We construct several denotational semantics for communicating processes that incorporate assumptions of strong (process) fairness. Strong fairness is the guarantee that every process enabled infinitely often will make progress infinitely often. Modeling fairness compositionally requires care: generally speaking, the fair computations of a command cannot be defined only in terms of the fair computations of its component commands. For this reason, we introduce the notion of parameterized fairness, which generalizes fairness sufficiently to admit a compositional characterization. In each of these semantics, a command's meaning is simply the set of fair traces representing its fair computations; each fair trace records the steps made along a computation as well as additional information made explicit by the definition of parameterized fairness. Each semantics obtains full abstraction with respect to a natural notion of strongly fair program behavior: two terms are given identical meanings precisely when they exhibit the same behaviors in all program contexts.

1. INTRODUCTION

The behavior of a parallel system depends not only on the properties of the individual components running in parallel but also on the interactions among those components. These interactions in turn depend on external factors (such as the relative speed of processors or the particular scheduler implementation) whose details can be complex or even unknown. By introducing appropriate fairness assumptions—roughly speaking, assurances that every sufficiently enabled component eventually proceeds—we can abstract away from these details without ignoring them completely. For this reason, fairness assumptions are often essential for reasoning about liveness properties of programs.
Unfortunately, the simplicity of fairness’s underlying theme belies the complexity of reasoning formally about fairness. The well-known relationship between fairness and unbounded nondeterminism has hampered both operational and denotational accounts of fairness, requiring the use of transfinite ordinals for proof rules and the use of noncontinuous semantic operators (Park, 1979; Apt and Plotkin, 1986). Brookes’ transition traces for a shared-variable language (Brookes, 1996b) evade these difficulties and provide a fully abstract model that incorporates fairness assumptions. In the same spirit, this paper presents fully abstract, trace-based semantics for strongly fair communicating processes.

As with the transition traces, the resulting semantics combine the advantages of both denotational and operational semantics: the use of traces permits formal, compositional reasoning that can be guided by operational intuition. However, the construction of fully abstract semantics requires significantly more care for communicating processes. In particular, it is significantly easier to reason about the “enabledness” of a process in the shared-variable setting: whether a given process is enabled depends only on the current state. In contrast, communication in the (synchronous) message-passing setting requires the active cooperation of at least two processes: whether a given process is enabled may depend on the ability of other processes to synchronize with it. As a result, a given transition sequence of a process may be both fair and unfair, depending on the execution of the processes running in parallel with it.

This contextual dependency complicates the task of providing a compositional characterization of fair computations: the fair computations of a command cannot be determined by considering only the fair computations of its component commands. The solution proposed in this paper is to introduce a generalized form of fairness—called parameterized fairness—that accounts for “almost fair” computations (that is, unfair computations that may contribute to fair computations). Intuitively, certain unfair computations can be tagged with information that indicates the conditions under which they contribute to fair computations. The definition of parameterized fairness makes explicit the semantic structure necessary to support compositional reasoning about strongly fair computations.

The semantics we construct are based on fair traces, which record the steps made along a computation as well as the additional fairness-related information prescribed by parameterized fairness. The meaning of a command is simply the set of fair traces corresponding to its fair computations, and we can give both operational and denotational characterizations of the semantics. These semantics can be viewed as extending the CSP failures model (Brookes et al., 1984) or acceptance trees (Hennessy, 1985) to support reasoning about infinite, fair computations. The precise structure of the fair traces depends on the particular notion of program behavior under consideration; in each case, however, the fairness-related information remains the same. By introducing appropriate closure conditions on trace sets, we achieve full abstraction with respect to natural notions of fair program behavior. The property of full abstraction provides an objective criterion for judging the utility of a semantics: a fully abstract semantics makes precisely the right distinctions for reasoning about a given notion of program behavior, identifying exactly those terms that induce the same behaviors in all program contexts.
2. IMPERATIVE COMMUNICATING PROCESSES

For the duration of this paper, we consider a simple language of communicating processes, originally introduced in (Brookes, 1994) and based on CSP (Hoare, 1978) and CCS (Milner, 1980). As in occam (INMOS Limited, 1984), processes have disjoint local states and communicate with one another via named channels; however, we impose no constraints on the number of processes that may use a given channel, nor do we prohibit a process from using a channel for both input and output.

2.1. Syntax

The abstract syntax of the language relies on the following seven syntactic domains: Ide, the set of identifiers, ranged over by \( i \); BExp, the set of boolean expressions, ranged over by \( b \); Exp, the set of (integer) arithmetic expressions, ranged over by \( e \); Chan, the set of channel names, ranged over by \( h \); Gua, the set of communication guards, ranged over by \( g \); Gcom, the set of guarded commands, ranged over by \( gc \); and Com, the set of commands, ranged over by \( c \).

We take for granted the syntax of identifiers, channel names, and boolean and arithmetic expressions. The syntax of guards, guarded commands, and commands is given by the following grammar:

\[
g ::= h?:i | h!e
\]

\[
gc ::= g \rightarrow e | gc_1 | gc_2
\]

\[
c ::= \text{skip} | i := e | c_1 ; c_2 | \text{if } b \text{ then } c_1 \text{ else } c_2 | \text{while } b \text{ do } c | gc | c_1 || c_2 | c^1 \parallel h
\]

In examples, as is conventional, we often use the abbreviation \( g \) for the guarded command \( g \rightarrow \text{skip} \).

We also impose the syntactic constraint that, for all commands of form \( c_1 || c_2 \), \( c_1 \) and \( c_2 \) have disjoint free identifiers. This constraint ensures that processes can alter one another’s local states only as the result of handshake communications.

2.2. Operational Semantics

A state is a finite partial function from identifiers to integers, and we define the set \( S \) of states as \( S = \{ \text{Ide} \rightarrow \mathbb{Z} \} \). For any state \( s \), \([s : i = n]\) is the state that agrees with \( s \) except that it assigns value \( n \) to identifier \( i \). The domain of a state \( s \), written \( \text{dom}(s) \), is the set of identifiers for which \( s \) has a value. Two states \( s_1 \) and \( s_2 \) are considered disjoint when their domains are disjoint: \( \text{dom}(s_1) \cap \text{dom}(s_2) = \emptyset \). In such cases, we write \( \text{disjoint}(s_1, s_2) \).

For simplicity, we assume that an evaluation semantics is given for arithmetic and boolean expressions, and that expression evaluation always terminates and produces no side effects. We write \( \langle e, s \rangle \rightarrow^{*} n \) to indicate that expression \( e \) in state \( s \) evaluates to value \( n \). We use a similar notation for the evaluation of boolean expressions, letting \( \mathbb{B} = \{ \text{tt}, \text{ff} \} \) represent the set of truth values.
We use a labeled transition system for commands, guards, and guarded commands; this approach is standard and follows that of (Plotkin, 1983). A configuration is a pair \( \langle c, s \rangle \) (or, more generally, \( \langle g, s \rangle \) or \( \langle gc, s \rangle \)) for which state \( s \) is defined on at least the free identifiers of \( c \) (or \( g \) or \( gc \)). We introduce the placeholder • to represent termination, allowing configurations with forms such as \( \langle •, s \rangle \), \( \langle • \parallel c_2, s \rangle \), and \( \langle • \setminus h, s \rangle \). A configuration \( \langle c, s \rangle \) is terminal if there is a label \( \lambda \) such that \( \langle c, s \rangle \xrightarrow{\lambda} \) can be proved from the axioms and inference rules in Fig. 1.

A label \( \lambda \) is a member of the set \( \mathcal{A} = \{ e \} \cup \{ h!n, h?n \mid h \in \text{Chan} \land n \in \mathbb{Z} \} \). Every transition has a label indicating the type of atomic action involved: \( e \) represents an internal action (e.g., assignment to a variable), \( h!n \) represents the transmission of the value \( n \) along channel \( h \), and \( h?n \) represents the receipt of value \( n \) from channel \( h \).

Two labels \( \lambda_1 \) and \( \lambda_2 \) match if and only if one has the form \( h!n \) and the other \( h?n \) for some channel \( h \) and value \( n \); in such cases, we write \( \text{match}(\lambda_1, \lambda_2) \). For a label \( \lambda \), \text{chan}(\lambda) \) is the channel associated with \( \lambda \); by convention, we define \( \text{chan}(e) = e \).

We write \( \langle c, s \rangle \xrightarrow{\lambda} \langle c', s' \rangle \) to indicate that the command \( c \) in state \( s \) can perform a transition labeled \( \lambda \), leading to the command \( c' \) in state \( s' \). The transition relations \( \xrightarrow{\lambda} (\lambda \in \mathcal{A}) \) are characterized by the axioms and inference rules given in Fig. 2 and Fig. 3.

A direction \( d \) is a member of the set \( \mathcal{A} = \{ h! \}, h? \mid h \in \text{Chan} \} \); we occasionally use the extended set of directions \( \mathcal{A}^* = \mathcal{A} \cup \{ \epsilon \} \). For a label \( \lambda \), \text{dir}(\lambda) \) is the direction associated with \( \lambda \); \( \text{dir}(h!n) = h! \), \( \text{dir}(h?n) = h? \), and (by convention) \( \text{dir}(e) = \epsilon \). Two directions match if and only if one has form \( h! \) and the other \( h? \) for some channel \( h \). We often write \( d \) for the unique direction that matches \( d \), and we write \( \hat{X} \) for the set of directions matching those in \( X \): \( \hat{X} = \{ d \mid d \in X \} \).

The set \( \text{inits}(c, s) \) contains the directions (possibly including \( e \)) that can be used on transitions from the configuration \( \langle c, s \rangle \): \( \text{inits}(c, s) = \{ \text{dir}(\lambda) \mid \exists c', s', c, s \xrightarrow{\lambda} \langle c', s' \rangle \} \). A configuration \( \langle c, s \rangle \) is enabled if \( \text{inits}(c, s) \) is nonempty. A configuration is blocked (or disabled) if it is neither enabled nor terminal; we write \( \langle c, s \rangle \xrightarrow{\epsilon} \) to indicate that the configuration \( \langle c, s \rangle \) is blocked. A computation is a

\[
\begin{align*}
\langle \text{skip}, s \rangle & \xrightarrow{\epsilon} \langle \bullet, s \rangle \\
\langle i := e, s \rangle & \xrightarrow{\epsilon} \langle \bullet, [s \mid i = n] \rangle \\
\langle c_1, s \rangle & \xrightarrow{\epsilon} \langle c_1', s' \rangle \\
\langle c_1; c_2, s \rangle & \xrightarrow{\epsilon} \langle c_1', c_2', s' \rangle \\
\langle b, s \rangle & \xrightarrow{\epsilon} \langle \text{tt} \rangle \\
\langle \text{if } b \text{ then } c_1 \text{ else } c_2, s \rangle & \xrightarrow{\epsilon} \langle \text{if } b \text{ then } c_1, c_2, s \rangle \\
\langle b, s \rangle & \xrightarrow{\epsilon} \langle \text{tt} \rangle \\
\langle \text{while } b \text{ do } c, s \rangle & \xrightarrow{\epsilon} \langle \bullet, s \rangle
\end{align*}
\]

FIG. 2. Transition rules for sequential constructs.
deadlocked configuration ending in a terminal configuration. We call a finite computation ending in a nonterminal configuration successful. A computation is strongly fair if every process that is enabled in some configuration of a computation, en\(\leftarrow\)d along that computation, contributes to some transition of a computation.

A direction \(d\) is enabled in a configuration \(\langle c, s \rangle\) if \(d\) \(\in\) inits\(\langle c, s \rangle\), and \(d\) is enabled along a computation \(p\) if it is enabled in some configuration of \(p\). An enabled direction is used along \(p\) if it contributes to some transition of \(p\). When \(p\) is a successful computation, en\(\leftarrow\)d\(\langle p \rangle\) and used\(\langle p \rangle\) are the sets of directions enabled and used (respectively) along \(p\). When \(p\) is an infinite computation, en\(\leftarrow\)d\(\langle p \rangle\) and used\(\langle p \rangle\) contain the directions enabled infinitely often and used infinitely often along \(p\).

2.2. Strong Fairness

Many notions of fairness have been considered for communicating processes, including weak and strong forms of process fairness, channel fairness, and communication fairness (Kuiper and de Roever, 1983). Of these, strong (process) fairness is often considered the most appropriate, because it alone is equivalence robust (Apt et al., 1988): the order in which independent actions occur does not affect the perceived fairness of a computation.

Informally, a computation is strongly fair if every process that is enabled infinitely often makes progress infinitely often. However, this intuition leaves unstated what a process is, as well as what it means for a process to be enabled or
to make progress. We therefore start out with some definitions that formalize the notion of strong fairness for our language.

Let $c$ and $q$ be commands, and let $\xi \in \{1, 2\}^*$ (where we use 0 as the empty string). The predicate $\text{sub}(q, c, \xi)$ is true if and only if it can be proven by the axioms and inference rules in Fig. 4. Intuitively, $\text{sub}(q, c, \xi)$ is true if $q$ is a subcomponent of $c$ with address $\xi$ and can (in terms of position if not enabledness) contribute to the next transition of $c$. The immediate subcomponents of a command $c$ are given by the set $\text{inits}(c) = \{ (q, \xi) \mid \text{sub}(q, c, \xi) \}$. Every transition of form $\langle c, s \rangle \xrightarrow{\delta} \langle c', s' \rangle$ can be further decorated with a set $A \subseteq \{1, 2\}^*$ indicating the addresses of the subcomponents involved in that transition; for example, the decorated transition $\langle c, s \rangle \xrightarrow{\delta} \{1, 2\} \langle c', s' \rangle$ indicates that $c$'s subcomponents with addresses 1 and 22 synchronized with one another, leading to configuration $\langle c', s' \rangle$. The set $\text{inits}(X, c, s)$ gives the subset of $c$'s actions to which $X = (q, \xi)$ contributes: $\text{inits}(X, c, s) = \{ \text{dir}(h) \mid \exists A, \langle c', s', c, s \rangle \xrightarrow{\delta} \langle c', s' \rangle \land (c', s') \in A \}$. An immediate subcomponent $X \in \text{sub}(c)$ is enabled in $\langle c, s \rangle$ if $\text{inits}(X, c, s)$ is nonempty, and blocked in $\langle c, s \rangle$ otherwise.

We give a precise definition of strong fairness as follows.

**Definition 2.1.** A computation is **strongly fair** (or simply **fair**) if it is finite or if it is an infinite computation

$$\rho = \langle c_0, s_0 \rangle \xrightarrow{\delta_0} \langle c_1, s_1 \rangle \xrightarrow{\delta_1} \cdots \xrightarrow{\delta_{k-1}} \langle c_k, s_k \rangle \xrightarrow{\delta_k} \cdots,$$

such that, for all indices $i$ and immediate subcomponents $X$ of $c_i$, either $X$ eventually contributes to a transition of $\rho$ or $X$ is blocked almost everywhere along $\rho$. That is, $\rho$ is fair if the following condition holds: $\forall i. \forall X \in \text{sub}(c_i), \exists k > i. (X \notin \text{sub}(c_k) \lor \forall \ell > k. \text{inits}(X, c_\ell, s_\ell) = \emptyset)$. The clause $X \notin \text{sub}(c_k)$ is intended to indicate that the subcomponent $X$ has finally contributed to a transition. The following lemma confirms that an unused subcomponent never “disappears.”

**Lemma 2.2.** Assume that $X = (q, \xi) \in \text{sub}(c)$. If $c$ has a transition $\langle c, s \rangle \xrightarrow{\delta} \langle c', s' \rangle$ that does not involve $X$, then $X \in \text{sub}(c')$ as well.

