mathcal{T}^\dagger[\![h?I]\!] &= \{\langle\{h?\},(s,h?n,[s|I=n])\rangle \mid n \in N\}^\dagger \\
&\quad \cup \{\langle \varepsilon_s, \{h?\}\rangle \mid I \in dom(s)\} \\
\mathcal{T}^\dagger[\![h!E]\!] &= \{\langle\{h!\},(s,h!n,s)\rangle \mid (s,n) \in \mathcal{E}[\![E]\!]\}^\dagger \\
&\quad \cup \{\langle \varepsilon_s, \{h!\}\rangle \mid free[\![E]\!] \subseteq dom(s)\} \\
\mathcal{T}^\dagger[\![G{\rightarrow}C]\!] &= (\mathcal{T}^\dagger[\![G]\!];\mathcal{T}^\dagger[\![C]\!])^\dagger \\
\mathcal{T}^\dagger[\![GC_1 \Box GC_2]\!] &= (\mathcal{T}^\dagger[\![GC_1]\!] \Box \mathcal{T}^\dagger[\![GC_2]\!])^\dagger \\
\mathcal{T}^\dagger[\![C_1 \| C_2]\!] &= (\mathcal{T}^\dagger[\![C_1]\!] \| \mathcal{T}^\dagger[\![C_2]\!])^\dagger \\
\mathcal{T}^\dagger[\![C\backslash h]\!] &= (\mathcal{T}^\dagger[\![C]\!]\backslash h)^\dagger.
\end{aligned}$$

The following proposition states that for any command C, the meaning given to C by the closed trace semantics $\mathcal{T}^\dagger$ is exactly the closure of $\mathcal{T}[\![C]\!]$.

Proposition 5.4 *For all C, $\mathcal{T}^\dagger[\![C]\!] = \mathcal{T}[\![C]\!]^\dagger$.*

Proof. By induction on the structure of C, using the following equalities:

$$\begin{aligned}
&(T_1;T_2)^\dagger = (T_1^\dagger;T_2^\dagger)^\dagger, \quad T^{\omega\dagger} = (T^\dagger)^{\omega\dagger}, \quad (T_1 \Box T_2)^\dagger = (T_1^\dagger \Box T_2^\dagger)^\dagger, \\
&(T_1 \| T_2)^\dagger = (T_1^\dagger \| T_2^\dagger)^\dagger, \quad (T\backslash h)^\dagger = (T^\dagger\backslash h)^\dagger, \quad (T_1 \cup T_2)^\dagger = (T_1^\dagger \cup T_2^\dagger)^\dagger.
\end{aligned}$$

□

Proposition 5.5 *The semantic function $\mathcal{T}^\dagger$ is (inequationally) fully abstract with respect to $\mathcal{M}$: for all commands C and C',*

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

Proof (Sketch) The forward implication follows from compositionality of the semantics, monotonicity of the operations on trace sets, and the fact that for all commands C,

$$\begin{aligned}
\mathcal{M}[\![C]\!] = &\{\alpha \mid \exists D.\ \langle D,\alpha\rangle \in \mathcal{T}^\dagger[\![C]\!] \ \&\ chans(\alpha) = \{\epsilon\}\ \} \\
&\cup \{\alpha\delta \mid \langle \alpha,\emptyset\rangle \in \mathcal{T}^\dagger[\![C]\!] \ \&\ chans(\alpha) = \{\epsilon\}\ \} \\
&\cup \{\alpha \mid \exists D.\ \langle \emptyset, D,\alpha\rangle \in \mathcal{T}^\dagger[\![C]\!] \ \&\ chans(\alpha) = \{\epsilon\}\ \}.
\end{aligned}$$

The reverse implication requires showing that whenever $\mathcal{T}^\dagger[\![C]\!] \not\subseteq \mathcal{T}^\dagger[\![C']\!]$, there is a context in which C and C' induce different sets of behaviors. The context chosen depends on the nature of the trace possible for C but not possible for C'. When C has an acceptance which is not possible for C', the proof is straightforward. When C has a finite trace which is impossible for C', a simple generalization of the approach given below for infinite traces suffices.

Whenever $\langle F, D, \alpha\rangle$ is possible for C but not for C', the closure conditions ensure that any trace of form $\langle F', D', \alpha\rangle$ which is possible for C' must have some fairness constraint or enabled direction which distinguishes it from $\langle F, D, \alpha\rangle$. Let $\langle F_1, D_1, \alpha\rangle, \ldots, \langle F_m, D_m, \alpha\rangle$ be the (finitely many) minimal α-traces of C' from which every $\langle F', D', \alpha\rangle$ in $\mathcal{T}^\dagger[\![C']\!]$ can be derived by closure. It suffices to build a context which distinguishes each of these minimal traces from $\langle F, D, \alpha\rangle$.

