A Denotational Framework for Fair Communicating Processes

Susan Older

December 1996

CMU-CS-96-204

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Stephen Brookes, Chair
Edmund Clarke
Jeannette Wing
Prakash Panangaden, McGill University

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

Keywords: Denotational semantics, fairness, communicating processes, traces, concurrency, full abstraction.

For my parents, who refereed my earliest discourses on things (un)fair

Abstract

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*—which, roughly speaking, states that every sufficiently enabled component eventually proceeds—we can abstract away from these details without ignoring them completely. However, modeling fairness for communicating processes is especially difficult: synchronization requires the cooperation and active participation of multiple processes, and hence the enabledness of a process depends on the ability of other processes to synchronize with it.

This dissertation introduces a general framework for modeling fairness for communicating processes, based on the notion of *fair traces*. Intuitively, a fair trace is an abstract representation of a fair computation, providing enough structure to capture the important essence of the computation (e.g., the sequences of states encountered or the communications made along it) as well as any contextual information necessary for compositionality. Within this framework, the meaning of a command is simply the set of fair traces that correspond to its possible fair computations. For each construct of the language, we define a corresponding operation on trace sets that reflects its operational behavior.

The use of traces provides a strong connection between the language's operational semantics and its denotational semantics, allowing operational intuition to guide formal, syntax-directed reasoning. Moreover, this trace framework is remarkably robust. By varying the structure of the traces, we can construct several different semantics that reflect different types of fairness assumptions for the same language of communicating processes.

Acknowledgments

I arrived at CMU with only vague ideas of what I was getting myself into: I had no research experience and knew only that CMU had a "good" computer-science program. To my good fortune, I found a wonderful advisor and mentor in Stephen Brookes. Steve was extremely patient as I developed the necessary foundations for doing research, giving me both encouragement and the space to figure things out on my own. What I know about research I learned from him.

I thank the other members of my committee—Ed Clarke, Jeannette Wing, and Prakash Panangaden—for reading my dissertation in pieces and on short notice. Despite these working conditions, they provided me with several good suggestions and insightful comments. I also thank Prakash for fitting my defense into his busy traveling schedule.

Many people offered their friendship and helped make my years in Pittsburgh enjoyable. Even during the most stressful times, my officemates—Maria, Matt and Sasha—made coming into the office a pleasant prospect, providing advice, commiseration, and plenty of laughter. The Brew Crew meant many evenings of good beer and cider and conversation; extra thanks to Gary and Bonnie for letting us destroy their kitchen on a weekly basis. The Cache Cows endured an occasionally volatile coach and provided a great excuse to put work aside and get some fresh air; thanks to Jim for taking over the Mad Cows for one final season. Other diversions through the years were provided by Bob, who organized summer croquet games, and Wayne, who—despite his protestations of innocence—led many of us astray as necessary. Phoebe and James (independently) gave me many pep talks, for which I am most appreciative. Finally, Jonathan provided both technical and (far more importantly) emotional support on a daily basis: I thank him with all my heart.

Contents

List of Figures

Chapter 1

Introduction

Reasoning about deterministic sequential programs is a relatively straightforward task: at any particular instant, there is only one thread of control, and its next action can be determined solely from the current state. The situation changes dramatically, however, when we start considering parallel programs. When a system comprises several components running in parallel, its behavior depends not only on properties of the individual components but also on the *interactions* among them. Any attempt to model or reason formally about parallel-program behavior must take these interactions into account [Mil75]. However, the interactions in turn depend on external factors, such as the relative speed of processors or the implementation of the scheduler, whose details can be complex or (in many cases) unknown. As a result, reasoning formally about parallel systems often requires abstracting away from these details without ignoring them completely. One common and useful abstraction, and the subject of this dissertation, is ***fairness***.

This chapter provides a brief introduction to the concept of fairness and the reasons for (and the arguments against) adopting fairness assumptions to reason about the behavior of parallel programs. It also describes the goal of this dissertation—namely, the construction of a denotational framework for fair communicating processes—and provides a sketch of the approach taken. The chapter concludes with a roadmap for the remainder of the dissertation.

1.1 The Case for Fairness

To be precise, fairness is not a single abstraction but rather a collection of abstractions that all express the same underlying theme: *no component should forever be denied its rightful opportunity to proceed.* This simple theme applies to many settings; both Francez [Fra86] and Kwiatkowska [Kwi89] provide extensive surveys. In each setting, the role of the fairness assumption is to simplify the task of reasoning about program behavior.

When we reason about programs, we typically want to prove that a program satisfies some combination of *safety* and *liveness* properties. Safety properties are those properties that state that "nothing bad" ever happens: deadlock-freedom, data consistency, and mutual exclusion are all examples of safety properties. Safety properties correspond to program invariants: proving that a program satisfies a safety property amounts to showing that every reachable state satisfies the necessary invariant. As a result, fairness assumptions are not necessary for proving safety properties.

In contrast, liveness properties state that "something good" eventually happens, such as termination, the granting of a request, or the occurrence of a particular event. Fairness itself is a liveness property: the "something good" guaranteed to occur is a component's eventual progress. Whereas safety properties represent features of individual states, liveness properties reflect characteristics of *sequences* of states. As a result, they depend on the particular events that occur and the order in which those events occur. For example, consider the following simple shared-variable program:

$$\mathsf{x}{:=}0;\mathsf{y}{:=}1;(\mathsf{while}\ \mathsf{y} \neq 0\ \mathsf{do}\ \mathsf{x}{:=}\mathsf{x}+1 \parallel \mathsf{y}{:=}0).$$

To determine whether the program terminates, we need to know how the two parallel subcomponents are scheduled. For instance, if we know that the assignment $\mathsf{y}{:=}0$ occurs before the first evaluation of the conditional $\mathsf{y} \neq 0$, then we can deduce that the program terminates with the value of x set to 0. More generally, if we know that the assignment $\mathsf{y}{:=}0$ occurs between the n^{th} and $[n+1]^{st}$ evaluations of the conditional, then we can deduce that the program terminates with the value of x set to n.

What can we deduce about the program's termination without such detailed knowledge? As first glance, we can deduce very little: a biased scheduler could prevent the assignment $\mathsf{y}{:=}0$ from ever occurring, in which case the program does not terminate. However, because every reasonable scheduler is fair, we can abstract away from the scheduler details by assuming fairness. Simply knowing that the scheduler is fair—that is, that the scheduler will eventually let the assignment occur—allows us to deduce that the program terminates. In this case, fairness allows us to prove a liveness property that we otherwise could not prove. Of course, assuming fairness leaves us with very little information about the final value of x: the most that we can say is that the final value is a nonnegative integer. This example illustrates the phenomenon of *unbounded nondeterminism* that often arises with fairness: although the program is guaranteed to terminate, there is an infinite number of possible final values for x.

1.2 Fairness: Complications and Criticisms

The underlying theme of fairness is simple yet powerful: by assuming only general features of a scheduler, we can prove liveness properties of parallel programs. However, this simplicity 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 [Par79, AP86]. Moreover, the halting problem for programs with unbounded nondeterminism is Π^1_1-complete [Cha78], as is predicate-satisfiability under fairness assumptions [EC80]. The complications inherent to fairness have led some people to discard it altogether; several arguments have been made against adopting fairness [Dij88, Hoa78]. We address the most common criticisms here:

- *Fairness is an unrealistic assumption, because no scheduler should be expected to generate all fair computations.*

  This criticism reflects a common misunderstanding. A fair scheduler does not need to generate *all* fair computations; rather, it must generate *only* fair computations. A simple round-robin scheduler is fair, because it guarantees each process an opportunity to proceed. Indeed, any reasonable scheduler is fair: a parallel system that ignores arbitrary processes is not much use.

- *No finite experiment can distinguish a fair implementation from an unfair implementation, and hence the distinction between fair and unfair computations is meaningless.*

  Indeed, there is no way to distinguish a fair implementation from a unfair implementation simply by looking at some finite portion of a resulting computation: such is the nature of liveness properties in general. The fact is that we often want to reason about liveness properties such as the eventual granting of all resource requests or the guaranteed message delivery: these properties cannot be determined solely by examining finite portions of computations either. Even proving termination of deterministic sequential programs is undecidable, and yet very few would argue that the distinction between terminating computations and nonterminating computations is meaningless.

- *Fairness should not be part of a language definition: it is the programmer's responsibility to prove her programs correct without relying on a fair implementation.*

  Fairness does not need to be part of the language definition to be a useful abstraction. Indeed, different implementations of the same language may provide different levels of fairness. However, proving programs correct often involves proving that they satisfy certain liveness properties, which in turn requires knowing general features or precise details of the scheduler. Without fairness, the programmer must understand the underlying implementation in detail or write her own scheduler.

In summary, fairness is a useful and often necessary abstraction, in spite of the technical difficulties that it introduces. Whereas discarding fairness may avoid technical complications, it does not reduce the complexity of reasoning about parallel programs.

1.3 Thesis Scope

Communicating processes represent an important (and still relevant) paradigm for parallel-program implementation in which processes communicate with one another through synchronous or asynchronous message passing; this paradigm is reflected in (among others) CCS [Mil80], CSP [Hoa78, Hoa85], occam [INM84], Ada [Uni80], and even the widely accepted MPI (Message Passing Interface) standard [Mes94]. In this dissertation, I explore the problem of modeling fairness for synchronously communicating processes, developing a denotational framework that incorporates a variety of fairness assumptions for these processes.

Modeling fairness for communicating processes is more difficult than for shared-variable programs. In the shared-memory paradigm, processes communicate with one another through changes to the shared global state. To avoid inadvertent (and inconsistent) simultaneous accesses to the shared state, shared-memory programs emphasize mutual exclusion. Whether a given process is enabled depends only on the global state: a process's ability to make progress is independent of the status of the processes in parallel with it. In contrast, the emphasis in the communicating-process paradigm is on synchronization, which requires the active cooperation and participation of two (or possibly more) processes. As a result, a process's ability to make progress is no longer a local property: it depends on the ability of other processes to synchronize with it. No matter how determined a process is to perform a particular communication, and regardless of how benevolent the scheduler is, the communication can occur only if some other process can synchronize with it. This dependence on other processes for progress has important consequences for modeling fairness: determining whether a process is treated fairly depends on knowing not only what the particular process is trying to do but also on what types of actions the processes in parallel with it can perform.

Complicating the problem is the number of fairness assumptions that are applicable for communicating processes. Several different types of fairness have been considered for communicating processes (see, for example, [Fra86] and [KdR83]), each one reflecting a different type of obligation that we might wish to impose on the implementation. For example, in addition to expecting that every process makes progress, we might require that certain pairs of processes communicate with one another or that particular communications eventually occur. Each of these different fairness assumptions affects the allowable program behavior and impacts the corresponding semantic model in some way. Can we construct a semantic framework that accounts for these different assumptions in a unified way, making only the distinctions necessary for dealing with the underlying differences in assumptions?

1.3.1 Thesis approach

Traces have long been used to model concurrency [Par79, Bro96b, Hoa81, BHR84, BR84, Hen85, Jon94, Rus90, Jos92]. In this dissertation, I show that traces can be extended with additional contextual information to support compositional reasoning about *fair* concurrency. Intuitively, a trace is an abstract record of a program execution, capturing the important aspects (e.g., communication sequences or state changes) of the execution while abstracting away from unimportant details such as program syntax. By adding appropriate structure that represents fairness-related contextual information (e.g., information about the communications that could have occurred along the computation) to yield *fair traces*, we can model the fair behaviors of communicating processes in a compositional manner.

The contextual components of the fair traces are essential for modeling fairness accurately, because they provide information about the type of situations in which the given trace represents a fair computation. However, determining exactly what type of structure these components require can be difficult: the extent to which program contexts affect the perceived fairness of (sub)computations depends on the particular notion of fairness under consideration. Generally speaking, the fair computations of a parallel command $c_1\|c_2$ cannot be defined only in terms of the fair computations of c_1 and c_2. The problem is that, because synchronous communications require the cooperation and participation of more than one process, a given (sub)computation of c_1 can be either fair or unfair when made part of a larger computation of $c_1\|c_2$, depending on what type of synchronization opportunities the component c_2 provides.

As a result, it is necessary to consider "almost fair" computations, which can be considered fair under certain assumptions (i.e., in certain contexts). By introducing notions of *parameterized fairness*, we can make precise this notion of "almost fair". Roughly speaking, these parameterized forms of fairness capture the features of program contexts that affect the fair progress of processes, such as the communications enabled along a computation and the types of communications that blocked processes are trying to perform. These parameterized forms of fairness are essential for the denotational (i.e., compositional) characterization of fairness.

Once the appropriate structure for the fair traces has been determined, the meaning of a command is given by the set of fair traces that correspond to its computations. To characterize this semantic function denotationally, we define operations on trace sets that reflect the operational behavior of the language constructs. For example, the computations of the sequential composition $c_1;c_2$ in essence arise from appending computations of c_2 to computations of c_1. The trace set of the command $c_1;c_2$ likewise can be created by appending traces of c_2 to traces of c_1.

The most difficult language construct to model is parallel composition. Generally speaking, the computations of $c_1\|c_2$ arise from merging and interleaving computations of c_1 with computations of c_2. However, not all pairs of computations can be merged and still reflect meaningful computations: for example, the progress made by one component may affect the perceived fairness of the other component's actions. The role of the fair traces' contextual components is to provide information sufficient for determining which merges are meaningful; we let a predicate *mergeable* indicate such combinations. It is also important that the merges of the traces are *fair merges* [Par79]: a fair merge of traces φ_1 and φ_2 consumes all of φ_1 and all of φ_2. To this end, we define a relation *fairmerge* $\subseteq \Phi \times \Phi \times \Phi$ on fair traces that guarantees a fair merging and acknowledges the potential of synchronization between components. This relation must also perform the necessary bookkeeping to maintain accurate information in the traces' contextual components. Intuitively, the triple $(\varphi_1, \varphi_2, \varphi)$ is in *fairmerge* if and only if φ represents a fair computation that can be obtained by merging the fair computations represented by φ_1 and φ_2. With these definitions in hand, we define parallel composition on trace sets in the following way:

$$T_1 \| T_2 = \{\varphi \mid \varphi_1 \in T_1 \ \&\ \varphi_2 \in T_2 \ \&\ (\varphi_1, \varphi_2, \varphi) \in \mathit{fairmerge} \ \&\ \mathit{mergeable}(\varphi_1, \varphi_2, \varphi)\}.$$

The particular definitions of *mergeable* and *fairmerge* vary depending on both the language and notion of fairness under consideration. However, they play the same roles in each setting. Indeed, the semantic functions in general are very similar from fairness notion to fairness notion, because the operational intuition underlying the operations remains the same in each case; only the bookkeeping operations vary, reflecting their dependence on the trace structure.

In this dissertation, I concentrate on modeling synchronously communicating processes. However, this description of the framework is general enough to suit other paradigms as well. Brookes' transition trace semantics [Bro96b] for shared-variable programs is a simple example of this general framework in which no additional contextual information is needed. In Chapter 7, we see that the framework also accommodates a hybrid language of communicating processes that includes features of shared-variable parallelism.

1.3.2 Thesis contributions

The primary contribution of this dissertation is the trace framework: it provides a general, extendible, modular approach for constructing semantics that support reasoning about fair program behavior. This framework can be viewed as an extension to existing trace models, identifying and adding the additional structure necessary for incorporating fairness assumptions.

Throughout this dissertation, I demonstrate the general robustness of the framework by constructing several different semantics that incorporate different types of fairness assumptions. In particular, I focus on a simple language of communicating processes and construct different semantics that incorporate assumptions of strong fairness (*every process that is enabled infinitely often makes progress infinitely often*), strong channel fairness (*every communication channel on which communication is enabled infinitely often is used infinitely often*), and weak fairness (*every process that is enabled continuously makes progress eventually*). The resulting semantics show that the same general approach can be applied for different notions of fairness: the main differences between the different semantics are the bookkeeping operations necessary for maintaining the fairness-related contextual information. By comparing these semantics, we can see how differences in fairness assumptions affect the type of semantic structure necessary for reasoning about program behavior.

In the case of *strong fairness*, this approach yields *fully abstract* semantics for several natural notions of program behavior: a semantics is fully abstract with respect to a notion of observable behavior if it identifies precisely the terms that behave identically in all program contexts. The full abstraction results reflect the suitability of the chosen contextual components for modeling strong fairness: in each of these strongly fair semantics, the contextual components of the fair traces remain the same.

1.4 Organization of the Dissertation

The remainder of this dissertation proceeds as follows:

- In Chapter 2, I introduce an imperative language of communicating processes that is based on Hoare's CSP [Hoa78] and Milner's CCS [Mil80]. Using this language as a backdrop, I also discuss and generalize the notions of fairness typically considered for communicating processes: process fairness, channel fairness, guard fairness, and communication fairness.

  The particular syntax and operational semantics are not important from a technical perspective. However, they provide a convenient foundation for the technical details of subsequent chapters.

- In Chapter 3, I describe a denotational semantics that incorporates assumptions of *strong process fairness*, which requires every infinitely enabled process to proceed infinitely often.

  Because strongly fair computation cannot be characterized in an immediately compositional way, I first introduce a new notion of *parameterized strong fairness* that can be characterized compositionally. This parameterization guides the construction of the strongly fair trace semantics. The meaning of a program is a set of traces that correspond to its possible executions; each trace is augmented by certain enabling information that is necessary for achieving compositionality.

  The main artifact of the chapter is the strongly fair semantics. However, this chapter also serves as the first illustration of the general trace framework, and many of the subsequent chapters build on ideas introduced here.

- In Chapter 4, I discuss the property of *full abstraction*, a well-known objective criterion for judging the utility of a semantics. Intuitively, a fully abstract semantics makes precisely the right distinctions to support compositional reasoning about program behavior.

The strongly fair semantics of Chapter 3 is not fully abstract. However, by introducing appropriate closure conditions on trace sets, I show how the semantics can be adapted to yield full abstraction with respect to a natural notion of strongly fair behavior. Moreover, small changes in the trace structure and the selection of closure conditions yield several other fully abstract semantics for other notions of strongly fair program behavior.

Having a common underlying framework significantly simplifies the construction of the additional semantics. In particular, the contextual components of the traces (that is, the portion that relates to strong fairness) remain the same in each case and facilitate the presentation and understanding of each new model. Moreover, because the contextual components of traces remain the same, many of the necessary lemmas for full abstraction can also be reused, greatly simplifying the subsequent full-abstraction proofs.

- In Chapter 5, I construct a semantics that incorporates assumptions of *strong channel fairness*. Roughly speaking, strong channel fairness requires not only the progress of infinitely enabled processes but also the infinite use of every infinitely enabled communication channel.

  Once again, this semantics depends on a parameterization of fairness that can be characterized in a compositional manner. The channel-fair semantics requires significantly more structure than the process-fair semantics of the previous two chapters, and it is not fully abstract. I discuss this lack of full abstraction and hint how full abstraction might be achieved.

- In Chapter 6, I consider *weak process fairness*, which requires all *continuously* enabled processes to make progress. Weak fairness is much easier to implement than strong fairness, but it is extremely sensitive to both the nuances of the operational semantics and the order in which independent actions occur. As a result, weak fairness is much harder than strong fairness to model semantically for communicating processes. In particular, the task of determining when processes are enabled continuously requires significantly more structure than determining when they are enabled infinitely often.

  As it turns out, the resulting weakly fair semantics is very similar in structure to the channel-fair semantics of Chapter 5. This similarity is rather surprising, given that strong process fairness is simultaneously stronger than weak process fairness and weaker than strong channel fairness. I discuss the underlying reasons for this similarity.

- Chapter 7 is the final technical chapter of the dissertation. In it, I introduce a language of hybrid distributed process that combines features of both the shared-variable and the communicating-process paradigms. By combining Brookes' transition trace semantics for shared-variable programs [Bro96b] with my strongly fair semantics for communicating processes, I construct a semantics for this hybrid language that incorporates a combination of weak and strong fairness assumptions. Moreover, suitable closure conditions on trace sets again yield full abstraction, the proof of which is a straightforward combination of the full-abstraction proofs for the original, independent semantics.

The ease with which these two distinct semantics can be combined reflects the generality of the trace framework. Despite the underlying differences of the paradigms, the two types of trace semantics can be combined in an intuitively appealing way.

- Finally, I conclude with a summary of the contributions of the thesis, some connections to related work, and suggestions for future work.

Chapter 2

Communicating Processes

In this chapter, we introduce a representative language of communicating processes, related to Hoare's CSP and Milner's CCS, in which processes have private local states and communicate with one another only via synchronous message passing. The particular syntax and operational semantics of this language are uninteresting from a technical standpoint, but they provide a convenient reference for the discussion of the relevant issues. In particular, throughout this dissertation we will show how different types of fairness assumptions can be incorporated into semantics for this same language. By modeling a single language, we can focus better on the similarities and differences of the various fairness assumptions.

After giving the syntax and operational semantics of the language, we introduce the standard notions of fairness for communicating processes: process fairness, channel fairness, guard fairness, and communication fairness. These notions of fairness have typically been identified with CSP; because our language's syntax differs from CSP in certain respects, we generalize the definitions to suit our language as well.

2.1 A Language of Communicating Processes

For most of this dissertation, we shall consider a simple imperative language of communicating processes originally introduced in [Bro94] and based on Hoare's CSP [Hoa78, Hoa85] and Milner's CCS [Mil80]. As in occam [INM84], processes have disjoint local states and communicate with one another via named *channels*.

2.1.1 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;
- 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:

$$\begin{array}{rcl} g & ::= & h?i \mid h!e \\ gc & ::= & g \to c \mid gc_1 \,\square\, gc_2 \\ c & ::= & \mathsf{skip} \mid i{:=}e \mid c_1; c_2 \mid \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ b\ \mathsf{do}\ c \\ & & \quad \mid gc \mid c_1 \| c_2 \mid c \backslash h \end{array}$$

As is common, we often abbreviate the guarded command $g \to \mathsf{skip}$ simply as g. We also use the notation $\sum_{i=1}^{n}(g_i \to c_i)$ to abbreviate guarded commands of the form

$$(g_1 \to c_1) \,\square\, (g_2 \to c_2) \,\square\, \cdots \,\square\, (g_n \to c_n).$$

As in the original CSP, processes have disjoint local states. We therefore impose an additional syntactic constraint to ensure that processes can affect one another's behavior only through handshake communication. We require that, for every command of form $c_1 \| c_2$, c_1 and c_2 have disjoint free identifiers; that is,

$$\mathsf{fv}[\![c_1]\!] \cap \mathsf{fv}[\![c_2]\!] = \emptyset,$$

where $\mathsf{fv}[\![c]\!]$ is the set of free identifiers of c. The set $\mathsf{fv}[\![c]\!]$ can be defined by structural induction in the standard way (see Figure 2.1), under the reasonable assumption that $\mathsf{fv}[\![b]\!]$ and $\mathsf{fv}[\![e]\!]$ are defined for boolean and arithmetic expressions. Likewise, the set of channel names occurring free in c—written $\mathsf{fc}[\![c]\!]$—can be defined inductively, as in Figure 2.2.

$$
\begin{array}{rcl}
\mathsf{fv}[\![\mathsf{skip}]\!] & = & \emptyset \\
\mathsf{fv}[\![i{:=}e]\!] & = & \{i\} \cup \mathsf{fv}[\![e]\!] \\
\mathsf{fv}[\![c_1 ; c_2]\!] & = & \mathsf{fv}[\![c_1]\!] \cup \mathsf{fv}[\![c_2]\!] \\
\mathsf{fv}[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] & = & \mathsf{fv}[\![b]\!] \cup \mathsf{fv}[\![c_1]\!] \cup \mathsf{fv}[\![c_2]\!] \\
\mathsf{fv}[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!] & = & \mathsf{fv}[\![b]\!] \cup \mathsf{fv}[\![c]\!] \\
\mathsf{fv}[\![h?i]\!] & = & \{i\} \\
\mathsf{fv}[\![h!e]\!] & = & \mathsf{fv}[\![e]\!] \\
\mathsf{fv}[\![g \rightarrow c]\!] & = & \mathsf{fv}[\![g]\!] \cup \mathsf{fv}[\![c]\!] \\
\mathsf{fv}[\![gc_1 \,\square\, gc_2]\!] & = & \mathsf{fv}[\![gc_1]\!] \cup \mathsf{fv}[\![gc_2]\!] \\
\mathsf{fv}[\![c_1 \| c_2]\!] & = & \mathsf{fv}[\![c_1]\!] \cup \mathsf{fv}[\![c_2]\!] \\
\mathsf{fv}[\![c \backslash h]\!] & = & \mathsf{fv}[\![c]\!].
\end{array}
$$

Figure 2.1: Inductive definition of $\mathsf{fv}[\![c]\!]$.

$$
\begin{array}{rcl}
\mathsf{fc}[\![\mathsf{skip}]\!] & = & \emptyset \\
\mathsf{fc}[\![i{:=}e]\!] & = & \emptyset \\
\mathsf{fc}[\![c_1 ; c_2]\!] & = & \mathsf{fc}[\![c_1]\!] \cup \mathsf{fc}[\![c_2]\!] \\
\mathsf{fc}[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] & = & \mathsf{fc}[\![c_1]\!] \cup \mathsf{fc}[\![c_2]\!] \\
\mathsf{fc}[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!] & = & \mathsf{fc}[\![c]\!] \\
\mathsf{fc}[\![h?i]\!] & = & \{h\} \\
\mathsf{fc}[\![h!e]\!] & = & \{h\} \\
\mathsf{fc}[\![g \rightarrow c]\!] & = & \mathsf{fc}[\![g]\!] \cup \mathsf{fc}[\![c]\!] \\
\mathsf{fc}[\![gc_1 \,\square\, gc_2]\!] & = & \mathsf{fc}[\![gc_1]\!] \cup \mathsf{fc}[\![gc_2]\!] \\
\mathsf{fc}[\![c_1 \| c_2]\!] & = & \mathsf{fc}[\![c_1]\!] \cup \mathsf{fc}[\![c_2]\!] \\
\mathsf{fc}[\![c \backslash h]\!] & = & \mathsf{fc}[\![c]\!] - \{h\}.
\end{array}
$$

Figure 2.2: Inductive definition of $\mathsf{fc}[\![c]\!]$.

$$\langle \bullet, s\rangle \mathsf{term} \qquad \frac{\langle c_1, s_1\rangle \mathsf{term} \quad \langle c_2, s_2\rangle \mathsf{term}}{\langle c_1 \| c_2, s_1 \cup s_2\rangle \mathsf{term}} \text{ if } \mathsf{disjoint}(s_1, s_2) \qquad \frac{\langle c, s\rangle \mathsf{term}}{\langle c \backslash h, s\rangle \mathsf{term}}$$

Figure 2.3: The predicate term.