**Proof.** This is proved by a simple induction on the structure of $c$. 

Note that the converse of this lemma does not hold: for example, $(\text{skip}, 0)$ is an immediate subcomponent of both the initial and final commands in the transition $\langle \text{skip}, \text{skip}, s \rangle \xrightarrow{\delta} \langle \text{skip}, s \rangle$. However, only finitely many such transitions can occur before the subcomponent $X$ disappears.
3. PARAMETERIZED STRONG FAIRNESS

Whether a given process is enabled (and therefore whether it is treated fairly) depends upon the context in which it appears. This contextual dependency has important consequences for characterizing fair computations compositionally. For example, consider the strongly fair computations of the program

\[ C = (C_1 \parallel (C_2 \parallel C_3)) \setminus \{a\} \backslash b, \]

where we define \( C_1 \equiv \text{while true do } a?x, \) \( C_2 \equiv \text{while true do } a!0, \) and \( C_3 \equiv \text{while true do } b!0 \rightarrow a!0. \) In the fair computations of \( C, \) the components \( C_1 \) and \( C_2 \) repeatedly synchronize with one another along channel \( a, \) while \( C_3 \) waits for the (never occurring) opportunity to communicate on channel \( b. \) However, these fair computations of \( C \) cannot be defined solely in terms of the fair computations of its components \( C_1, \) \( C_2, \) and \( C_3; \) in every strongly fair computation of \( C_3 \) (and of \( C_2 \parallel C_3 \)), \( C_3 \) makes infinitely many transitions. The problem is that \( C \) permits channel \( b \) to be used only for synchronization, whereas \( C_3 \) allows the unrestricted use of channel \( b. \) Merely knowing the channels on which \( C \) requires synchronization, however, is insufficient for identifying which subcommands are enabled along a given computation of \( C \): communication is also restricted on channel \( a \) in the command \( C, \) but \( C_2 \) can make continual progress by synchronizing with \( C_1 \) infinitely often.

The key to characterizing fair computations compositionally is to introduce a generalized notion of fairness, parameterized by sets that represent conditions under which certain unfair computations may contribute to truly fair computations. For every finite set \( F \) of directions, we characterize those computations that are strongly fair modulo \( F. \) Roughly speaking, a computation \( \rho \) of the command \( c \) is strongly fair modulo \( F \) if every infinitely enabled process either (1) makes progress infinitely often (just as in traditional strong fairness) or (2) eventually stops in a configuration in which its only possible transitions are labeled by directions in \( F \) and \( it \) cannot synchronize with any other processes. Intuitively, \( \rho \) represents \( c \)'s contribution to a strongly fair computation of \( P[c], \) where \( P[\_] \) is some program context that both restricts communication on the channels in \( F \) and fails to provide sufficient synchronization opportunities for members of \( F. \) For example, the (unfair) infinite computation of \( C_2 \parallel C_3 \) that never performs output along channel \( b \) can be characterized as fair modulo \( \{b!\} \): the context \((C_1 \parallel \_)/a \backslash b \) restricts communication on channel \( b \) and provides no synchronization opportunities for \( C_3 \)'s \( b!0 \) action. We formalize this intuition with the following definition.

**Definition 3.1.** Let \( F \) be a finite set of directions. A (possibly partial) computation

\[ \rho = \langle c_0, s_0 \rangle \xrightarrow{a} \langle c_1, s_1 \rangle \xrightarrow{a} \ldots \xrightarrow{a} \langle c_k, s_k \rangle \xrightarrow{a} \ldots \]

is strongly fair modulo \( F \) (or fair mod \( F \)) if \( \rho \) is a successfully terminating computation, if \( \rho \) is a partial computation with a final configuration \( \langle c, s \rangle \) such that
The condition \( Y \cap \text{used}(\rho) = \emptyset \) prohibits processes from using a direction infinitely often when other processes are blocking on that same direction. Under fairness, no program context can provide one process infinitely many opportunities to use a direction \( d \) without providing those opportunities to all processes trying to use \( d \).

Two properties should be immediately clear from this definition. First, a computation that is strongly fair mod \( F \) is also fair mod \( F' \) for all \( F \subseteq F' \). Second, a computation is strongly fair if and only if it is strongly fair modulo \( \emptyset \). Unlike traditional formulations of strong fairness (Francez, 1986), however, parameterized strong fairness can be characterized compositionally, as shown by the following proposition.

**Proposition 3.2.** Let \( F \) be a finite set of directions. A computation \( \rho \) of command \( c \) is strongly fair modulo \( F \) if and only if \( \rho \) satisfies one of the following conditions:

- \( \rho \) is a finite, successfully terminating computation;
- \( \rho \) is a partial computation with a final configuration \( \langle c, s \rangle \) such that \( \text{inits}(c, s) \subseteq F \);
- \( \rho \) is an infinite computation, \( c \) has form \( (c_1; c_2) \) or \( (\text{if } b \text{ then } c_1 \text{ else } c_2) \), and the underlying infinite computation of \( c_1 \) or \( c_2 \) is fair mod \( F \);
- \( \rho \) is an infinite computation, \( c \) has form \( (\text{while } b \text{ do } c') \) or \( (g c') \), and each of the underlying computations of \( c' \) is fair mod \( F \);
- \( \rho \) is an infinite computation, \( c \) has form \( (g c_1 \square g c_2) \), and the underlying computation of the selected \( gc_i \) is fair mod \( F \);
- \( \rho \) is an infinite computation, \( c \) has form \( c' \setminus h \), and the underlying computation of \( c' \) is fair modulo \( F \cup \{h!, h?\} \);
- \( \rho \) is an infinite computation, \( c \) has form \( c_1 \parallel c_2 \), and there exist sets \( F_1 \) and \( F_2 \) and computations \( \rho_1 \) of \( c_1 \) and \( \rho_2 \) of \( c_2 \) such that \( \rho_1 \) is fair mod \( F_1 \), \( \rho_2 \) is fair mod \( F_2 \), \( F \supseteq F_1 \cup F_2 \), \( \rho \) can be obtained by merging and synchronizing \( \rho_1 \) and \( \rho_2 \), neither \( \rho_1 \) enables infinitely often any direction matching a member of \( F_3 \), and neither \( \rho_1 \) uses a direction in \( F_3 \) infinitely often.

**Proof.** The cases for finite and partial computations are immediate. For infinite computations, the proof is based on a straightforward induction on the structure of the command \( c \).
4. FAIR TRACES

We begin the construction of our semantics with a set of steps $\Sigma = S \times A \times S$: intuitively, each step $(s, \lambda, s')$ corresponds to a transition of form $\langle e, s \rangle \xrightarrow{\lambda} \langle e', s' \rangle$. A (simple) trace is a finite or infinite sequence of steps representing a sequence of uninterrupted transitions. The set $\Sigma^0 = \{e_s | s \in S\}$ of empty traces provides a collection of local units for concatenation, and the set of simple traces is $\Sigma^\omega = \Sigma^* \cup \Sigma^\omega$, where $\Sigma^*$ and $\Sigma^\omega$ are the sets of finite and infinite traces, respectively:

$$\Sigma^* = \Sigma^0 \cup \{(s_0, \lambda_0, s_1)(s_1, \lambda_1, s_2) \cdots (s_k, \lambda_k, s_{k+1}) | k \geq 0 \& \forall i \leq k. (s_i, \lambda_i, s_{i+1}) \in \Sigma\},$$

$$\Sigma^\omega = \{(s_0, \lambda_0, s_1)(s_1, \lambda_1, s_2) \cdots (s_k, \lambda_k, s_{k+1}) \cdots | \forall i \geq 0. (s_i, \lambda_i, s_{i+1}) \in \Sigma\}.$$

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

$$\langle e, s_0 \rangle \xrightarrow{\lambda_0} \langle e_1, s_1 \rangle \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_k} \langle e_{k+1}, s_{k+1} \rangle \text{ term},$$

then $\text{trace}(\rho) = (s, \lambda_0, s_1)(s_1, \lambda_1, s_2) \cdots (s_k, \lambda_k, s_{k+1})$.

These simple traces are clearly insufficient for reasoning about fairness: they record only the events that occurred along a computation, providing no information about events that could have occurred but did not. Guided by the definition of parameterized fairness, we augment simple traces with additional fairness-related contextual information, yielding the set $\Phi$ of strongly fair traces:

$$\Phi = \Sigma^\omega \times (\mathcal{P}(A^+) \times \mathcal{P}(A^+) \times \{i, f, p\}).$$

The tags $\{i, f, p\}$ are included to aid readability by indicating the type of computation (i.e., infinite, finite, or partial) being represented. Intuitively, the trace $\langle x, F, E, i \rangle$ represents an infinite, strongly fair mod $F$ computation with simple trace $x$ and set $E$ of infinitely enabled directions. Similarly, the trace $\langle x, (F, E, f) \rangle$ represents a successfully terminating (and necessarily fair mod $F$) computation having simple trace $x$ and enabled directions $E$. Finally, the trace $\langle x, (F, E, p) \rangle$ (with $F \supseteq E$) represents a partial computation whose final configuration has the set $E$ of directions (possibly including $z$) enabled; when $e \notin E$, the partial computation is necessarily fair mod $E$ (and therefore fair mod $F$). These partial traces support reasoning about deadlock and blocking and are the obvious analogues of acceptances (Hennessy, 1985) and failures (Brookes et al., 1984). Technically, the fairness sets $F$ are unnecessary for finite and partial traces; however, their inclusion allows a uniform trace structure that facilitates certain semantic definitions. Likewise, we allow $e$ to appear in a partial trace’s fairness set $F$, maintaining the invariant that $E \subseteq F$ for all partial traces; in contrast, the fairness sets for finite and infinite traces will never contain $e$. 


We then give an operational characterization of a strongly fair trace semantics $T$: \[ \text{Com}(\Phi) \]

\[ T_c = \{ \langle \text{trace}(\rho), (F, \text{en}(\rho), \xi), s_0 \rangle | F \in \mathcal{P}(A) \ \& \ \rho = \langle c, s_0 \rangle \xrightarrow{\Delta c} \langle c_1, s_1 \rangle \xrightarrow{\Delta c} \cdots \xrightarrow{\Delta c} \langle c_k, s_k \rangle \text{ term} \} \]

\[ \cup \{ \langle \text{trace}(\rho), (F, E, \xi) \rangle | E = \text{inits}(c_k, s_k) \ \& \ F \supseteq E \ \& \rho = \langle c, s_0 \rangle \xrightarrow{\Delta c} \langle c_1, s_1 \rangle \xrightarrow{\Delta c} \cdots \xrightarrow{\Delta c} \langle c_k, s_k \rangle \ \& \ \neg \langle c_k, s_k \rangle \text{ term} \} \]

\[ \cup \{ \langle \text{trace}(\rho), (F, \text{en}(\rho), \xi) \rangle | \rho = \langle c, s_0 \rangle \xrightarrow{\Delta c} \langle c_1, s_1 \rangle \xrightarrow{\Delta c} \cdots \xrightarrow{\Delta c} \text{ is strongly fair mod } F \}. \]

That is, $T[c]$ is the set of traces corresponding to the strongly fair (modulo appropriate sets $F$) computations of $c$.

5. DENOTATIONAL SEMANTICS

The trace semantics $T$ can also be characterized denotationally: for each construct of the language, we define a corresponding operation on trace sets that reflects its operational behavior. These operations perform bookkeeping operations on the contextual components of the traces in addition to the obvious manipulations on the simple-trace components.

We assume semantic functions $B: \text{BExp} \rightarrow \mathcal{P}(S \times B)$ and $E: \text{Exp} \rightarrow \mathcal{P}(S \times Z)$ characterized operationally by

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

We also introduce a semantic function $T: \text{BExp} \rightarrow \mathcal{P}(\Phi)$ such that

\[ T[b] = \{ \langle (s, e, s), (F, \emptyset, \xi) \rangle, \langle e, (F \cup \{ e \}, \emptyset) \rangle \ |

(\forall s, \forall t) \in B[b] \ \& \ F \in \mathbb{P}(A) \}. \]

Intuitively, $T[b]$ contains the idle steps possible from states satisfying the boolean expression $b$.

Based on the operational characterization of $T$, it should be easy to see that

\[ T[\text{skip}] = \{ \langle (s, e, s), (F, \emptyset, \xi) \rangle | s \in S \ \& \ F \in \mathbb{P}(A) \}

\cup \{ \langle e, (F, \{ e \}, \emptyset) \rangle | s \in S \ \& \ F \supseteq \{ e \} \}, \]

\[ T[i := e] = \{ \langle (s, e, [s | i = n]), (F, \emptyset, \xi) \rangle | f e[i := e] \subseteq \text{dom}(s) \ \& \ F \in \mathbb{P}(A) \ \& \ (s, n) \in E[e] \}

\cup \{ \langle e, (F, \{ e \}, \emptyset) \rangle | f e[i := e] \subseteq \text{dom}(s) \ \& \ F \supseteq \{ e \} \}. \]
Similarly, for guards we obtain

\[ \mathcal{F}[h?] = \{ \langle (s, h?n, [s \mid i = n]), (F, \{ h? \}), \bar{\tau} \rangle \mid i \in \text{dom}(s) \& n \in \mathbb{Z} \& F \in \mathcal{P}_{a}(A) \} \]

\[ \cup \{ \langle s, (F, \{ h? \}), p \rangle \mid i \in \text{dom}(s) \& F \ni \{ h? \} \}, \]

\[ \mathcal{F}[h!e] = \{ \langle (s, h!n, s), (F, \{ h! \}), \bar{\tau} \rangle \mid (s, n) \in \mathcal{E}[e] \& F \in \mathcal{P}_{a}(A) \} \]

\[ \cup \{ \langle s, (F, \{ h! \}), p \rangle \mid fr[e] \subseteq \text{dom}(s) \& F \ni \{ h! \} \} . \]

Two fair traces \( \varphi_1 \) and \( \varphi_2 \) are composable—written \( \text{composable}(\varphi_1, \varphi_2) \)—if \( \varphi_1 \) is an infinite or partial trace or if the final state of \( \varphi_1 \) is the initial state of \( \varphi_2 \). When \( \varphi_1 = \langle x, (F_1, E_1, R_1) \rangle \) and \( \varphi_2 = \langle \beta, (F_2, E_2, R_2) \rangle \) are composable fair traces, their concatenation \( \varphi_1 \varphi_2 \) is defined by

\[ \varphi_1 \varphi_2 = \begin{cases} \varphi_1, & \text{if } R_1 \in \{ i, p \}, \\ \langle x\beta, (F_2, E_1 \cup E_2, \bar{\tau}) \rangle, & \text{if } R_1 = R_2 = \bar{\tau}, \\ \langle x\beta, (F_2, E_2, R_2) \rangle, & \text{if } R_1 = \bar{\tau} \text{ and } R_2 \in \{ i, p \} . \end{cases} \]

When \( \varphi_1 \) represents an infinite or partial computation, \( \varphi_2 \)'s contextual information is irrelevant: the computation represented by \( \varphi_2 \) never begins, because the computation represented by \( \varphi_1 \) does not terminate. We define sequential composition on trace sets \( T_1 \) and \( T_2 \) by

\[ T_1; T_2 = \{ \varphi_1 \varphi_2 \mid \varphi_1 \in T_1 \& \varphi_2 \in T_2 \& \text{composable}(\varphi_1, \varphi_2) \} , \]

so that

\[ \mathcal{F}[c_1; c_2] = \mathcal{F}[c_1]; \mathcal{F}[c_2] , \]

\[ \mathcal{F}[g \to c] = \mathcal{F}[g]; \mathcal{F}[c] , \]

\[ \mathcal{F}[\text{if } b \text{ then } c_1 \text{ else } c_2] = \mathcal{F}[b]; \mathcal{F}[c_1] \cup \mathcal{F}[\neg b]; \mathcal{F}[c_2] . \]

The semantics of loops relies on notions of iteration on trace sets, which in turn require some auxiliary notation. We denote the infinite sequence of fair traces \( \varphi_0, \varphi_1, \ldots, \varphi_n, \ldots \) by \( \langle \varphi_i \mid i \geq 0 \rangle \). For a collection \( \{ X_i \mid i \geq 0 \} \) of finite sets, \( \bigcup_{i=0}^{\infty} X_i \) is the set of elements that appear in infinitely many sets \( X_i \): \( \bigcup_{i=0}^{\infty} X_i = \{ d \mid \exists k \geq 0.3k > j.d \in X_k \} \). The sequence \( \langle \varphi_i = \langle x_0, (F_i, E_i, R_i) \rangle \mid i \geq 0 \rangle \) is composable if the sets \( \bigcup_{i=0}^{\infty} F_i \) and \( \bigcup_{i=0}^{\infty} E_i \) are finite and (for each \( i \)) the traces \( \varphi_0 \varphi_1 \ldots \varphi_{i-1} \) and \( \varphi_i \) are composable; we write \( \text{composable}(\langle \varphi_i \mid i \geq 0 \rangle) \) in such cases. We then define infinite concatenation as

\[ \varphi_0 \varphi_1 \varphi_2 \ldots = \begin{cases} \langle x_0 x_1 \cdots x_n \cdots, \left( \bigcup_{i=0}^{\infty} F_i, \bigcup_{i=0}^{\infty} E_i, \bar{\tau} \right) \rangle, & \text{if } \forall i. R_i = \bar{\tau}, \\ \langle x_0 x_1 \cdots x_k, (F_k, E_k, R_k) \rangle, & \text{if } \forall i < k. R_i = \bar{\tau} \text{ and } R_k \in \{ i, p \} . \end{cases} \]
When each \( \varphi_i \) is finite, the infinitely enabled directions of the resulting trace are those directions that appear in infinitely many of the sets \( E_i \).