For each $i \in 1..m$ we can choose a direction d_i which is either in $F_i - F$ or in $D_i - D$. Moreover, we can always choose these directions in such a way

that the set $\{d_i \mid 1 \leq i \leq m\}$ can be partitioned into sets X and Y, with each member of X in some $F_i - F$, each member of Y in some $D_i - D$, and $\neg match(X, Y)$.

We let $H = \{h_1, \ldots, h_k\}$ represent the set of channels which appear in C or C', and we let x, y, $f1$ and $f2$ be fresh identifiers. We use x and y to create guards corresponding to the directions in X and Y; $f1$ and $f2$ serve as flags indicating whether certain synchronizations have occurred. We let $G(X, x)$ be a set of guards in one-to-one correspondence with X as follows: $h!0$ is in $G(X, x)$ when $h?$ is in X, and $h?x$ is in $G(X, x)$ when $h!$ is in X. We define $G(Y, y)$ in a similar fashion.

We next construct a program $guess(x, f1)$ which repeatedly "guesses" how to synchronize with α by nondeterministically choosing a channel on which to communicate, choosing whether to perform input or output, and (in the case of output) choosing a value to send. Because these choices are recorded in the state, it is always possible to determine whether or not the command in parallel with $guess(x, f1)$ actually performed the trace α. Each "guessed" communication is offerred alongside the guards of $G(X, x)$ as a guarded choice; consequently, in any infinite computation of $guess(x, f1)$, synchronization with directions in X is enabled infinitely often. If one of the guards in $G(X, x)$ is ever chosen, $guess(x, f1)$ sets $f1$ to 1.

We can now define a distinguishing program context $P[-]$ as follows, where we assume $G(Y, y) = \{g_1, \ldots, g_j\}$:

$$([-] \parallel guess(x, f1) \parallel ((g_1 \rightarrow f2{:=}1) \,\Box \cdots \Box\, (g_j \rightarrow f2{:=}1))\backslash h_1 \ldots \backslash h_k.$$

Under the assumption of strong fairness, $P[C]$ has a behavior corresponding to α in which $f1$ and $f2$ are never set to 1, whereas $P[C']$ does not. □

6 Other Notions of Behavior

The state trace behavior $\mathcal{M}$ reflects the assumptions that external communication is prohibited and that every state change can be detected. In this section, we consider behaviors which relax one or both of these assumptions.

6.1 Simple traces

If we assume that a program is an arbitrary command, and that external communications are observable, it is natural to consider the *simple trace* behavior function $\mathcal{S} : \mathbf{Com} \rightarrow \mathcal{P}(\Sigma^\infty \cup \Sigma^*\delta)$, defined by:

$$\begin{aligned}\mathcal{S}[\![C]\!] = &\{trace(\rho) \mid \rho = \langle C, s_0\rangle \xrightarrow{\lambda_0} \langle C_1, s_1\rangle \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_{k-1}} \langle C_k, s_k\rangle term\} \\ &\cup \{trace(\rho)\delta \mid \rho = \langle C, s_0\rangle \xrightarrow{\lambda_0} \langle C_1, s_1\rangle \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_{k-1}} \langle C_k, s_k\rangle dead\} \\ &\cup \{trace(\rho) \mid \\ &\qquad \rho = \langle C, s_0\rangle \xrightarrow{\lambda_0} \langle C_1, s_1\rangle \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_{k-1}} \langle C_k, s_k\rangle \xrightarrow{\lambda_k} \cdots \text{ is fair}\}.\end{aligned}$$

As before, we assume that it is possible to distinguish between deadlock and success and that every single transition can be detected.

The proof of Proposition 5.5 can be adapted easily to show that that $\mathcal{T}^\dagger$ is also inequationally fully abstract with respect to $\mathcal{S}$. Thus $\mathcal{M}$ and $\mathcal{S}$ induce the same notion of contextual equivalence: two commands have the same state trace behaviors in all contexts if and only if they have the same simple trace behaviors in all contexts.

6.2 Stuttering and mumbling

So far we have assumed an "omniscient" observer capable of detecting every state change made during a computation. If instead we suppose that an observer sees only a subsequence of the states during a computation, we obtain corresponding notions of behavior based on $\longrightarrow^*$, the reflexive, transitive closure of the one-step transition relation. One then needs to impose closure conditions on trace sets corresponding to "stuttering" and "mumbling" [12,3].