2.1.2 Operational semantics

A state is a finite partial function from identifiers to integers. Letting $\mathbb{Z}$ represent the set of integers, the set S of states can be defined as

$$S = [\mathsf{Ide} \rightharpoonup \mathbb{Z}].$$

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 $\mathsf{dom}(s)$, 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: $\mathsf{dom}(s_1) \cap \mathsf{dom}(s_2) = \emptyset$. In such cases, we write $\mathsf{disjoint}(s_1, s_2)$.

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 $\langle e, s\rangle \longrightarrow^* n$ to indicate that expression e in state s evaluates to value n. Implicit in this notation is the assumption that the free identifiers of e are included in the domain of s: that is, $\mathsf{fv}[\![e]\!] \subseteq \mathsf{dom}(s)$. We use a similar notation for the evaluation of boolean expressions, and we let $\mathbb{B} = \{\mathtt{tt}, \mathtt{ff}\}$ 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 [Plo83]. 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 place-holder $\bullet$ to represent termination, and allow configurations with forms such as $\langle \bullet, s\rangle$, $\langle \bullet \| c_2, s\rangle$ and $\langle \bullet \backslash h, s\rangle$. A configuration $\langle c, s\rangle$ is **terminal** if the predicate $\langle c, s\rangle \mathsf{term}$ can be proved from the axioms and inference rules in Figure 2.3.

A **label** λ is a member of the set

$$\Lambda = \{\varepsilon\} \cup \{h!n,\ h?n \mid h \in \mathsf{Chan} \;\&\; n \in \mathbb{Z}\}.$$

Every transition has a label indicating the type of atomic action involved: ε represents an internal action (e.g., assignment to a variable), $h!n$ represents the transmission of value n along channel h, and $h?n$ represents the receipt of value n from channel h. Two labels λ_1 and λ_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 a case, we write $\mathsf{match}(\lambda_1, \lambda_2)$. For a label λ, $\mathsf{chan}(\lambda)$ is the channel associated with λ; by convention, we define $\mathsf{chan}(\varepsilon) = \varepsilon$.

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

$$\frac{\langle c_1, s\rangle \xrightarrow{\lambda} \langle c_1', s'\rangle \quad \neg\langle c_1', s'\rangle\mathsf{term}}{\langle c_1; c_2, s\rangle \xrightarrow{\lambda} \langle c_1'; c_2, s'\rangle} \qquad \frac{\langle c_1, s\rangle \xrightarrow{\lambda} \langle c_1', s'\rangle\mathsf{term}}{\langle c_1; c_2, s\rangle \xrightarrow{\lambda} \langle c_2, s'\rangle}$$

$$\frac{\langle b, s\rangle \longrightarrow^* \mathtt{tt}}{\langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, s\rangle \xrightarrow{\varepsilon} \langle c_1, s\rangle} \qquad \frac{\langle b, s\rangle \longrightarrow^* \mathtt{ff}}{\langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, s\rangle \xrightarrow{\varepsilon} \langle c_2, s\rangle}$$

$$\frac{\langle b, s\rangle \longrightarrow^* \mathtt{tt}}{\langle \mathsf{while}\ b\ \mathsf{do}\ c, s\rangle \xrightarrow{\varepsilon} \langle c; \mathsf{while}\ b\ \mathsf{do}\ c, s\rangle} \qquad \frac{\langle b, s\rangle \longrightarrow^* \mathtt{ff}}{\langle \mathsf{while}\ b\ \mathsf{do}\ c, s\rangle \xrightarrow{\varepsilon} \langle \bullet, s\rangle}$$

Figure 2.4: Transition rules for sequential constructs.

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 λ, leading to the command c' in state s'. The transition rules for the sequential constructs are standard and appear in Figure 2.4.

The transition rules for guards and guarded commands appear in Figure 2.5. The guard $h?i$ represents the ability to receive a value for identifier i on channel h, and the guard $h!e$ represents the ability to transmit the value of expression e along channel h. The guarded command $g \to c$ is a command that, after performing the action associated with guard g, behaves like command c. The guarded command $gc_1 \Box gc_2$ represents a nondeterministic choice[1] between the guarded commands gc_1 and gc_2: on its first step, $gc_1 \Box gc_2$ can perform any action that either gc_1 or gc_2 can, and afterwards behaves like the chosen gc_i.

The transition rules for the parallel composition and channel restriction appear in Figure 2.6. The command $c_1 \| c_2$ represents the parallel composition of commands c_1 and c_2, and it can perform any action that either component can perform. Additionally, if one component can perform output and the other receive input on the same channel, then the two components can synchronize, resulting in a single ε-transition of the parallel command; such handshakes correspond to "distributed" assignments. Finally, the command $c \backslash h$ behaves like the command c, except that communication on the channel h is restricted to handshakes.

In many situations, we will be interested in the general properties of a communication (i.e., whether it is input or output, and on which channel it occurs) without caring for the particular value transmitted. In such cases, we consider the set of *directions*. A **direction** is a member of

[1]This choice is an *external choice*, in that it can be influenced by the environment.

$$\langle h?i, s\rangle \xrightarrow{h?n} \langle \bullet, [s \mid i = n]\rangle \text{ for each } n \in \mathbb{Z} \qquad \frac{\langle e, s\rangle \longrightarrow^* n}{\langle h!e, s\rangle \xrightarrow{h!n} \langle \bullet, s\rangle}$$

$$\frac{\langle g, s\rangle \xrightarrow{\lambda} \langle \bullet, s'\rangle}{\langle g \rightarrow c, s\rangle \xrightarrow{\lambda} \langle c, s'\rangle}$$

$$\frac{\langle gc_1, s\rangle \xrightarrow{\lambda} \langle c, s'\rangle}{\langle gc_1 \Box gc_2, s\rangle \xrightarrow{\lambda} \langle c, s'\rangle} \qquad \frac{\langle gc_2, s\rangle \xrightarrow{\lambda} \langle c, s'\rangle}{\langle gc_1 \Box gc_2, s\rangle \xrightarrow{\lambda} \langle c, s'\rangle}$$

Figure 2.5: Transition rules for guards and guarded commands.

$$\frac{\langle c_2, s_2\rangle \xrightarrow{\lambda} \langle c_2', s_2'\rangle}{\langle c_1 \| c_2, s_1 \cup s_2\rangle \xrightarrow{\lambda} \langle c_1 \| c_2', s_1 \cup s_2'\rangle} \text{ if } \mathsf{disjoint}(s_1, s_2)$$

$$\frac{\langle c_1, s_1\rangle \xrightarrow{\lambda_1} \langle c_1', s_1'\rangle \quad \langle c_2, s_2\rangle \xrightarrow{\lambda_2} \langle c_2', s_2'\rangle}{\langle c_1 \| c_2, s_1 \cup s_2\rangle \xrightarrow{\varepsilon} \langle c_1' \| c_2', s_1' \cup s_2'\rangle} \text{ if } \mathsf{disjoint}(s_1, s_2) \;\&\; \mathsf{match}(\lambda_1, \lambda_2)$$

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

Figure 2.6: Transition rules for parallel constructs.

the set

$$\Delta = \{h!, h? \mid h \in \mathsf{Chan}\}.$$

Occasionally we will also be concerned with the extended set of directions

$$\Delta^+ = \Delta \cup \{\varepsilon\}.$$

We write $\mathsf{match}(d_1, d_2)$ when the directions d_1 and d_2 match: that is, whenever one has the form $h!$ and the other $h?$ for some channel h. We often write $\bar{d}$ for the unique direction that matches d, and we write $\overline{X}$ for the set of matching directions of the set X: $\overline{X} = \{\bar{d} \mid d \in X\}$. Similarly, we write $\mathsf{match}(X_1, X_2)$ for sets $X_1, X_2 \subseteq \Delta$ if there exist directions $d_1 \in X_1$ and $d_2 \in X_2$ such that $\mathsf{match}(d_1, d_2)$. For any direction d, $\mathsf{chan}(d)$ is the channel associated with d. For a label λ, $\mathsf{dir}(\lambda)$ is the direction associated with λ. Again, by convention, we let $\mathsf{dir}(\varepsilon) = \varepsilon$.

A configuration $\langle c, s\rangle$ is **enabled** if there exists a transition $\langle c, s\rangle \xrightarrow{\lambda} \langle c', s'\rangle$ for some command c', state s' and label λ. A configuration is **blocked** (or **disabled**) if it is neither enabled nor terminal. We write $\langle c, s\rangle\mathsf{dead}$ to indicate that the configuration $\langle c, s\rangle$ is blocked. We define a set $\mathsf{inits}(c,s)$ that contains the directions (possibly including ε) that can be used on transitions from the configuration $\langle c,s\rangle$:

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

A **computation** is a finite or infinite, maximal sequence of transitions; a **partial computation** is a finite sequence of transitions. We call a finite computation ending in a terminal configuration **successful** and one ending in a blocked configuration **deadlocked**.

2.1.3 Processes

As a program executes, it has one or more *processes* associated with it; each process is a thread of control in that execution. At every step along a computation, the active processes can be determined from the syntactic portion of the current configuration. Although processes are technically features of program executions, it is convenient to associate them with program syntax. For example, in the command

$$\mathsf{b!0} \parallel (\mathsf{a?x} \Box \mathsf{a!1}),$$

we say that there are two processes: $\mathsf{b!0}$ and $(\mathsf{a?x} \Box \mathsf{a!1})$.

The number of processes can increase or decrease dynamically as a program executes. For example, the following computation has one active process initially, two active processes after the first transition, and no active processes in the final configuration:

$$\begin{aligned}\langle \mathsf{a!x} \to (\mathsf{y}{:=}1 \| \mathsf{x}{:=}1), [\mathsf{x}=0,\mathsf{y}=0]\rangle &\xrightarrow{\mathsf{a!0}} \langle \mathsf{y}{:=}1 \| \mathsf{x}{:=}1, [\mathsf{x}=0,\mathsf{y}=0]\rangle \\ &\xrightarrow{\varepsilon} \langle \mathsf{y}{:=}1 \| \bullet, [\mathsf{x}=1,\mathsf{y}=0]\rangle \\ &\xrightarrow{\varepsilon} \langle \bullet \| \bullet, [\mathsf{x}=1,\mathsf{y}=1]\rangle.\end{aligned}$$

A process is *enabled* in a given configuration if it can contribute to a transition from that configuration. That is, a process is enabled if it can perform an internal action, if it can perform an external communication along an unrestricted channel, or if it is able to synchronize with some other process. As a result, whether a process is enabled can depend upon the status of the processes running in parallel with it: a process trying to communicate along a restricted channel is enabled only if another process can synchronize with it.

Example 2.1.1 Consider the program

$$(Q_1 \| Q_2 \| Q_3 \| Q_4 \| Q_5)\backslash\mathsf{a}\backslash\mathsf{b},$$

where the processes Q_1, Q_2, Q_3, Q_4 and Q_5 are defined as follows:

$$\begin{array}{rcl} Q_1 & \equiv & \mathsf{x{:=}x} - 1, \\ Q_2 & \equiv & \mathsf{a?y} \rightarrow \mathsf{y{:=}y} + 1, \\ Q_3 & \equiv & \mathsf{a!z} \rightarrow \mathsf{skip}, \\ Q_4 & \equiv & \mathsf{b!w} \rightarrow \mathsf{w{:=}w} + 1, \\ Q_5 & \equiv & (\mathsf{b!5} \rightarrow \mathsf{skip}) \,\square\, (\mathsf{c!5} \rightarrow \mathsf{skip}). \end{array}$$

1. Process Q_1 is enabled, because it can perform an internal action that decrements the value of x.

2. Processes Q_2 and Q_3 are both enabled, because they are able to synchronize with one another along channel a.

3. Process Q_4 is disabled: its only potential transition requires synchronization on channel b, and no other process can synchronize with it.

4. Process Q_5 is enabled, because it can communicate along channel c. $\diamond$

2.2 Fairness for Communicating Processes

Most of the common notions of fairness—and all of the ones discussed in this dissertation—share the same general form:

> Every entity that is enabled sufficiently often will eventually make progress.

Varying the interpretations of *entity* and *sufficiently often* leads to different notions of fairness. In the context of communicating processes, there are many different kinds of entity to consider, each choice leading to a different notion of fairness. In particular, Francez [Fra86] and Kuiper and de Roever [KdR83] have collectively identified a hierarchy of fairness notions for CSP that includes the following forms of fairness: process fairness, channel fairness, guard fairness, and communication fairness. Each of these fairness notions have weak and strong varieties, which differ in the interpretation of *sufficiently often*: weak forms of fairness are concerned with *continuously* enabled entities, whereas strong forms of fairness are concerned with the *infinitely* enabled entities.

The hierarchy of fairness assumptions for CSP is sketched in Figure 2.7. Each link of form $A \rightarrow B$ can be interpreted as "fairness notion A is subsumed by fairness notion B" or (equivalently) "Every B-fair computation is also A-fair." For example, every weakly process-fair computation is also weakly channel-fair, as well as strongly process-fair. Moreover, for

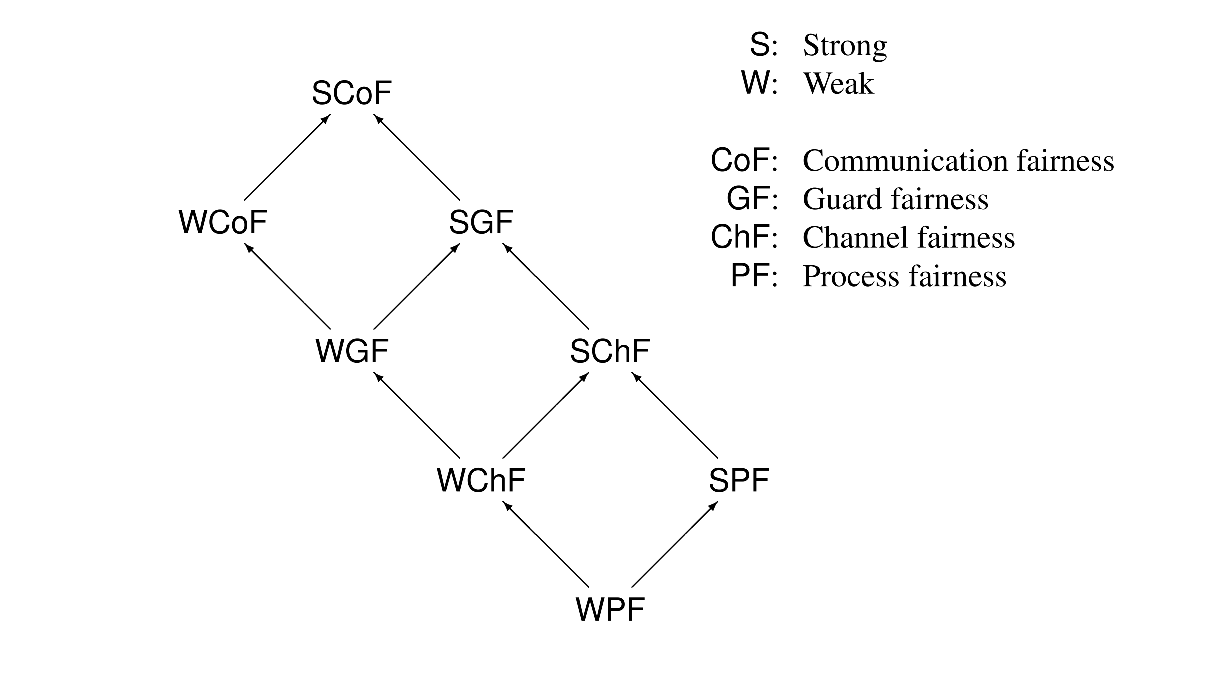

Figure 2.7: The hierarchy of fairness notions for CSP.

each link $A \rightarrow B$, there is a program that always terminates under the assumption of B-fairness but has nonterminating computations under the weaker assumption of A-fairness [KdR83].

In this section, we define each of these fairness notions, first as defined originally for CSP and then adapted to suit the more general syntax of our communicating processes. Process and channel fairness figure prominently in subsequent chapters. Guard and communication fairness—which are more strongly tied to program syntax—seem less reasonable as abstractions, because they are much more impractical to implement: they require the scheduler to keep track of all (syntactic) communication points of a program and to ensure that each communication point enabled sufficiently often is used sufficiently often. As a result, we discuss guard and communication fairness only in this section, to provide a more complete overview of the hierarchy of fairness notions.

2.2.1 Process fairness

Process fairness is by far the most common notion from this hierarchy, due to its applicability to contexts besides communicating processes and to the relative ease of implementing process-fair schedulers.

Weak (process) fairness (also known as **justice** [LPS81]) states that every process enabled *continuously* will eventually make progress. Intuitively, weak fairness ensures that the sched-

$$\begin{array}{l} \mathsf{while\ true\ do}\ (\\ \quad \textit{non-critical-section}_i; \\ \quad \mathsf{sem}?x_i;\ \textit{critical-section}_i; \\ \quad \mathsf{sem!1} \\ ) \end{array}$$

Figure 2.8: The processes Q_i.

uler will never forget a process forever. It is straightforward to implement weak fairness as a scheduling policy, using a simple round-robin scheduling queue.

Despite the ease of implementing weak fairness, sometimes a stronger notion of fairness is warranted. For example, consider the use of a semaphore *sem*, which we can implement as the process

$$Sem \equiv \mathsf{while\ true\ do\ (sem!1} \rightarrow \mathsf{sem?s}),$$

to prevent processes Q_1 and Q_2 (sketched in Figure 2.8) from being in their critical sections at the same time. In this scenario, it is reasonable to expect that each of Q_1 and Q_2 will eventually enter its critical section. However, weak fairness is not a strong enough assumption to ensure such an outcome. A process waiting for the semaphore becomes disabled whenever the other process successfully enters its critical section. A computation in which Q_1 repeatedly enters its critical section while Q_2 never gains admission to its critical section is weakly fair, because Q_2 is not enabled *continuously* but only *infinitely often*.

Another problem with weak process fairness for communicating processes is that, in the vocabulary of Apt and colleagues, it is not *equivalence robust* [AFK88]. That is, weak fairness is very sensitive to the order in which independent actions are scheduled. For example, consider the following program

$$(\mathsf{b!0} \parallel Q_3 \parallel Q_4)\backslash\mathsf{b},$$

where the processes Q_3 and Q_4 are defined as follows:

$$Q_3 \equiv \mathsf{while\ true\ do\ (b?x} \Box \mathsf{a!1}), \qquad Q_4 \equiv \mathsf{while\ true\ do\ (b?y} \Box \mathsf{a!2}).$$

In the following computation, the process b!0 makes no progress, while Q_3 and Q_4 repeatedly perform the same sequence of actions:

$$\begin{array}{lll} \langle(\mathsf{b!0} \parallel Q_3 \parallel Q_4)\backslash\mathsf{b}, s\rangle & \xrightarrow{\varepsilon} & \langle(\mathsf{b!0} \parallel (\mathsf{b?x}\Box\mathsf{a!1}); Q_3 \parallel Q_4)\backslash\mathsf{b}, s\rangle \\ & \xrightarrow{\varepsilon} & \langle(\mathsf{b!0} \parallel (\mathsf{b?x}\Box\mathsf{a!1}); Q_3 \parallel (\mathsf{b?y}\Box\mathsf{a!2}); Q_4)\backslash\mathsf{b}, s\rangle \\ & \xrightarrow{\mathsf{a!1}} & \langle(\mathsf{b!0} \parallel Q_3 \parallel (\mathsf{b?y}\Box\mathsf{a!2}); Q_4)\backslash\mathsf{b}, s\rangle \\ & \xrightarrow{\mathsf{a!2}} & \langle(\mathsf{b!0} \parallel Q_3 \parallel Q_4)\backslash\mathsf{b}, s\rangle \\ & \xrightarrow{\varepsilon} & \cdots \end{array}$$

This computation is weakly process-fair, because the process $\mathsf{b!0}$ does not have synchronization enabled continuously. In contrast, consider the following computation, in which processes Q_3 and Q_4 make exactly the same transitions as in the preceding computation, but the order in which the components' transitions are interleaved varies:

$$\begin{array}{rcl}
\langle(\mathsf{b!0} \parallel Q_3 \parallel Q_4)\backslash\mathsf{b}, s\rangle & \xrightarrow{\varepsilon} & \langle(\mathsf{b!0} \parallel (\mathsf{b?x}\Box\mathsf{a!1}); Q_3 \parallel Q_4)\backslash\mathsf{b}, s\rangle \\
& \xrightarrow{\varepsilon} & \langle(\mathsf{b!0} \parallel (\mathsf{b?x}\Box\mathsf{a!1}); Q_3 \parallel (\mathsf{b?y}\Box\mathsf{a!2}); Q_4)\backslash\mathsf{b}, s\rangle \\
& \xrightarrow{\mathsf{a!1}} & \langle(\mathsf{b!0} \parallel Q_3 \parallel (\mathsf{b?y}\Box\mathsf{a!2}); Q_4)\backslash\mathsf{b}, s\rangle \\
& \xrightarrow{\varepsilon} & \langle(\mathsf{b!0} \parallel (\mathsf{b?x}\Box\mathsf{a!1}); Q_3 \parallel (\mathsf{b?y}\Box\mathsf{a!2}); Q_4)\backslash\mathsf{b}, s\rangle \\
& \xrightarrow{\mathsf{a!2}} & \langle(\mathsf{b!0} \parallel (\mathsf{b?x}\Box\mathsf{a!1}); Q_3 \parallel Q_4)\backslash\mathsf{b}, s\rangle \\
& \xrightarrow{\varepsilon} & \langle(\mathsf{b!0} \parallel (\mathsf{b?x}\Box\mathsf{a!1}); Q_3 \parallel (\mathsf{b?y}\Box\mathsf{a!2}); Q_4)\backslash\mathsf{b}, s\rangle \\
& \xrightarrow{\mathsf{a!1}} & \cdots
\end{array}$$

This computation is not weakly process-fair, because the process $\mathsf{b!0}$ is enabled for synchronization continuously from the second configuration onwards. Thus weak process fairness relies not only on the actions of the components running in parallel but also on the manner in which those actions are scheduled.

As an alternative to weak fairness, **strong (process) fairness** states that every *infinitely enabled* process makes progress *infinitely often*. Strong fairness is equivalence robust, and hence does not depend on the order in which individual transitions are scheduled. As a result, strong process fairness is a much more natural notion of fairness to consider for communicating processes.

Because strong fairness reflects a stronger expectation of scheduler behavior, it is more difficult than weak fairness to implement as a scheduling policy. One way to implement strong fairness is to employ a priority queue scheme involving two process queues A and B. All processes originate in the lower priority queue (B), which behaves like the simple round-robin scheduler for weak fairness. However, if a process cycles through this queue too many times (for some previously determined value of *too many*) without making progress, it transfers to the higher priority queue (A). Processes in A are given preference whenever they have transitions enabled, and they retain their position in A until they make progress, at which point they return to the end of B. In particular, a process in queue A is scheduled immediately upon becoming enabled (assuming it has the highest priority among all enabled processes in queue A.) Because processes in A are given preference until they make progress, a process can fail to make infinite progress only if it becomes permanently disabled. A scheduler that implements this policy for some fixed value of *too many* is strongly fair, because every execution that it generates is strongly fair. Strong fairness is the abstraction that lets us ignore the specific value of *too many*.