Iteration on trace sets then follows directly from the definitions of concatenation and sequential composition. Finite and infinite iteration on the trace set \( T \) are defined by \( T^* \) and \( T^n \), respectively, as follows, where \( T^0 = \{ \langle s, (\emptyset, \emptyset, \varepsilon) \rangle \mid s \in S \} \) and \( T^{n+1} = T^n \cdot T \):

\[
T^* = \bigcup_{i=0}^{\infty} T^i
\]

\[
T^n = \{ \varphi_0 \varphi_1 \cdots \varphi_n \cdots \mid (\forall i \geq 0. \varphi_i \in T) \land \text{composable}(\langle \varphi_i \mid i \geq 0 \rangle) \}.
\]

Note that \( T^n \) is a fixed point of the functional \( F(X) = T; X \), but it is generally neither greatest nor least: the greatest fixed point contains traces with impossible enabling information (e.g., \( \langle (s, h!0, s)^n, (\emptyset, \emptyset, \varepsilon) \rangle \)), while the least fixed point is empty. We define the semantics of loops by

\[
F[\text{while } b \text{ do } c] = (F[b]; F[c])^\omega \cup (F[b]; F[c])^*; F[-b].
\]

A command \( gc_1 \square gc_2 \) represents a choice to be made on the first step between the guarded commands \( gc_1 \) and \( gc_2 \). Every fair trace \( \varphi \) that represents an infinite computation (or a partial computation involving at least one step) of \( gc_1 \) or \( gc_2 \) therefore also represents a computation of \( gc_1 \square gc_2 \). In contrast, every fair trace \( \varphi \) that represents a finite computation (or an initial partial computation) of \( gc_1 \) or \( gc_2 \) must be augmented with information about those directions that were enabled initially by the unselected \( gc_i \). This additional enabling information can be generated by looking at the "empty" partial traces of the appropriate \( gc_i \). We therefore define guarded choice on trace sets by

\[
T_1 \square T_2 = \{ \langle x, (F, E, i) \rangle \in T_1 \cup T_2 \mid x \in \Sigma^\omega \} \cup \{ \langle x, (F, E, p) \rangle \in T_1 \cup T_2 \mid x \in \Sigma^+ \} \\
\cup \{ \langle e_r, (F_1 \cup F_2, E_1 \cup E_2, p) \rangle \mid \langle e_r, (F_1, E_1, p) \rangle \in T_1 \land \langle e_r, (F_2, E_2, p) \rangle \in T_2 \} \\
\cup \{ \langle e_r, (F_1, E_1 \cup E_2, \varepsilon) \rangle \mid \langle e_r, (F_1, E_1, \varepsilon) \rangle \in T_1 \land \langle e_r, (F_2, E_2, \varepsilon) \rangle \in T_2 \land \varepsilon \notin E_2 \} \\
\cup \{ \langle e_r, (F_2, E_2 \cup E_1, \varepsilon) \rangle \mid \langle e_r, (F_2, E_2, \varepsilon) \rangle \in T_2 \land \langle e_r, (F_1, E_1, \varepsilon) \rangle \in T_1 \land \varepsilon \notin E_1 \},
\]

so that \( F[gc_1 \square gc_2] = F[gc_1] \square F[gc_2] \).

The computations of \( c \backslash h \) arise from the computations of \( c \) that do not use channel \( h \) for visible communications. Correspondingly, \( T \backslash h \) can be obtained from \( T \) by first removing those traces in which \( h \) is visible and then deleting \( h? \) and \( h! \) from the enabling and fairness sets of the remaining traces. For a trace \( \alpha \), \( \text{chans}(\alpha) \) is the
set of channels appearing along \(\pi\). For a set \(X\) of directions, \(X \setminus h\) is the set \(X\) with references to channel \(h\) removed: \(X \setminus h = X - \{h!, h?\}\). We thus define

\[
T \setminus h = \{ \langle \pi, (F', E \setminus h, R) \rangle | \langle \pi, (F, E, R) \rangle \in T \land F' \supseteq F \setminus h \land h \notin \text{chans}(\pi) \},
\]

so that \(T[c/h] = T[c] \setminus h\).

Defining fair parallel composition on trace sets requires care for two reasons: (1) only certain pairs of traces can be meaningfully combined, and (2) those pairs must be combined in a manner that “consumes” all of their steps. To handle the first constraint, we introduce a predicate \(\text{mergeable}(\varphi_1, \varphi_2)\) based directly on the compositional characterization of parameterized fairness given by Proposition 3.2. Letting \(\text{vis}(\pi)\) be the set of directions visible infinitely often along \(\pi\), we define the predicate \(\text{mergeable}(\varphi_1, \varphi_2)\) for fair traces \(\varphi_1 = \langle \pi_1, (F_1, E_1, R_1) \rangle\) and \(\varphi_2 = \langle \pi_2, (F_2, E_2, R_2) \rangle\) as

\[
\text{mergeable}(\varphi_1, \varphi_2) \iff (R_1 = \top) \text{ or } (R_2 = \top) \text{ or } (R_1 = R_2 = \bot) \text{ or } (E_1 \neq E_2) \text{ and } \varphi_1 \cap \varphi_2 = \emptyset \land F_1 \cap \text{vis}(\pi_1) = \emptyset.
\]

Note that it suffices to consider the infinitely often visible (rather than used) directions of each \(\pi_j\): whenever a direction \(d\) is used (but not visible) infinitely often, \(d\) must also be used (and therefore enabled) infinitely often.

To satisfy the second constraint, we define a ternary relation \(\text{fairmerge} \subseteq \Phi \times \Phi \times \Phi\) on fair traces, adapting Park’s \text{fairmerge} relation (Park, 1979) to account for the possibility of synchronization and to perform the necessary bookkeeping on the contextual components of the traces. The definition of \(\text{fairmerge}\) relies on two different sets of triples: both, whose triples represent finite sequences of transitions made while both components are active, and one, whose triples represent transition sequences made by one component after the other has terminated. Before defining these sets, we introduce some auxiliary definitions and operations.

Two simple traces \(\pi\) and \(\beta\) are disjoint—written \(\text{disjoint}(\pi, \beta)\)—if each state along \(\pi\) is disjoint from every state along \(\beta\); in particular, \(\pi\) and \(e_\pi\) are disjoint if \(s\) is disjoint from each state along \(\pi\). In such cases, \(\pi \parallel e_\pi\) is the trace obtained by propagating state \(s\) through \(\pi\): for example, if \(\pi = (s_0, \lambda_\pi, s_1)(s_1, \lambda_1, s_2) \cdots (s_k, \lambda_k, s_{k+1})\), then

\[
\pi \parallel e_\pi = (s_0 \cup s, \lambda_\pi, s_1 \cup s)(s_1 \cup s, \lambda_1, s_2 \cup s) \cdots (s_k \cup s, \lambda_k, s_{k+1} \cup s).
\]

More generally, for disjoint \(\pi\) and \(\beta\) we define \(\pi \parallel \beta = (\pi \parallel e_\pi)(\beta \parallel e_\beta)\), where \(s\) and \(t\) are the final state of \(\pi\) and initial state of \(\beta\), respectively. Intuitively, \(\pi \parallel \beta\) represents the transitions made by a parallel command if one component performs the transitions \(\pi\), followed by the other component performing the transitions \(\beta\).

Two nonempty, finite traces \(\pi = (s_0, \lambda_\pi, s_1) \cdots (s_k, \lambda_k, s_{k+1})\) and \(\beta = (t_0, \mu_\beta, t_1) \cdots (t_n, \mu_n, t_{n+1})\) match—written \(\text{match}(\pi, \beta)\)—if they have the same length and each step of \(\pi\) matches the corresponding step of \(\beta\) (that is, if \(k = n\) and
match(\(\lambda_i, \mu_i\)) for each \(i\). When \(\pi\) and \(\beta\) match, \(\pi \parallel \beta\) is the trace in which \(\pi\) and \(\beta\) synchronize at each step:

\[
\pi \parallel \beta = (s_0 \cup t_0, \varepsilon, s_1 \cup t_1) \cdots (s_k \cup t_k, \varepsilon, s_{k+1} \cup t_{k+1}).
\]

Similarly, the fair traces \(\varphi_1 = \langle \pi, (F_1, E_1, \varepsilon) \rangle\) and \(\varphi_2 = \langle \beta, (F_2, E_2, \varepsilon) \rangle\) match when their simple-trace components \(\pi\) and \(\beta\) match.

To extend these interleaving and merging operators to fair traces, we first introduce an operator \(\theta_1 \parallel \theta_2\), with the intuition that each \(\theta \in \theta_1 \parallel \theta_2\) (\(= \theta_2 \parallel \theta_1\)) provides valid contextual information for a computation that arises from merging computations with contextual triples \(\theta_1\) and \(\theta_2\):

\[
(F_1, E_1, \varepsilon) \parallel (F_2, E_2, \varepsilon) = \{(F, E_1 \cup E_2, \varepsilon) \mid F \supseteq F_1 \cup F_2\}
\]

\[
(F_1, E_1, \varphi) \parallel (F_2, E_2, \varphi) = \{(F, E_1 \cup E_2, \varphi) \mid F \supseteq F_1 \cup F_2\}
\]

\[
(F_1, E_2, \varphi) \parallel (F_2, E_2, \varphi) = \{(F, E_1 \cup E_2, \varphi) \mid F \supseteq F_1 \cup F_2\}
\]

\[
(F_1, E_1, \varepsilon) \parallel (F_2, E_2, \varepsilon) = \{(F, E_1 \cup E_2, \varepsilon) \mid F \supseteq F_1 \cup F_2\}
\]

\[
(F_1, E_1, \varphi) \parallel (F_2, E_2, \varepsilon) = \{(F, E_1 \cup E_2, \varepsilon) \mid F \supseteq F_1 \cup F_2\}
\]

\[
(F_1, E_1, \varepsilon) \parallel (F_2, E_2, \varphi) = \{(F, E_1 \cup E_2, \varphi) \mid F \supseteq F_1 \cup F_2\}
\]

\[
(F_1, E_1, \varphi) \parallel (F_2, E_2, \varphi) = \{(F, E_1 \cup E_2, \varphi) \mid F \supseteq F_1 \cup F_2\}.
\]

Note that no definition for \((F_1, E_1, \varepsilon) \parallel (F_2, E_2, \varepsilon)\) is necessary, because we shall never merge two infinite traces directly; rather, the definition of \(\text{fairmerge}\) will merge two infinite traces by merging finite portions of one with finite portions of another. For fair traces \(\varphi_1 = \langle \pi, \theta_1 \rangle\) and \(\varphi_2 = \langle \beta, \theta_2 \rangle\), such that \(\pi \parallel \beta\) or \(\pi \parallel \beta\) is defined, we then define \(\varphi_1 \parallel \varphi_2 = \{\langle \pi \parallel \beta, \theta \rangle \mid \theta \in \theta_1 \parallel \theta_2\}\) and \(\varphi_2 \parallel \varphi_1 = \{\langle \pi \parallel \beta, \theta \rangle \mid \theta \in \theta_1 \parallel \theta_2\}\).

We can now define the sets \(\text{both} \subseteq \Phi \times \Phi \times \Phi\) and \(\text{one} \subseteq \Phi \times \Phi \times \Phi\). The set \(\text{both}\) captures the intuition that, as long as both components remain active, neither component can be forever ignored; its triples reflect interleavings (or synchronizations) of finite portions of possibly infinite traces:

\[
\text{both} = \{(\varphi_1, \varphi_2, \varphi) \mid \varphi_1, \varphi_2 \in \Phi_{\text{fin}} \& \text{disjoint}(\varphi_1, \varphi_2) \& \varphi \in (\varphi_1 \parallel \varphi_2 \cup \varphi_2 \parallel \varphi_1)\}
\]

\[
\cup \{(\varphi_1, \varphi_2, \varphi) \mid \varphi_1, \varphi_2 \in \Phi_{\text{fin}} \& \text{disjoint}(\varphi_1, \varphi_2) \& \text{match}(\varphi_1, \varphi_2) \& \varphi \in \varphi_1 \parallel \varphi_2\}.
\]

The set \(\text{one}\) captures the intuition that, once one component terminates or becomes permanently blocked, the other component can proceed uninterrupted; its triples reflect the uninterrupted progress of one component while the other component idles:

\[
\text{one} = \{(\varphi_1, \varphi_2, \varphi), (\varphi_2, \varphi_1, \varphi) \mid \varphi_1 \in \Phi \& \varphi_2 = \langle \varepsilon, \theta_2 \rangle \& \text{disjoint}(\varphi_1, \varphi_2) \& \varphi \in \varphi_1 \parallel \varphi_2\}.
\]

We then define \(\text{fairmerge}\) to be the greatest fixed point of the functional

\[
F(Y) = \text{both} \cdot Y \cup \text{one},
\]
where \( Y_1 \cdot Y_2 \) is the obvious extension of trace concatenation to sets of trace triples:

\[
Y_1 \cdot Y_2 = \{ (\varphi_1, \varphi_1', \varphi_2, \varphi_2', \varphi_3, \varphi_3') \mid (\varphi_1, \varphi_2, \varphi_3) \in Y_1 \land (\varphi_1', \varphi_2', \varphi_3') \in Y_2 \\
\text{and composable}(\varphi_1, \varphi_1') \land \text{composable}(\varphi_2, \varphi_2') \land \text{composable}(\varphi_3, \varphi_3') \}
\]

Equivalently, fair merging is \( \text{both}^* \cup \text{both} \cdot \text{one} \), where \( Y^* \) and \( Y^\omega \) are the corresponding finite and infinite iterations on the set \( Y \).

Finally, we define fair parallel composition on trace sets as

\[
T_1 \parallel T_2 = \{ \varphi \mid \varphi, \varphi' \in T_1 \land \varphi, \varphi' \in T_2 \land \text{mergeable}(\varphi_1, \varphi_2) \land (\varphi_1, \varphi_2, \varphi) \in \text{fairmerge} \},
\]

so that \( \mathcal{F}[c_1 || c_2] = \mathcal{F}[c_1] || \mathcal{F}[c_2] \).

**Proposition 5.1.** The denotational and operational characterizations of the fair trace semantics \( \mathcal{F} \) coincide.

**Proof.** By a straightforward but tedious induction on the structure of commands.

Most of the details concern parallel composition and make precise the connection with the operational characterization of parameterized fairness given by Proposition 3.2.