Definition 6.1 The relation $\rightsquigarrow \subseteq \Sigma^\infty \times \Sigma^\infty$ is the least reflexive, transitive relation on simple traces satisfying the following conditions:

- Stuttering: $\alpha\varepsilon_s\beta \rightsquigarrow \alpha(s,\epsilon,s)\beta$.
- Mumbling: $\alpha(s,\epsilon,s')(s',\lambda,s'')\beta \rightsquigarrow \alpha(s,\lambda,s'')\beta$ and $\alpha(s,\lambda,s')(s',\epsilon,s'')\beta \rightsquigarrow \alpha(s,\lambda,s'')\beta$.

To be precise, each condition may be applied finitely often at each position along a trace.

Definition 6.2 Given a trace set T, let T_* be the smallest set containing T and closed under the conditions below:

- If $\langle D,\alpha\rangle$ is in T_* and $\alpha \rightsquigarrow \alpha'$, then $\langle D,\alpha'\rangle$ is also in T_*.
- If $\langle F,D,\alpha\rangle$ is in T_* and $\alpha \rightsquigarrow \alpha'$, then $\langle F,D,\alpha'\rangle$ is also in T_*.
- If $\langle \alpha, X\rangle$ is in T_* and $\alpha \rightsquigarrow \alpha'$, then $\langle \alpha', X\rangle$ is also in T_*.
- If $\langle \alpha, X\rangle$ is in T_*, then $\langle \alpha, \{\epsilon\}\rangle$ is also in T_*.
- If $\langle \alpha(s,\epsilon,s'), X\rangle$ is in T_*, then $\langle \alpha, X\cup\{\epsilon\}\rangle$ is also in T_*.

The fourth condition reflects the fact that each stuttering step performed on a "complete" computation leads to a new partial computation as well. The fifth closure condition arises because acceptances must now account for those actions which are possible after some finite number of ϵ-transitions.

Stuttering and mumbling correspond to generalized relations $\stackrel{\lambda}{\Longrightarrow}$ $(\lambda \in \Lambda)$, where $\stackrel{\epsilon}{\Longrightarrow}$ is the reflexive, transitive closure of $\stackrel{\epsilon}{\longrightarrow}$ and $\stackrel{\lambda}{\Longrightarrow}$ (for $\lambda \neq \epsilon$) is defined so that $\langle C,s\rangle \stackrel{\lambda}{\Longrightarrow} \langle C',s'\rangle$ if and only if there exist C_1, C_2, s_1, s_2 for which $\langle C,s\rangle \stackrel{\epsilon}{\Longrightarrow} \langle C_1,s_1\rangle \stackrel{\lambda}{\longrightarrow} \langle C_2,s_2\rangle \stackrel{\epsilon}{\Longrightarrow} \langle C',s'\rangle$.

Given a trace set T, we define $T_*^\dagger = (T_*)^\dagger$, so that $T_*^\dagger$ is closed under stuttering, mumbling, and the closure conditions of Definition 5.2.

We let $\mathcal{P}_*^\dagger\Xi$ be the set of closed sets of traces. Much as before, we can define a denotational semantic function $\mathcal{T}_*^\dagger : \mathbf{Com} \to \mathcal{P}_*^\dagger\Xi$ such that for all commands C, $\mathcal{T}_*^\dagger[\![C]\!] = (\mathcal{T}[\![C]\!])_*^\dagger$. Moreover, $\mathcal{T}_*^\dagger$ is inequationally fully abstract with respect to both the generalized state transition trace behav-

ior $\mathcal{M}_* : \mathbf{Com} \to \mathcal{P}(S^\infty \cup S^*\delta)$ and the generalized simple trace behavior $\mathcal{S}_* : \mathbf{Com} \to \mathcal{P}(\Sigma^\infty \cup \Sigma^*\delta)$ defined below:

$$\begin{aligned}\mathcal{M}_*[\![C]\!] = &\{s_0 s_1 \ldots s_k \mid \langle C, s_0\rangle \stackrel{\epsilon}{\Longrightarrow} \langle C_1, s_1\rangle \stackrel{\epsilon}{\Longrightarrow} \cdots \stackrel{\epsilon}{\Longrightarrow} \langle C_k, s_k\rangle term\}\\ &\cup \{s_0 s_1 \ldots s_k \delta \mid \langle C_0, s_0\rangle \stackrel{\epsilon}{\Longrightarrow} \langle C_1, s_1\rangle \stackrel{\epsilon}{\Longrightarrow} \cdots \stackrel{\epsilon}{\Longrightarrow} \langle C_k, s_k\rangle dead\}\\ &\cup \{s_0 s_1 \ldots s_k \ldots \mid \langle C_0, s_0\rangle \stackrel{\epsilon}{\Longrightarrow} \cdots \stackrel{\epsilon}{\Longrightarrow} \langle C_k, s_k\rangle \stackrel{\epsilon}{\Longrightarrow} \cdots \text{ is fair}\},\end{aligned}$$