2.2.2 Channel fairness

Definitions for channel fairness appear in both [Fra86] and [KdR83]. Although the two definitions differ, both formulations are intrinsically tied to the syntax of original CSP. In this subsection, we present Francez's definition and adapt it to suit our language. Kuiper and de Roever's definition of channel fairness—which coincides with what Francez terms *communication fairness*—is discussed in Subsection 2.2.4.

In the original CSP, processes have names and communicate by name, so that (for example) the process Q_i uses the guard $Q_j!e$ to represent its willingness to transmit the value of expression e to process Q_j. Similarly, the process Q_j uses the guard $Q_i?x$ to indicate its willingness to receive a value for identifier x from Q_i. As a result, Francez interprets a channel as simply a pair of processes, and he defines strong channel fairness as the following assumption:

> Every pair of processes that are infinitely often able to synchronize with one another will do so infinitely often.

This definition for channel fairness includes an implicit minimal liveness assumption [OL82]: a process will never block if it can perform an internal action such as assignment.

Every strongly channel-fair computation is also strongly process-fair, because channel fairness ensures the eventual progress of every infinitely enabled process. Every infinitely enabled process has either infinitely many opportunities for internal actions or infinitely many opportunities to synchronize with other processes. In the former case, the minimal liveness assumption ensures that the process makes progress. In the latter case, there must be at least one process with which the process has infinitely many opportunities to synchronize, and channel fairness ensures that the synchronization happens.

Because CSP processes communicate by name, each channel corresponds precisely to a pair of processes: only two processes communicate along any given channel, and only one channel is used between any two processes. In our language, any number of processes may communicate along a given channel, and two processes may communicate along any number of channels. As a result, it is possible for a particular channel to be used infinitely often and yet for another process to become blocked while trying to use that same channel. For example, the program

$$[\ (\mathsf{while\ true\ do\ a!0}) \parallel (\mathsf{while\ true\ do\ a?x}) \parallel \mathsf{a!2}\ ]\backslash\mathsf{a}$$

has an infinite computation in which the channel a is used infinitely often and yet the process $\mathsf{a!2}$ remains blocked.

This example raises an interesting question: should an infinitely enabled process be allowed to block along a strongly channel-fair computation? If we answer *yes*, then we must forfeit the "hierarchy" concept of fairness notions, because process fairness will no longer be subsumed by channel fairness. If we answer *no*, then we must build strong process fairness into our notion of strong channel fairness. We take the latter approach and incorporate strong fairness into our definition of strong fairness; this choice not only preserves the fairness hierarchy but also satisfies the obligation to include a minimal liveness assumption.

Bearing these considerations in mind, we arrive at the following generalized definition for strong channel fairness. A computation is **strongly channel-fair** if it satisfies the following two conditions:

- Every process enabled infinitely often makes progress infinitely often.
- Every channel on which communication is enabled infinitely often is used infinitely often.

This definition of strong channel fairness generalizes the Francez definition, while preserving the notion of channel fairness as it applies to CSP programs. In particular, every CSP program P can be translated into a program P' in our language in a straightforward manner. In each case, the (Francez-defined) channel-fair behaviors of P correspond precisely to the (generalized) channel-fair behaviors of P'.

If we view different channels as representing different types of messages, then channel fairness ensures that every type of message that is infinitely often deliverable gets delivered infinitely often. Implementing strong channel fairness requires a mechanism similar to that described for strong process fairness, suitably updated to ensure that infinitely enabled channels are used infinitely often.

To understand the extra strength over process fairness provided by channel fairness, consider the program

$$(P \| Q \| R) \backslash \mathsf{a} \backslash \mathsf{b} \backslash \mathsf{c},$$

where the processes P, Q, and R are defined as in Figure 2.9. (Following [AFK88], we assume for now that communications are possible only when all three processes are inside their loops. We impose this assumption only to simplify the exposition here; we remove this assumption in subsequent chapters. Moreover, we return to this matter in Chapter 5, particularly in Examples 5.4.2 and 5.4.3.) Termination of the program cannot be guaranteed under strong process fairness: it is perfectly acceptable for each of P and Q to communicate only with process R, each doing so infinitely often. However, in any infinite computation, synchronization is enabled infinitely often on each of the channels a, b and c. As a result, in any channel-fair computation, process P must eventually transmit the value 0 along channel a, an action that eventually leads to the termination of the entire program.

$$\begin{array}{l} \mathsf{while}\ (\mathsf{x} \neq 0)\ \ \mathsf{do} \\ \quad (\mathsf{a!0} \rightarrow \mathsf{x{:=}0} \ \Box \ \mathsf{b!1} \rightarrow \mathsf{skip}) \end{array}$$

(a) Process P

$$\begin{array}{l} \mathsf{n{:=}1;} \\ \mathsf{while}\ (\mathsf{w} \neq 0)\ \ \mathsf{do} \\ \quad (\mathsf{a?w} \rightarrow \mathsf{c!w} \ \Box \ \mathsf{c!n} \rightarrow \mathsf{n{:=}n+1}) \end{array}$$

(b) Process Q

$$\mathsf{while}\ (\mathsf{v} \neq 0)\ \ \mathsf{do}\ (\mathsf{c?v} \rightarrow \mathsf{skip} \ \Box \ \mathsf{b?v} \rightarrow \mathsf{skip})$$

(c) Process R

Figure 2.9: Channel fairness example.

2.2.3 Guard fairness

Guard fairness places even more restrictions on what types of computations can be considered fair. Informally, strong guard fairness states that every *guard* that is enabled infinitely often will be chosen infinitely often.[2] A guard is enabled in a given configuration if it can contribute to a transition from that configuration. Hence, a guard of process P is enabled if it involves communication on an unrestricted channel or if it involves communication on a restricted channel and a "matching" guard of another process is also enabled. For example, the guard $\mathsf{a!0}$ is enabled in the configurations

$$\langle \mathsf{a!0} \Box \mathsf{b!0}, s \rangle \qquad \text{and} \qquad \langle (\mathsf{a!0} \| \mathsf{a?x}) \backslash \mathsf{a}, s \rangle,$$

but not in the configuration

$$\langle (\mathsf{a!0} \Box \mathsf{b?x}) \backslash \mathsf{a}, s \rangle.$$

Strong guard fairness provides a stronger assumption than strong channel fairness, as illustrated by the following example. Consider the program

$$(P' \| Q \| R) \backslash \mathsf{a} \ \backslash \mathsf{b} \ \backslash \mathsf{c},$$

where Q and R are as defined in Figure 2.9 and P' is defined in Figure 2.10. (We again suppose that communications occur only when all three processes are inside their loops.) This program does not always terminate under strong channel fairness. Although channel a must be used infinitely often along any infinite computation, it is permissible under channel fairness for the $\mathsf{a!0}$ guard of P' to be ignored while the guard $\mathsf{a!1}$ synchronizes continually with Q's $\mathsf{a?n}$ guard. In such a computation, none of the variables x, w, or v ever gets set to 0, and hence the program never terminates. In contrast, under strong guard fairness, the guard $\mathsf{a!0}$ must eventually be involved in a handshake communication with $\mathsf{a?w}$, leading to termination of the program.

[2] To be precise, we also assume a minimal liveness property that ensures that no process becomes stuck in a configuration in which it can perform an internal action will block.

$$\begin{array}{l} \mathsf{while}\ (\mathsf{x} \neq 0)\ \ \mathsf{do} \\ \quad (\mathsf{a!0} \rightarrow \mathsf{x} = 0 \,\Box\, \mathsf{a!1} \rightarrow \mathsf{skip} \,\Box\, \mathsf{b!1} \rightarrow \mathsf{skip}) \end{array}$$

Figure 2.10: The process P'.

2.2.4 Communication fairness

Communication fairness[3] provides an even stronger assumption than guard fairness. Informally, communication fairness states that every communication enabled infinitely often will occur infinitely often. For CSP, a communication corresponds to two "matching" guards, which necessarily appear in two processes. Thus communication fairness for CSP can be stated as follows:

> For every pair of processes Q_i and Q_j, and for every pair $\langle g_i, g_j\rangle$ of (syntactically) matching guards from the two processes, if g_i and g_j are *jointly enabled* infinitely often, then they will synchronize infinitely often.

The notion of communication fairness—like guard fairness—has a very strong syntactic flavor: a scheduler must be able to distinguish two separate occurrences of the same guard in a program. The syntactic requirements behind this fairness notion seem inappropriate for a practical abstraction, and hence we will not discuss communication fairness in the rest of this dissertation. However, to complete the overview of the hierarchy, we introduce an appropriate generalization of communication fairness for our language.

The interpretation of *communication* is trickier for our language than for CSP, again because any number of processes may communicate along a given channel. In particular, we need to consider not only synchronizations among the processes that we know about but also possible interactions with the external environment. For example, compare the program

$$P \equiv (\mathsf{while\ true\ do}\ (\mathsf{a?x} \rightarrow \mathsf{skip}\ \Box \mathsf{a?y} \rightarrow\ \mathsf{skip}) \parallel \mathsf{while\ true\ do\ a!0})\backslash \mathsf{a},$$

which is (in effect) a CSP program translated directly into our language, with the program

$$P' \equiv \mathsf{while\ true\ do}\ (\mathsf{a?x} \rightarrow \mathsf{skip}\ \Box \mathsf{a?y} \rightarrow\ \mathsf{skip}) \parallel \mathsf{while\ true\ do\ a!0}.$$

For program P, the synchronizations between the guards $\mathsf{a?x}$ and $\mathsf{a!0}$ and between the guards $\mathsf{a?y}$ and $\mathsf{a!0}$ are the only possible communications. Thus every strongly communication-fair computation of program P should include infinitely many synchronizations on each pair. In contrast, program P' also permits three types of external communication: reading a value into x, reading a value into y, and transmitting the value 0. These external communications must also

[3] Kuiper and de Roever call this notion of fairness *channel fairness*.

```
n:=1;
while (w ≠ 0) do
    (a?w → c!w
      □ c!n → n:=n+1
      □ a?n → c!1)
```

Figure 2.11: The process Q'.

occur infinitely often along any strongly communication-fair computation of P', to represent the potential for communication with processes placed in parallel with P'.

We therefore introduce the following generalized notion of strong communication fairness. A computation is considered **strongly communication-fair** if it satisfies the following two conditions:

- Every handshake communication enabled infinitely often is chosen infinitely often.
- Every external action enabled infinitely often is chosen infinitely often.

When only closed programs (i.e., programs having no free channels) are considered, this definition of communication fairness coincides with the original notion of communication fairness introduced for CSP-style programs.

To distinguish communication fairness from guard fairness, consider the program

$$(P' \| Q' \| R)\backslash \mathsf{a} \backslash \mathsf{b} \backslash \mathsf{c},$$

where processes P' and R are as defined previously, and process Q' is defined as in Figure 2.11. (Once again, we assume that communications occur only when all three processes are inside their loops.) Under strong guard fairness, the program does not necessarily terminate. Each of the guards a!0, a!1, a?w and a?n must be used infinitely often in any infinite computation, but it is permissible for the two guards a!0 and a?n to synchronize only with one another, and likewise for the guards a!1 and a?w. In such an execution, the value 0 will never be transmitted to process R in such a way that the value of w gets set to 0. In contrast, under strong communication fairness, each of the guards a!0 and a!1 must synchronize with each of the guards a?w and a?n, resulting in the eventual termination of the program.

As the preceding discussion illustrates, the choice of fairness assumption affects what we can prove about program behavior: for example, there are programs that necessarily terminate under assumptions of strong channel fairness but may not terminate under strong process fairness. In the next several chapters, we see how the choice of fairness assumption also affects the semantic structure that is necessary for modeling fair behavior. We concentrate on three of these fairness assumptions: strong process fairness, strong channel fairness, and weak process fairness. We show how the framework adapts for each fairness notion, discussing the differences in semantic structure for each case. Perhaps surprisingly, the complexity of the semantic structure for a given notion of fairness is not linked directly to that notion's place in the hierarchy: as we shall see, strong process fairness is much simpler to model than either strong channel fairness or weak process fairness, despite falling between them in the hierarchy.

Chapter 3

Strong Process Fairness

In this chapter, we show how assumptions of strong process fairness can be incorporated into the general denotational framework described in Section 1.3. Modeling fairness in a compositional way is tricky, because the fairness of a subcomponent is context-dependent: whether a process can become blocked along a fair computation depends on the processes running in parallel with it. To model this dependence accurately, we must first introduce a parameterized form of strong fairness that take contexts into account.

After introducing parameterized strong fairness, we show how fair computations can be represented by traces, and we construct a denotational semantics based on these traces that incorporates assumptions of strong process fairness. This strongly fair semantics first appeared in [BO95], with a slightly different formulation. The chapter concludes with some simple examples illustrating how the semantics can be used to reason about program behavior.

3.1 Parameterized Strong Fairness

The enabledness of a process depends upon the context in which it appears. This contextual dependency has important consequences for any attempt to define fair computations in a compositional way. For example, consider the program

$$C \equiv (C_1 \| (C_2 \| C_3)) \backslash \mathsf{a} \backslash \mathsf{b},$$

where C_1, C_2 and C_3 are defined as follows:

$$C_1 \equiv \mathsf{while\ true\ do\ a?x}, \quad C_2 \equiv \mathsf{while\ true\ do\ a!0}, \quad C_3 \equiv \mathsf{while\ true\ do\ (b!0 \rightarrow a!1)}.$$

Any compositional treatment of fairness must allow the fair computations of C to be defined in terms of the fair computations of C_1 and $C_2 \| C_3$. In turn, the fair computations of $C_2 \| C_3$

must be defined in terms of the fair computations of C_2 and C_3. Because C_2 and C_3 are both enabled infinitely often along any computation of $C_2||C_3$, every strongly fair computation of $C_2||C_3$ must contain infinitely many outputs along each of the channels a and b. When $C_2||C_3$ is placed in the larger context of program C, however, the process C_3 becomes blocked when trying to perform output on channel b: communication on the channel is restricted, and no matching input is ever available. In contrast, channel a is also restricted in this context, but C_2 is repeatedly enabled for synchronization with C_1. Thus the program C has an infinite, strongly fair execution in which C_3 becomes permanently blocked, but none in which C_1 or C_2 ever becomes permanently blocked.

This example highlights two problems that arise in trying to characterize strongly fair computations in a compositional way. First, the strongly fair computations of a command cannot always be determined solely from the strongly fair computations of its component commands. In the preceding example, for instance, the strongly fair computations of C could not be determined solely from the strongly fair computations of C_1 and $C_2||C_3$. In particular, simply omitting the occurrences of channel b that appear along the fair computations of $C_2||C_3$ would lead to impossible computations for the larger command: each action a!1 that appears along the fair computations of $C_2||C_3$ is possible *only* when the action b!0 appears first. Second, the restricted channels alone are insufficient for identifying which subcommands will be enabled along any given computation: even though communication was restricted on channel a, C_2 could make continual progress by synchronizing with C_1 infinitely often.

To address these problems, we introduce generalized notions of enabledness and fairness, parameterizing each by a set of directions representing fairness constraints. In effect, we can talk about "almost blocked" configurations and "almost fair" computations, and the sets of directions provide a precise interpretation of "almost". Moreover, these sets of directions provide a description of those program contexts[1] $P[-]$ for which the "almost fair" computations will represent the transitions of c in a truly fair computation of $P[c]$.

For every finite set F of directions, we characterize those computations that are *strongly fair modulo F*. Roughly speaking, a computation ρ of the command c is strongly fair modulo F if every process enabled infinitely often either makes progress infinitely often (just as in traditional strong fairness) or eventually stops in a configuration in which its only possible transitions are labeled by directions in F and it cannot synchronize with any other process. Intuitively, even though the directions of F may be enabled infinitely often along ρ, it is possible to construct a program context $P[-]$ that restricts communication on the channels in F and fails to provide synchronization opportunities for members of F; for such contexts, the computation ρ will represent c's contribution to a strongly fair computation of $P[c]$. In particular, those processes can be ignored fairly in any program context that restricts communication

[1] A program context $P[-]$ is simply a program with a "hole", and $P[c]$ is the program that results from filling the hole with command c.

on the channels of F and does not provide sufficient opportunities for them to synchronize. For example, the infinite computation of $C_2||C_3$ that never performs output along channel b can be characterized as fair modulo $\{\mathsf{b}!\}$: the context $(C_1||-)\backslash\mathsf{a}\backslash\mathsf{b}$ restricts communication on channel b and provides no synchronization opportunities for C_3's $\mathsf{b}!0$ action.

Unlike the traditional notion of strong fairness, parameterized fairness can be characterized compositionally. Before doing so formally, however, we introduce some auxiliary definitions and give an informal explanation.

Definition 3.1.1 Let F be a finite set of directions. A configuration $\langle c, s\rangle$ is **enabled modulo** F if $\mathsf{inits}(c,s) - F$ is nonempty, and **blocked modulo** F if $\mathsf{inits}(c,s) \subseteq F$. $\diamond$

Thus a configuration is enabled modulo F if it can perform an action (either internal or otherwise) not labeled by a direction in F, and blocked modulo F otherwise. Any configuration that is blocked modulo F is necessarily blocked modulo F' for all $F' \supseteq F$.

Unlike strong fairness, parameterized strong fairness can be characterized compositionally. Just as every finite computation is strongly fair, every finite computation is strongly fair modulo F, for all sets F. A partial computation is strongly fair modulo F provided its final configuration is blocked modulo F. The fairness of an infinite computation ρ of a command c depends on the syntactic structure of c and on the form of ρ, as follows.

In general, an infinite computation of a command c inherits its fairness constraints from the underlying computations of c's component commands. For example, an infinite computation ρ of the command $c_1;c_2$ arises either from an infinite computation of c_1 or from a finite computation of c_1 followed by an infinite computation of c_2. The computation ρ is fair mod F whenever the infinite computation of c_1 or c_2 is fair mod F; any subcomponent that is blocked mod F along ρ must also be blocked mod F along the corresponding infinite computation of c_1 or c_2. Similarly, an infinite computation of the command $\mathsf{while\ true\ do}\ c$ arises either from infinitely many finite computations of c or from finitely many finite computations of c followed by an infinite computation of c. The computation ρ is fair mod F when all of these component computations of c are fair mod F: thus ρ is fair mod F whenever it contains infinitely many finite computations of c or when the single infinite computation of c is fair mod F.

Similar reasoning governs the fairness conditions for most of the remaining nonparallel commands. An infinite computation of $g \to c$ is fair mod F when the sequence of transitions made by c is fair mod F, and an infinite computation of $\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$ is fair mod F when the sequence of transitions made by the selected branch c_i is fair mod F. An infinite computation of $gc_1 \Box gc_2$ is fair mod F if, after making its choice of components gc_i on the first step, it behaves like a fair mod F computation of the selected gc_i.

Placing a command within the scope of channel restriction has the effect of discharging any context assumptions involving the newly restricted channel. For example, suppose ρ is an infinite computation of the command $c\backslash h$. If the i^{th} transition of ρ is $\langle c_i, s_i\rangle \xrightarrow{\lambda_i} \langle c_{i+1}, s_{i+1}\rangle$, then there is a corresponding computation ρ' of c such that the i^{th} transition of ρ' is $\langle c'_i, s_i\rangle \xrightarrow{\lambda_i} \langle c'_{i+1}, s_{i+1}\rangle$, with $c_i \equiv c'_i\backslash h$. If the computation ρ' is fair mod $F \cup \{h!, h?\}$, then there may be subprocesses of c that are willing to communicate on channel h and yet fail to make progress along ρ'. However, when c appears in the context $[-]\backslash h$, those subprocesses no longer have communication enabled along channel h and are no longer treated unfairly with respect to h. In effect, placing c in the context $[-]\backslash h$ discharges the assumption that c will eventually appear in a context that restricts communication on h. Hence a computation ρ of $c\backslash h$ is fair mod F whenever its underlying computation of c is fair mod $F \cup \{h!, h?\}$.

Determining the fairness of parallel commands requires more care. Every computation ρ of the command $c_1 \| c_2$ arises from interleaving and merging a computation ρ_1 of c_1 with a computation ρ_2 of c_2. Intuitively, when ρ_1 is fair mod F_1 and ρ_2 is fair mod F_2, ρ should inherit fairness constraints from both and therefore be fair mod $F_1 \cup F_2$: processes blocked mod F_1 along ρ_1 do not make progress along ρ, and likewise for F_2 and ρ_2. However, this analysis is valid only when neither component violates the assumptions incorporated in the other component's fairness set. For example, suppose a process Q of c_1 becomes (and remains) blocked mod F_1 along ρ_1. If the computation ρ_2 provides Q with infinitely many opportunities to synchronize, then the implicit assumption that Q will have insufficient opportunities to make progress is violated, and hence ρ cannot be fair (mod any F). It is also essential to ensure that none of the directions in F_1 appear infinitely often along ρ_2, for the following reason. The fairness set F_1 reflects the assumption that c_1 (and therefore $c_1 \| c_2$) will appear in a context that restricts communication on the channels associated with F_1. If a direction in F_1 appears infinitely often along ρ_2, then ρ_2 can represent c_2's transitions only if the context provides infinitely many opportunities to synchronize with c_2 on that direction. In such a case, however, the context would also be enabling synchronization with any processes of c_1 that were blocked in configurations in which they could use that direction, violating the assumptions inherent in F_1.

We can now give a formal, inductive characterization of strongly fair computation modulo F. When $F = \emptyset$, this characterization coincides with the traditional notion of strong process fairness, as given in [Fra86, AO91].

Definition 3.1.2 A computation ρ of command c is **strongly fair modulo** F (or, **fair mod** F) provided ρ satisfies one of the following conditions:

- ρ is a finite, successfully terminating computation;
- ρ is a partial computation whose final configuration is blocked modulo F;
- ρ is an infinite computation, c has form $(c_1; c_2)$ or $(\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)$, and the underlying infinite computation of c_1 or c_2 is fair mod F;

- ρ is an infinite computation, c has form $(\mathsf{while}\ b\ \mathsf{do}\ c')$ or $(g \to c')$, and each of ρ's component computations of c' is fair mod F;
- ρ is an infinite computation, c has form $(g \to c')$, and the underlying computation of c' is fair mod F;
- ρ is an infinite computation, c has form $(gc_1 \Box gc_2)$, and the underlying computation of the selected gc_i is fair mod F;
- ρ is an infinite computation, c has form $c' \backslash h$, and the underlying computation of c' is fair modulo $F \cup \{h!, h?\}$;
- ρ is an infinite computation, c has form $c_1 \| c_2$, and there exist sets F_1 and F_2 and computations ρ_1 of c_1 and ρ_2 of c_2 such that ρ_1 is fair mod F_1, ρ_2 is fair mod F_2, $F \supseteq F_1 \cup F_2$, ρ can be obtained by merging and synchronizing ρ_1 and ρ_2, neither ρ_i enables infinitely often any direction matching a member of F_j $(i \neq j)$, and neither ρ_i uses a direction in F_j infinitely often. $\diamond$

The following example highlights the compositional aspect of this characterization.

Example 3.1.3 Let C be the program $\mathsf{while\ true\ do\ c!1}$, and consider the computation

$$\rho = \langle((\mathsf{a!0} \to \mathsf{b!0}) \parallel C)\backslash \mathsf{b}, s\rangle \xrightarrow{\mathsf{a!0}} \langle(\mathsf{b!0} \parallel C)\backslash \mathsf{b}, s\rangle \xrightarrow{\varepsilon} \langle(\mathsf{b!0} \parallel \mathsf{c!1};C)\backslash \mathsf{b}, s\rangle \xrightarrow{\mathsf{c!1}} \cdots$$

in which the $\mathsf{b!0}$ action never occurs. ρ is strongly fair (that is, strongly fair mod $\emptyset$), for the following reasons:

1. The partial computation $\rho_1 = \langle \mathsf{a!0} \to \mathsf{b!0}, s\rangle \xrightarrow{\mathsf{a!0}} \langle \mathsf{b!0}, s\rangle$ is fair modulo $\{\mathsf{b!}\}$.
2. The infinite computation
$$\rho_2 = \langle C, s\rangle \xrightarrow{\varepsilon} \langle(\mathsf{c!1};C), s\rangle \xrightarrow{\mathsf{c!1}} \langle C, s\rangle \xrightarrow{\varepsilon} \langle(\mathsf{c!1};C), s\rangle \xrightarrow{\mathsf{c!1}} \langle C, s\rangle \xrightarrow{\varepsilon} \cdots$$
is fair mod $\emptyset$. Moreover, the only direction enabled infinitely often along ρ_2 is $\mathsf{c!}$.
3. Let ρ' be the infinite computation
$$\begin{aligned} \langle(\mathsf{a!0} \to \mathsf{b!1}) \parallel C, s\rangle &\xrightarrow{\mathsf{a!0}} \langle \mathsf{b!1} \parallel C, s\rangle \xrightarrow{\varepsilon} \langle \mathsf{b!1} \parallel (\mathsf{c!1};C), s\rangle \\ &\xrightarrow{\mathsf{c!1}} \langle \mathsf{b!1} \parallel C, s\rangle \xrightarrow{\varepsilon} \langle \mathsf{b!1} \parallel (\mathsf{c!1};C), s\rangle \xrightarrow{\mathsf{c!1}} \cdots \end{aligned}$$
in which the $\mathsf{b!0}$ action never occurs. This computation can be obtained by merging ρ_1 and ρ_2. Because ρ_2 does not use or enable synchronization with $\mathsf{b!}$ infinitely often, ρ' is fair modulo $\{\mathsf{b!}\}$.