---

6. FULL ABSTRACTION

A semantics is *sound* with respect to a given notion of behavior if, whenever it gives two terms the same meaning, those terms induce the same behaviors in all program contexts. A semantics is *fully abstract* (Milner, 1975) with respect to a notion of behavior if the converse also holds: two terms are given the same meaning if and only if they induce the same behaviors in all program contexts.

For communicating processes, there are several natural notions of behavior to consider. We focus first on the following state-trace behavior.

**Definition 6.1.** The state trace behavior \( \mathcal{M} : \text{Com} \rightarrow \mathcal{P}(S^* \cup S^\delta) \) is defined by

\[
\mathcal{M}[c] = \{ s_0, s_1, \ldots, s_k \mid \langle c, s_0 \rangle \xrightarrow{c_1, s_1} \cdots \xrightarrow{c_k, s_k} \text{term} \}
\]

\[
\text{or } \{ s_0, s_1, \ldots, s_k \delta \mid \langle c, s_0 \rangle \xrightarrow{c_1, s_1} \cdots \xrightarrow{c_k, s_k} \text{dead} \}
\]

\[
\text{or } \{ s_0, s_1, \ldots, s_k \delta \mid \langle c, s_0 \rangle \xrightarrow{c_1, s_1} \cdots \xrightarrow{c_k, s_k} \text{is fair} \},
\]

where \( S^* \delta = \{ s_0, s_1, \ldots, s_k \delta \mid s_0, s_1, \ldots, s_k \in S \} \).

The behavior \( \mathcal{M} \) incorporates the assumptions that a program is a closed system (that is, no external communication is permitted) and that an observer can detect each and every state change. It also reflects the assumption that deadlock is
distinguishable both from successful termination and from infinite chattering. In Section 7, we show that full abstraction is also possible for behaviors that relax one or more of these assumptions.

6.1. Closed Trace Sets

The semantics \( F \) is sound with respect to \( \mathcal{M} \): for all commands \( c \) and \( c' \),

\[
F[c] = F[c'] \Rightarrow \forall \mathcal{M}[P[c]] = \mathcal{M}[P[c']].
\]

However, \( F \) is not fully abstract with respect to \( \mathcal{M} \), because it distinguishes programs that behave identically in all contexts. For example, consider the commands

\[
C_1 \equiv (a!0 \rightarrow b!0) \square (a!0 \rightarrow c!0),
\]

\[
C_2 \equiv (a!0 \rightarrow b!0) \square (a!0 \rightarrow c!0) \square (a!0 \rightarrow (b!0 \square c!0)).
\]

The traces \( \langle s, a!0, s \rangle, \langle b!, c! \rangle, \langle b!, c! \rangle, \langle b!, c!, \rangle \rangle \) and \( \langle s, a!0, s, b!0, s \rangle, \langle \emptyset, \langle a!0, b!, c! \rangle, \rangle \rangle \) are both possible for \( C_2 \) but not for \( C_1 \). However, the two commands behave identically in all program contexts: after performing an \( a!0 \), each command may perform \( b!0 \) or \( c!0 \), and each command may refuse either one of these actions (but not both). That \( C_2 \) can enable each of \( b! \) and \( c! \) along the same computation is not directly observable: any behavior possible when both are enabled is also possible when only one of them is enabled.

A similar situation arises with the commands \( C_3 \) and \( C_4 \), defined as

\[
C_3 \equiv (a!0 \rightarrow b!0) \square (a!0 \rightarrow (b!0 \square c!0 \square d!0))
\]

\[
C_4 \equiv (a!0 \rightarrow b!0) \square (a!0 \rightarrow (b!0 \square c!0 \square d!0)) \square (a!0 \rightarrow (b!0 \square c!0 \square d!0)).
\]

The partial traces \( \langle s, a!0, s \rangle, \langle b!, \rangle, \langle b!, \rangle \rangle \) and \( \langle s, a!0, s \rangle, \langle b!, c!, d! \rangle, \langle b!, c!, d! \rangle \rangle \) are possible for both \( C_1 \) and \( C_4 \), whereas the partial trace \( \langle s, a!0, s \rangle, \langle b!, c!, \rangle, \langle b!, c!, \rangle \rangle \) is possible only for \( C_4 \). However, the two commands behave the same in all program contexts, because no context can detect the simultaneous enabling of \( c! \) and \( d! \).

Similar observations led to the introduction of closure conditions for acceptance trees (Hennessy, 1985) and the CSP failures model (Brookes et al., 1984). The need for such closure conditions arises from the desire to model deadlock and blocking and is orthogonal to the matter of fairness. However, fairness does introduce a need for additional closure conditions due to interactions between traces' fairness sets \( F \) and enabling sets \( E \). To understand why, recall that, in an infinite trace \( \langle \pi, (F, E, \xi) \rangle \), the sets \( F \) and \( E \) play dual roles: \( F \) represents constraints on the type of context in which \( \pi \) can represent a fair transition sequence, while \( E \) provides enabling information necessary for determining whether other components' fairness constraints are satisfied. Therefore, distinguishing between a process with the trace

\[
\langle s, a!0, s \rangle, \langle b!, c!, d! \rangle, \langle b!, c!, d! \rangle \rangle
\]

and another process with the trace

\[
\langle s, a!0, s \rangle, \langle b!, c!, \rangle, \langle b!, c!, \rangle \rangle
\]

without regard to their respective fairness constraints would be undecidable in general.
\(\langle \pi, (F, E, \varepsilon) \rangle\) and one with the trace \(\langle \pi, (F, E', \varepsilon) \rangle\) requires a context with a sub-component \(Q\) whose constraints are satisfied by \(E\) but not by \(E'\) (or vice versa); when placed in such a context, one process can perform \(\pi\) fairly while \(Q\) blocks, whereas the other process cannot perform \(\pi\) without eventually synchronizing with \(Q\). In contrast, distinguishing a process with trace \(\langle \pi, (F, E, \varepsilon) \rangle\) from one with trace \(\langle \pi, (F', E, \varepsilon) \rangle\) requires a context that satisfies the constraints of \(F\) but not \(F'\) (or vice versa); in particular, the context must enable some direction in \(F\) or \(F'\) (but not both) infinitely often without becoming blocked itself, thereby providing infinitely many synchronization opportunities to a previously blocked mod \(F\) (or \(F'\)) subcomponent of one of the processes.

Bearing these considerations in mind, we now consider two more commands that behave the same in all program contexts and yet have different meanings under the semantics \(\mathcal{F}\):

\[C_5 \equiv (a!0 \rightarrow b!0 \rightarrow c!0)\Box(a!0 \rightarrow b?x)\Box(a!0 \rightarrow (b!0\Box b?x)),\]

\[C_6 \equiv (a!0 \rightarrow b!0 \rightarrow c!0)\Box(a!0 \rightarrow b?x)\Box(a!0 \rightarrow (b!0\Box b?x))\Box(a!0 \rightarrow b!0).\]

The only potential difference between these commands is that \(C_6\) can perform the successfully terminating sequence of actions \([a!0 \ b!0]\) without enabling input on channel \(b\). That is, the trace \(\varphi = \langle(s, a!0, s)(s, b!0, s), (\mathcal{G}, \{a!, b!?, \varepsilon\})\rangle\) is possible for both \(C_5\) and \(C_6\), whereas the trace \(\varphi' = \langle(s, a!0, s)(s, b!0, s), (\mathcal{G}, \{a!, b!\}, \varepsilon)\rangle\) is possible only for \(C_6\). To distinguish \(C_6\) from \(C_5\), we must distinguish \(\varphi'\) from \(\varphi\), which requires an argument based on fairness. In particular, we need a context that allows each \(C_i\) to repeatedly perform the sequence \([a!0 \ b!0]\) and permits an observer to determine when the direction \(b?\) is enabled only finitely often along the infinite computation. Such a context must include the following three components, placed in parallel with one another:

1. A loop \(L\) that iterates the relevant \(C_i\) infinitely many times. Intuitively, when \(C_6\) is placed in this loop, it can repeatedly perform the sequence \([a!0 \ b!0]\) without ever enabling the direction \(b?\). In contrast, when \(C_5\) is placed in this loop and performs the same sequence of actions, it must enable \(b?\) infinitely often.

2. A component \(B\) that can block fairly only when \(L\) does not enable \(b?\) infinitely often. This component must contain a guard \(b!e\) that can be ignored (thereby letting \(B\) block) only if it has insufficient synchronization opportunities. Because blocking can happen only when synchronization is required, both \(B\) and \(L\) must appear in the scope of restriction on channel \(b\).

3. A synchronizing component \(S\) that offers input opportunities for each of \(L\)'s \(b!0\) actions. \(L\) needs to perform the action \(b!0\) infinitely often, and yet communication on channel \(b\) is restricted. Therefore, there must be a component in parallel with \(L\) that can synchronize with each of \(L\)'s \(b!0\) actions.

Unfortunately, \(S\) also provides \(B\)'s guard \(b!e\) (which is intended to block in certain situations) with infinitely many synchronization opportunities. As a result, \(B\) cannot block fairly, regardless of whether \(C_5\) or \(C_6\) is inserted into the context. In effect, \(C_5\)'s enabling (but nonuse) of \(b?\) is obscured by its use of the matching
direction $b$! Because every possible distinguishing context must have the same general form (and therefore the same conflict), $C_7$ and $C_8$ are behaviorally indistinguishable. More generally, a trace set containing the finite or infinite trace $\langle x, (F, E \cup X, R) \rangle$, with $X \cap \text{vis}(x) = \emptyset$ and $\overline{X} \subseteq \text{vis}(x)$, cannot be distinguished from one that also contains the trace $\langle x, (F, E, R) \rangle$.

Finally, consider the commands $C_7 \equiv G_1 \sqcap G_2$ and $C_8 \equiv G_1 \sqcap G_2 \sqcap G_3$, where $G_1$, $G_2$, and $G_3$ are defined as

\[
\begin{align*}
G_1 & \equiv b!0 \rightarrow \text{while true do } (b!0 \sqcap a?x \sqcap a!0) \\
G_2 & \equiv b!0 \rightarrow ((\text{while true do } b!0) \sqcap (a?x \rightarrow \text{while true do } a?x)) \\
G_3 & \equiv b!0 \rightarrow \text{while true do } (b!0 \sqcap a?x).
\end{align*}
\]

Letting $x$ represent the simple trace $[(s, b!0, s)(s, e, s)]^\omega$, the trace sets of $C_7$ and $C_8$ both contain the traces $\varphi_1 = \langle x, (\emptyset, \{b!, a?!, a!\}, i) \rangle$ and $\varphi_2 = \langle x, (\{a?\}, \{b!, a?!, i\}) \rangle$ but the trace $\varphi_3 = \langle x, (\emptyset, \{b!, a!\}, i) \rangle$ is possible only for $C_8$. To distinguish between $C_7$ and $C_8$, a context must distinguish $\varphi_3$ from both $\varphi_1$ and $\varphi_2$ at the same time. Distinguishing $\varphi_3$ from $\varphi_1$ requires a context that places the relevant $C_i$ in parallel with a component $B$ that tries to perform input on channel $a$ but blocks; distinguishing $\varphi_3$ from $\varphi_2$ requires a context that places the relevant $C_i$ in parallel with a component $NB$ that enables output on channel $a$ infinitely often and yet does not block. As a result, any distinguishing context for the commands $C_7$ and $C_8$ must contain both of these components running in parallel, one continuously attempting to perform input and the other repeatedly offering matching output. In such a context, the “blocking” component $B$ is enabled infinitely often by $NB$, regardless of which command is inserted. It follows that no context can possibly distinguish the commands $C_7$ and $C_8$. More generally, whenever the traces $\langle x, (F \cup \{d\}, E, i) \rangle$ and $\langle x, (F, E \cup \{d\}, i) \rangle$ are in a trace set $T$, it is impossible to determine whether the trace $\langle x, (F, E, i) \rangle$ is in $T$ as well.

We formalize these observations by imposing the following closure conditions on trace sets.

**Definition 6.2.** Given a fair trace set $T$, the **closure** of $T$ (written $T^+$) is the smallest set containing $T$ and satisfying the following conditions:

- **Union:** If $\langle x, (F_1, E_1, \wp) \rangle$ and $\langle x, (F_2, E_2, \wp) \rangle$ are in $T$, then $\langle x, (F_1 \cup F_2, E_1 \cup E_2, \wp) \rangle$ is in $T^+$.
- **Convexity:** If $\langle x, (F_1, E_1, \wp) \rangle$ and $\langle x, (F_2, E_2, \wp) \rangle$ are in $T$, $E_1 \subseteq E \subseteq E_2$, $F_1 \subseteq F \subseteq F_2$, and $F \supseteq E$, then $\langle x, (F, E, \wp) \rangle$ is in $T^+$.
- **Superset:** If $\langle x, (F, E, R) \rangle$ is in $T^+$, $R \in \{\emptyset, i\}$, $F \subseteq F'$, and $E \subseteq E'$, then $\langle x, (F', E', R) \rangle$ is in $T^+$.
- **Displacement:** If $\langle x, (F, E \cup X, R) \rangle$ is in $T^+$, $R \in \{\emptyset, i\}$, $X \cap \text{vis}(x) = \emptyset$, and $X \subseteq \text{vis}(x)$, then $\langle x, (F, E, R) \rangle$ is in $T^+$.
- **Contention:** If $\langle x, (F \cup \{d\}, E, i) \rangle$ and $\langle x, (F, E \cup \{d\}, i) \rangle$ are in $T^+$, then $\langle x, (F, E, i) \rangle$ is in $T^+$. 


As we shall see soon, these closure conditions are precisely what is needed to obtain a fully abstract semantics. We let $\mathcal{P}$ be the set of closed sets of fair traces and define a \emph{closed trace semantics} $\mathcal{T}^\circ$: $	ext{Com} \to \mathcal{P}$ denotationally, modifying the semantic equations given for $\mathcal{T}$ in Section 5 by building the closure into each clause. Letting $\mathcal{T}^\circ[h] = \mathcal{T}[h]^\circ$, we define $\mathcal{T}^\circ$ as in Fig. 5.

The following proposition asserts that, for all commands $c$, $\mathcal{T}^\circ[c]$ is precisely the closure of $\mathcal{T}[c]$. The proof of this proposition, which requires a detour that examines general properties of commands’ trace sets, appears in Section A.1 of the Appendix.

\textbf{Proposition 6.3.} \textit{For all commands $c$, $\mathcal{T}^\circ[c] = (\mathcal{T}[c])^\circ$.}

\textbf{6.2. Inequational Full Abstraction}

We can now prove full abstraction of the semantics $\mathcal{T}^\circ$ for the behavior $\mathcal{M}$.

\textbf{Proposition 6.4.} \textit{The closed trace semantics $\mathcal{T}^\circ$ is inequationally fully abstract with respect to $\mathcal{M}$: for all commands $c$ and $c'$,}

\[ \mathcal{T}^\circ[c] \subseteq \mathcal{T}^\circ[c'] \iff \forall P \in [-].\mathcal{M} \left[ P[c] \right] \subseteq \mathcal{M} \left[ P[c'] \right]. \]

\textbf{Proof.} The forward implication follows from the compositionality of $\mathcal{T}^\circ$, the monotonicity of operations on trace sets, and the fact that, when $\mathcal{T}^\circ[c] \subseteq \mathcal{T}^\circ[c']$, \[
\mathcal{M} \left[ P[c] \right] = \{ \text{states}(x) \mid \exists E. \langle x, (\emptyset, E, R) \rangle \in \mathcal{T}^\circ[[P[c]]] \& R \in \{ ?, i \} \& \text{chans}(x) = \{ e \} \}
\cup \{ \text{states}(x) \delta \mid \langle x, (\emptyset, R, p) \rangle \in \mathcal{T}^\circ[[P[c]]] \& \text{chans}(x) = \{ e \} \}
\subseteq \{ \text{states}(x) \mid \exists E. \langle x, (\emptyset, E, R) \rangle \in \mathcal{T}^\circ[[P[c']]] \& R \in \{ ?, i \} \& \text{chans}(x) = \{ e \} \}
\cup \{ \text{states}(x) \delta \mid \langle x, (\emptyset, R, p) \rangle \in \mathcal{T}^\circ[[P[c']]] \& \text{chans}(x) = \{ e \} \}
= \mathcal{M} \left[ P[c'] \right].
\]

(We write $\text{states}(x)$ to indicate the sequence of states encountered along $x$: for example, if $x = (s_0, e, s_1 | s_2, s_3) \cdots | s_k, e, s_{k+1}$, then $\text{states}(x) = s_0, s_1, s_2 \cdots s_k, s_{k+1}$.)