$$\begin{aligned}\mathcal{S}_*[\![C]\!] = &\{trace(\rho) \mid \rho = \langle C, s_0\rangle \stackrel{\lambda_0}{\Longrightarrow} \langle C_1, s_1\rangle \stackrel{\lambda_1}{\Longrightarrow} \cdots \stackrel{\lambda_{k-1}}{\Longrightarrow} \langle C_k, s_k\rangle term\}\\ &\cup \{trace(\rho)\delta \mid \rho = \langle C, s_0\rangle \stackrel{\lambda_0}{\Longrightarrow} \langle C_1, s_1\rangle \stackrel{\lambda_1}{\Longrightarrow} \cdots \stackrel{\lambda_{k-1}}{\Longrightarrow} \langle C_k, s_k\rangle dead\}\\ &\cup \{trace(\rho) \mid \rho = \langle C, s_0\rangle \stackrel{\lambda_0}{\Longrightarrow} \cdots \stackrel{\lambda_{k-1}}{\Longrightarrow} \langle C_k, s_k\rangle \stackrel{\lambda_k}{\Longrightarrow} \cdots \text{ is fair}\}.\end{aligned}$$

The proofs of full abstraction are analogous to that presented in the previous section.

7 Related and Future Work

Our semantics is related to the *failures* model of CSP [2,11], Roscoe's failures model of *occam* [17], and Hennessy's *acceptance tree* models of communicating processes [7–9]. Of these only [17] and [9] consider communicating processes with local state; neither of these papers deals with fairness. Roscoe [17] assumes that divergence (the ability to perform an infinite sequence of ϵ-moves) is "catastrophic": his semantics identifies all pairs of possibly divergent processes. As a result his model is tailored to reasoning only about total correctness properties of programs. Hennessy [9] achieves full abstraction with respect to a notion of behavior based on "testing" [6]. Of the remaining papers, only [8] models a form of fairness, giving an acceptance tree model for a language of asynchronous processes (without local state) and achieving full abstraction with respect to a notion of fair testing. The notion of fairness considered there is rather limited: roughly speaking, an infinite parallel computation is fair iff both of its projected computations are infinite and fair. The fair trace semantics for communicating processes developed in [4] uses a form of simple trace and does not account for the possibility of deadlock. We regard our work as putting these earlier models into a wider perspective, showing how to incorporate state, fairness, synchronization and deadlock in a single framework.

Our work achieves somewhat similar net results to [3], where full abstraction was achieved for a fair shared-variable parallel programming language. As remarked earlier, the trace models used there do not simply adapt to the setting of communication-based languages. The two approaches seem to be orthogonal, however, and we plan to develop a hybrid form of trace semantics which combines the two approaches to provide a model for a language allowing processes both to communicate by message-passing and to share memory.

Our semantics incorporates an assumption of strong process fairness. We expect that our approach can be adapted to incorporate other notions of (strong) fairness, such as channel fairness or communication fairness [5]. We also plan to develop a semantics incorporating weak fairness, i.e. every process

which is continuously enabled will eventually proceed.

References

[1] K. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer Verlag, New York, 1991.

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

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

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

[5] N. Francez. *Fairness.* Springer-Verlag, New York, 1986.

[6] M. Hennessy and R. DeNicola. Testing equivalences for processes. *Theoretical Computer Science*, 34, 1984.

[7] M. Hennessy. Acceptance trees. *JACM*, 32(4), 1985.

[8] M. Hennessy. An algebraic theory of fair asynchronous communicating processes. *Theoretical Computer Science*, 49, 1987.

[9] M. Hennessy and A. Ingólfsdóttir. Communicating processes with value-passing and assignments. *Formal Aspects of Computing*, 3, 1993.

[10] C.A.R. Hoare. Communicating Sequential Processes. *CACM*, 21(8):666–677, August 1978.

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

[12] L. Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress.* IFIP, North Holland, September 1983.

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

[14] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[15] D. Park. On the semantics of fair parallelism. In D. Bjørner, editor, *Abstract Software Specifications*, number 86 in Lecture Notes in Computer Science, pages 504–526. Springer Verlag, 1980.

[16] G.D. Plotkin. An operational semantics for CSP. In D. Bjørner, editor, *Formal Description of Programming Concepts II*, pages 199–225. North-Holland, 1982.

[17] A.W. Roscoe. Denotational semantics for occam. In *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science.* Springer-Verlag, 1984.