4. Because the underlying computation of ρ is ρ', ρ is fair modulo $\emptyset$. $\diamond$

The next example illustrates the role that the fairness sets F play in determining those contexts in which a given computation can be considered fair.

Example 3.1.4

1. Let C be the program $\mathsf{while\ true\ do}\ (\mathsf{a!1} \Box \mathsf{b!1})$, and consider the computation

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

   that never outputs along channel b.

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

2. Define $C_1 \equiv \mathsf{while\ true\ do\ a!1}$ and $C_2 \equiv \mathsf{b!1} \to (\mathsf{while\ true\ do\ b!1})$, and consider the computation

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

   that never outputs along channel b.

   The set of infinitely enabled directions of ρ is also $\{\mathsf{a!}, \mathsf{b!}\}$. The computation ρ is not fair mod $\emptyset$, because the component C_2 remains blocked mod $\{\mathsf{b!}\}$. However, ρ is fair mod $\{\mathsf{b!}\}$.

3. Let C_p be the program $\mathsf{while\ true\ do}\ (\mathsf{a!0}\Box\mathsf{b?z})$, and let ρ_p be the computation

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

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

   Let $P[-]$ be the program context $([-] \parallel C_p)\backslash\mathsf{b}$. There is a fair (mod $\emptyset$) computation of $P[c]$ that corresponds to a merging of ρ_c and ρ_p and hence involves no synchronizations on channel b. In contrast, every fair (mod $\emptyset$) computation of $P[C_1 \| C_2]$ must eventually synchronize on channel b, because it is unfair for C_2 to be forced to block on b! when a matching direction is enabled infinitely often. Thus there is no fair execution of $P[C_1 \| C_2]$ in which the C_p component performs ρ_p. $\diamond$

3.2 Strongly Fair Traces

We define a set of **steps**

$$\Sigma = S \times \Lambda \times S;$$

intuitively, the step (s,λ,s') corresponds to a transition of the form $\langle c,s\rangle \xrightarrow{\lambda} \langle c',s'\rangle$. Thus each step (s,λ,s') records the initial and final states of a transition, as well as the label of the action that occurred. We also introduce a set of empty traces $\Sigma^0 = \{\varepsilon_s \mid s \in S\}$, with each ε_s corresponding to configurations of form $\langle c,s\rangle$. The set of finite traces is $\Sigma^* = \Sigma^0 \cup \Sigma^+$, where

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

is the set of nonempty finite traces. We let $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$, where the set Σ^ω of infinite traces is defined by

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

Each trace $\alpha \in \Sigma^\infty$ represents a finite or infinite transition sequence.

Two traces α and β are **composable** if α is infinite or if the final state of α is the first state of β; we write $\mathit{composable}(\alpha,\beta)$ in such cases. For composable traces α and β, the trace $\alpha\beta$ is their (string-like) concatenation. For example, if $\alpha = (s_0,\lambda_0,s_1)(s_1,\lambda_1,s_2)$ and $\beta = (s_2,\lambda_2,s_3)$, then

$$\alpha\beta = (s_0,\lambda_0,s_1)(s_1,\lambda_1,s_2)(s_2,\lambda_2,s_3).$$

The traces of Σ^0 serve as local units for concatenation: $\alpha\varepsilon_s = \alpha$ and $\varepsilon_s\beta = \beta$ when s is the final state of α and the first state of β. Infinite concatenation is the obvious extension of finite concatenation. An infinite sequence of traces $\alpha_0,\alpha_1,\alpha_2,\dots$ is composable if, for every $i \geq 0$, the traces $\alpha_0\alpha_1\dots\alpha_i$ and α_{i+1} are composable; their concatenation is the trace

$$\alpha_0\alpha_1\alpha_2\dots\alpha_n\alpha_{n+1}\dots.$$

These simple traces are insufficient for reasoning about strong process fairness compositionally, because they fail to record the necessary contextual information made explicit in Definition 3.1.2. For any infinite computation ρ, we need to know which directions are enabled infinitely often along ρ. We also need to know for which contexts ρ will represent a fair computation; that is, we need to know for which sets F the computation ρ is fair modulo F. Every finite computation is fair mod F for all sets F. However, because a finite computation may be used to generate an infinite computation, we also need to know which directions are enabled along a finite computation. Finally, to reason about deadlock and blocking, we need information about partial computations. For a partial computation ρ, we need to know what type of actions (including ε) are possible from the final configuration of ρ. Because a partial computation will never be iterated in a looping context, we do not need to record the directions enabled along that computation.

We combine simple traces with this additional contextual information to yield **fair traces**. Letting

$$\Gamma = \mathcal{P}_{\text{fin}}(\Delta^+) \times \mathcal{P}_{\text{fin}}(\Delta^+) \times \{\mathtt{f}, \mathtt{i}, \mathtt{p}\}$$

capture the necessary contextual information, we define the set $\Phi \subseteq \Sigma^\infty \times \Gamma$ of **fair traces** as

$$\begin{array}{rcl} \Phi & = & \Sigma^* \times (\mathcal{P}_{\text{fin}}(\Delta) \times \mathcal{P}_{\text{fin}}(\Delta) \times \{\mathtt{f}\}) \\ & \cup & \Sigma^\omega \times (\mathcal{P}_{\text{fin}}(\Delta) \times \mathcal{P}_{\text{fin}}(\Delta) \times \{\mathtt{i}\}) \\ & \cup & \Sigma^* \times (\mathcal{P}_{\text{fin}}(\Delta^+) \times \mathcal{P}_{\text{fin}}(\Delta^+) \times \{\mathtt{p}\}). \end{array}$$

For convenience, we occasionally use Φ_{fin}, Φ_{inf}, and Φ_{par} to refer to the subsets of Φ with tags $\mathtt{f}$, $\mathtt{i}$, and $\mathtt{p}$, respectively.

Intuitively, the fair trace $\langle \alpha, (F, E, \mathtt{f}) \rangle$ represents a fair mod F, successfully terminating computation with enabled directions E; the tag "f" merely indicates that the trace represents a finite computation. Similarly, the fair trace $\langle \alpha, (F, E, \mathtt{i}) \rangle$ represents an infinite, fair mod F computation with infinitely often enabled directions E; the tag "i" indicates that the trace represents an infinite computation. The fair trace $\langle \alpha, (F, E, \mathtt{p}) \rangle$ (with $F \supseteq E$) represents a partial computation for which the directions E (possibly including ε) are enabled in the final configuration. When ε is not in E, the blocked computation is necessarily fair mod E, and therefore fair mod F as well. Again, the tag "p" merely indicates that the trace represents a partial computation. Technically, the F-component of the contextual tuple is unnecessary for finite traces because every finite computation is necessarily fair. Similarly, the F-component of a partial computation does not provide any essential information not already incorporated in the E-component. However, the inclusion of these components allows a consistent representation for all fair traces, which will be convenient for subsequent definitions.

For every (possibly partial) computation ρ, $\mathsf{trace}(\rho)$ is the simple trace that corresponds to the transitions made along ρ. For example, if ρ is the computation

$$\langle c, s_0 \rangle \xrightarrow{\lambda_0} \langle c_1, s_1 \rangle \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_{k-1}} \langle c_k, s_k \rangle,$$

then $\mathsf{trace}(\rho) = (s_0, \lambda_0, s_1)(s_1, \lambda_1, s_2) \ldots (s_{k-1}, \lambda_{k-1}, s_k)$. The set $\mathsf{en}(\rho)$ contains the "relevant" directions enabled along ρ: when ρ is a finite computation, $\mathsf{en}(\rho)$ contains the directions enabled along ρ; when ρ is an infinite computation, $\mathsf{en}(\rho)$ contains the directions enabled infinitely often ρ.

We can give an operational characterization of a fair trace semantics $\mathcal{T}_s : \mathsf{Com} \to \mathcal{P}(\Phi)$ as follows:

$$\begin{aligned}
\mathcal{T}_s[\![c]\!] = \{\langle \mathsf{trace}(\rho), (F, \mathsf{en}(\rho), \mathtt{f})\rangle \mid \\
&\rho = \langle c, s_0\rangle \xrightarrow{\lambda_0} \langle c_1, s_1\rangle \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_{k-1}} \langle c_k, s_k\rangle \mathsf{term} \text{ is fair mod } F\} \\
\cup \{\langle \mathsf{trace}(\rho), (F, E, \mathtt{p})\rangle \mid E = \mathsf{inits}(c_k, s_k) \ \&\ F \supseteq E \\
&\rho = \langle c, s_0\rangle \xrightarrow{\lambda_0} \langle c_1, s_1\rangle \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_{k-1}} \langle c_k, s_k\rangle \ \&\ \neg\langle c_k, s_k\rangle \mathsf{term}\} \\
\cup \{\langle \mathsf{trace}(\rho), (F, \mathsf{en}(\rho), \mathtt{i})\rangle \mid \\
&\rho = \langle c, s_0\rangle \xrightarrow{\lambda_0} \langle c_1, s_1\rangle \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_k} \cdots \text{is strongly fair mod } F\}.
\end{aligned}$$

3.3 Strongly Fair Trace Semantics

In the previous section, we gave an operational characterization of a fair trace semantics $\mathcal{T}_s$. In this section, we show how to give a denotational characterization of this same semantic function. We do this by defining, for each construct in the language, a corresponding operation on trace sets.

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

$$\mathcal{B}[\![b]\!] = \{(s, v) \mid \langle b, s\rangle \longrightarrow^* v\}, \qquad \mathcal{E}[\![e]\!] = \{(s, n) \mid \langle e, s\rangle \longrightarrow^* n\}.$$

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

$$\mathcal{T}_s[\![b]\!] = \{\langle (s, \varepsilon, s), (F, \emptyset, \mathtt{f})\rangle,\ \langle \varepsilon_s, (F \cup \{\varepsilon\}, \{\varepsilon\}, \mathtt{p})\rangle \mid (s, \mathtt{tt}) \in \mathcal{B}[\![b]\!] \ \&\ F \in \mathcal{P}_{\mathrm{fin}}(\Delta)\}.$$

Intuitively, $\mathcal{T}_s[\![b]\!]$ contains the idle steps possible from states satisfying the boolean expression b. Note that, for any boolean expression b,

$$\begin{aligned}
\mathcal{T}_s[\![\neg b]\!] &= \{\langle (s, \varepsilon, s), (F, \emptyset, \mathtt{f})\rangle,\ \langle \varepsilon_s, (F \cup \{\varepsilon\}, \{\varepsilon\}, \mathtt{p})\rangle \mid (s, \mathtt{tt}) \in \mathcal{B}[\![\neg b]\!] \ \&\ F \in \mathcal{P}_{\mathrm{fin}}(\Delta)\} \\
&= \{\langle (s, \varepsilon, s), (F, \emptyset, \mathtt{f})\rangle,\ \langle \varepsilon_s, (F \cup \{\varepsilon\}, \{\varepsilon\}, \mathtt{p})\rangle \mid (s, \mathtt{ff}) \in \mathcal{B}[\![b]\!] \ \&\ F \in \mathcal{P}_{\mathrm{fin}}(\Delta)\}.
\end{aligned}$$

Consequently, both $\mathcal{T}_s[\![b]\!]$ and $\mathcal{T}_s[\![\neg b]\!]$ can be defined solely in terms of b.

Based on the operational characterization of $\mathcal{T}_s$, it should be easy to see that

$$\begin{aligned}
\mathcal{T}_s[\![\mathsf{skip}]\!] = &\{\langle (s, \varepsilon, s), (F, \emptyset, \mathtt{f})\rangle \mid s \in S \ \&\ F \in \mathcal{P}_{\mathrm{fin}}(\Delta)\} \\
\cup\ &\{\langle \varepsilon_s, (F, \{\varepsilon\}, \mathtt{p})\rangle \mid s \in S \ \&\ F \supseteq \{\varepsilon\}\}
\end{aligned}$$

and

$$\begin{aligned}
\mathcal{T}_s[\![i{:=}e]\!] = &\{\langle (s, \varepsilon, [s|i = n]), (F, \emptyset, \mathtt{f})\rangle \mid \mathsf{fv}[\![i{:=}e]\!] \subseteq \mathsf{dom}(s) \ \&\ F \in \mathcal{P}_{\mathrm{fin}}(\Delta) \ \&\ (s, n) \in \mathcal{E}[\![e]\!]\} \\
\cup &\{\langle \varepsilon_s, (F, \{\varepsilon\}, \mathtt{p})\rangle \mid \mathsf{fv}[\![i{:=}e]\!] \subseteq \mathsf{dom}(s) \ \&\ F \supseteq \{\varepsilon\}\}.
\end{aligned}$$

Similarly, for guards we obtain

$$\begin{aligned}\mathcal{T}_s[\![h?i]\!] = \{\langle(s,h?n,[s|i=n]),(F,\{h?\},\mathtt{f})\rangle \mid i \in \mathsf{dom}(s) \;\&\; n \in \mathbb{Z} \;\&\; F \in \mathcal{P}_{\text{fin}}(\Delta)\} \\ \cup\,\{\langle\varepsilon_s,(F,\{h?\},\mathtt{p})\rangle \mid i \in \mathsf{dom}(s) \;\&\; F \supseteq \{h?\}\}\end{aligned}$$

and

$$\begin{aligned}\mathcal{T}_s[\![h!e]\!] = \{\langle(s,h!n,s),(F,\{h!\},\mathtt{f})\rangle \mid (s,n) \in \mathcal{E}[\![e]\!] \;\&\; F \in \mathcal{P}_{\text{fin}}(\Delta)\} \\ \cup\,\{\langle\varepsilon_s,(F,\{h!\},\mathtt{p})\rangle \mid \mathsf{fv}[\![e]\!] \subseteq \mathsf{dom}(s) \;\&\; F \supseteq \{h!\}\}.\end{aligned}$$

Sequential composition

The command $c_1;c_2$ represents the sequential composition of commands c_1 and c_2: each computation of $c_1;c_2$ corresponds to a computation of c_1 that, if successful, is followed by a computation of c_2. If the computation of c_1 terminates successfully in state s, then the computation of c_2 must begin from state s; if the computation of c_1 instead is infinite or becomes blocked, then the computation of c_2 never begins. We can construct the traces of $c_1;c_2$ by combining traces of c_1 with traces of c_2 in a similar way.

Two fair traces φ_1 and φ_2 are composable whenever φ_1 is an infinite or partial trace, or when their simple trace components are composable (that is, when the final state of the first trace is the initial state of the second trace). We write *composable*(φ_1,φ_2) when φ_1 and φ_2 are composable. When $\varphi_1 = \langle\alpha,(F_1,E_1,R_1)\rangle$ 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:

$$\varphi_1\varphi_2 = \begin{cases}\varphi_1, & \text{if } R_1 \in \{\mathtt{i},\mathtt{p}\},\\ \langle\alpha\beta,(F_2,E_1\cup E_2,\mathtt{f})\rangle, & \text{if } R_1 = R_2 = \mathtt{f},\\ \langle\alpha\beta,(F_2,E_2,R_2)\rangle, & \text{if } R_1 = \mathtt{f} \text{ and } R_2 \in \{\mathtt{i},\mathtt{p}\}.\end{cases}$$

As is evident from this definition, the necessary contextual information for the resulting trace depends on the form of the individual traces. When α represents an infinite or partial computation, the contextual information of β becomes irrelevant: the computation represented by β never begins, because the computation represented by α does not terminate. When α represents a finite, successful computation, its fairness constraints (as represented by the fairness set F_1) become irrelevant; however, the finite enabling information provided by E_1 must be preserved when the resulting trace also represents a finite, successful computation.

Thus we define sequential composition on trace sets T_1 and T_2 by

$$T_1;T_2 \quad = \quad \{\varphi_1\varphi_2 \mid \varphi_1 \in T_1 \;\&\; \varphi_2 \in T_2 \;\&\; \mathit{composable}(\varphi_1,\varphi_2)\}.$$

We can then define

$$\mathcal{T}_s[\![c_1;c_2]\!] = \mathcal{T}_s[\![c_1]\!];\mathcal{T}_s[\![c_2]\!],$$
$$\mathcal{T}_s[\![g \rightarrow c]\!] = \mathcal{T}_s[\![g]\!];\mathcal{T}_s[\![c]\!],$$

and

$$\mathcal{T}_s[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] = \mathcal{T}_s[\![b]\!];\mathcal{T}_s[\![c_1]\!] \cup \mathcal{T}_s[\![\neg b]\!];\mathcal{T}_s[\![c_2]\!].$$

Iteration

Loops correspond to the finite or infinite iteration of a single command. Thus we base our semantics for loops on the iteration of trace sets.

When $\{X_i \mid i \geq 0\}$ is a collection of finite sets, we let $\overset{\infty}{\underset{i=0}{\Cup}} X_i$ be the set of elements appearing in infinitely many sets X_i. That is,

$$\overset{\infty}{\underset{i=0}{\Cup}} X_i = \{d \mid \forall j \geq 0. \exists k > j.\ d \in X_k\}.$$

We then introduce composability criteria for infinite collections of fair traces. Let $\langle \varphi_i \rangle_{i=0}^{\infty}$ represent an infinite sequence of fair traces

$$\varphi_0, \varphi_1, \ldots, \varphi_n, \ldots,$$

such that, for each $i \geq 0$, $\varphi_i = \langle \alpha_i, (F_i, E_i, R_i) \rangle$. The sequence $\langle \varphi_i \rangle_{i=0}^{\infty}$ is composable, written *composable*$(\langle \varphi_i \rangle_{i=0}^{\infty})$, if, for each i, the traces $\varphi_0 \varphi_1 \ldots \varphi_{i-1}$ and φ_i are composable and the sets $\overset{\infty}{\underset{i=0}{\Cup}} F_i$ and $\overset{\infty}{\underset{i=0}{\Cup}} E_i$ are finite. (These sets must be finite to ensure that the resulting trace is well-formed.) We then define infinite concatenation as follows:

$$\varphi_0\varphi_1\varphi_2\ldots \quad = \quad \begin{cases} \langle \alpha_0\alpha_1 \ldots \alpha_n \ldots, (\overset{\infty}{\underset{i=0}{\Cup}} F_i, \overset{\infty}{\underset{i=0}{\Cup}} E_i, \mathtt{i}) \rangle, & \text{if } \forall i. R_i = \mathtt{f}, \\ \langle \alpha_0\alpha_1 \ldots \alpha_k, (F_k, E_k, R_k) \rangle, & \text{if } \forall i < k. R_i = \mathtt{f} \text{ and } R_k \in \{\mathtt{i}, \mathtt{p}\}. \end{cases}$$

When each φ_i is finite, the infinitely enabled directions of the resulting trace are those directions that appear in infinitely many of the sets E_i, and similarly for the infinitely visible directions. When at least one φ_i is an infinite or partial trace, the infinite concatenation is simply the finite concatenation $\varphi_0\varphi_1 \ldots \varphi_k$, where φ_k is the first infinite or partial trace of the series.

The definitions for finite and infinite iteration on trace sets follow directly from the definitions of concatenation and sequential composition. Finite iteration on the trace set T is defined by

$$T^* = \bigcup_{i=0}^{\infty} T^i,$$

where $T^0 = \{\langle \varepsilon_s, (\emptyset, \emptyset, \mathtt{f})\rangle \mid s \in S\}$ and $T^{n+1} = T^n;T$. Infinite iteration on the trace set T is defined as follows:

$$T^{\omega} = \{\varphi_0\varphi_1 \ldots \varphi_k \ldots \mid (\forall i \geq 0.\varphi_i \in T) \ \& \ composable(\langle \varphi_i \rangle_{i=0}^{\infty})\}.$$

We can give the semantics of loops using these definitions of iteration:

$$\mathcal{T}_s[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!] = (\mathcal{T}_s[\![b]\!];\mathcal{T}_s[\![c]\!])^{\omega} \cup (\mathcal{T}_s[\![b]\!];\mathcal{T}_s[\![c]\!])^{*};\mathcal{T}_s[\![\neg b]\!].$$

Guarded choice

The command $gc_1 \Box gc_2$ represents a choice, to be made on the first step, between the guarded commands gc_1 and gc_2. Every computation of gc_1 and of gc_2 therefore gives rise to a corresponding computation of $gc_1 \Box gc_2$ that, on its initial step, can perform any action enabled by either component. Whenever a fair trace φ represents an infinite computation (or a partial computation involving at least one step) of gc_1 or gc_2, φ necessarily also represents a computation of $gc_1 \Box gc_2$. When φ represents a finite computation (or a partial computation involving no steps) of gc_1 or gc_2, however, the enabling component E must be augmented with those directions that were enabled initially by the unchosen component. This additional enabling information can be generated by looking at the "empty" partial traces of the unchosen component: if $\langle \varepsilon_s, (F, E, \mathtt{p})\rangle$ is a trace of gc_i, then gc_i must be able to perform the actions E on its first step. Thus we define guarded choice on trace sets as follows:

$$\begin{aligned}
T_1 \Box T_2 = &\{\langle \alpha, (F,E,\mathtt{i})\rangle \in T_1 \cup T_2 \mid \alpha \in \Sigma^{\omega}\} \cup \{\langle \alpha, (F,E,\mathtt{p})\rangle \in T_1 \cup T_2 \mid \alpha \in \Sigma^{+}\} \\
&\cup \{\langle \varepsilon_s, (F_1 \cup F_2, E_1 \cup E_2, \mathtt{p})\rangle \mid \langle \varepsilon_s, (F_1, E_1, \mathtt{p})\rangle \in T_1 \ \& \ \langle \varepsilon_s, (F_2, E_2, \mathtt{p})\rangle \in T_2\} \\
&\cup \{\langle \alpha, (F_1, E_1 \cup E_2, \mathtt{f})\rangle \mid \langle \varepsilon_s\alpha, (F_1, E_1, \mathtt{f})\rangle \in T_1 \ \& \ \langle \varepsilon_s, (F_2, E_2, \mathtt{p})\rangle \in T_2 \ \& \ \varepsilon \notin E_2\} \\
&\cup \{\langle \alpha, (F_2, E_1 \cup E_2, \mathtt{f})\rangle \mid \langle \varepsilon_s\alpha, (F_2, E_2, \mathtt{f})\rangle \in T_2 \ \& \ \langle \varepsilon_s, (F_1, E_1, \mathtt{p})\rangle \in T_1 \ \& \ \varepsilon \notin E_1\}.
\end{aligned}$$

The final two clauses impose conditions of form $\varepsilon \notin E_i$ when $\langle \varepsilon_s, (F_i, E_i, \mathtt{p})\rangle$ is a trace of the unchosen component. Technically, these conditions are moot: we perform the operation $T_1 \Box T_2$ only when $T1$ and T_2 are trace sets of guarded commands, and ε is never enabled on the first step of guarded commands. However, in Chapter 4 we introduce semantic variations in which ε may appear to be enabled on the initial step, and these conditions maintain the integrity of the resulting traces' sets of enabled directions.

We define $\mathcal{T}_s[\![gc_1 \Box gc_2]\!] = \mathcal{T}_s[\![gc_1]\!] \Box \mathcal{T}_s[\![gc_2]\!]$.

Channel restriction

The computations of $c \backslash h$ are 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 α, $\mathsf{chans}(\alpha)$ is the set of channels appearing along α. For a set X of directions, we let $X\backslash h$ be the set X with references to channel h removed: $X\backslash h = X - \{h!, h?\}$. We then define $T\backslash h$ by

$$T\backslash h = \{\langle \alpha, (F', E\backslash h, R)\rangle \mid \langle \alpha, (F, E, R)\rangle \in T \ \& \ \ \& \ F' \supseteq F\backslash h \ \& \ h \notin \mathsf{chans}(\alpha)\}.$$

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

Parallel composition

The command $c_1 \| c_2$ represents the parallel execution of the commands c_1 and c_2. The computations of $c_1 \| c_2$ can be derived from interleavings and synchronizations of computations of c_1 with computations of c_2. Likewise, the fair traces of $c_1 \| c_2$ can be derived from interleavings and synchronizations of traces of c_1 with traces of c_2.