For the reverse implication, consider $\varphi = \langle x, (F, E, ?) \rangle$ in $\mathcal{T}^\circ[c] - \mathcal{T}^\circ[c']$.

\textit{Case: $\varphi = \langle x, (F, E, ?) \rangle$. Because $\varphi$ is a finite trace, we can assume without loss of generality that $F = \emptyset$. Let $\langle x, (E_1, ?), ..., \langle x, (E_m, ?) \rangle$ be the (necessarily finite number of) minimal $x$ traces in $\mathcal{T}^\circ[c']$. Closure under superset ensures that $E_i \not\subseteq E$ for each $i \in m$; thus for each $i$ we can choose a direction $d_i \in E_i - E$.

Let $x_1, ..., x_n$ be the free identifiers of $c$, and let $h_1, ..., h_k$ be the channel names appearing in $c$. We let $x$, $y$, $\text{flag}$, $\text{step}$, $v_1, ..., v_n$ be fresh identifiers, and we define}
track of the number of steps performed along the way.

Because guards never enable synchronization with any of the guards when\(d_i = h?\), and \(g_i = h?x\) when \(d_i = h!.\) We also define a command \(\text{Match}_{y,i}(\alpha)\) inductively as

\[
\text{Match}_{y,i}(x, e, s') = \text{step} := i
\]

\[
\text{Match}_{y,i}(s, h!n, s') = h?y \rightarrow \text{step} := i
\]

\[
\text{Match}_{y,i}(x, h?n, s') = h!n \rightarrow \text{step} := i
\]

\[
\text{Match}_{y,i}(\sigma \beta) = \text{Match}_{y,i+1}(\sigma) \land \text{Match}_{y,i+1}(\beta).
\]

Intuitively, the command \(\text{Match}_{y,i}(\alpha)\) can synchronize with the trace \(\alpha\), keeping track of the number of steps performed along the way.

We now let \(P[\_]\) be the context

\[
\begin{align*}
\text{while true do} & \\
& (v_1 := x_1; v_2 := x_2; \ldots; v_n := x_n; \\
& (\text{[\_]} || \text{Match}_{y,1}(\alpha)); \\
& x_1 := e_1; v_2 := x_2; \ldots; x_n := e_n) \\
& \sum_{i=1}^{m} (g_i \rightarrow \text{flag} := 1) \\
& \text{\text{\textbackslash\textbeta}_{1} \ldots \text{\textbackslash\textbeta}_{k}.}
\end{align*}
\]

Because \(\phi\) never enables synchronization with any of the guards \(g_i\), \(\text{\textbackslash A}[P[e]]\) has a behavior corresponding to the infinite iteration of \(\alpha\) in which the variable \(\text{flag}\) is never set to 1. The existence of this behavior can be confirmed at the trace level as follows:

The left component (i.e., the while loop) has a trace \(\langle \delta, (\emptyset, E \cup V, 1) \rangle\), where \(V\) is the set of visible actions appearing along \(\alpha\) and \(\delta\) is the trace obtained when
c repeatedly performs \( x \) by synchronizing with \( \text{Match}_{x,1}(x) \). The right component has an empty partial trace \( \langle s, (\emptyset, Y, p) \rangle \), where \( Y = \{ d_i \mid 1 \leq i \leq m \} \).

These traces are mergeable if and only if the sets \( (E \cup \hat{Y}) \) and \( \hat{Y} \) do not match. The construction of the set \( \hat{Y} \) ensures that \( E \) and \( \hat{Y} \) do not match; thus, the minimality of each \( \langle x, (\emptyset, E_i, \iota) \rangle \) ensure that \( \hat{V} \) and \( \hat{Y} \) do not match.

In contrast, every computation of \( P[c'] \) that iterates \( x \) infinitely many times must also enable synchronization infinitely often with at least one guard \( g_i \). It follows that any behavior in \( \mathcal{A}[P[c']] \) corresponding to the infinite iteration of \( x \) must eventually set \( \text{flag} \) to 1.

**Case:** \( \varphi = \langle x, (F, E, p) \rangle \). Without loss of generality, we can assume that \( F = E \).

We let \( x, y, \text{flag}, \text{step} \) be fresh identifiers, and we let \( h_1, ..., h_k \) be the channel names appearing in \( c \). We let \( a \) be a fresh channel name not appearing in \( c \) or \( c' \).

Let \( \langle x, (E_1, E_1, p) \rangle, ..., \langle x, (E_m, E_m, p) \rangle \) be the finite number of minimal partial \( x \) traces in \( \mathcal{T}[c'] \), and let \( Z = \bigcup_{i=1}^{m} E_i \). Closure under union ensures that \( \langle x, (Z, Z, p) \rangle \) is in \( \mathcal{F}[c'] \); by convex closure, it must be that (for each \( i \leq m \))

\[ \neg(E_i \subseteq E \subseteq Z). \]

Therefore, either \( E \not\subseteq Z \) or for each \( i, E_i \not\subseteq E \).

If \( E \not\subseteq Z \), then there exists a direction \( d \in E - Z \). Let \( g \) be a matching guard for \( d \) if \( d \not\subseteq \emptyset \), and let \( P[\_] \) be the context

\[ ([\_] \mid \text{Match}_{x,1}(x); \text{flag} := 1; g \rightarrow \text{flag} := 2) \backslash h_1 \ldots h_k. \]

(When \( d = \emptyset \), replace the code fragment “\( g \rightarrow \text{flag} := 2 \)” by “\( \text{flag} := 2 \)”.) \( \mathcal{A}[P[c]] \) has a behavior that begins with a correspondence to \( x \), followed by \( \text{flag} \) being set to 1 and then, exactly two steps later, being set to 2. In contrast, \( \mathcal{A}[P[c']] \) has no such behavior.

If, instead, each \( E_i \not\subseteq E \), then for each \( i \) choose a direction \( d_i \in E_i - E \). Let \( g_i \) be a matching guard for \( d_i \) whenever \( d_i \not\subseteq \emptyset \), and let \( g_i \) be the guard \( a' \! 0 \) when \( d_i = \emptyset \).

Let \( P[\_] \) be the context

\[ ([\_] \mid \text{Match}_{x,1}(x); y := 0; \bigoplus_{i=1}^{m} g_i \rightarrow \text{flag} := 1) \backslash h_1 \ldots h_k \backslash a. \]

\( \mathcal{A}[P[c]] \) has a deadlocked behavior corresponding to \( x \) in which the final step involves setting \( y \) to 0. In contrast, every deadlocked behavior in \( \mathcal{A}[P[c']] \) corresponding to \( x \) must take at least one step after setting \( y \) to 0.

**Case:** \( \varphi = \langle x, (F, E, \iota) \rangle \). Without loss of generality, assume that \( \varphi \) is minimal in \( \mathcal{F}[c] \).

We let \( x, y, f_1, f_2, t_1, t_2, \text{synch}, \text{value}, \text{comm}, \text{and count} \) be fresh identifiers, \( h_1, ..., h_k \) be the channel names appearing in \( c \), and \( a \) be a fresh channel name not appearing in \( c \) or \( c' \).

Let \( \varphi_1 = \langle x, (F_1, E_1, \iota) \rangle, ..., \varphi_m = \langle x, (F_m, E_m, \iota) \rangle \) be the minimal \( x \) traces in \( \mathcal{F}[c'] \). Closure under superset ensures that, for each \( i \leq m \),

\[ \neg((F_i \subseteq F \& E_i \subseteq E). \]

Thus, for each \( i \) we can choose a \( d_i \in (F_i - F) \cup (E_i - E) \). Moreover, contention ensures that we can always choose these directions in such a way that the set \( \{ d_i \mid 1 \leq i \leq m \} \) is partitionable into sets \( X \subseteq \{ d_i \in (F_i - F) \mid 1 \leq i \leq m \} \), \( Y \subseteq \{ d_i \in (E_i - E) \mid 1 \leq i \leq m \} \) for which \( \neg\text{match}(X, Y) \) (see Lemma A.10 in the Appendix).
\[
\text{count} := \text{count} + 1; \text{synch} := 1; \\
\text{while true do} \\
\text{Pick}_\text{Int}(\text{comm}, t1, t2); \\
\text{Pick}_\text{Int}(\text{value}, t1, t2); \\
\text{case} (\text{comm} \mod (2k + 1)) \text{ of} \\
1: \text{synch} := 0; ((h_1! \text{value} \rightarrow \text{synch} := 1) \parallel \sum_{g \in G_x} (g \rightarrow f1 := 1)) \\
2: \text{synch} := 0; ((h_1? \text{value} \rightarrow \text{synch} := 1) \parallel \sum_{g \in G_x} (g \rightarrow f1 := 1)) \\
\vdots \\
2k - 1: \text{synch} := 0; ((h_k! \text{value} \rightarrow \text{synch} := 1) \parallel \sum_{g \in G_x} (g \rightarrow f1 := 1)) \\
2k: \text{synch} := 0; ((h_k? \text{value} \rightarrow \text{synch} := 1) \parallel \sum_{g \in G_x} (g \rightarrow f1 := 1)) \\
0: \text{synch} := 0; (((a? \text{value} \rightarrow \text{synch} := 1) \parallel \sum_{g \in G_x} (g \rightarrow f1 := 1)) \parallel a!0) \cdot a \\
\text{endcase;}
\text{count} := \text{count} + 1.
\]

**FIG. 6.** The program \(\text{Guess}(H, G_x, f1)\).

Define sets \(G_x = \{h!0 \mid h? \in X\} \cup \{h?x \mid h! \in X\}\) and \(G_y = \{h!0 \mid h? \in Y\} \cup \{h?y \mid h! \in Y\}\) so that each direction in \(X\) has a matching guard in \(G_x\) and each direction in \(Y\) has a matching guard in \(G_y\). Let \(\text{Guess}(H, G_x, f1)\) abbreviate the command in Fig. 6, where the case construct is syntactic sugar for the corresponding series of nested if-statements and each command \(\text{Pick}_\text{Int}(\text{var}, t1, t2)\) (as defined in Fig. 7) sets \(\text{var}\) to an arbitrary integer. Intuitively, the program \(\text{Guess}(H, G_x, f1)\) can synchronize with any computation of any program that uses only the channels \(h_1, \ldots, h_k\) for visible communication. For each synchronization, \(\text{Guess}(H, G_x, f1)\) “guesses” the particular communication necessary for synchronization; the case where \(\text{comm} \mod (2k + 1) = 0\) is necessary when \(\alpha\) involves only finitely many visible communications (e.g., \((s, e, s)\)). A crucial point here is that, for any infinite computation \(\rho\) of \(\text{Guess}(H, G_x, f1)\), the directions enabled infinitely often along \(\rho\) are precisely those that either are visible infinitely often along \(\rho\) or are associated with the guards in \(G_x\).

Let \(P[x] = \cdot [-] \parallel \text{Guess}(H, G_x, f1) \parallel \sum_{g \in G_x} g \rightarrow f2 := 1) \cdot h_1 \cdots \cdot h_k\).

\(\#(P[e])\) has a behavior corresponding to \(\alpha\) in which neither \(f1\) nor \(f2\) is ever set to 1. The existence of this behavior can be confirmed at the trace level as follows:

\(\text{Guess}(H, G_x, f1)\) has a trace \(\langle \bar{s}, (\bar{X} \cup \bar{Y}, \bar{z})\rangle\), where \(V = \text{vis}(\bar{x})\) and \(\bar{s}\) is the trace obtained when \(\text{Guess}(H, G_x, f1)\) provides exactly the matching actions for \(\alpha\). The right component has the empty partial trace \(\langle e, (Y, Y, y)\rangle\).

**FIG. 7.** The command \(\text{Pick}_\text{Int}(\text{var}, t1, t2)\).
The mergeability of these traces along with \( \varphi = \langle \pi, (F, E, \downarrow) \rangle \) hinges on the following relationships: \( \neg \text{match}(F, X \cup \hat{V}), F \cap \hat{V} = \emptyset, \neg \text{match}(F, \hat{Y}), \) and \( \neg \text{match}(\hat{Y}, E \cup X \cup \hat{V}) \). The first three facts follow from the construction of the sets \( X \) and \( Y \) and from the relationship between the sets \( F, E, \) and \( \text{vis}(\hat{X}) \) for any minimal trace; the fourth fact depends on contention (for \( \neg \text{match}(\hat{Y}, \hat{V}) \)) and displacement (for \( \neg \text{match}(\hat{Y}, \hat{V}) \)).

In contrast, every behavior of \( \text{M}[P[c']] \) corresponding to \( \pi \) must eventually set at least one of the flags \( f_1 \) and \( f_2 \) to 1.

### 6.3. Program Equivalences

By virtue of full abstraction, the semantics \( \mathcal{T}^+ \) validates many natural laws of program equivalence with respect to the behavior \( \text{M} \). For example, we can prove the following (in)equivalences, where \( c \equiv c' \) indicates that \( \mathcal{T}^+[c] = \mathcal{T}^+[c'] \):

\[
\begin{align*}
&c_1 \parallel c_2 \equiv c_2 \parallel c_1 \\
&(c_1 \parallel c_2) \parallel c_3 \equiv (c_2 \parallel c_3) \\
&c \setminus h_1 \parallel h_2 \equiv c \setminus h_2 \parallel h_1 \\
&(c_1 \parallel c_2) \setminus h \equiv c_1 \parallel (c_2 \setminus h), \quad \text{provided } h \notin \text{chans}(c_1) \\
&c \setminus h \equiv c, \quad \text{provided } h \notin \text{chans}(c) \\
&a!0 \parallel b!0 \equiv (a!0 \rightarrow b!0) \bowtie (b!0 \rightarrow a!0) \\
&(\text{if } b \text{ then } c_1 \text{ else } c_2); c \equiv \text{if } b \text{ then } c_1; c \text{ else } c_2; c \\
&(\text{if } b \text{ then } c_1 \text{ else } c_2) \parallel c \not\equiv \text{if } b \text{ then } (c_1 \parallel c) \text{ else } (c_2 \parallel c) \\
&(\text{if } b \text{ then } c_1 \text{ else } c_2) \parallel (\text{skip}; c) \equiv \text{skip}; ((\text{if } b \text{ then } c_1 \text{ else } c_2) \parallel c).
\end{align*}
\]

Each of the program equivalences also holds for all of the semantic variations considered in the next section, and the inequivalence becomes a valid equivalence when we introduce stuttering and mumbling.

There are certain laws that hold under the standard (unfair) models that do not hold under the fair-trace semantics. For example, the standard models validate the refinement

\[
\text{while true do skip} \subseteq \text{while true do skip} \parallel (\text{skip}; a!0).
\]

That is, every trace or failure of \( \text{while true do skip} \) is also a trace or failure of \( \text{while true do skip} \parallel (\text{skip}; a!0) \). However, this relationship does not hold in the fair semantics \( \mathcal{T}^+ \): every infinite trace of the latter command either includes the action \( a!0 \) or has an additional fairness constraint.
In fact, the semantics $\mathcal{F}^\dagger$ distinguishes even finite commands (i.e., commands having no infinite behaviors) that the failures model and acceptance trees do not. For example, the commands
g_1 ≜ (a!0 → (c!0 ⇒ c? x ⇒ c!0)) □ (a!0 → (c!0 || c? x) □ (a!0 → x := 1)),
g_2 ≜ (a!0 → (c!0 ⇒ c? x ⇒ c!0)) □ (a!0 → x := 0) □ (a!0 → x := 1)

have precisely the same failures, but $\mathcal{F}^\dagger$ distinguishes them: in the context

(while true do [ - ] || c! 1 → y := 1) \(c,\)

only $g_2$ can repeatedly set the variable $x$ to 0 without the variable $y$ ever being set to 1.