Of course, only certain pairs of computations—and, correspondingly, traces—can be merged in a meaningful way. For example, merging a partial computation represented by the fair trace $\langle \alpha, (\{h!\}, \{h!\}, \mathtt{p})\rangle$ with an infinite computation represented by the fair trace $\langle \beta, (\emptyset, \{h?\}, \mathtt{i})\rangle$ does not yield a fair computation of the parallel command: the first component cannot remain blocked if it is enabled for synchronization infinitely often. For this reason, we introduce a predicate *mergeable* that indicates when a potential merging of fair traces is "meaningful": the predicate $\mathit{mergeable}(\varphi_1, \varphi_2)$ is true precisely when merging computations represented by φ_1 and φ_2 would yield a fair (modulo an appropriate set F) computation of the corresponding parallel command. The criteria for determining whether two traces are mergeable follow directly from the parallel clause of the parameterized fairness definition in Section 3.1. We let $\mathsf{vis}(\alpha)$ be the set of directions visible infinitely often along the simple trace α: for example, if $\alpha = (s, \mathtt{b!0}, s)[(s, \mathtt{a!0}, s)]^\omega$, then $\mathsf{vis}(\alpha) = \{\mathtt{a!}\}$. We then define the predicate $\mathit{mergeable}(\varphi_1, \varphi_2)$ for fair traces $\varphi_1 = \langle \alpha_1, (F_1, E_1, R_1)\rangle$ and $\varphi_2 = \langle \alpha_2, (F_2, E_2, R_2)\rangle$ as follows:

$$\begin{aligned}&\mathit{mergeable}(\varphi_1, \varphi_2) \iff (R_1 = \mathtt{f}) \text{ or } (R_2 = \mathtt{f}) \text{ or } (R_1 = R_2 = \mathtt{p}) \text{ or}\\ &(\varepsilon \notin F_1 \cup F_2 \ \& \ \neg\mathsf{match}(F_1, E_2) \ \& \ \neg\mathsf{match}(F_2, E_1) \ \& \ F_1 \cap \mathsf{vis}(\alpha_2) = \emptyset \ \& \ F_2 \cap \mathsf{vis}(\alpha_1) = \emptyset).\end{aligned}$$

Any trace can be merged safely with a finite, successful trace; hence two traces are mergeable if either trace is finite. Additionally, two partial traces can always be merged to yield a partial trace of the parallel command. The final clause specifies when an infinite trace can be merged with another infinite trace or a partial trace; its individual conditions correspond precisely to the conditions incorporated into the parallel-composition clause for parameterized strong fairness in Definition 3.1.2. A partial trace represents a computation that can become blocked, provided that no ε-transition is possible from its final configuration. The conditions $\neg\mathsf{match}(F_1, E_2)$ and $\neg\mathsf{match}(F_2, E_1)$ ensure that neither component enables synchronization infinitely often with any direction in the other component's fairness set. Similarly, the conditions $F_1 \cap \mathsf{vis}(\alpha_2) = \emptyset$ and $F_2 \cap \mathsf{vis}(\alpha_1) = \emptyset$ ensure that neither component uses infinitely often a direction in the other component's fairness set.

Given two mergeable computations (or traces), only certain mergings of them will represent fair computations (or traces) of the corresponding parallel command. In particular, every fair merge of the traces φ_1 and φ_2 should "consume" all of φ_1 and φ_2. That is, every step of each φ_i should be accounted for in any fair merge of φ_1 and φ_2. We can capture this intuition by defining a ternary relation *fairmerge* $\subseteq \Phi \times \Phi \times \Phi$ on fair traces, adapted from Park's fairmerge relation [Par79] to account for the possibility of synchronization, with the idea that $(\varphi_1, \varphi_2, \varphi) \in$ *fairmerge* if and only if φ arises from a fair interleaving (and synchronization) of φ_1 and φ_2. The definition of *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 interleaving and merging operators on both simple and fair traces.

Consider a parallel program $C_1 \| C_2$, and suppose that C_1 can perform a finite transition sequence represented by the simple trace $\alpha = (s_0, \lambda_0, s_1)(s_1, \lambda_1, s_2) \ldots (s_k, \lambda_k, s_{k+1})$. If s is a local state of C_2, then the simple trace

$$\alpha \rfloor\!\lfloor \varepsilon_s = (s_0 \cup s, \lambda_0, s_1 \cup s)(s_1 \cup s, \lambda_1, s_2 \cup s) \ldots (s_k \cup s, \lambda_k, s_{k+1} \cup s)$$

represents a finite transition sequence of the parallel command in which C_1 makes the transitions represented by α and C_2 idles in its local state. The trace $\alpha \rfloor\!\lfloor \varepsilon_s$ is similarly defined for infinite traces α, capturing the intuition that C_1 can perform α uninterrupted when C_2 has no transitions possible from state s. For finite, nonempty, disjoint[2] traces α and β, we also define

$$\alpha \rfloor\!\lfloor \beta = (\alpha \rfloor\!\lfloor \varepsilon_t)(\beta \rfloor\!\lfloor \varepsilon_s),$$

where s and t are the final state of α and initial state of β, respectively. That is, $\alpha \rfloor\!\lfloor \beta$ is the trace that looks like α (with the first state of β propagated), followed by β (with the final state of α propagated). Intuitively, if α and β represent finite transition sequences of C_1 and C_2 respectively, then $\alpha \rfloor\!\lfloor \beta$ represents a transition sequence of $C_1 \| C_2$ in which C_1 makes the transitions represented by α, followed by C_2 making the transitions represented by β. For example, if $\alpha = (s_0, \lambda_0, s_1)(s_1, \lambda_1, s_2)$ and $\beta = (t_0, \mu_0, t_1)(t_1, \mu_1, t_2)$, then

$$\alpha \rfloor\!\lfloor \beta = (s_0 \cup t_0, \lambda_0, s_1 \cup t_0)(s_1 \cup t_0, \lambda_1, s_2 \cup t_0)(s_2 \cup t_0, \mu_0, s_2 \cup t_1)(s_2 \cup t_1, \mu_1, s_2 \cup t_2).$$

[2] Two traces α and β are disjoint if each state along α is disjoint from every state along β; in such cases we write $\mathsf{disjoint}(\alpha, \beta)$. Likewise, two fair traces $\varphi_1 = \langle \alpha, \theta_1 \rangle$ and $\varphi_2 = \langle \beta, \theta_2 \rangle$ are disjoint when their simple-trace components α and β are disjoint.

The parallel command $C_1 \| C_2$ may also have transition sequences in which the two components repeatedly synchronize. Two nonempty, finite simple traces $\alpha = (s_0, \lambda_0, s_1) \dots (s_k, \lambda_k, s_{k+1})$ and $\beta = (t_0, \mu_0, t_1) \dots (t_n, \mu_n, t_{n+1})$ **match**—and we write $\mathsf{match}(\alpha, \beta)$—if the two traces have the same length and each step of α matches the corresponding step of β (that is, if $k = n$ and $\mathsf{match}(\lambda_i, \mu_i)$ for each i). When α and β match, $\alpha \| \beta$ is the trace in which α and β synchronize at each step:

$$\alpha \| \beta = (s_0 \cup t_0, \varepsilon, s_1 \cup t_1) \dots (s_k \cup t_k, \varepsilon, s_{k+1} \cup t_{k+1}).$$

Similarly, the fair traces $\varphi_1 = \langle \alpha, (F_1, E_1, \mathtt{f}) \rangle$ and $\varphi_2 = \langle \beta, (F_2, E_2, \mathtt{f}) \rangle$ match when their simple-trace components α and β match.

When computations ρ_1 of C_1 and ρ_2 of C_2 are merged fairly to yield a computation ρ of $C_1 \| C_2$, the order in which their steps are interleaved and synchronized does not affect the general properties (that is, the set of infinitely enabled directions or the relative fairness set F) of ρ. Instead, these properties can be determined solely from the corresponding properties of the original computations ρ_1 and ρ_2. Thus we define an operator $\theta_1 \| \theta_2$ for contextual triples $\theta_1, \theta_2 \in \Gamma$ as follows, with the intuition that each $\theta \in \theta_1 \| \theta_2$ provides valid contextual information for a computation that arises from merging computations with contextual information θ_1 and θ_2.

The result of merging two finite transition sequences is yet another finite transition sequence, and the set of directions enabled along that transition sequence is the union of the sets enabled along each of the original sequences. Thus we define

$$(F_1, E_1, \mathtt{f}) \| (F_2, E_2, \mathtt{f}) = \{ (F, E_1 \cup E_2, \mathtt{f}) \mid F \supseteq F_1 \cup F_2 \}.$$

Merging a finite (successful) transition sequence and a partial computation that can next perform actions E_2 results in a partial computation that can next perform actions E_2; thus we define

$$(F_1, E_1, \mathtt{f}) \| (F_2, E_2, \mathtt{p}) = (F_2, E_2, \mathtt{p}) \| (F_1, E_1, \mathtt{f}) = \{ (F, E_2, \mathtt{p}) \mid F \supseteq F_1 \cup F_2 \}.$$

Merging two partial computations—one of which can next perform actions E_1 and the other of which can next perform actions E_2— results in a third partial computation that, on its next step, can perform any of the actions $E_1 \cup E_2$. In addition, when the sets E_1 and E_2 match, the resulting computation can also perform an internal action corresponding to a synchronization. Thus we define

$$(F_1, E_1, \mathtt{p}) \| (F_2, E_2, \mathtt{p}) = \{ (F, E_1 \cup E_2 \cup \{ \varepsilon \mid \mathsf{match}(E_1, E_2) \}, \mathtt{p}) \mid F \supseteq F_1 \cup F_2 \}.$$

Merging a finite computation and an infinite computation with infinitely enabled directions E_2 yields another infinite computation with infinitely enabled directions E_2:

$$(F_1, E_1, \mathtt{f}) \| (F_2, E_2, \mathtt{i}) = (F_2, E_2, \mathtt{i}) \| (F_1, E_1, \mathtt{f}) = \{ (F, E_2, \mathtt{i}) \mid F \supseteq F_1 \cup F_2 \}.$$

Finally,[3] merging a partial computation that can next perform actions E_1 and an infinite computation with infinitely enabled directions E_2 results in an infinite computation with infinitely enabled directions $E_1 \cup E_2$; thus we define

$$(F_1, E_1, \mathtt{p}) \| (F_2, E_2, \mathtt{i}) = (F_2, E_2, \mathtt{i}) \| (F_1, E_1, \mathtt{p}) = \{(F, E_1 \cup E_2, \mathtt{i}) \mid F \supseteq F_1 \cup F_2\}.$$

Note that this last definition safely ignores the possibility of synchronization between the two components: the sets E_1 and E_2 are guaranteed not to match, because we perform this operation only on traces φ_1 and φ_2 for which the predicate $\mathit{mergeable}(\varphi_1, \varphi_2)$ is true.

Using this parallel operator on contextual triples, we can extend the interleaving ($\rfloor\!\lfloor$) and merging ($\|$) operators to fair traces in the obvious way. For fair traces $\varphi_1 = \langle \alpha, \theta_1 \rangle$ and $\varphi_2 = \langle \beta, \theta_2 \rangle$ such that $\alpha \rfloor\!\lfloor \beta$ or $\alpha \| \beta$ is defined, we define $\varphi_1 \rfloor\!\lfloor \varphi_2$ and $\varphi_1 \| \varphi_2$ (respectively) as follows:

$$\varphi_1 \rfloor\!\lfloor \varphi_2 = \{\langle \alpha \rfloor\!\lfloor \beta, \theta \rangle \mid \theta \in \theta_1 \| \theta_2\}, \qquad \varphi_1 \| \varphi_2 = \{\langle \alpha \| \beta, \theta \rangle \mid \theta \in \theta_1 \| \theta_2\}.$$

Thus the fair trace φ is in $\varphi_1 \rfloor\!\lfloor \varphi_2$ if its simple trace component is the interleaving $\alpha \rfloor\!\lfloor \beta$ and its contextual information corresponds to the merging $\theta_1 \| \theta_2$. Similarly, φ is in $\varphi_1 \| \varphi_2$ if it captures the information inherent in a synchronization of φ_1 and φ_2.

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

$$\begin{array}{rcl} \mathit{both} & = & \{(\varphi_1, \varphi_2, \varphi), (\varphi_2, \varphi_1, \varphi) \mid \varphi_1, \varphi_2 \in \Phi_{\mathrm{fin}} \;\&\; \mathsf{disjoint}(\varphi_1, \varphi_2) \;\&\; \varphi \in \varphi_1 \rfloor\!\lfloor \varphi_2\} \\ & \cup & \{(\varphi_1, \varphi_2, \varphi) \mid \varphi_1, \varphi_2 \in \Phi_{\mathrm{fin}} \;\&\; \mathsf{disjoint}(\varphi_1, \varphi_2) \;\&\; \mathsf{match}(\varphi_1, \varphi_2) \;\&\; \varphi \in \varphi_1 \| \varphi_2\}. \end{array}$$

Once one component terminates (or becomes permanently blocked), the other component can proceed uninterrupted. Thus the set *one* contains triples that reflect the uninterrupted progress of one component while the other component idles (and hence *one* involves no synchronizations):

$$\mathit{one} = \{(\varphi_1, \varphi_2, \varphi),\ (\varphi_2, \varphi_1, \varphi) \mid \varphi_1 \in \Phi \;\&\; \varphi_2 = \langle \varepsilon_s, \theta_2 \rangle \;\&\; \varphi \in \varphi_1 \rfloor\!\lfloor \varphi_2 \;\&\; \mathsf{disjoint}(\varphi_1, \varphi_2)\}.$$

To define *fairmerge* from *both* and *one*, we define a dot operator ($\cdot$) that extends concatenation of traces to sets of triples of traces in the obvious way. For example, when Y_1 and Y_2 are sets of triples of traces,

$$\begin{array}{rcl} Y_1 \cdot Y_2 & = & \{(\varphi_1\varphi_1', \varphi_2\varphi_2', \varphi_3\varphi_3') \mid (\varphi_1, \varphi_2, \varphi_3) \in Y_1 \;\&\; (\varphi_1', \varphi_2', \varphi_3') \in Y_2 \\ & & \quad \&\; \mathit{composable}(\varphi_1, \varphi_1') \;\&\; \mathit{composable}(\varphi_2, \varphi_2') \;\&\; \mathit{composable}(\varphi_3, \varphi_3')\}. \end{array}$$

[3] We do not provide a definition for $(F_1, E_1, \mathtt{i}) \| (F_2, E_2, \mathtt{i})$, because we never merge an infinite trace with another infinite trace directly. Rather, we merge two infinite traces by merging finite portions of one with finite portions of the other.

Likewise, Y^* and Y^ω represent (respectively) the finite and infinite iterations of this dot operator on the set Y. We then define *fairmerge* to be the greatest fixed point of the functional

$$F(Y) = both \cdot Y \cup one,$$

so that

$$fairmerge = both^\omega \cup both^* \cdot one.$$

The triple $(\varphi, \varphi', \psi)$ is in $both^\omega$ if and only if the traces φ, φ', and ψ can be written as infinite concatenations of finite nonempty traces

$$\varphi = \varphi_0\ \varphi_1\ \varphi_2\ \varphi_3\ \dots, \qquad \varphi' = \varphi'_0\ \varphi'_1\ \varphi'_2\ \varphi'_3\ \dots, \qquad \psi = \psi_0\ \psi_1\ \psi_2\ \psi_3\ \dots,$$

such that each ψ_i is in $(\varphi_i \rfloor\!\rfloor \varphi'_i\ \cup\ \varphi'_i \rfloor\!\rfloor \varphi_i\ \cup\ \varphi_i \| \varphi_i)$. Such triples represent the merging of two infinite traces. Likewise, the triple $(\varphi, \varphi', \psi)$ is in $both^* \cdot one$ if and only if the traces φ, φ', and ψ can be written as finite concatenations

$$\varphi = \varphi_0\ \varphi_1\ \varphi_2\ \varphi_3\ \dots\ \varphi_n, \quad \varphi' = \varphi'_0\ \varphi'_1\ \varphi'_2\ \varphi'_3\ \dots\ \varphi'_n, \quad \psi = \psi_0\ \psi_1\ \psi_2\ \psi_3\ \dots\ \psi_n,$$

such that each φ_i, φ'_i and ψ_i (for $i < n$) is a nonempty finite trace, each ψ_i (for $i < n$) is a member of the set $(\varphi_i \rfloor\!\rfloor \varphi'_i\ \cup\ \varphi'_i \rfloor\!\rfloor \varphi_i\ \cup\ \varphi_i \| \varphi_i)$, at least one of φ_n and φ'_n has form $\langle \varepsilon_s, \theta \rangle$, and ψ_n is a member of the set $(\varphi_n \rfloor\!\rfloor \varphi'_n\ \cup\ \varphi'_n \rfloor\!\rfloor \varphi_n)$.

We can now define fair parallel composition on trace sets as follows:

$$T_1 \| T_2 = \{\varphi \mid \varphi_1 \in T_1\ \&\ \varphi_2 \in T_2\ \&\ mergeable(\varphi_1, \varphi_2)\ \&\ (\varphi_1, \varphi_2, \varphi) \in fairmerge\}.$$

The traces of $T_1 \| T_2$ are those traces that result from fair merges of mergeable traces from T_1 and T_2. We therefore define $\mathcal{T}_s[\![c_1 \| c_2]\!] = \mathcal{T}_s[\![c_1]\!] \| \mathcal{T}_s[\![c_2]\!]$.

We summarize the preceding discussion and give the following complete denotational characterization of the trace semantics $\mathcal{T}_s$.

Definition 3.3.1 The trace semantic function $\mathcal{T}_s : \mathsf{Com} \to \mathcal{P}(\Phi)$ is defined by:

$$
\begin{aligned}
\mathcal{T}_s[\![\mathsf{skip}]\!] &= \{\langle (s,\varepsilon,s),(F,\emptyset,\mathtt{f})\rangle \mid s \in S \;\&\; F \in \mathcal{P}_{\mathrm{fin}}(\Delta)\} \\
&\quad \cup \{\langle \varepsilon_s,(F,\{\varepsilon\},\mathtt{p})\rangle \mid s \in S \;\&\; F \supseteq \{\varepsilon\}\} \\
\mathcal{T}_s[\![i{:=}e]\!] &= \{\langle (s,\varepsilon,[s|i=n]),(F,\emptyset,\mathtt{f})\rangle \mid \\
&\qquad\qquad \mathsf{fv}[\![i{:=}e]\!] \subseteq \mathsf{dom}(s) \;\&\; F \in \mathcal{P}_{\mathrm{fin}}(\Delta) \;\&\; (s,n) \in \mathcal{E}[\![e]\!]\} \\
&\quad \cup \{\langle \varepsilon_s,(F,\{\varepsilon\},\mathtt{p})\rangle \mid \mathsf{fv}[\![i{:=}e]\!] \subseteq \mathsf{dom}(s) \;\&\; F \supseteq \{\varepsilon\}\} \\
\mathcal{T}_s[\![c_1;c_2]\!] &= \mathcal{T}_s[\![c_1]\!];\mathcal{T}_s[\![c_2]\!] \\
\mathcal{T}_s[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] &= \mathcal{T}_s[\![b]\!];\mathcal{T}_s[\![c_1]\!] \cup \mathcal{T}_s[\![\neg b]\!];\mathcal{T}_s[\![c_2]\!] \\
\mathcal{T}_s[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!] &= (\mathcal{T}_s[\![b]\!];\mathcal{T}_s[\![c]\!])^{\omega} \cup (\mathcal{T}_s[\![b]\!];\mathcal{T}_s[\![c]\!])^{*};\mathcal{T}_s[\![\neg b]\!] \\
\mathcal{T}_s[\![h?i]\!] &= \{\langle (s,h?n,[s|i=n]),(F,\{h?\},\mathtt{f})\rangle \mid i \in \mathsf{dom}(s) \;\&\; n \in \mathbb{Z} \;\&\; F \in \mathcal{P}_{\mathrm{fin}}(\Delta)\} \\
&\quad \cup \{\langle \varepsilon_s,(F,\{h?\},\mathtt{p})\rangle \mid i \in \mathsf{dom}(s) \;\&\; F \supseteq \{h?\}\} \\
\mathcal{T}_s[\![h!e]\!] &= \{\langle (s,h!n,s),(F,\{h!\},\mathtt{f})\rangle \mid (s,n) \in \mathcal{E}[\![e]\!] \;\&\; F \in \mathcal{P}_{\mathrm{fin}}(\Delta)\} \\
&\quad \cup \{\langle \varepsilon_s,(F,\{h!\},\mathtt{p})\rangle \mid \mathsf{fv}[\![e]\!] \subseteq \mathsf{dom}(s) \;\&\; F \supseteq \{h!\}\} \\
\mathcal{T}_s[\![g \to c]\!] &= \mathcal{T}_s[\![g]\!];\mathcal{T}_s[\![c]\!] \\
\mathcal{T}_s[\![gc_1 \,\square\, gc_2]\!] &= \mathcal{T}_s[\![gc_1]\!] \,\square\, \mathcal{T}_s[\![gc_2]\!] \\
\mathcal{T}_s[\![c_1 \| c_2]\!] &= \mathcal{T}_s[\![c_1]\!] \| \mathcal{T}_s[\![c_2]\!] \\
\mathcal{T}_s[\![c \backslash h]\!] &= \mathcal{T}_s[\![c]\!] \backslash h.
\end{aligned}
$$

$\diamond$

The following result shows that the denotational semantics accurately reflects the operational behavior of programs executing under the assumption of strong fairness.

Proposition 3.3.2 *The denotational and operational characterizations of the fair trace semantics $\mathcal{T}_s$ 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 in Definition 3.1.2. ∎

3.4 Examples

In this section, we sketch how the semantics $\mathcal{T}_s$ can support reasoning about the behavior of programs.

Example 3.4.1 Recall Example 3.1.3, where we defined $C \equiv \mathsf{while\ true\ do\ c!1}$ and considered a computation of the command

$$((\mathsf{a!0} \rightarrow \mathsf{b!1}) \parallel C)\backslash\mathsf{b}.$$

$\mathcal{T}_s[\![\mathsf{a!0} \rightarrow \mathsf{b!1}]\!]$ contains the partial trace $\varphi_1 = \langle (s, \mathsf{a!0}, s), (\{\mathsf{b!}\}, \{\mathsf{b!}\}, \mathtt{p})\rangle$, which represents the blocked mod $\{\mathsf{b!}\}$ computation

$$\rho_1 = \langle \mathsf{a!0} \rightarrow \mathsf{b!0}, s\rangle \xrightarrow{\mathsf{a!0}} \langle \mathsf{b!0}, s\rangle.$$

$\mathcal{T}_s[\![C]\!]$ contains the infinite trace $\varphi_2 = \langle [(s,\varepsilon,s)(s,\mathsf{c!1},s)]^\omega, (\emptyset, \{\mathsf{c!}\}, \mathtt{i})\rangle$, which represents the fair mod $\emptyset$ computation

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

The traces φ_1 and φ_2 are mergeable, because $\neg\mathsf{match}(\{\mathsf{b!}\}, \{\mathsf{c!}\})$ and $\{\mathsf{b!}\} \cap \{\mathsf{c!}\} = \emptyset$. Moreover, $(\varphi_1, \varphi_2, \varphi)$ is in *fairmerge*, where we let φ be the trace

$$\varphi = \langle (s, \mathsf{a!0}, s)[(s,\varepsilon,s)(s,\mathsf{c!1},s)]^\omega, (\{\mathsf{b!}\}, \{\mathsf{b!}, \mathsf{c!}\}, \mathtt{i})\rangle.$$

As a result, φ is in $\mathcal{T}_s[\![(\mathsf{a!0} \rightarrow \mathsf{b!1}) \| C]\!]$; not surprisingly, φ corresponds to the computation

$$\begin{aligned}\rho' = \langle (\mathsf{a!0} \rightarrow \mathsf{b!1}) \parallel C, s\rangle &\xrightarrow{\mathsf{a!0}} \langle \mathsf{b!1} \parallel C, s\rangle \xrightarrow{\varepsilon} \langle \mathsf{b!1} \parallel (\mathsf{c!1};C), s\rangle \\ &\xrightarrow{\mathsf{c!1}} \langle \mathsf{b!1} \parallel C, s\rangle \xrightarrow{\varepsilon} \langle \mathsf{b!1} \parallel (\mathsf{c!1};C), s\rangle \xrightarrow{\mathsf{c!1}} \cdots,\end{aligned}$$

which can be obtained by interleaving ρ_1 and ρ_2. It follows that $\mathcal{T}_s[\![((\mathsf{a!0} \rightarrow \mathsf{b!1}) \| C)\backslash\mathsf{b}]\!]$ contains the trace $\langle (s, \mathsf{a!0}, s)[(s,\varepsilon,s)(s,\mathsf{c!1},s)]^\omega, (\emptyset, \{\mathsf{c!}\}, \mathtt{i})\rangle$, which corresponds to the computation ρ of Example 3.1.3. $\diamond$

Example 3.4.2 Recall Example 3.1.4, which introduced the following programs:

$$\begin{aligned} C &\equiv \mathsf{while\ true\ do\ (a!1} \Box \mathsf{b!1)}, \\ C_1 \| C_2 &\equiv (\mathsf{while\ true\ do\ a!1}) \parallel (\mathsf{b!1} \rightarrow (\mathsf{while\ true\ do\ b!1})), \\ C_p &\equiv \mathsf{while\ true\ do\ (a!0} \Box \mathsf{b?z)}. \end{aligned}$$

$\mathcal{T}_s[\![C]\!]$ contains the trace $\varphi = \langle [(s,\varepsilon,s)(s,\mathsf{a!1},s)]^\omega, (\emptyset, \{\mathsf{a!}, \mathsf{b!}\}, \mathtt{i})\rangle$, corresponding to its fair mod $\emptyset$ computation that enables the directions $\mathsf{a!}$ and $\mathsf{b!}$ infinitely often and yet uses only $\mathsf{a!}$ infinitely often. In contrast, $\mathcal{T}_s[\![C_1 \| C_2]\!]$ contains the trace

$$\varphi' = \langle [(s,\varepsilon,s)(s,\mathsf{a!1},s)]^\omega, (\{\mathsf{b!}\}, \{\mathsf{a!}, \mathsf{b!}\}, \mathtt{i})\rangle$$

but not the trace φ, because its only computations that do not use $\mathsf{b!}$ are fair mod $\{\mathsf{b!}\}$ but not fair mod $\emptyset$.

$\mathcal{T}_s[\![C_p]\!]$ contains the trace $\varphi_p = \langle [(s,\varepsilon,s)(s,\mathsf{a}!0,s)]^\omega, (\emptyset, \{\mathsf{a}!,\mathsf{b}?\}, \mathtt{i})\rangle$, which corresponds to its fair mod $\emptyset$ computation that enables the directions a! and b? infinitely often and repeatedly performs the action a!0. The traces φ and φ_p are mergeable; letting ψ be the trace

$$\psi = \langle [(s\cup t,\varepsilon,s\cup t)(s\cup t,\varepsilon,s\cup t)(s\cup t,\mathsf{a}!0,s\cup t)(s\cup t,\mathsf{a}!1,s\cup t)]^\omega,\ (\emptyset, \{\mathsf{a}!,\mathsf{b}!,\mathsf{b}?\}, \mathtt{i})\rangle,$$

the triple (φ,φ_p,ψ) is in *fairmerge*, and hence ψ is in $\mathcal{T}_s[\![C\|C_p]\!]$. The trace ψ corresponds to the following fair mod $\emptyset$ merging of computations ρ and ρ_p:

$$\begin{array}{rcl}
\langle C \parallel C_p,\ s\cup t\rangle & \xrightarrow{\varepsilon} & \langle(\mathsf{a}!1\,\Box\,\mathsf{b}!1);C \parallel C_p,\ s\cup t\rangle \\
& \xrightarrow{\varepsilon} & \langle(\mathsf{a}!1\,\Box\,\mathsf{b}!1);C \parallel (\mathsf{a}!0\,\Box\,\mathsf{b}?\mathsf{z});C_p,\ s\cup t\rangle \\
& \xrightarrow{\mathsf{a}!0} & \langle(\mathsf{a}!1\,\Box\,\mathsf{b}!1);C \parallel C_p,\ s\cup t\rangle \\
& \xrightarrow{\mathsf{a}!1} & \langle C \parallel C_p,\ s\cup t\rangle \xrightarrow{\varepsilon} \cdots
\end{array}$$

In contrast, the traces φ' and φ_p are not mergeable, because the fairness set $\{\mathsf{b}!\}$ of φ' matches φ_p's set $\{\mathsf{a}!,\mathsf{b}?\}$ of infinitely enabled directions. The lack of mergeability reflects C_p's inability to refuse to synchronize on channel b when $C_1\|C_2$ has a process blocked on the direction b!. $\diamond$

Example 3.4.3 Consider the program $(\mathsf{Stream}_1 \parallel \mathsf{Stream}_2 \parallel \mathsf{Merge})\backslash\mathsf{left}\backslash\mathsf{right}$, where the processes Stream_1, Stream_2 and Merge are defined as follows:

$$\begin{array}{rcl}
\mathsf{Stream}_1 & \equiv & \mathsf{while\ true\ do\ left!1}, \\
\mathsf{Stream}_2 & \equiv & \mathsf{while\ true\ do\ right!2}, \\
\mathsf{Merge} & \equiv & \mathsf{while\ true\ do\ (left?x \to out!x\ \Box\ right?x \to out!x)}.
\end{array}$$

None of the commands have any successful finite traces.

Every infinite trace of Merge has form $\langle\alpha,\ (F, \{\mathsf{left?},\mathsf{right?},\mathsf{out!}\}, \mathtt{i})\rangle$. Therefore, the only traces of $(\mathsf{Stream}_1 \parallel \mathsf{Stream}_2)$ that can be merged with traces of Merge are those whose fairness sets do not contain the directions left!, right! or out?. The only such traces are those that represent computations in which both Stream_1 and Stream_2 make infinite progress. These traces necessarily have form $\langle\beta, (F', \{\mathsf{left!},\mathsf{right!}\}, \mathtt{i})\rangle$, where $\{\mathsf{left!},\mathsf{right!}\}\cap F' = \emptyset$ and β contains infinitely many left!1 actions and infinitely many right!2 actions.

As a consequence, every trace (and therefore every fair computation) of

$$(\mathsf{Stream}_1 \parallel \mathsf{Stream}_2 \parallel \mathsf{Merge})\backslash\mathsf{left}\backslash\mathsf{right}$$

must contain infinitely many out!1 actions and infinitely many out!2 actions. Therefore Merge represents a fair merger of the streams created by Stream_1 and Stream_2. $\diamond$

The following example highlights the connection between fairness and unbounded nondeterminism, using the trace semantics to prove that a single program can terminate with any possible integer value for the identifier x. (The program will also prove useful in certain proofs in Chapter 4.)

Example 3.4.4 Let $\mathsf{Pick\_Int}(\mathsf{x},\mathsf{y},\mathsf{w})$ be the command

$$(\mathsf{Data}(\mathsf{x},\mathsf{y}) \parallel \mathsf{Control}(\mathsf{w}))\backslash \mathsf{a}\backslash \mathsf{b},$$

where $\mathsf{Data}(\mathsf{x},\mathsf{y})$ and $\mathsf{Control}(\mathsf{w})$ are the following programs:

$$\mathsf{Data}(\mathsf{x},\mathsf{y}) \;\equiv\; \begin{array}{l} \mathsf{x}{:=}0;\ \mathsf{y}{:=}1; \\ \mathsf{while}\ \mathsf{y} \neq 0\ \mathsf{do}\ (\mathsf{a}!1 \to \mathsf{x}{:=}\mathsf{x}+1 \;\square\; \mathsf{b}?\mathsf{y} \to \mathsf{skip} \;\square\; \mathsf{a}!1 \to \mathsf{x}{:=}-\mathsf{x}), \end{array}$$

$$\mathsf{Control}(\mathsf{w}) \;\equiv\; \mathsf{w}{:=}1;\ \mathsf{while}\ \mathsf{w} \neq 0\ \mathsf{do}\ (\mathsf{a}?\mathsf{w} \to \mathsf{skip} \square\ \mathsf{b}!0 \to \mathsf{w}{:=}0).$$

For all integers m and n, let s_n abbreviate the state $[\mathsf{x}=n,\mathsf{y}=0]$ and t_n abbreviate the state $[\mathsf{x}=n,\mathsf{y}=1]$, and let u_m abbreviate the state $[\mathsf{w}=m]$. For each $n \in \mathbb{Z}$, let the traces α_n^+, α_n^-, and $\alpha_n^\bullet$ be defined by:

$$\begin{array}{rcl} \alpha_n^+ &=& (t_n,\mathsf{a}!1,t_n)(t_n,\varepsilon,t_{n+1})(t_{n+1},\varepsilon,t_{n+1}), \\ \alpha_n^- &=& (t_n,\mathsf{a}!1,t_n)(t_n,\varepsilon,t_{-n})(t_{-n},\varepsilon,t_{-n}), \\ \alpha_n^\bullet &=& (t_n,\mathsf{b}?0,s_n)(s_n,\varepsilon,s_n)(s_n,\varepsilon,s_n). \end{array}$$

Intuitively, α_n^+ represents the transitions made by $\mathsf{Data}(\mathsf{x},\mathsf{y})$ in executing the code fragment

$$\mathsf{a}!1 \to \mathsf{x}{:=}\mathsf{x}+1$$

from the state $[\mathsf{x}=n,\mathsf{y}=1]$, and then entering the loop again by verifying that the condition $\mathsf{y} \neq 0$ holds. Similarly, α_n^- represents the transitions made by $\mathsf{Data}(\mathsf{x},\mathsf{y})$ in executing the fragment

$$\mathsf{a}!1 \to \mathsf{x}{:=}-\mathsf{x}$$

from the same state and reentering the loop. Finally, the trace $\alpha_n^\bullet$ represents the transitions made by $\mathsf{Data}(\mathsf{x},\mathsf{y})$ in which it receives the value 0 along channel b and finally terminates.

For every nonnegative integer n, we can also define the simple traces

$$\begin{array}{rcl} \gamma_n^+ &=& (s_0,\varepsilon,s_0)(s_0,\varepsilon,t_0)(t_0,\varepsilon,t_0)\alpha_0^+\alpha_1^+\ldots\alpha_{n-1}^+\alpha_n^\bullet, \\ \gamma_n^- &=& (s_0,\varepsilon,s_0)(s_0,\varepsilon,t_0)(t_0,\varepsilon,t_0)\alpha_0^+\alpha_1^+\ldots\alpha_{n-1}^+\alpha_n^-\alpha_{-n}^\bullet. \end{array}$$

Intuitively, γ_n^+ (respectively, γ_n^-) represents a computation of $\mathsf{Data}(\mathsf{x},\mathsf{y})$ that sets x to n (respectively, $-n$) and then terminates. Moreover, for each $n \geq 0$, the traces $\langle \gamma_n^+, (\emptyset, \{\mathsf{a}!,\mathsf{b}?\}, \mathtt{f})\rangle$ and $\langle \gamma_n^-, (\emptyset, \{\mathsf{a}!,\mathsf{b}?\}, \mathtt{f})\rangle$ are in $\mathcal{T}_s[\![\mathsf{Data}(\mathsf{x},\mathsf{y})]\!]$.

We can also define simple traces

$$\beta = (u_1, \mathsf{a?1}, u_1)(u_1, \varepsilon, u_1)(u_1, \varepsilon, u_1), \qquad \beta^\bullet = (u_1, \mathsf{b!0}, u_1)(u_1, \varepsilon, u_0)(u_0, \varepsilon, u_0),$$

and (for each $n \geq 0$) $\xi_n = (u_0, \varepsilon, u_1)(u_1, \varepsilon, u_1)\beta^{n-1}\beta^\bullet$. The trace ξ_n represents a computation of $\mathsf{Control(w)}$ that executes the guard $\mathsf{a?0}$ $n-1$ times and then executes the guard $\mathsf{b!0}$ and terminates. For each nonnegative n, the trace $(\xi_n, (\emptyset, \{\mathsf{a?}, \mathsf{b!}\}, \mathtt{f}))$ is in $\mathcal{T}_s[\![\mathsf{Control(w)}]\!]$.

Finally, using the notation $(s, \lambda, s)^3$ to abbreviate the trace $(s, \lambda, s)(s, \lambda, s)(s, \lambda, s)$, we can define the following simple traces, for all integer values n:

$$\begin{aligned}
\theta_n^+ &= (t_n \cup u_1, \varepsilon, t_n \cup u_1)(t_n \cup u_1, \varepsilon, t_{n+1} \cup u_1)[(t_{n+1} \cup u_1, \varepsilon, t_{n+1} \cup u_1)]^3 \\
\theta_n^- &= (t_n \cup u_1, \varepsilon, t_n \cup u_1)(t_n \cup u_1, \varepsilon, t_{-n} \cup u_1)[(t_{-n} \cup u_1, \varepsilon, t_{-n} \cup u_1)]^3 \\
\theta_n^\bullet &= (t_n \cup u_1, \varepsilon, s_n \cup u_1)(s_n \cup u_1, \varepsilon, s_n \cup u_0)[(s_n \cup u_0, \varepsilon, s_n \cup u_0)]^3.
\end{aligned}$$

Intuitively, θ_n^+ arises from a merge of α_n^+ and β, θ_n^- arises from a merge of α_n^- and β, and $\theta_n^\bullet$ arises from a merge of $\alpha_n^\bullet$ and $\beta^\bullet$. Letting ι be the trace

$$(s_0 \cup u_0, \varepsilon, s_0 \cup u_0)(s_0 \cup u_0, \varepsilon, t_0 \cup u_0)(t_0 \cup u_0, \varepsilon, t_0 \cup u_0)(t_0 \cup u_0, \varepsilon, t_0 \cup u_1)(t_0 \cup u_1, \varepsilon, t_0 \cup u_1),$$

representing one possible merging of the "initial" portions of γ_n^+ and ξ_n, it follows that, for each nonnegative n, the traces

$$\langle \iota\theta_0^+\theta_1^+ \ldots \theta_{n-1}^+\theta_n^\bullet, (\emptyset, \emptyset, \{\mathsf{a!}, \mathsf{a?}, \mathsf{b!}, \mathsf{b?}\}, \mathtt{f})\rangle$$

and

$$\langle \iota\theta_0^+\theta_1^+ \ldots \theta_{n-1}^+\theta_n^-\theta_{-n}^\bullet, (\emptyset, \emptyset, \{\mathsf{a!}, \mathsf{a?}, \mathsf{b!}, \mathsf{b?}\}, \mathtt{f})\rangle$$

are in $\mathcal{T}_s[\![\mathsf{Data(x,y)} \| \mathsf{Control(w)}]\!]$. Consequently, the traces

$$\langle \iota\theta_0^+\theta_1^+ \ldots \theta_{n-1}^+\theta_n^\bullet, (\emptyset, \emptyset, \mathtt{f})\rangle \quad \text{and} \quad \langle \iota\theta_0^+\theta_1^+ \ldots \theta_{n-1}^+\theta_n^-\theta_{-n}^\bullet, (\emptyset, \emptyset, \mathtt{f})\rangle$$

are in $\mathcal{T}_s[\![\mathsf{Pick\_Int(x,y,w)}]\!]$. These traces reflect the fact that, for every integer n, there is a strongly fair computation of $\mathsf{Pick\_Int(x,y,w)}$ that terminates in a state where the identifier x has value n. ◇

Example 3.4.5 As a postscript to the previous example, let γ be the infinite simple trace

$$(s_0, \varepsilon, s_0)(s_0, \varepsilon, t_0)(t_0, \varepsilon, t_0)\alpha_0^+\alpha_1^+ \ldots \alpha_{n-1}^+ \ldots,$$

so that γ represents a computation of $\mathsf{Data(x,y)}$ that continually increments x. Similarly, let $\xi = (u_0, \varepsilon, u_1)(u_1, \varepsilon, u_1)\beta^\omega$, so that ξ represents a computation of $\mathsf{Control(w)}$ in which w is never set to 0.

The trace $\langle \gamma, (\emptyset, \{\mathtt{a!}, \mathtt{b?}\}, \mathtt{i}) \rangle$ is in $\mathcal{T}_s[\![\mathsf{Data(x,y)}]\!]$, and the trace $\langle \xi, (\emptyset, \{\mathtt{a?}, \mathtt{b!}\}, \mathtt{i}) \rangle$ is in $\mathcal{T}_s[\![\mathsf{Control(w)}]\!]$. It follows that the trace

$$\langle \iota\theta_0^+\theta_1^+ \ldots \theta_n^+ \ldots, (\emptyset, \{\mathtt{a!}, \mathtt{a?}, \mathtt{b!}, \mathtt{b?}\}, \mathtt{i}) \rangle$$

is in $\mathcal{T}_s[\![\mathsf{Data(x,y)} \| \mathsf{Control(w)}]\!]$, and hence the trace $\langle \iota\theta_0^+\theta_1^+ \ldots \theta_n^+ \ldots, (\emptyset, \emptyset, \mathtt{i}) \rangle$ is in $\mathcal{T}_s[\![\mathsf{Pick\_Int(x,y,w)}]\!]$. The existence of this trace reflects the fact that $\mathsf{Pick\_Int(x,y,w)}$ has strongly fair computations that never terminate. $\diamond$

These examples all illustrate how the strongly fair trace semantics can be used to reason about strongly fair program behavior. What we have not yet addressed, however, is whether a simpler semantics (that is, a semantics constructed at a higher level abstraction) would also support such reasoning: are the fairness sets F and sets E of enabled directions really necessary for reasoning about strong process fairness? We address this question in the next chapter, where we discuss *full abstraction*. Intuitively a semantics is fully abstract if it provides precisely the right level of detail to support compositional reasoning about program behavior. We show in the next chapter that the semantics $\mathcal{T}_s$ can be made fully abstract and that the sets F and E play a vital role in modeling strongly fair computations.

Chapter 4

Full Abstraction for Strong Fairness

A single language can have several different semantics, each suited for reasoning about a different type of program behavior. The struggle for each semantics is to find a balance between supporting compositional reasoning and maintaining an appropriate level of abstraction. For example, a semantics intended to support reasoning about the sequence of states encountered along a computation must capture intermediate states in some fashion. In contrast, that same semantics may be unnecessarily complex for reasoning about a behavior that ignores intermediate states; a semantics that also ignores intermediate states may provide a better level of abstraction.

Given a semantics and a notion of program behavior, how do we determine whether—and, if so, how well—the semantics supports reasoning about the behavior? One well-known criterion for judging the utility of a semantics is *full abstraction* [Mil75]. Informally, a semantics is fully abstract with respect to a given notion of behavior if it gives identical meanings to program terms exactly when those terms exhibit identical behaviors in all program contexts. In essence, a fully abstract semantics supplies precisely the right level of detail to support compositional reasoning about a given notion of behavior.

In this chapter, we introduce a natural notion of strongly fair behavior, and we show how the semantics $\mathcal{T}_s$ introduced in Chapter 3 can be adapted—through the introduction of suitable closure conditions on trace sets—to yield full abstraction with respect to this behavior. We also introduce several additional notions of strongly fair behavior and show how the same general framework, with small changes to the specific semantics, yields full abstraction with respect to these behaviors as well. Having a common underlying framework significantly simplifies the construction of the additional semantics: the different traces share the same general structure, the semantic operators represent the same type of operational behavior, and the full-abstraction proofs rely on the same observations and necessary lemmas.

4.1 Soundness and Full Abstraction

A **program context** $P[-]$ is a program with one or more "holes" into which a command can be inserted. $P[c]$ is the program that results from filling the holes of $P[-]$ with command c, provided c "makes sense" in the given hole. For example, if $P[-]$ is the context $([-] \to \mathsf{skip})$, then $P[\mathsf{a!0}]$ is the command $(\mathsf{a!0} \to \mathsf{skip})$, whereas $P[\mathsf{skip}]$ is undefined.[1]

A **behavior notion** is the set of program actions assumed to be visible to an external observer. For example, the input–output behavior of a program provides a black-box view of programs: a program's initial input and final result are considered observable, but its intermediate states are not. For communicating processes, there are several natural notions of behavior to consider, such as a program's sequences of communications or the states encountered along its possible executions. For most of this chapter, we focus on the following form of *state trace behavior*; in Section 4.5, we discuss several other notions of strongly fair behavior.

Definition 4.1.1 The state trace behavior $\mathcal{M} : \mathsf{Com} \to \mathcal{P}(S^\infty \cup S^*\delta)$ is defined by:

$$\begin{array}{rcl}\mathcal{M}[\![c]\!] &=& \{s_0s_1\ldots s_k \mid \langle c, s_0\rangle \xrightarrow{\varepsilon} \langle c_1, s_1\rangle \xrightarrow{\varepsilon} \cdots \xrightarrow{\varepsilon} \langle c_k, s_k\rangle \mathsf{term}\} \\ &\cup& \{s_0s_1\ldots s_k\delta \mid \langle c_0, s_0\rangle \xrightarrow{\varepsilon} \langle c_1, s_1\rangle \xrightarrow{\varepsilon} \cdots \xrightarrow{\varepsilon} \langle c_k, s_k\rangle \mathsf{dead}\} \\ &\cup& \{s_0s_1\ldots s_k\ldots \mid \langle c_0, s_0\rangle \xrightarrow{\varepsilon} \langle c_1, s_1\rangle \xrightarrow{\varepsilon} \cdots \xrightarrow{\varepsilon} \langle c_k, s_k\rangle \xrightarrow{\varepsilon} \cdots \text{ is fair}\},\end{array}$$

where we define $S^*\delta = \{s_0s_1\ldots s_k\delta \mid \forall i \in 0..k.\ s_i \in S\}$ ◇

The state trace behavior $\mathcal{M}$ incorporates the assumption 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. This notion of behavior captures exactly the information necessary for reasoning about the linear-time temporal logic properties of programs; the assumption that every state change is detectable corresponds to the inclusion of a next-time operator in the temporal logic. Finally, this behavior reflects the assumption that deadlock is distinguishable both from successful termination and from infinite chattering.

A semantics is **sound** with respect to a given notion of behavior if whenever two terms have the same meaning, they induce the same behaviors in all program contexts. Thus, whenever a sound semantics identifies two terms, either term can always be substituted for the other in any program without affecting the program's observable behavior. However, when a sound semantics gives different meanings to program terms, the terms may or may not be safely interchangeable: they may have different meanings either because they induce different behaviors

[1] To be more precise and pedantic, each context should be tagged with a label that indicates whether the hole takes guards or commands and a set of identifiers that are forbidden to be free in any command filling the hole. For example, the context $([-] \to \mathsf{skip} \parallel \mathsf{x}{:=}1)$ would be tagged to indicate that it accepts guards that do not have free identifier x.

in some program context or because the semantics provides too low a level of abstraction. For example, the semantics that maps each term to its own syntactic representation is sound for any notion of program behavior, but it is not very useful: two terms have the same meaning in this semantics if and only if they are identical, and hence they necessarily behave the same in all program contexts.

A semantics is **(equationally) fully abstract** [Mil75] with respect to a notion of behavior if it assigns two terms the same meaning exactly when they induce the same behaviors in all program contexts. A fully abstract semantics faithfully captures Morris-style contextual equivalence [Mor68], identifying two terms if and only if they are contextually equivalent. Thus a fully abstract semantics makes precisely the right distinctions and retains just enough detail to support compositional reasoning about the given behavior. When the semantic and behavioral domains both come equipped with approximation orderings, we can also speak of a stronger property called *inequational full abstraction*: a semantics is **inequationally fully abstract** with respect to a notion of behavior provided that the meaning of a term c approximates that of c' exactly when the behavior of c approximates that of c' in all program contexts. Inequational full abstraction necessarily implies equational full abstraction.

4.2 Closed Trace Sets

The semantics $\mathcal{T}_s$ introduced in Chapter 3 is *sound* with respect to $\mathcal{M}$: for all commands c and c',

$$\mathcal{T}_s[\![c]\!] = \mathcal{T}_s[\![c']\!] \implies \forall P[-].\mathcal{M}[\![P[c]]\!] = \mathcal{M}[\![P[c']]\!].$$

The soundness of $\mathcal{T}_s$ for $\mathcal{M}$ follows directly from the compositionality of $\mathcal{T}_s$, the monotonicity of the semantic operators, and the fact that the state traces in each $\mathcal{M}[\![P[c]]\!]$ correspond to the traces of $P[c]$ that contain only ε-transitions. However, $\mathcal{T}_s$ is not fully abstract with respect to $\mathcal{M}$, because it makes distinctions between programs that behave equivalently in all contexts. These inappropriate distinctions arise because certain combinations of traces convey exactly the same information as do certain other combinations.