### 7. Semantic Variations

We now consider several other notions of behavior that relax one or more of the assumptions underlying $\mathcal{M}$, in each case showing how the semantics can be adapted to yield full abstraction. The changes to the semantics affect only the simple-trace components of the fair traces. The underlying notion of parameterized strong fairness—and thus the extra contextual information necessary to incorporate fairness assumptions—remains the same.

#### 7.1. Simple-Trace Behavior

The state-trace behavior $\mathcal{M}$ adopts a view of programs as closed systems that cannot communicate with the external world. However, if we wish to reason about the possible interactions a command may have with its environment, then it is essential to relax this assumption. In such cases, it is natural to consider the simple-trace behavior function $\mathcal{S}: \text{Com} \rightarrow \mathcal{P}(\Sigma^* \cup \Sigma^* \delta)$ defined by

\[
\mathcal{S}[c] = \{ \text{trace}(\rho) \mid \rho = \langle c, s_0 \rangle \xrightarrow{\delta_1} \langle c_1, s_1 \rangle \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_k} \langle c_k, s_k \rangle \text{ term} \}
\cup \{ \text{trace}(\rho) \mid \rho = \langle c, s_0 \rangle \xrightarrow{\delta_1} \langle c_1, s_1 \rangle \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_k} \langle c_k, s_k \rangle \text{ dead} \}
\cup \{ \text{trace}(\rho) \mid \rho = \langle c, s_0 \rangle \xrightarrow{\delta_1} \langle c_1, s_1 \rangle \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_k} \langle c_k, s_k \rangle \text{ is fair} \}.
\]

The behavior $\mathcal{S}$ clearly includes more information about a command’s possible computations than $\mathcal{M}$ does: for any command $c$, $\mathcal{S}[c]$ represents a superset of the computations represented by $\mathcal{M}[c]$. However, as the following full abstraction result attests, the two behaviors induce exactly the same notion of contextual equivalence: two programs exhibit the same $\mathcal{M}$-behaviors in all program contexts if and only if they exhibit the same $\mathcal{S}$-behaviors in all program contexts.

**Proposition 7.1.** The closed trace semantics $\mathcal{F}^\dagger$ is inequationally fully abstract with respect to $\mathcal{S}$: for all commands $c$ and $c'$,

\[\mathcal{F}^\dagger[c] \subseteq \mathcal{F}^\dagger[c'] \iff \forall P[-].\mathcal{S}[P[c]] \subseteq \mathcal{S}[P[c']].\]
Proof. The forward implication follows from the compositionality of \( \mathcal{F} \), the monotonicity of operations on trace sets, and the fact that, for all commands \( c \), \( \mathcal{F}[c] \) can be extracted from \( \mathcal{F}[c'] \).

For the reverse implication, assume \( \mathcal{F}[c] \not\subseteq \mathcal{F}[c'] \). By Proposition 6.4, there exists a context \( P[\_\_] \) and a behavior \( \beta \) that is in \( \mathcal{M}[P[c]] - \mathcal{M}[P[c']] \). Because

\[
\mathcal{M}[P[c]] = \{ \text{states}(x) \mid x \in \mathcal{F}[P[c]] \land \text{chans}(x) = \{e\} \}
\]

(and likewise for \( \mathcal{M}[P[c']] \)), there must be a trace \( x \) in \( \mathcal{F}[P[c]] - \mathcal{F}[P[c']] \).

7.2. Stuttering and Mumbling

The behaviors \( \mathcal{M} \) and \( \mathcal{F} \) assume that an observer can detect every state change made along a computation. In many cases, however, the concept of “next state” is ill-defined, such as when a program is distributed across multiple machines with different clock speeds. As a result, it is often appropriate to assume only that the observer is capable of seeing some subsequence of the states encountered during a computation. In doing so, we make use of generalized relations \( \delta \) and \( \delta^* \) that reflect the possibility of introducing idle steps immediately following : |

\[
\delta = \delta' \circ \delta \circ \delta \circ \cdots
\]

These relations induce the generalized behaviors \( \mathcal{M}_* \) and \( \mathcal{F}_* \) as

\[
\mathcal{M}_*[c] = \{ s_0s_1 \cdots s_k \mid \langle c, s_0 \rangle \overset{s_0}{\rightarrow} \langle c_1, s_1 \rangle \overset{s_1}{\rightarrow} \cdots \overset{s_k}{\rightarrow} \langle c_k, s_k \rangle \ \text{term} \}
\]

\[
\cup \{ s_0s_1 \cdots s_k \mid \langle c, s_0 \rangle \overset{s_0}{\rightarrow} \langle c_1, s_1 \rangle \overset{s_1}{\rightarrow} \cdots \overset{s_k}{\rightarrow} \langle c_k, s_k \rangle \ \text{is fair} \},
\]

\[
\mathcal{F}_*[c] = \{ trace(p) \mid p = \langle c, s_0 \rangle \overset{s_0}{\rightarrow} \langle c_1, s_1 \rangle \overset{s_1}{\rightarrow} \cdots \overset{s_k}{\rightarrow} \langle c_k, s_k \rangle \ \text{term} \}
\]

\[
\cup \{ trace(p) \mid p = \langle c, s_0 \rangle \overset{s_0}{\rightarrow} \langle c_1, s_1 \rangle \overset{s_1}{\rightarrow} \cdots \overset{s_k}{\rightarrow} \langle c_k, s_k \rangle \ \text{is fair} \}
\]

To model the relations \( \delta \) accurately in our trace sets, we impose closure conditions corresponding to “stuttering” and “mumbling” (Lamport, 1983; Brookes, 1996b). Stuttering captures the reflexivity of \( \delta \) and has the effect of introducing idle steps into traces. We define the relation \( \text{stut} \subseteq \Phi \times \Phi \) as

\[
\text{stut} = \{( \langle x; \beta, \theta \rangle, \langle x(s, e, s) \beta, \theta \rangle ) \mid \beta \in \Sigma^+ \cup \Sigma^* \land s \in S' \}
\]

\[
\cup \{( \langle \alpha \beta, \theta \rangle, \langle x_0, \{e\}, \{e\}, p \rangle ) \mid x \in \Sigma^* \}
\]

Stuttering steps from \( \langle x; \beta, \theta \rangle \) to \( \langle x_0, \{e\}, \{e\}, p \rangle \) introduce the relevant partial traces for every possible idle-step introduction: the sets \( \{e\} \) reflect the possibility of an idle step immediately following \( x \).
Mumbling has the effect of absorbing \( \varepsilon \)-steps, just as the \( \Rightarrow \) relations absorb \( \varepsilon \)-transitions. We define the relation \( \text{mumb} \subseteq \Phi \times \Phi \) as

\[
\text{mumb} = \{ (\langle s, \varepsilon, s' \rangle (s', \varepsilon, s''), \beta, 0 \rangle, \langle s, \varepsilon, s'' \rangle (s', \varepsilon, s''), \beta, 0 \rangle) \mid s(s, \varepsilon, s'') \beta \in \Sigma^\infty \}
\]

\[
\cup \{ (\langle s, \varepsilon, s' \rangle (s', \varepsilon, s''), \beta, 0 \rangle, \langle s, \varepsilon, s'' \rangle (s', \varepsilon, s''), \beta, 0 \rangle) \mid s(s, \varepsilon, s'') \beta \in \Sigma^\infty \}
\]

\[
\cup \{ (\langle s, \varepsilon, s' \rangle (F, E, \varepsilon), \langle s, \varepsilon, s'' \rangle (E, \varepsilon, \varepsilon), \beta, 0 \rangle) \mid s(s, \varepsilon, s'') \beta \in \Sigma^\infty \}.
\]

Mumbling steps of the last form capture the intuition that, if \( \pi \) represents a transition sequence ending in configuration \( \langle c, s \rangle \), then each direction in \( E \cup \{ \varepsilon \} \) represents some \( \Rightarrow \)-transition possible from the configuration \( \langle c, s \rangle \).

Letting \( \text{id} = \{ (\varphi, \varphi') \mid \varphi \in \Phi \} \) be the identity relation on fair traces and following the approach of Brookes (1996a), we define \( \text{stut}^\infty \) and \( \text{mumb}^\infty \) to be the (respectively) greatest fixed points of the functionals \( F(R) = \text{stut} \cdot R \cup \text{id} \) and \( G(R) = \text{mumb} \cdot R \cup \text{id} \). That is, we define

\[
\text{stut}^\infty = \text{stut}^* \cdot \text{id} \cup \text{stut}^\infty, \quad \text{mumb}^\infty = \text{mumb}^* \cdot \text{id} \cup \text{mumb}^\infty.
\]

Intuitively, the pair \((\varphi, \varphi')\) is in \( \text{stut}^\infty \) (respectively, \( \text{mumb}^\infty \)) if \( \varphi' \) can be obtained by inserting an idle step (respectively, eliding an \( \varepsilon \)-step) at some of the positions along \( \varphi \)’s simple-trace component. In particular, when \( \varphi \) is an infinite trace, the stuttering and mumbling operations can be applied at potentially infinitely many places along \( \varphi \) but only finitely many times at any particular place along \( \varphi \). This point is essential for avoiding the accidental introduction of divergence: stuttering should never transform the finite trace \( \langle (s, \varepsilon, s'), (\emptyset, \emptyset, \varepsilon) \rangle \) into the infinite trace \( \langle (s, \varepsilon, s)', (\emptyset, \emptyset, \varepsilon) \rangle \).

**Definition 7.2.** Given a trace set \( T \), \( T^* \) is the smallest set containing \( T \) and closed under stuttering and mumbling:

- If \( \varphi \) is in \( T^* \) and \((\varphi, \varphi')\) \( \in \text{stut}^\infty \), then \( \varphi' \) is also in \( T^* \).
- If \( \varphi \) is in \( T^* \) and \((\varphi, \varphi')\) \( \in \text{mumb}^\infty \), then \( \varphi' \) is also in \( T^* \).

These closure conditions can be combined with the conditions introduced in Definition 6.2. For a trace set \( T \), we define \( T^* = (T^*)_1 \), so that \( T^* \) is closed under stuttering and mumbling, as well as superset, union, convexity, displacement, and contention.

Letting \( \mathcal{P}_\Phi \) be the set of closed sets of traces, we define a denotational semantic function \( \mathcal{F} : \text{Com} \rightarrow \mathcal{P}_\Phi \) such that, for all commands \( c \), \( \mathcal{F}_c^{(c)} = (\mathcal{F}[c])^{(c)} \). The addition of the stuttering and mumbling closure conditions is sufficient to yield full abstraction with respect to the generalized behaviors \( \cdot \) and \( \cdot \), as shown by the following results.

**Proposition 7.3.** The semantics \( \mathcal{F}^T_c \) is inequationally fully abstract with respect to \( \cdot \): for all commands \( c \) and \( c' \),

\[
\mathcal{F}_c^{(c)} \subseteq \mathcal{F}_c^{(c)} \Rightarrow \forall P[\cdot]. \cdot \mathcal{F}_c[P[c]] \subseteq \mathcal{F}_c[P[c']].
\]
Proof. The proof follows that of Proposition 6.4; the only difference occurs in the case of partial traces where each $E_i \not\subseteq E$. In this case, we choose (for each $i$) a direction $d_i \in E_i - E$ such that (when possible) $d_i \neq e$. Let $g_i$ be a matching guard for $d_i$ whenever $d_i \neq e$, and define the set $G = \{ d_i | d_i \neq e \& 1 \leq i \leq m \}$. Let $P[\_]$ be the context

\[
\left( [ - ] \| \text{Match}_{x, y}(x); y := 0; \sum_{g \in G} g \rightarrow \text{flag} := 1 \right) \| h_1 \ldots h_k.
\]

The only difference between this distinguishing context and that used for the same case in the proof of Proposition 6.4 is that we do not include an arbitrary guard $a!0$ for chosen directions $d_i = e$. The cases where $d_i = e$ can be ignored, because such steps are either idle steps (in which case some other chosen $d_j$ is appropriate), steps in which the state changes (and are therefore noticeable), or steps that lead to divergence.

**Proposition 7.4.** The semantics $\mathcal{F}_a^i$ is inequationally fully abstract with respect to $\mathcal{S}_a^*$: for all commands $c$ and $c'$,

\[
\mathcal{F}_a^i [c] \equiv \mathcal{F}_a^i [c'] \iff \forall P[\_]. \mathcal{S}_a^* [P[c]] \equiv \mathcal{S}_a^* [P[c']]\]

Proof: This is proven by obvious analogy with the proof of Proposition 7.1.

### 7.3. Busy-Waiting Behavior

The behaviors $\mathcal{H}$ and $\mathcal{S}$ (and their generalized forms $\mathcal{H}_a$ and $\mathcal{S}_a$) assume that deadlock can be distinguished from both successful termination and infinite chattering. The semantics $\mathcal{F}_a^i$ and $\mathcal{F}_a^*$ are well suited to this assumption, using different forms of traces to represent successfully terminating, infinite, and deadlocked computations. From an implementation point of view, however, deadlock and blocking often appear in the guise of busy-waiting. Because a scheduler cannot always detect whether a process has become blocked, it may continue to allocate processor cycles to a process that has no transitions enabled. This view of the world can be captured by the following busy-waiting trace behavior $\mathcal{W}_c$: $\text{Com} \rightarrow \mathcal{P}(\text{S}^\omega)$, in which deadlock is modeled as busy-waiting:

\[
\begin{align*}
\mathcal{W}_c[c] &= \{ s_0, s_1, \ldots, s_k | \langle c, s_0 \rangle \xrightarrow{e} \langle c_1, s_1 \rangle \xrightarrow{e} \cdots \xrightarrow{e} \langle c_k, s_k \rangle \text{ term} \} \\
&\cup \{ s_0, s_1, \ldots, s_k | \langle c_0, s_0 \rangle \xrightarrow{e} \langle c_1, s_1 \rangle \xrightarrow{e} \cdots \xrightarrow{e} \langle c_k, s_k \rangle \text{ dead} \} \\
&\cup \{ s_0, s_1, \ldots | \langle c_0, s_0 \rangle \xrightarrow{e} \cdots \xrightarrow{e} \langle c_k, s_k \rangle \xrightarrow{e} \cdots \text{is strongly fair} \}.
\end{align*}
\]

This behavior does not distinguish between deadlock and infinite idle chattering: for example, $\mathcal{W}_c[a!0] = \mathcal{W}_c[\text{while true do skip}] = \{ s^\omega | s \in S \}$.

To reason compositionally about $\mathcal{W}_c$, we introduce a semantics that is related to $\mathcal{F}_a^i$ but that represents blocked computations by infinite traces. Intuitively, a partial
computation that becomes blocked mod $F$ in a configuration $\langle c, s \rangle$ can be represented by the fair trace

$$\langle x(s, \epsilon, s)^\omega, (F, E, \bot) \rangle,$$

where $x$ is the finite trace corresponding to the transitions made before the computation became blocked and $E \subseteq F$ is the set of directions on which $c$ was trying to communicate. Intuitively, a computation that is blocked mod $E$ is fair mod $F$, and the infinitely enabled directions are the elements of $E$.