For example, consider the following commands C_1 and C_2:

$$\begin{aligned} C_1 &\equiv (\mathsf{a!0} \to \mathsf{b!0}) \,\Box\, (\mathsf{a!0} \to \mathsf{c!0}), \\ C_2 &\equiv (\mathsf{a!0} \to \mathsf{b!0}) \,\Box\, (\mathsf{a!0} \to \mathsf{c!0}) \,\Box\, (\mathsf{a!0} \to (\mathsf{b!0} \Box \mathsf{c!0})). \end{aligned}$$

The traces $\varphi_p = \langle (s, \mathsf{a!0}, s), (\{\mathsf{b!}, \mathsf{c!}\}, \{\mathsf{b!}, \mathsf{c!}\}, \mathsf{p})\rangle$ and $\varphi_f = \langle (s, \mathsf{a!0}, s)(s, \mathsf{b!0}, s), (\emptyset, \{\mathsf{a!}, \mathsf{b!}, \mathsf{c!}\}, \mathtt{f})\rangle$ 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. In essence, the partial trace φ_p conveys no information that is not already conveyed by the partial traces

$$\varphi_1 = \langle(s,\mathsf{a!0},s),(\{\mathsf{b!}\},\{\mathsf{b!}\},\mathtt{p})\rangle \qquad \text{and} \qquad \varphi_2 = \langle(s,\mathsf{a!0},s),(\{\mathsf{c!}\},\{\mathsf{c!}\},\mathtt{p})\rangle,$$

both of which are possible for C_1 as well as for C_2. More generally, the information provided by the combination of partial traces $\langle\alpha,(F_1,E_1,\mathtt{p})\rangle$ and $\langle\alpha,(F_2,E_2,\mathtt{p})\rangle$ encompasses any information provided by the partial trace $\langle\alpha,(F_1\cup F_2,E_1\cup E_2,\mathtt{p})\rangle$. Consequently, it is safe to assume that the latter trace exists in any trace set that contains the first two. Likewise, the finite trace φ_f above conveys no more information than that conveyed by the finite trace

$$\langle(s,\mathsf{a!0},s)(s,\mathsf{b!0},s),(\emptyset,\{\mathsf{a!},\mathsf{b!}\},\mathtt{f})\rangle,$$

which is possible for both C_1 and C_2. More generally, it is safe to assume that the finite or infinite trace $\langle\alpha,(F',E',R)\rangle$ is in any trace set containing the trace $\langle\alpha,(F,E,R)\rangle$, provided $E\subseteq E'$, $F\subseteq F'$, and $R\in\{\mathtt{f},\mathtt{i}\}$.

A similar situation arises with the following guarded commands C_3 and C_4:

$$\begin{array}{lcl} C_3 & \equiv & (\mathsf{a!0}\rightarrow\mathsf{b!0})\,\square\,(\mathsf{a!0}\rightarrow(\mathsf{b!0}\,\square\,\mathsf{c!0}\,\square\,\mathsf{d!0})) \\ C_4 & \equiv & (\mathsf{a!0}\rightarrow\mathsf{b!0})\,\square\,(\mathsf{a!0}\rightarrow(\mathsf{b!0}\,\square\,\mathsf{c!0}\,\square\,\mathsf{d!0}))\,\square\,(\mathsf{a!0}\rightarrow(\mathsf{b!0}\,\square\,\mathsf{c!0})). \end{array}$$

The two partial traces

$$\varphi_1 = \langle(s,\mathsf{a!0},s),(\{\mathsf{b!}\},\{\mathsf{b!}\},\mathtt{p})\rangle, \qquad \varphi_2 = \langle(s,\mathsf{a!0},s),(\{\mathsf{b!},\mathsf{c!},\mathsf{d!}\},\{\mathsf{b!},\mathsf{c!},\mathsf{d!}\},\mathtt{p})\rangle$$

are possible for both C_3 and C_4, but the partial trace $\varphi = \langle(s,\mathsf{a!0},s),(\{\mathsf{b!},\mathsf{c!}\},\{\mathsf{b!},\mathsf{c!}\},\mathtt{p})\rangle$ is possible only for C_4. However, for reasons similar to those above, the two commands behave the same in all program contexts. Any information conveyed by the trace φ is also conveyed by the combination of traces φ_1 and φ_2, both of which are possible for C_3 and C_4. More generally, the combination of partial traces

$$\langle\alpha,(F_1,E_1,\mathtt{p})\rangle \qquad \text{and} \qquad \langle\alpha,(F_2,E_2,\mathtt{p})\rangle$$

encompasses any information conveyed by the partial trace $\langle\alpha,(F,E,\mathtt{p})\rangle$, provided $E_1\subseteq E\subseteq E_2$, $F_1\subseteq F\subseteq F_2$, and $F\supseteq E$.

Similar observations led to the imposition of saturation closure conditions in Hennessy's acceptance trees model [Hen85] and downwards and convex closure conditions for refusal sets in the failures model for CSP [BHR84]. The need for these closure conditions arises from our desire to model deadlock and is orthogonal to our attempts to model fairness. However, other fairness-related difficulties also arise, due to the interactions between traces' fairness sets F and enabling sets E.

To understand why, recall that, in the infinite trace $\langle\alpha,(F,E,\mathtt{i})\rangle$, the set F represents constraints on the type of context in which α will represent a fair transition sequence, and the set E indicates which directions are enabled infinitely often along that sequence. Therefore, distinguishing between a process with the trace $\langle\alpha,(F,E,\mathtt{i})\rangle$ and one with the trace $\langle\alpha,(F,E',\mathtt{i})\rangle$ requires a context with a subcomponent Q that can be enabled infinitely by E but not E' (or vice versa). When placed in such a context, one process can perform α fairly while Q blocks, whereas the other process cannot perform α without eventually synchronizing with Q. For example, suppose that the command C (but not C') has the trace $\langle\alpha,(\emptyset,\{\mathsf{a}!,\mathsf{b}!\},\mathtt{i})\rangle$ and that C' has the trace $\langle\alpha,(\emptyset,\{\mathsf{a}!,\mathsf{b}!,\mathsf{c}!\},\mathtt{i})\rangle$. The two commands can be distinguished by a context like the following:

$$P[-]\equiv([-]\parallel \mathsf{c?x}\rightarrow \mathsf{flag{:=}1})\backslash\mathsf{c}.$$

The program $P[C]$ has a fair behavior in which C performs the transition sequence α and in which the identifier flag never gets set to 1. In contrast, $P[C']$ has no such behavior: if C' tries to perform the sequence α, the context's guard c?x will be enabled infinitely often, thereby forcing a synchronization that leads to flag getting set to the value 1.

Distinguishing a process with trace $\langle\alpha,(F,E,\mathtt{i})\rangle$ from one with trace $\langle\alpha,(F',E,\mathtt{i})\rangle$ requires a different approach. 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. For example, recall the commands C, $C_1\|C_2$ and C_p from Example 3.1.4:

$$\begin{array}{rcl} C & \equiv & \mathsf{while\ true\ do\ (a!0\,\Box\,b!1)},\\ C_1\|C_2 & \equiv & \mathsf{(while\ true\ do\ a!0)}\parallel\mathsf{(b!1\rightarrow while\ true\ do\ b!1)},\\ C_p & \equiv & \mathsf{while\ true\ do\ (a!0\,\Box\,b?z)}. \end{array}$$

Letting $\alpha=[(s,\varepsilon,s)(s,\mathsf{a}!1,s)]^{\omega}$, the commands C and $C_1\|C_2$ have (respectively) the traces $\langle\alpha,(\emptyset,\{\mathsf{a}!,\mathsf{b}!\},\mathtt{i})\rangle$ and $\langle\alpha,(\{\mathsf{b}!\},\{\mathsf{a}!,\mathsf{b}!\},\mathtt{i})\rangle$. The context

$$([-]\parallel\mathsf{while\ true\ do\ (a!0\,\Box\,b?z)})\backslash\mathsf{b}$$

can distinguish these commands, because the b?z command appears within a guarded choice. The context's infinite enabling of b? is sufficient to force $C_1\|C_2$ to synchronize on channel b, and yet C may refrain fairly from using b at all.

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 $\mathcal{T}_s$:

$$\begin{array}{rcl} C_5 & \equiv & \mathsf{(a!0\rightarrow b!0\rightarrow c!0)\,\Box\,(a!0\rightarrow b?x)\,\Box\,(a!0\rightarrow (b!0\,\Box\,b?x))},\\ C_6 & \equiv & \mathsf{(a!0\rightarrow b!0\rightarrow c!0)\,\Box\,(a!0\rightarrow b?x)\,\Box\,(a!0\rightarrow (b!0\,\Box\,b?x))\,\Box\,(a!0\rightarrow b!0)}. \end{array}$$

These commands exhibit the same potential for deadlock (i.e., they share the same partial traces), and they can perform the same sequences of communications. The only potential difference between these commands is that C_6 can perform the successfully terminating sequence of actions $[\mathsf{a!0\ b!0}]$ without enabling input on channel b. This potential difference is reflected in their trace sets: the trace $\varphi = \langle (s, \mathsf{a!0}, s)(s, \mathsf{b!0}, s), (\emptyset, \{\mathsf{a!}, \mathsf{b!}, \mathsf{b?}\}, \mathtt{f})\rangle$ is possible for both C_5 and C_6, whereas the trace $\varphi' = \langle (s, \mathsf{a!0}, s)(s, \mathsf{b!0}, s), (\emptyset, \{\mathsf{a!}, \mathsf{b!}\}, \mathtt{f})\rangle$ is possible only for C_6. As a result, the only possible way to distinguish C_6 from C_5 is to distinguish φ' from φ, which requires an argument based on fairness. In particular, distinguishing between C_6 and C_5 requires a context that allows each C_i to repeatedly perform a!0 followed by b!0, while permitting an observer to determine when the direction b? is enabled only finitely often along the infinite computation. Therefore, any potential distinguishing context must have at least the following three separate components:

1. A loop that repeats the relevant C_i infinitely many times.

   Intuitively, when C_6 is placed in this loop, it can repeatedly perform a!0 followed by 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 necessarily enables b? infinitely often.

2. A component—placed in parallel with the aforementioned loop—that can block only when b? is not enabled by the relevant C_i infinitely often.

   To block when b? is enabled only finitely often, this component must contain a guard that blocks when trying to perform output on channel b. Because blocking can happen only when synchronization is required, both this component and the loop must appear in the scope of channel restriction on channel b.

3. A component that repeatedly offers input opportunities for each of the loop's attempted b!0 actions.

   The loop has communication on channel b restricted and yet needs to perform the action b!0 infinitely often. Consequently, it requires an additional component that repeatedly offers input opportunities on channel b, thus permitting synchronization.

Consequently, any distinguishing context must have the following general form:

$$(\mathsf{while\ true\ do}\ [-] \parallel \mathsf{b!0} \rightarrow \mathsf{flag}{:=}1 \parallel \mathsf{while\ true\ do\ b?v})\backslash \mathsf{b}.$$

However, the second component (which is intended to block in certain situations) is always provided infinitely many synchronization opportunities by the third component. As a result, it can never become permanently blocked, regardless of whether C_5 or C_6 is inserted into the context. In effect, C_5's enabling (but non-use) of b? is obscured by C_5's use of the matching direction b!. Because every possible distinguishing context must have the same general form,

C_5 and C_6 are behaviorally indistinguishable. More generally, a trace set containing the finite or infinite trace $\varphi = \langle \alpha, (F, E \cup X, R) \rangle$, with $X \cap \mathsf{vis}(\alpha) = \emptyset$ and $\overline{X} \subseteq \mathsf{vis}(\alpha)$, cannot be distinguished from one that also contains the trace $\langle \alpha, (F, E, R) \rangle$.

The final source of inappropriate distinction arises from pairs of traces whose fairness and enabling sets conflict with one another. For example, consider the commands $C_7 \equiv G_1 \Box G_2$ and $C_8 \equiv G_1 \Box G_2 \Box G_3$, where the guarded commands G_1, G_2, and G_3 are defined as follows:

$$\begin{array}{lcl} G_1 & \equiv & \mathsf{b!0} \rightarrow \mathsf{while\ true\ do\ (b!0} \Box \mathsf{a?x} \Box \mathsf{a!0)} \\ G_2 & \equiv & \mathsf{b!0} \rightarrow \mathsf{((while\ true\ do\ b!0)} \parallel \mathsf{(a?x} \rightarrow \mathsf{while\ true\ do\ a?x))} \\ G_3 & \equiv & \mathsf{b!0} \rightarrow \mathsf{while\ true\ do\ (b!0} \Box \mathsf{a?x).} \end{array}$$

Letting α represent the simple trace $[(s, \mathtt{b!0}, s)(s, \varepsilon, s)]^{\omega}$, the trace sets of C_7 and C_8 both contain the traces $\varphi_1 = \langle \alpha, (\emptyset, \{\mathtt{b!}, \mathtt{a?}, \mathtt{a!}\}, \mathtt{i}) \rangle$ and $\varphi_2 = \langle \alpha, (\{\mathtt{a?}\}, \{\mathtt{b!}, \mathtt{a?}\}, \mathtt{i}) \rangle$, but the trace $\varphi_3 = \langle \alpha, (\emptyset, \{\mathtt{b!}, \mathtt{a?}\}, \mathtt{i}) \rangle$ is possible only for C_8. To distinguish between C_7 and C_8 requires a context in which φ_3 can be distinguished from both φ_1 and φ_2 *at the same time.* As discussed previously, distinguishing φ_3 from φ_1 requires a context that places the relevant C_i in parallel with a component that blocks while trying to perform input on channel a. In contrast, distinguishing φ_3 from φ_2 requires a context that places the relevant C_i in parallel with a component that enables output on channel a infinitely often and yet does not block. Therefore, 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 intended "blocking" component is enabled infinitely often by the second component, regardless of which command is inserted. Thus, no context can possibly distinguish the commands C_7 and C_8. More generally, whenever the traces $\langle \alpha, (F \cup \{d\}, E, \mathtt{i}) \rangle$ and $\langle \alpha, (F, E \cup \{\bar{d}\}, \mathtt{i}) \rangle$ are in a trace set T, it is impossible to determine whether (and it is safe to assume that) the trace $\langle \alpha, (F, E, \mathtt{i}) \rangle$ is in T as well.

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

Definition 4.2.1 Given a fair trace set T, the **closure** of T (written $T^{\dagger}$) is the smallest set containing T and satisfying the following conditions:

- *Union*: If $\langle \alpha, (F_1, E_1, \mathtt{p}) \rangle$ and $\langle \alpha, (F_2, E_2, \mathtt{p}) \rangle$ are in $T^{\dagger}$, then $\langle \alpha, (F_1 \cup F_2, E_1 \cup E_2, \mathtt{p}) \rangle$ is in $T^{\dagger}$.
- *Convexity*: If $\langle \alpha, (F_1, E_1, \mathtt{p}) \rangle$ and $\langle \alpha, (F_2, E_2, \mathtt{p}) \rangle$ are in $T^{\dagger}$, $E_1 \subseteq E \subseteq E_2$, $F_1 \subseteq F \subseteq F_2$, and $F \supseteq E$, then $\langle \alpha, (F, E, \mathtt{p}) \rangle$ is in $T^{\dagger}$.
- *Superset*: If $\langle \alpha, (F, E, R) \rangle$ is in $T^{\dagger}$, $R \in \{\mathtt{f}, \mathtt{i}\}$, $F \subseteq F'$, and $E \subseteq E'$, then $\langle \alpha, (F', E', R) \rangle$ is in $T^{\dagger}$.
- *Displacement*: If $\langle \alpha, (F, E \cup X, R) \rangle$ is in $T^{\dagger}$, $R \in \{\mathtt{f}, \mathtt{i}\}$, $X \cap \mathsf{vis}(\alpha) = \emptyset$, and $\overline{X} \subseteq \mathsf{vis}(\alpha)$, then $\langle \alpha, (F, E, R) \rangle$ is in $T^{\dagger}$.

- *Contention*: If $\langle \alpha, (F \cup \{d\}, E, \mathtt{i})\rangle$ and $\langle \alpha, (F, E \cup \{\bar{d}\}, \mathtt{i})\rangle$ are in $T^\dagger$, then $\langle \alpha, (F, E, \mathtt{i})\rangle$ is in $T^\dagger$. $\diamond$

Closure is obviously monotonic (if $T_1 \subseteq T_2$, then $T_1^\dagger \subseteq T_2^\dagger$) and idempotent ($T^\dagger = (T^\dagger)^\dagger$). Moreover, any trace introduced by a closure condition has the same tag as the trace(s) that led to its introduction: for example, convexity introduces partial traces when certain partial traces are in the set, and contention introduces infinite traces when certain infinite traces are in the set. As a result, if the sets T_f, T_p and T_i contain only finite, partial and infinite traces, respectively, then $(T_f \cup T_p \cup T_i)^\dagger = T_f^\dagger \cup T_p^\dagger \cup T_i^\dagger$. We use this fact in many subsequent definitions.

As we shall see in Section 4.4, these closure conditions are precisely what is needed to obtain full abstraction. Let $\mathcal{P}^\dagger\Phi$ be the set of closed sets of fair traces. We define a *closed trace semantic function* $\mathcal{T}_s^\dagger : \mathsf{Com} \to \mathcal{P}^\dagger\Phi$ denotationally, modifying the semantic equations given for $\mathcal{T}_s$ in Definition 3.3.1 by building the closure property into each clause. Letting $\mathcal{T}_s^\dagger[\![b]\!] = \mathcal{T}_s[\![b]\!]^\dagger$, we define $\mathcal{T}_s^\dagger$ as follows.

Definition 4.2.2 The closed trace semantic function $\mathcal{T}_s^\dagger : \mathsf{Com} \to \mathcal{P}^\dagger\Phi$ is defined by:

$$
\begin{aligned}
\mathcal{T}_s^\dagger[\![\mathsf{skip}]\!] &= \{\langle (s,\varepsilon,s),(F,\emptyset,\mathtt{f})\rangle \mid s \in S \;\&\; F \in \mathcal{P}_{\text{fin}}(\Delta)\}^\dagger,\\
&\quad \cup \{\langle \varepsilon_s,(F,\{\varepsilon\},\mathtt{p})\rangle \mid s \in S \;\&\; F \supseteq \{\varepsilon\}\}^\dagger,\\
\mathcal{T}_s^\dagger[\![i{:=}e]\!] &= \{\langle (s,\varepsilon,[s|i=n]),(F,\emptyset,\mathtt{f})\rangle \mid \\
&\qquad\qquad \mathsf{fv}[\![i{:=}e]\!] \subseteq \mathsf{dom}(s) \;\&\; F \in \mathcal{P}_{\text{fin}}(\Delta) \;\&\; (s,n) \in \mathcal{E}[\![e]\!]\}^\dagger\\
&\quad \cup \{\langle \varepsilon_s,(F,\{\varepsilon\},\mathtt{p})\rangle \mid \mathsf{fv}[\![i{:=}e]\!] \subseteq \mathsf{dom}(s) \;\&\; F \supseteq \{\varepsilon\}\}^\dagger\\
\mathcal{T}_s^\dagger[\![c_1;c_2]\!] &= (\mathcal{T}_s^\dagger[\![c_1]\!];\mathcal{T}_s^\dagger[\![c_2]\!])^\dagger\\
\mathcal{T}_s^\dagger[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] &= (\mathcal{T}_s^\dagger[\![b]\!];\mathcal{T}_s^\dagger[\![c_1]\!] \cup \mathcal{T}_s^\dagger[\![\neg b]\!];\mathcal{T}_s^\dagger[\![C_2]\!])^\dagger\\
\mathcal{T}_s^\dagger[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!] &= ((\mathcal{T}_s^\dagger[\![b]\!];\mathcal{T}_s^\dagger[\![c]\!])^\omega \cup (\mathcal{T}_s^\dagger[\![b]\!];\mathcal{T}_s^\dagger[\![c]\!])^*;\mathcal{T}_s^\dagger[\![\neg b]\!])^\dagger\\
\mathcal{T}_s^\dagger[\![h?i]\!] &= \{\langle (s,h?n,[s|i=n]),(F,\{h?\},\mathtt{f})\rangle \mid i \in \mathsf{dom}(s) \;\&\; n \in \mathbb{Z} \;\&\; F \in \mathcal{P}_{\text{fin}}(\Delta)\}^\dagger\\
&\quad \cup \{\langle \varepsilon_s,(F,\{h?\},\mathtt{p})\rangle \mid i \in \mathsf{dom}(s) \;\&\; F \supseteq \{h?\}\}^\dagger\\
\mathcal{T}_s^\dagger[\![h!e]\!] &= \{\langle (s,h!n,s),(F,\{h!\},\mathtt{f})\rangle \mid (s,n) \in \mathcal{E}[\![e]\!] \;\&\; F \in \mathcal{P}_{\text{fin}}(\Delta)\}^\dagger\\
&\quad \cup \{\langle \varepsilon_s,(F,\{h!\},\mathtt{p})\rangle \mid \mathsf{fv}[\![e]\!] \subseteq \mathsf{dom}(s) \;\&\; F \supseteq \{h!\}\}^\dagger\\
\mathcal{T}_s^\dagger[\![g \to c]\!] &= (\mathcal{T}_s^\dagger[\![g]\!];\mathcal{T}_s^\dagger[\![c]\!])^\dagger\\
\mathcal{T}_s^\dagger[\![gc_1 \,\square\, gc_2]\!] &= (\mathcal{T}_s^\dagger[\![gc_1]\!] \,\square\, \mathcal{T}_s^\dagger[\![gc_2]\!])^\dagger\\
\mathcal{T}_s^\dagger[\![c_1\|c_2]\!] &= (\mathcal{T}_s^\dagger[\![c_1]\!]\|\mathcal{T}_s^\dagger[\![c_2]\!])^\dagger\\
\mathcal{T}_s^\dagger[\![c\backslash h]\!] &= (\mathcal{T}_s^\dagger[\![c]\!]\backslash h)^\dagger.
\end{aligned}
$$

$\diamond$

For all commands c, $\mathcal{T}_s^\dagger[\![c]\!]$ is precisely the closure of $\mathcal{T}_s[\![c]\!]$: that is, for all commands c, $\mathcal{T}_s^\dagger[\![c]\!] = (\mathcal{T}_s[\![c]\!])^\dagger$. Proving this fact, however, requires a detour provided by the following section. In particular, the obvious inductive proof requires that we prove that

$$(\mathcal{T}_s^\dagger[\![c_1]\!] \| \mathcal{T}_s^\dagger[\![c_2]\!])^\dagger = (\mathcal{T}_s[\![c_1]\!]^\dagger \| \mathcal{T}_s[\![c_2]\!]^\dagger)^\dagger = (\mathcal{T}_s[\![c_1]\!] \| \mathcal{T}_s[\![c_2]\!])^\dagger.$$

Although this equality holds, we can prove it only by referring to particular properties of the trace sets $\mathcal{T}_s[\![c_1]\!]$ and $\mathcal{T}_s[\![c_2]\!]$: the property $(T_1^\dagger \| T_2^\dagger)^\dagger = (T_1 \| T_2)^\dagger$ does not hold for arbitrary trace sets. For example, consider the following two trace sets:

$$\begin{array}{rcl} T_1 & = & \{\langle \alpha_1, (\emptyset, \{d, \bar{d}, e\}, \mathtt{i})\rangle,\ \langle \alpha_1, (\{d\}, \{d, e\}, \mathtt{i})\rangle\}, \\ T_2 & = & \{\langle \alpha_2, (\{d\}, \emptyset, \{d, \bar{d}\}, \mathtt{i})\rangle\}. \end{array}$$

The set $(T_1 \| T_2)^\dagger$ is empty, because the single trace in T_2 is not mergeable with either trace in T_1. However, it is mergeable with the trace $\langle \alpha_1, (\emptyset, \{d, e\}, \mathtt{i})\rangle$, which is in $T_1^\dagger$ by contention, and hence $(T_1^\dagger \parallel T_2^\dagger)^\dagger$ is not empty.

4.3 Computational Feasibility

For any command c, the trace set $\mathcal{T}_s[\![c]\!]$ necessarily satisfies certain properties that an arbitrary trace set may not. These properties stem from the nature of programs, computations, and the definition of parameterized fairness. Several of these properties are essential for proving full abstraction and hence are worth making explicit.

Because every successfully terminating computation is fair mod $\emptyset$, the trace $\langle \alpha, (\emptyset, E, \mathtt{f})\rangle$ is in $\mathcal{T}_s[\![c]\!]$ whenever any trace $\langle \alpha, (F, E, \mathtt{f})\rangle$ is. Similarly, because a fair mod F computation is also fair mod F' for all $\langle \alpha, (F', E, R)\rangle$ is in $\mathcal{T}_s[\![c]\!]$ whenever $\langle \alpha, (F, E, R)\rangle$ is in $\mathcal{T}_s[\![c]\!]$ and $F' \supseteq F$.

A partial computation with final configuration $\langle c, s\rangle$ is fair mod F if and only if $F \supseteq \mathsf{inits}(c, s)$. In particular, if $E = \mathsf{inits}(c, s)$, then the computation is blocked mod E but not blocked E' for any $E' \subset E$. As a result, the trace $\langle \alpha, (E, E, \mathtt{p})\rangle$ is in $\mathcal{T}_s[\![c]\!]$ whenever any trace $\langle \alpha, (F, E, \mathtt{p})\rangle$ is in $\mathcal{T}_s[\![c]\!]$. Similarly, the trace $\langle \alpha, (F, E, \mathtt{p})\rangle$ is in $\mathcal{T}_s[\![c]\!]$ whenever $F \supseteq E$ and $\langle \alpha, (E, E, \mathtt{p})\rangle$ is in $\mathcal{T}_s[\![c]\!]$.