Employing the closure operators defined in Definitions 7.2 and 6.2 (and ignoring the conditions for partial traces), we can give an operational characterization of the trace semantics $\mathcal{T}_w$: Com $\rightarrow \mathcal{P}^*_\Phi(\Phi)$ as

$$\mathcal{T}_w[c] = (\{ \langle trace(p), (F, en(p), \bot) \rangle \mid
$$

$$\rho = \langle c, s_0 \rangle \xrightarrow{\delta_0} \langle c_1, s_1 \rangle \xrightarrow{\delta_1} \cdots \xrightarrow{\delta_k} \langle c_k, s_k \rangle \text{ term is fair mod } F \}
$$

$$\cup \{ \langle trace(p)(s_k, \epsilon, s_j)^\omega, (F, E, \bot) \rangle \mid F \supseteq E = \text{ inits} (c_k, s_k) \& \epsilon \notin E \&
$$

$$\rho = \langle c, s_0 \rangle \xrightarrow{\delta_0} \langle c_1, s_1 \rangle \xrightarrow{\delta_1} \cdots \xrightarrow{\delta_k} \langle c_k, s_k \rangle \& \neg \langle c_k, s_k \rangle \text{ term} \}
$$

$$\cup \{ \langle trace(p), (F, en(p), \bot) \rangle \mid
$$

$$\rho = \langle c, s_0 \rangle \xrightarrow{\delta_0} \langle c_1, s_1 \rangle \xrightarrow{\delta_1} \cdots \xrightarrow{\delta_k} \cdots \text{ is strongly fair mod } F \})^\dagger_w$$

The denotational characterization of $\mathcal{T}_w$ is similar to that given for $\mathcal{T}^\dagger$.

**Proposition 7.5.** The semantics $\mathcal{T}_w$ is inequationally fully abstract with respect to $\mathcal{W}$; for all commands $c$ and $c'$,

$$\mathcal{T}_w[c] \subseteq \mathcal{T}_w[c'] \Leftrightarrow \forall P[-]. \mathcal{W}^[-][P[c]] \subseteq \mathcal{W}^[-][P[c']]$$

**Proof.** The forward implication follows from the compositionality of $\mathcal{T}_w$, the monotonicity of operations on trace sets, and the fact that, for all commands $c$, $\mathcal{W}^[-][c]$ can be extracted from $\mathcal{T}_w[c]$.

The reverse implication uses an abbreviated version of the case analysis in the proof of Proposition 6.4. In particular, the cases for finite and infinite traces remain the same, and the case for partial traces disappears.

### 7.4. Communication Traces

Each of the behaviors considered so far incorporates the assumption that intermediate states encountered along a computation are observable. In many cases, however, it is appropriate to consider programs (or the processors on which they run) as black boxes whose internal states are private and whose only observable characteristics are their interactions with their environment. We now consider a communication-trace behavior that incorporates the assumption that states are truly
private and that only the sequence of visible communications that occur along a
computation is observable.

We redefine $A = \{ h!n, h?n \mid h \in \text{Chan} \land n \in \mathbb{Z} \}$ to be the set of visible communications, and we let $A^* = \{ \varepsilon \} \cup A^*$ be the set of finite communication sequences. The set of all communication sequences is $A^\omega = A^* \cup A^*\{\varepsilon\}^\omega \cup A^\omega$. For each $\eta \in A^\omega$, the generalized relation $\xrightarrow{\eta}$ is defined as follows:

- When $\eta$ is finite, $\langle c, s \rangle \xrightarrow{\eta} \langle c', s' \rangle$ indicates that the command $c$ in state $s$ can perform the sequence of visible communications $\eta$ (possibly with some intermediate $\varepsilon$-transitions) leading to the command $c'$ in state $s'$.

- When $\eta$ is infinite, $\langle c, s \rangle \xrightarrow{\eta} \langle c', s' \rangle$ indicates that there is a strongly fair computation of the command $c$, originating in state $s$, with the sequence of communications $\eta$. When $\eta$ has the form $\varepsilon^\omega$, the computation diverges after $\varepsilon$ with internal chatting.

Note that the empty sequence $\varepsilon$ is distinct from the communication sequence $\varepsilon^\omega$: the former represents a finite (possibly empty) sequence of internal actions, whereas the latter represents an infinite sequence of internal actions. Using these definitions, we define the communication trace behavior $\mathcal{C}: \text{Com} \rightarrow \mathcal{P}(A^\omega \cup A^*\delta)$ as

$$\mathcal{C}[c] = \{ \eta \mid \exists s, s', \langle c, s \rangle \xrightarrow{\eta} \langle c', s' \rangle \text{ term} \}$$

$$\cup \{ \eta\delta \mid \exists s, s', \langle c, s \rangle \xrightarrow{\eta\delta} \langle c', s' \rangle \text{ dead} \}$$

$$\cup \{ \eta \mid \exists s, \langle c, s \rangle \xrightarrow{\eta} \text{ is strongly fair} \}.$$  

Our semantics relies on the sets $\Sigma^*$ and $\Sigma^\omega$ of finite and infinite simple traces, redefined as

$$\Sigma^* = (S \times A^* \times S) \cup (S \times A^*), \quad \Sigma^\omega = S \times A^\omega.$$  

Finite traces with form $(s, \eta, s')$ represent successful computations, while finite traces with form $(s, \eta)$ represent partial computations. Successful traces require a final state for two reasons. First, the final state is necessary for determining which traces are composable: the traces $\varphi_1$ of $c_1$ and $\varphi_2$ of $c_2$ can be composed to generate a trace of $c_1; c_2$ only if $\varphi_2$ originates in the final state of $\varphi_1$. Second, the final state of a successful computation can be “observed” by transmitting the values of the finite number of variables along some channel. In contrast, the final state of partial traces is unnecessary for determining compositability, and there is no reliable way to interrupt a computation to observe intermediate states. Similarly, infinite traces contain no final state, because the computations they represent have no final state. We let $\Sigma^\omega = \Sigma^* \cup \Sigma^\omega$ be the set of all traces and—using the same contextual information as before—define the set $\Phi$ of fair communication traces by

$$\Phi = \Sigma^\omega \times (\mathcal{P}_\text{in}(A^*) \times \mathcal{P}_\text{in}(A^*) \times \{ f, p, i \}).$$
We define a trace semantic function \( T_c : \text{Com} \rightarrow \mathcal{P}(\Phi) \) characterized operationally as

\[
T_c[c] = \{ \langle (s, \eta, s'), (F, \text{en}(p), \xi) \rangle \mid p = \langle c, s \rangle \overset{\delta}{\Rightarrow} \langle c', s' \rangle \text{ term is fair mod } F \}
\]

\[
\cup \{ \langle (s, \eta), (F, E, \eta) \rangle \mid p = \langle c, s \rangle \overset{\delta}{\Rightarrow} \langle c', s' \rangle \& \sim \langle c', s' \rangle \text{ term \&} \\
F \equiv E = \text{inits}(c', s') \}
\]

\[
\cup \{ \langle (s, \eta), (F, \text{en}(p), \xi) \rangle \mid p = \langle c, s \rangle \overset{\delta}{\Rightarrow} \text{ is fair mod } F \}
\]

The denotational characterization of \( T_c \) is straightforward.

To achieve full abstraction, we again employ the closure conditions introduced in Definition 6.2; stuttering and mumbling are unnecessary, because there are no intermediate states. As before, we can define a closed trace semantic function \( T_c^\dagger : \text{Com} \rightarrow \mathcal{P}(\Phi) \) denotationally so that \( T_c^\dagger[c] = T_c[c] \) for all commands \( c \).

**Proposition 7.6.** The closed trace semantics \( T_c^\dagger \) is (inequationally) fully abstract with respect to \( \mathcal{C} \); for all commands \( c \) and \( c' \),

\[
T_c^\dagger[c] \subseteq T_c^\dagger[c'] \iff \forall P \vdash \mathcal{C}[P[c]] \subseteq \mathcal{C}[P[c']]\]

**Proof (Sketch).** As in the previous full abstraction proofs, the forward implication follows from the composition of \( T_c^\dagger \), the monotonicity of operations on trace sets, and the fact that, for all commands \( c \), \( \mathcal{C}[c] \) can be extracted from \( T_c^\dagger[c] \).

The reverse implication follows from a case analysis similar to that used in the proof of Proposition 6.4. The main difference is that the distinguishing context must now signal the occurrence of particular events with visible communications rather than with state changes.

For example, suppose that \( c \) has an infinite trace \( \langle (s, \eta), (F, E, \xi) \rangle \) that \( c' \) does not. Roughly speaking, each pair of lines

\[
\begin{align*}
2i-1: \text{synch} := 0; \quad (h_i!\text{value} \rightarrow \text{synch} := 1) \quad \square \sum_{g \in G} (g \rightarrow f1 := 1) \\
2i: \quad \text{synch} := 0; \quad (h_i?\text{value} \rightarrow \text{synch} := 1) \quad \square \sum_{g \in G} (g \rightarrow f1 := 1)
\end{align*}
\]

in \texttt{Guess}(H, G, f1) of Fig. 6 can be replaced by the following pair of lines, where \( c_1, \ldots, c_k \) and \( b \) are fresh channels:

\[
\begin{align*}
2i-1: \quad (h_i!\text{value} \rightarrow c_i!0 \rightarrow c_i!\text{value}) \quad \square \sum_{g \in G} (g \rightarrow b!0) \\
2i: \quad (h_i?\text{value} \rightarrow c_i!1 \rightarrow c_i!\text{value}) \quad \square \sum_{g \in G} (g \rightarrow b!0)
\end{align*}
\]
Each communication along channel $h_i$ is signaled by two outputs along channel $c_i$, the first indicating whether input or output occurred and the second indicating the transmitted value. The guard $b!0$ plays the same role that the variable flag played in the previous proof.

We then let $P[\_]$ be the following context, where we use communications on the fresh channel $a$ to record the initial state:

$$\left(a!x_1 \rightarrow a!x_2 \rightarrow \cdots a!x_n \rightarrow [\_] \parallel \text{Guess}(H, G_x, b!0) \parallel \sum_{g \in G_y} g \rightarrow b!0 \right) \backslash h_1 \backslash \cdots \backslash h_k.$$ 

$\%[P[c]]$ contains a behavior corresponding to $(s, \eta)$ in which the communication $b!0$ never occurs. In contrast, every behavior of $\%[P[c']]$ corresponding to $x$ must eventually perform the action $b!0$.

8. CONCLUSIONS

This paper describes several fully abstract semantics for a language of communicating processes, with each semantics suited to reasoning about a particular notion of strongly fair program behavior. In each semantics, standard traces are augmented with additional information that supports compositional reasoning about strong fairness. This necessary contextual information, which remains the same across the range of semantics, is made explicit by the definition of parameterized strong fairness. This shared structure simplifies the proofs of full abstraction for the semantic variations: many of the necessary definitions and lemmas can be reused directly or with only very minor modifications.

These semantics can be viewed as extensions to the CSP failures model (Brookes et al., 1984) and Hennessy’s acceptance trees (Hennessy, 1985) for dealing with strong fairness. Other authors have also used traces to model various forms of fairness in the message-passing setting. For data flow and asynchronous networks, Jonsson provides a fully abstract trace model that incorporates assumptions of weak fairness (Jonsson, 1994). The traces of this model are effectively communication traces without additional fairness-related information such as our sets $E$ and $F$. Essential for modeling weak fairness without this type of extra structure are the asynchronous nature of communication and the following assumptions: each channel is used for input by at most one node, each channel is used for output by at most one node, and no channel is used for both input and output by any node. Under these assumptions, a process $P$ trying to receive input from a channel is enabled only if some other process first sends a value on that channel; moreover, because no other process can consume that value, $P$ remains enabled until it finally makes some transition.

In the setting of synchronous communication, both Hennessy and Brookes have given semantics for notions of unconditional fairness. In (Hennessy, 1987), Hennessy extends acceptance trees with limit points that indicate which infinite paths are fair: roughly speaking, an infinite computation is considered fair if every
process makes infinitely many transitions along that computation. In particular, certain commands like \((\text{skip} \mid \text{while true do skip})\) have no fair computations at all: skip cannot make infinitely many transitions and \(\text{while true do skip}\) can never terminate. Brookes (1994) achieves similar net results for a slightly more liberal notion of fairness: an infinite computation is considered fair if every process either makes infinitely many transitions or terminates successfully. By adding infinite traces to Hoare’s (1981) trace semantics and adapting Park’s (1979) fairmerge operator to handle the potential of synchronization between parallel components, he obtains a fully abstract semantics for a notion of behavior that ignores the possibility of deadlock.

Neither of these semantics is sufficient for reasoning about more general notions of fairness in which processes may become blocked, such as weak or strong process fairness. The problem is that synchronous communication requires the active cooperation and participation of more than one process: a process’s ability to make progress can depend on the processes in parallel with it and their willingness to synchronize. To support reasoning about these types of fairness, it is essential to augment traces with additional information about the types of communications that are possible (even if never taken) along a given computation.

This observation provides the foundation for Darondeau’s (1985) fully abstract, strongly fair semantics for a stateless, CCS-like language. In this semantics, a term’s meaning is given by a set of histories; these histories capture the same type of information embedded in our fair traces. Whereas we obtain full abstraction by introducing closure conditions, Darondeau obtains full abstraction by considering only minimal sets of normalized histories (which, roughly speaking, correspond to traces \(\langle x, (F; E, z) \rangle\) for which \(E = F = E \cap F\)). Thus the two full-abstraction results are obtained in dual manners: in one case, additional traces are considered through closure; in the other, certain histories are ignored through normalization.

The semantics of this paper place Darondeau’s work in a more general light. In addition to being stateless, the language he considers has no notion of sequential composition and contains only a limited form of iteration: the iterative construct generates only infinite computations, and no other language constructs may appear inside it. Although our language is also limited to iteration (the inability to define \(T^\omega\) as a distinguished fixed point prevents a generalization to arbitrary recursion), its operators are common and less ad hoc. More important, however, our development makes explicit the underlying concept of parameterized strong fairness, which can be used either to aid operational reasoning or to ease the task of developing semantics for other notions of fairness. Similarly, the closed trace sets have a strongly operational interpretation, whereas the precise role of normalization for histories is unclear.

The characterization of parameterized fairness appears similar to Costa and Stirling’s (1987) operational transition rules for fair CCS, in that careful accounting of blocked processes is required. However, parameterized fairness requires only a record of the actions possible for blocked processes, rather than explicit records of the processes themselves. In contrast, Costa and Stirling’s transition rules employ a labeled CCS syntax that keeps track of subcomponents via unique addressing labels; this scheme is similar to that used in Subsection 2.3 but also attaches the
labels to individual actions and operators. The transitions are then decorated with sets (or sequences, in the case of strong fairness) representing the live processes that participate in action sequences. This additional bookkeeping complexity is necessary because Costa and Stirling’s rules are specifically designed to generate directly (i.e., in a single pass) all and only the fair computations. In contrast, parameterized fairness is based on the more typical two-level approach of generating computations and then filtering out the unfair ones in a second pass.

The communicating processes described in this paper have disjoint local states. Because a process’s external environment cannot alter its private state, state changes between steps of a fair trace are disallowed. Furthermore, because a process’s fair progress depends on the synchronization abilities of other processes, the traces must record relevant enabling information. In contrast, Brookes’ (1996b) transition traces for shared-variable programs permit intermediate state changes, reflecting the ability of a process’s external environment to affect the shared global state; no additional fairness-related information needs to be recorded. These two different kinds of trace structure are intuitively orthogonal, representing distinct but compatible aspects of computation. As shown in the author’s dissertation (Older, 1996), the two forms of trace can be combined in a very straightforward fashion to yield a fully abstract semantics for a hybrid language of processes that communicate through both message passing and shared memory. Moreover, the full-abstraction proof is a natural amalgam of the full-abstraction proofs of the original two semantics. The resulting semantics is similar to that of Horita et al. (1994) for a similar hybrid language but also incorporates fairness assumptions.

Finally, the semantics described in this paper are interleaving models, but their fairness-related structure seems appropriate also for noninterleaving models of concurrency, such as event structures or pomsets. In particular, the parameterization of strong fairness relies only on blocked processes and sets of infinitely enabled communications, features that are also relevant for “true” concurrency. Parameterized forms of fairness also support the construction of semantics for other fairness notions, such as weak fairness and strong channel fairness (Older, 1996, 1997). However, the resulting semantics are significantly more complex than that for strong fairness, due to their lack of equivalence robustness (Apt et al., 1988): these fairness notions are intrinsically dependent on the order in which independent actions occur, and their semantics reflect this dependency.

**APPENDIX: AUXILIARY PROOFS**

This appendix contains supplemental proofs and explanations of properties that are necessary for main results (such as full abstraction) but whose proofs require detours from the main path of the paper.

**A. Computational Feasibility**

The obvious inductive proof of the property \( \mathcal{F}^*[c] = \mathcal{F}[c]^* \) requires that we prove

\[
(\mathcal{F}^*[c_1] \parallel \mathcal{F}^*[c_2])^* = (\mathcal{F}[c_1]^* \parallel \mathcal{F}[c_2]^*)^* = (\mathcal{F}[c_1] \parallel \mathcal{F}[c_2])^*.
\]
Although this equality holds, we can prove it only by referring to particular properties of the trace sets $\mathcal{T}[c_1]$ and $\mathcal{T}[c_2]$: the property $(T_1 \parallel T_2)^f = (T_1 \parallel T_2)^f$ does not hold for arbitrary trace sets, as evidenced by the sets

$$T_1 = \{ (\alpha_1, (\emptyset, \{d, d, e\}, \dagger)), (\alpha_1, (\{d\}, \{d, e\}, \dagger)) \},$$

$$T_2 = \{ (\alpha_2, (\{d\}, \{d, d\}, \dagger)) \}.$$

We therefore consider a subset of properties that may not hold for arbitrary trace sets but do hold for every trace set $\mathcal{T}[c]$. These properties stem from the nature of programs, computations and the definition of parameterized fairness. For example, the trace $\langle \alpha, (\emptyset, E, \dagger) \rangle$ is in $\mathcal{T}[c]$ whenever any trace $\langle \alpha, (F, E, \dagger) \rangle$ is, because every successfully terminating computation is fair mod $\emptyset$. The following definition summarizes several of these properties that are essential for proving full abstraction.

**Definition A.1.** A fair trace set $T$ is computationally feasible if it satisfies the following properties:

- If the trace $\langle \alpha, (F, E, \dagger) \rangle$ is in $T$, then the trace $\langle \alpha, (\emptyset, E, \dagger) \rangle$ is in $T$.  
- If the trace $\langle \alpha, (F, E, R) \rangle$ is in $T$, $R \in \{\dagger\}$, and $F \equiv F'$, then $\langle \alpha, (F', E, R) \rangle$ is in $T$.

The trace $\langle \alpha, (F, E, \dagger) \rangle$ is in $T$ if and only if $F \equiv E$ and the trace $\langle \alpha, (E, E, \dagger) \rangle$ is in $T$.

- If the trace $\langle \alpha, (F, E, \dagger) \rangle$ is in $T$, then $\text{vis}(\alpha) \equiv E$.

The final property is subtle but extremely important. Intuitively, a trace $\varphi = \langle \alpha, (F \cup \{d\}, E \cup \{d, d\}, \dagger) \rangle$ represents a computation $\rho$ that enables the directions $d$ and $\bar{d}$ infinitely often and is fair mod $F \cup \{d\}$. Any subcomponent of $c$ that is blocked mod $(F \cup \{d\})$ along $\rho$ must be blocked in a configuration where its only transitions involve the directions $F \cup \{d\}$. If we assume that $d \notin F$, no blocked subcomponent is capable of using $\bar{d}$; therefore, because $\bar{d}$ is enabled infinitely often, no fairly blocked (mod $F \cup \{d\}$) process is capable of using $d$ either. It follows that every blocked subcomponent is restricted to using directions in $F$, and hence $\rho$ is also fair mod $F$. As a result, the trace $\langle \alpha, (F \cup \{d, d\}, \dagger) \rangle$ must be in the set $\mathcal{T}[c]$ whenever $\varphi$ is. Once we start considering the closed trace set $\mathcal{T}[c]^f$, however, we must account for the possibility that $\varphi$ appears in $\mathcal{T}[c]$. For example, the trace $\langle \alpha, (F \cup \{d\}, E \cup \{d, d\}, \dagger) \rangle$ in $\mathcal{T}[c]$.
The following lemma confirms that the definition of computational feasibility indeed captures general properties of commands’ trace sets.

**Lemma A.2.** For all commands \( c \), \( \mathcal{F}[c] \) is computationally feasible.

**Proof.** This is proven by a straightforward but tedious induction on the structure of \( c \).

The following two lemmas show that closure preserves computational feasibility and distributes over the various semantic operators when applied to computationally feasible trace sets.

**Lemma A.3.** If the trace set \( T \) is computationally feasible, then \( T^\dagger \) is also computationally feasible.

**Proof.** This is proven by a straightforward but tedious case analysis showing that each possible trace introduced by closure respects computational feasibility.

**Lemma A.4.** For all computationally feasible trace sets \( T, T_1 \) and \( T_2 \), the following properties hold:

\[
\begin{align*}
(T_1; T_2)^! &= (T_1^!; T_2^!)^! \\
(T^*)^! &= (T^!)^* \\
(T^\dagger)^! &= (T^\dagger)^h \\
(T \ominus T_2)^! &= (T_1^! \ominus T_2^!)^! \\
(T^m)^! &= (T^m)^! \\
(T_1 \parallel T_2)^! &= (T_1^! \parallel T_2^!)^! \\
(T_1 \boxdot T_2)^! &= (T_1^! \boxdot T_2^!)^!.
\end{align*}
\]

**Proof.** In general, the proof of each property is based on a simple case analysis that shows that whenever a trace is in \( T_1^! \) (for each relevant operator \( \odot \)), the trace is also in \( (T_1 \odot T_2)^! \). Because closure is monotonic and idempotent, it follows that \( (T_1^! \odot T_2^!)^! = (T_1^! \odot T_2^!)^! \).

The following result shows that, for all commands \( c \), the meaning given to \( c \) by the closed trace semantics \( \mathcal{F}^\dagger \) is exactly the closure of \( \mathcal{F}[c] \).

**Proposition A.5.** For all commands \( c \), \( \mathcal{F}^\dagger[c] = \mathcal{F}[c]^! \).

**Proof.** This is proved by a straightforward induction on the structure of \( c \), using the properties of Lemma A.4. For example, the case for parallel composition proceeds as follows, relying on the inductive hypothesis that \( \mathcal{F}^\dagger[c_i] = \mathcal{F}[c_i]^! \) for each \( i \):

\[
\mathcal{F}^\dagger[c_1 \parallel c_2] = (\mathcal{F}^\dagger[c_1] \parallel \mathcal{F}^\dagger[c_2])^! = (\mathcal{F}[c_1]^! \parallel \mathcal{F}[c_2]^!)^!
= (\mathcal{F}^\dagger[c_1] \parallel \mathcal{F}^\dagger[c_2])^! = \mathcal{F}[c_1 \parallel c_2]^!.
\]

### A.2. Conflict-Free Resolutions

**Definition A.6.** An element \( \varphi = \langle \alpha, (F, E, R) \rangle \) of a trace set \( T \) is **minimal** if for every \( \varphi' = \langle \alpha, (F', E', R') \rangle \) in \( T \), \( (F' \subseteq F & E' \subseteq E) \Rightarrow \varphi = \varphi' \).
Thus a finite or infinite trace $\varphi \in T$ is minimal if there is no trace $\varphi' \in T$ that would yield $\varphi$ through closure under subset; a partial trace $\varphi = (\alpha, (F, E, \varpi)) \in T$ is minimal if $F = E$ and every other partial trace $\varphi' = (\alpha, (F', E', \varpi'))$ in $T$ has a direction $d \in E' - E$. A closed trace set is uniquely characterized by its set of minimal traces.

**Definition A.7.** Let $T$ be a trace set, and let $\alpha$ be an infinite trace. The set $\text{min}(T, \alpha)$ is the set of minimal $\alpha$-traces in $T$: $\text{min}(T, \alpha) = \{ \varphi = (\alpha, (F, E, \varpi)) \in T \mid \varphi$ is minimal in $T \}$.

If the infinite trace $\varphi = (\alpha, (F, E, \varpi))$ is minimal in a computationally feasible trace set, then $F \subseteq E$, because directions enabled only finitely often do not introduce fairness constraints. Moreover, if the direction $d$ is in the set $F$ (representing a fairness constraint of some component), then either $d$ is also in $F$ (indicating that exactly one subcomponent enables each of $d$ and $\bar{d}$, with insufficient synchronization opportunities) or $\bar{d}$ is not enabled infinitely often. We call infinite traces that satisfy these criteria potentially minimal.

**Definition A.8.** An infinite trace $\varphi = (\alpha, (F, E, \varpi))$ is potentially minimal if $F \subseteq E$ and, for all directions $d \in F, \bar{d} \in F \Rightarrow d \in E$.

Every minimal trace of a computationally feasible trace set is potentially minimal.

Now, suppose that a closed, computationally feasible trace set $T$ does not contain the potentially minimal trace $\varphi = (\alpha, (F, E, \varpi))$. If $\text{min}(T, \alpha) \neq \emptyset$, then each minimal trace $(\alpha, (F_i, E_i, \varpi))$ in $T$ must have an additional fairness constraint (represented by a direction $d \in F_i - F$) or enable an additional direction infinitely often (represented by a direction $d \in E_i - E$). Intuitively, by carefully selecting one of these fairness constraints $d_i \in F_i - F$ or infinitely enabled directions $d_i \in E_i - E$ for each minimal $\varphi_i$, we can construct a context that distinguishes the trace $\varphi$ from the traces in $T$. For reasons similar to those that motivated the introduction of the contention closure condition, it is important that none of the selected fairness constraints matches any of the selected infinitely enabled directions. We formalize this “careful selection” as a conflict-free resolution, as given in the following definition.

**Definition A.9.** Let $T$ be a trace set not containing the trace $\varphi = (\alpha, (F, E, \varpi))$. A conflict-free resolution of $T$ for $\varphi$ is a total function $\mathcal{R}: \text{min}(T, \alpha) \rightarrow (A \times \{F, E\})$ satisfying the following two conditions:

- For all traces $\varphi_i \in \text{min}(T, \alpha)$,
  $$\mathcal{R}(\varphi_i) = (d_i, F) \Rightarrow d_i \in F_i - F \quad \& \quad \mathcal{R}(\varphi_i) = (d_i, E) \Rightarrow d_i \in E_i - E.$$

- For all traces $\varphi_i, \varphi_j \in \text{min}(T, \alpha)$,
  $$\mathcal{R}(\varphi_i) = (d_i, F) \& \mathcal{R}(\varphi_j) = (d_j, E) \Rightarrow \neg \text{match}(d_i, d_j).$$

As a consequence of the following lemma, a conflict-free resolution of $\mathcal{T}^\dagger[\ell]$ for $\varphi$ can always be constructed, for any command $\ell$ and any potentially minimal trace $\varphi \notin \mathcal{T}^\dagger[\ell]$. That is, the necessary “careful selection” is always possible.
Lemma 4.10. Let $T$ be a closed, computationally feasible trace set not containing the potentially minimal trace $\varphi = \langle x, (F, E, \hat{i}) \rangle$, and suppose that $\text{min}(T, x)$ is finite. There is a conflict-free resolution of $T$ for $\varphi$.

Proof. Let $\mathcal{R}$ be a total function $\mathcal{R}: \text{min}(T, x) \rightarrow (A \times \{ \emptyset, E \})$ such that, for all traces $\varphi_i \in \text{min}(T, x)$,

$$\mathcal{R}(\varphi_i) = (d_i, \emptyset) \Rightarrow d_i \in F_i - F \quad \& \quad \mathcal{R}(\varphi_i) = (d_i, E) \Rightarrow d_i \in E_i - E.$$

We say that $\mathcal{R}$ has conflicts on channel $h$ if there exist traces $\varphi_i, \varphi_j \in \text{min}(T, x)$ and a direction $d$ such that $\mathcal{R}(\varphi_i) = (d, \emptyset)$, $\mathcal{R}(\varphi_j) = (d, E)$, and $\text{chan}(d) = h$. We introduce an arbitrary well-ordering $<$ on channels, and we show that $\mathcal{R}$ can be transformed into a conflict-free resolution by removing conflicts in a systematic way, using the channel ordering.

Suppose $h$ is the least channel on which $\mathcal{R}$ has conflicts. There must be traces $\varphi_x = \langle x, (F_x, E_x, \hat{i}) \rangle$ and $\varphi_y = \langle x, (F_y, E_y, \hat{i}) \rangle$ in $\text{min}(T, x)$ such that $\mathcal{R}(\varphi_x) = (d, \emptyset)$, $\mathcal{R}(\varphi_y) = (d, E)$, and $\text{chan}(d) = h$. Exactly one of the following cases must hold:

Case: $d \notin E$. Because $T$ is computationally feasible and $\varphi_x$ is minimal, it must be that $d \notin E_x - E$ as well. Thus every mapping to $(d, \emptyset)$ in $\mathcal{R}$ can be replaced by a mapping to $(d, \emptyset)$; likewise, every mapping to $(d, E)$ can be replaced by a mapping to $(d, \emptyset)$. The resulting resolution has no conflicts on channels $k < h$ or on channel $h$.

Case: $d \in E$ and $(d \notin F_x$ or $d \notin F_y$). Because $d \in E$, $\mathcal{R}$ does not map any trace to the pair $(d, \emptyset)$. As a result, replacing $\mathcal{R}(\varphi_x)$ or $\mathcal{R}(\varphi_y)$ (or both, when possible) by a mapping to $(d, \emptyset)$ will remove at least one conflict on channel $h$ without introducing any conflicts on channels $k < h$.

Case: $d \in E$ and $d \notin F_x$ and $d \notin F_y$. Because $\varphi_x$ and $\varphi_y$ are minimal, we know that $d \notin E_x$ and $d \notin F_y$. By superset closure, $T$ contains the traces $\langle x, (F_x \cup F_y, E_x \cup E_y) - \{d\}, \hat{i} \rangle$ and $\langle x, ((F_x \cup F_y) - \{d\}, E_x \cup E_y, \hat{i}) \rangle$ via $\varphi_x$ and $\varphi_y$, respectively. It follows that the trace

$$\langle x, ((F_x \cup F_y) - \{d\}, E_x \cup E_y) - \{d\}, \hat{i} \rangle$$

is in $T$ by contention, and thus there must be some minimal trace $\varphi_r = \langle x, (F_r, E_r, \hat{i}) \rangle$ in $T$ such that $F_r \subseteq (F_x \cup F_y) - \{d\}$ and $E_r \subseteq (E_x \cup E_y) - \{d\}$.

If $\mathcal{R}(\varphi_r) = (e, E)$ (for some direction $e$), then $e \in E_r - E$, and hence $e \in E_x - E$ or $e \in E_y - E$. Likewise, if $\mathcal{R}(\varphi_r) = (e, \emptyset)$, then $e \in F_x - F$, and $e \in (F_y - F) \cup (F_y - F)$. Thus at least one of $\mathcal{R}(\varphi_x)$ and $\mathcal{R}(\varphi_y)$ can be replaced by a mapping to $\mathcal{R}(\varphi_r)$. This change cannot introduce any new conflicts on channels $k < h$ and reduces the number of conflicts on channel $h$.

Because $\text{min}(T, x)$ is finite, repeating the preceding analysis eventually removes all conflicts on channel $h$, without introducing any conflicts on any channel $k < h$. Moreover, because only finitely many channels can be mentioned in the set $\text{min}(T, x)$, a finite number of iterations must eventually result in a conflict-free resolution for $\varphi$. 

ACKNOWLEDGMENTS

Stephen Brookes coauthored a preliminary account of this work (Brookes and Older, 1995), and several discussions with him have greatly improved the formulation of these semantics. This research was sponsored in part by the office of Naval Research under Grant N00014-92-J-1298.

Received December 30, 1997; final manuscript received July 19, 1999

REFERENCES