The remaining properties concern the relationships between a computation's infinitely enabled directions, infinitely used directions, and blocked processes. The directions that are used in visible communications infinitely often along a computation are clearly enabled infinitely often. As a result, for any trace $\langle \alpha, (F, E, \mathtt{i})\rangle$ in $\mathcal{T}_s[\![c]\!]$, it must be that $\mathsf{vis}(\alpha) \subseteq E$. Similarly, no process can become blocked while capable of using a direction that is used infinitely often by some other process: if a fair mod $(F \cup X)$ computation uses the directions in X infinitely often, then the computation must also be fair mod F. Therefore, whenever the trace $\langle\alpha,(F \cup X, E, \mathtt{i})\rangle$ is in $\mathcal{T}_s[\![c]\!]$ and $X \subseteq \mathsf{vis}(\alpha)$, the trace $\langle\alpha,(F,E,\mathtt{i})\rangle$ also must be in $\mathcal{T}_s[\![c]\!]$. Moreover, the set of directions enabled infinitely often along a computation provide an upper bound on the directions on which processes are permanently blocked: if a fair mod $(F \cup X)$ computation has infinitely enabled directions E and $X \cap E = \emptyset$, then no blocked mod $(F \cup X)$ process can actually used the directions in X, and thus the computation is also fair mod F. As a result, it is safe to remove the set X from the trace $\langle\alpha,(F \cup X, E, \mathtt{i})\rangle$ in $\mathcal{T}_s[\![c]\!]$ whenever $X \cap E = \emptyset$.

The final property is subtle but extremely important. Intuitively, a trace with form

$$\langle\alpha,(F \cup \{d\}, E \cup \{d,\bar{d}\}, \mathtt{i})\rangle$$

represents a computation ρ that enables the directions d and $\bar{d}$ (among others) infinitely often and is fair mod $F \cup \{d\}$. Thus any subcomponent of c that is blocked mod $(F \cup \{d\})$ along ρ must be blocked in a configuration in which its only transitions involve the directions $F \cup \{d\}$. If we assume that $\bar{d} \notin F$, then any process capable of using direction d has infinitely many opportunities to synchronize, because the matching direction $\bar{d}$ is also enabled infinitely often. Therefore, any subcomponent blocked mod $F \cup \{d\}$ must also be blocked mod F, and hence the computation is also fair mod F. As a result, the trace

$$\langle\alpha,(F, E \cup \{d,\bar{d}\}, \mathtt{i})\rangle$$

must be in the set $\mathcal{T}_s[\![c]\!]$ whenever the original trace is. However, once we start considering the closed trace set $\mathcal{T}_s^{\dagger}[\![c]\!]$, we need to account for the possibility that the trace

$$\langle\alpha,(F \cup \{d\}, E \cup \{d,\bar{d}\}, \mathtt{i})\rangle$$

is in $(\mathcal{T}_s[\![c]\!])^{\dagger}$ by superset closure from the trace $\langle\alpha,(F \cup \{d\}, E \cup \{d\}, \mathtt{i})\rangle$ in $\mathcal{T}_s[\![c]\!]$.

There are, of course, other general properties that are true for all sets $\mathcal{T}_s[\![c]\!]$ that are not incorporated into the following definition of *computational feasibility*. The properties that are included suffice for proving that $\mathcal{T}_s^{\dagger}[\![c]\!] = \mathcal{T}_s[\![c]\!]^{\dagger}$ for all commands c and that $\mathcal{T}_s^{\dagger}$ is fully abstract.

Definition 4.3.1 A fair trace set T is **computationally feasible** if it satisfies the following properties:

- If the trace $\langle\alpha,(F,E,\mathtt{f})\rangle$ is in T, then the trace $\langle\alpha,(\emptyset,E,\mathtt{f})\rangle$ is in T.
- If the trace $\langle\alpha,(F,E,R)\rangle$ is in T, $R \in \{\mathtt{f},\mathtt{i}\}$, and $F \subseteq F'$, then $\langle\alpha,(F',E,R)\rangle$ is in T.
- The trace $\langle\alpha,(F,E,\mathtt{p})\rangle$ is in T if and only if $F \supseteq E$ and the trace $\langle\alpha,(E,E,\mathtt{p})\rangle$ is in T.
- If the trace $\langle\alpha,(F,E,\mathtt{i})\rangle$ is in T, then $\mathsf{vis}(\alpha) \subseteq E$.

- If the trace $\langle \alpha, (F \cup X, E, \mathtt{i}) \rangle$ is in T and $X \subseteq \mathsf{vis}(\alpha)$, then the trace $\langle \alpha, (F, E, \mathtt{i}) \rangle$ is in T.
- If $\langle \alpha, (F \cup X, E, \mathtt{i}) \rangle$ is in T and $X \cap E = \emptyset$, then $\langle \alpha, (F, E, \mathtt{i}) \rangle$ is in T.
- If $\langle \alpha, (F \cup \{d\}, E \cup \{d, \bar{d}\}, \mathtt{i}) \rangle$ is in T and $\bar{d} \notin F$, then at least one of the traces $\langle \alpha, (F, E \cup \{d, \bar{d}\}, \mathtt{i}) \rangle$ and $\langle \alpha, (F \cup \{d\}, E \cup \{d\}, \mathtt{i}) \rangle$ is in T. $\diamond$

The following lemma shows that the definition of computational feasibility indeed captures general properties of commands' trace sets.

Lemma 4.3.2 *For all commands c, $\mathcal{T}_s[\![c]\!]$ is computationally feasible.*

Proof: By a straightforward but tedious induction on the structure of c. To give a flavor of the proof, we prove that $\mathcal{T}_s[\![c_1 \| c_2]\!]$ satisfies the sixth condition: if $\langle \alpha, (F \cup X, E, \mathtt{i}) \rangle$ is in $\mathcal{T}_s[\![c_1 \| c_2]\!]$ and $X \cap E = \emptyset$, then $\langle \alpha, (F, E, \mathtt{i}) \rangle$ is in $\mathcal{T}_s[\![c_1 \| c_2]\!]$.

Suppose that $\varphi = \langle \alpha, (F \cup X, E, \mathtt{i}) \rangle$ is in $\mathcal{T}_s[\![c_1 \| c_2]\!]$ and that $X \cap E = \emptyset$. By definition of parallel composition, there exist traces

$$\varphi_1 = \langle \alpha_1, (F_1, E_1, R_1) \rangle \in \mathcal{T}_s[\![c_1]\!], \qquad \varphi_2 = \langle \alpha_2, (F_2, E_2, R_2) \rangle \in \mathcal{T}_s[\![c_2]\!]$$

such that $(\varphi_1, \varphi_2, \varphi) \in$ *fairmerge*. At least one of φ_1, φ_2 is infinite; without loss of generality, we assume that φ_1 is infinite. As a result, we know that $(F \cup X) \supseteq F_1 \cup F_2$ and that $E = E_1$ (if $R_2 = \mathtt{f}$) or $E = E_1 \cup E_2$ (if $R_2 \neq \mathtt{f}$).

Because $X \cap E = \emptyset$ and $E \supseteq E_1$, we know that $E_1 \cap X = \emptyset$ (and $E_2 \cap X = \emptyset$ if $R_2 \neq \mathtt{f}$). By the inductive hypothesis, $\mathcal{T}_s[\![c_1]\!]$ and $\mathcal{T}_s[\![c_2]\!]$ are computationally feasible, and hence

$$\varphi_1' = \langle \alpha_1, (F_1 - X, E_1, \mathtt{i}) \rangle \in \mathcal{T}_s[\![c_1]\!], \qquad \varphi_2' = \langle \alpha_2, (F_2 - X, E_2, R_2) \rangle \in \mathcal{T}_s[\![c_2]\!].$$

(The existence of φ_2' follows because either $R_2 = \mathtt{f}$ (in which case any choice of F is permissible for φ_2') or $E_2 \cap X = \emptyset$.) Letting $\varphi' = \langle \alpha, (F, E, \mathtt{i}) \rangle$, it follows that $(\varphi_1', \varphi_2', \varphi') \in$ *fairmerge*, and hence φ' is in $\mathcal{T}_s[\![c_1]\!] \| \mathcal{T}_s[\![c_2]\!] = \mathcal{T}_s[\![c_1 \| c_2]\!]$ as required. ∎

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 4.3.3 *If the trace set T is computationally feasible, then $T^\dagger$ is also computationally feasible.*

Proof: By a straightforward but tedious case analysis showing that each possible trace introduced by closure respects computational feasibility. To give a flavor of the proof, we show that displacement preserves the final property of the definition of computational feasibility.

Let T be a computationally feasible trace set, and let $\varphi = \langle \alpha, (F \cup \{d\}, E \cup \{d, \bar{d}\}, \mathtt{i})\rangle$ be a trace of $T^\dagger$ that arises by displacement from one of T's traces

$$\langle \alpha, (F \cup \{d\}, E \cup \{d, \bar{d}\} \cup Y, \mathtt{i})\rangle,$$

where $Y \cap \mathsf{vis}(\alpha) = \emptyset$ and $\overline{Y} \subseteq \mathsf{vis}(\alpha)$. We need to show that $T^\dagger$ also contains either $\varphi' = \langle \alpha, (F, E \cup \{d, \bar{d}\}, \mathtt{i})\rangle$ or $\varphi'' = \langle \alpha, (F \cup \{d\}, E \cup \{d\}, \mathtt{i})\rangle$.

Because T is computationally feasible, we know that $\mathsf{vis}(\alpha) \subseteq E \cup \{d, \bar{d}\}$ and that T also contains at least one of the following two traces:

$$\langle \alpha, (F, E \cup \{d, \bar{d}\} \cup Y, \mathtt{i})\rangle, \qquad \langle \alpha, (F \cup \{d\}, E \cup \{d\} \cup Y, \mathtt{i})\rangle.$$

It follows by displacement that $T^\dagger$ contains either φ' or φ'' as required. ∎

Lemma 4.3.4 *For all computationally feasible trace sets T, T_1 and T_2, the following properties hold:*

$$(T_1;T_2)^\dagger = (T_1^\dagger;T_2^\dagger)^\dagger \qquad (T^*)^\dagger = (T^\dagger)^{*\dagger} \qquad (T\backslash h)^\dagger = (T^\dagger\backslash h)^\dagger$$
$$(T_1 \cup T_2)^\dagger = (T_1^\dagger \cup T_2^\dagger)^\dagger \qquad (T^\omega)^\dagger = (T^\dagger)^{\omega\dagger} \qquad (T_1 \| T_2)^\dagger = (T_1^\dagger \| T_2^\dagger)^\dagger$$
$$(T_1 \,\square\, T_2)^\dagger = (T_1^\dagger \,\square\, T_2^\dagger)^\dagger$$

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

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

Proposition 4.3.5 *For all commands c, $\mathcal{T}_s^\dagger[\![c]\!] = \mathcal{T}_s[\![c]\!]^\dagger$.*

Proof: By a straightforward induction on the structure of c, using the properties of Lemma 4.3.4. For example, the case for parallel composition proceeds as follows, relying on the inductive hypothesis that $\mathcal{T}_s^\dagger[\![c_i]\!] = \mathcal{T}_s[\![c_i]\!]^\dagger$ for each i:

$$\begin{aligned} \mathcal{T}_s^\dagger[\![c_1 \| c_2]\!] &= (\mathcal{T}_s^\dagger[\![c_1]\!] \,\|\, \mathcal{T}_s^\dagger[\![c_2]\!])^\dagger \;=\; (\mathcal{T}_s[\![c_1]\!]^\dagger \,\|\, \mathcal{T}_s[\![c_2]\!]^\dagger)^\dagger \\ &= (\mathcal{T}_s[\![c_1]\!] \,\|\, \mathcal{T}_s[\![c_2]\!])^\dagger \;=\; \mathcal{T}_s[\![c_1 \| c_2]\!]^\dagger. \end{aligned}$$

∎

4.4 Full Abstraction for the Behavior $\mathcal{M}$

In this section, we prove that the semantics $\mathcal{T}_s^\dagger$ is fully abstract with respect to the state trace behavior $\mathcal{M}$: $\mathcal{T}_s$ gives identical meanings to two program terms if and only if they exhibit the same state trace behaviors in all program contexts. We begin with some definitions and necessary lemmas.

Definition 4.4.1 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, $(F' \subseteq F \;\&\; E' \subseteq E) \implies \varphi = \varphi'$. ◇

Thus a finite or infinite trace $\varphi \in T$ is minimal if there is no trace $\varphi' \in T$ that would yield φ through closure under subset; a partial trace $\varphi = \langle \alpha, (F, E, \mathtt{p}) \rangle \in T$ is minimal if $F = E$ and every other partial trace $\varphi' = \langle \alpha, (F', E', \mathtt{p}) \rangle$ in T has a direction $d \in E' - E$. A closed trace set is uniquely characterized by its set of minimal traces: each of its finite or infinite traces can be obtained from minimal traces by superset closure, and every partial trace can be obtained by a combination of union and convex closure.

For a trace set T and a simple trace α, it is often necessary to talk about the (minimal) traces of T with the simple component α. In the following definition, we concern ourselves only with infinite traces α; clearly, a similar definition can be given for finite traces α, as well as a distinction between successful and partial α-traces.

Definition 4.4.2 Let T be a trace set. The set $\mathsf{min}(T, \alpha)$ is the set of minimal (nonpartial) α-traces in T; that is, $\mathsf{min}(T, \alpha) = \{\varphi = \langle \alpha, (F, E, R) \rangle \in T \mid \varphi \text{ is minimal in } T \;\&\; R \in \{\mathtt{f}, \mathtt{i}\}\}$. ◇

The minimal traces of a computationally feasible trace set all satisfy certain conditions relating the fairness set F to the enabling set E. For a minimal finite trace $\varphi = \langle \alpha, (F, E, \mathtt{f}) \rangle$, the set F is necessarily empty; for any minimal partial trace $\varphi = \langle \alpha, (F, E, \mathtt{p}) \rangle$, it must be the case that $F = E$. If the infinite trace $\varphi = \langle \alpha, (F, E, \mathtt{i}) \rangle$ 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 $\bar{d}$ 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 4.4.3 An infinite trace $\varphi = \langle \alpha, (F, E, \mathtt{i}) \rangle$ is **potentially minimal** if $F \subseteq E$ and, for all directions $d \in F$, $\bar{d} \in F \iff \bar{d} \in E$. ◇

Every potentially minimal trace φ is a minimal trace of some computationally feasible trace set T; in particular, φ is a minimal trace of the computationally feasible trace set $\{\varphi\}^\dagger$. Moreover, every minimal trace of a computationally feasible trace set is potentially minimal.

Suppose a closed, computationally feasible trace set T does not contains the potentially minimal trace $\varphi = \langle\alpha,(F,E,\mathtt{i})\rangle$. If T does contain other α-traces (that is, if $\mathsf{min}(T,\alpha) \neq \emptyset$), then each minimal trace $\langle\alpha,(F_i,E_i,\mathtt{i})\rangle$ 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 $d \in E_i - E$). The idea is that, 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 φ_i, we can construct a context that distinguishes the trace φ 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 4.4.4 Let T be a trace set not containing the trace $\varphi = \langle\alpha,(F,E,\mathtt{i})\rangle$. A **conflict-free resolution of** T **for** φ is a total function

$$\mathcal{R} : \mathsf{min}(T,\alpha) \to (\Delta \times \{\mathtt{F},\mathtt{E}\})$$

satisfying the following two conditions:

- For all traces $\varphi_i \in \mathsf{min}(T,\alpha)$,

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

- For all traces $\varphi_i,\varphi_j \in \mathsf{min}(T,\alpha)$, $\quad \varphi_i = (d_i,\mathtt{F})\ \&\ \varphi_j = (d_j,\mathtt{E}) \implies \neg\mathsf{match}(d_i,d_j)$. $\diamond$

As a consequence of the following lemma, a conflict-free resolution of $\mathcal{T}_s^\dagger[\![c]\!]$ for φ can always be constructed, for any command c and any potentially minimal trace $\varphi \notin \mathcal{T}_s^\dagger[\![c]\!]$. That is, the necessary "careful selection" is always possible. This fact will be necessary for proving full abstraction.

Lemma 4.4.5 *Let T be a closed, computationally feasible trace set not containing the potentially minimal trace $\varphi = \langle\alpha,(F,E,\mathtt{i})\rangle$. If the set $\mathsf{min}(T,\alpha)$ is finite, then there is a conflict-free resolution of T for φ.*

Proof: Assume that $\mathsf{min}(T,\alpha)$ is finite, and let $\mathcal{R}$ be a total function $\mathcal{R} : \mathsf{min}(T,\alpha) \to (\Delta \times \{\mathtt{F},\mathtt{E}\})$ such that, for all traces $\varphi_i \in \mathsf{min}(T,\alpha)$,

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

We say that $\mathcal{R}$ has conflicts on channel h if there exist traces $\varphi_i,\varphi_j \in \mathsf{min}(T,\alpha)$ and a direction d such that $\mathcal{R}(\varphi_i) = (d,\mathtt{F})$, $\mathcal{R}(\varphi_j) = (\bar{d},\mathtt{E})$, and $\mathsf{chan}(d) = h$. We introduce a 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 $\varphi_x = \langle \alpha, (F_x, E_x, \mathtt{i}) \rangle$ and $\varphi_y = \langle \alpha, (F_y, E_y, \mathtt{i}) \rangle$ in $\mathsf{min}(T, \alpha)$ such that $\mathcal{R}(\varphi_x) = (d, \mathtt{F})$, $\mathcal{R}(\varphi_y) = (\bar{d}, \mathtt{E})$, and $\mathsf{chan}(d) = h$. Exactly one of the following cases must hold:

Case: $d \notin E$

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

Case: $d \in E$ and ($\bar{d} \in F_y$ or $\bar{d} \in F_x$)

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

Case: $d \in E$ and $\bar{d} \notin F_y$ and $\bar{d} \notin F_x$

Because φ_y is minimal, we know that $d \notin F_y$. Because T is closed under superset, T contains the traces

$$\langle \alpha, (F_x \cup F_y, (E_x \cup E_y) - \{\bar{d}\}, \mathtt{i}) \rangle \quad \text{and} \quad \langle \alpha, ((F_x \cup F_y) - \{d\}, E_x \cup E_y, \mathtt{i}) \rangle,$$

via φ_x and φ_y, respectively. It follows that the trace

$$\langle \alpha, ((F_x \cup F_y) - \{d\}, (E_x \cup E_y) - \{\bar{d}\}, \mathtt{i}) \rangle$$

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

If $\mathcal{R}(\varphi_r) = (e, \mathtt{E})$ (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 $\mathcal{R}(\varphi_r) = (e, \mathtt{F})$, then $e \in F_r - F$, and hence $e \in (F_x - F) \cup (F_y - F)$. 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 $\mathsf{min}(T, \alpha)$ is finite, repeating the preceding analysis eventually removes all conflicts on channel h, without introducing any conflicts on any channel $k < h$. Moreover, because there can be only finitely many channels mentioned in the set $\mathsf{min}(T, \alpha)$, the analysis must be applied for only a finite number of channels, eventually resulting in a conflict-free resolution for φ. ∎

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

Proposition 4.4.6 *The closed trace semantics $\mathcal{T}_s^\dagger$ is inequationally fully abstract with respect to $\mathcal{M}$: for all commands c and c',*

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

Proof: The forward implication follows from the compositionality of $\mathcal{T}_s^\dagger$, the monotonicity of operations on trace sets, and the fact that, when $\mathcal{T}_s^\dagger[\![c]\!] \subseteq \mathcal{T}_s^\dagger[\![c']\!]$,

$$\begin{aligned}
\mathcal{M}[\![P[c]]\!] &= \{\mathsf{states}(\alpha) \mid \exists E.\ \langle \alpha, (\emptyset, E, R)\rangle \in \mathcal{T}_s^\dagger[\![P[c]]\!]\ \&\ R \in \{\mathtt{f}, \mathtt{i}\}\ \&\ \mathsf{chans}(\alpha) = \{\varepsilon\}\} \\
&\cup\ \{\mathsf{states}(\alpha)\delta \mid \langle \alpha, (\emptyset, \emptyset, \mathtt{p})\rangle \in \mathcal{T}_s^\dagger[\![P[c]]\!]\ \&\ \mathsf{chans}(\alpha) = \{\varepsilon\}\} \\
&\subseteq \{\mathsf{states}(\alpha) \mid \exists E.\ \langle \alpha, (\emptyset, E, R)\rangle \in \mathcal{T}_s^\dagger[\![P[c']]\!]\ \&\ R \in \{\mathtt{f}, \mathtt{i}\}\ \&\ \mathsf{chans}(\alpha) = \{\varepsilon\}\} \\
&\cup\ \{\mathsf{states}(\alpha)\delta \mid \langle \alpha, (\emptyset, \emptyset, \mathtt{p})\rangle \in \mathcal{T}_s^\dagger[\![P[c']]\!]\ \&\ \mathsf{chans}(\alpha) = \{\varepsilon\}\} \\
&= \mathcal{M}[\![P[c']]\!].
\end{aligned}$$

[We write $\mathsf{states}(\alpha)$ to indicate the sequence of states encountered along α: for example, if $\alpha = (s_0, \varepsilon, s_1)(s_1, \varepsilon, s_2)\ldots(s_k, \varepsilon, s_{k+1})$, then $\mathsf{states}(\alpha) = s_o s_1 s_2 \ldots s_k s_{k+1}$.]

For the reverse implication, consider $\varphi = \langle \alpha, (F, E, R)\rangle$ in $\mathcal{T}_s^\dagger[\![c]\!] - \mathcal{T}_s^\dagger[\![c']\!]$.

Case: $\varphi = \langle \alpha, (F, E, \mathtt{f})\rangle$

Because $\mathcal{T}_s^\dagger[\![c]\!]$ and $\mathcal{T}_s^\dagger[\![c']\!]$ are computationally feasible, we can assume without loss of generality that $F = \emptyset$. Let $\langle \alpha, (\emptyset, E_1, \mathtt{f})\rangle, \ldots, \langle \alpha, (\emptyset, E_m, \mathtt{f})\rangle$ be the (necessarily finite number of) minimal α-traces in $\mathcal{T}_s^\dagger[\![c']\!]$. Closure under superset ensures that $E_i \not\subseteq E$ for each $i \leq m$; thus for each i we can choose a direction $d_i \in E_i - E$.

Let $x_1, \ldots, x_n$ be the free identifiers of c, and let $h_1, \ldots, h_k$ be the channel names appearing in c. We let $\mathsf{x}, \mathsf{y}, \mathsf{flag}, \mathsf{step}, v_1, \ldots, v_n$ be fresh identifiers, and we define guards g_i (for each $i \leq m$) so that each guard g_i "matches" the direction d_i: $g_i = h!0$ when $d_i = h?$, and $g_i = h?\mathsf{x}$ when $d_i = h!$. We also define a command $\mathsf{Match}_{\mathsf{y},i}(\alpha)$ inductively as follows:

$$\begin{aligned}
\mathsf{Match}_{\mathsf{y},i}((s, \varepsilon, s')) &\equiv \mathsf{step}{:=}i \\
\mathsf{Match}_{\mathsf{y},i}((s, h!n, s)) &\equiv h?\mathsf{y} \rightarrow \mathsf{step}{:=}i \\
\mathsf{Match}_{\mathsf{y},i}((s, h?n, s)) &\equiv h!n \rightarrow \mathsf{step}{:=}i \\
\mathsf{Match}_{\mathsf{y},i}(\sigma\beta) &\equiv \mathsf{Match}_{\mathsf{y},i}(\sigma); \mathsf{Match}_{\mathsf{y},i+1}(\beta).
\end{aligned}$$

Intuitively, the command $\mathsf{Match}_{\mathsf{y},1}(\alpha)$ can synchronize with the trace α, keeping track of the number of steps performed along the way.

We now let $P[-]$ be the following context:

$$\left[\begin{array}{l} \mathsf{while\ true\ do} \\ \quad (v_1{:=}x_1; v_2{:=}x_2; \cdots; v_n{:=}x_n; \\ \quad ([-] \parallel \mathsf{Match}_{\mathsf{y},1}(\alpha)); \\ \quad x_1{:=}v_1; x_2{:=}v_2; \cdots; x_n{:=}v_n) \end{array} \ \left\| \ \sum_{i=1}^{m} (g_i \rightarrow \mathsf{flag}{:=}1) \right. \right] \backslash h_1 \backslash \cdots \backslash h_k.$$

Because φ never enables synchronization with any of the guards g_i, $\mathcal{M}[\![P[c]]\!]$ has a behavior that corresponds to the infinite iteration of α in which the variable flag is never set to 1. In contrast, every computation of $P[c']$ that iterates α infinitely many times must enable synchronization infinitely often with at least one guard g_i; consequently, any behavior in $\mathcal{M}[\![P[c']]\!]$ corresponding to the infinite iteration of α must eventually set flag to 1.

Case: $\varphi = \langle \alpha, (F, E, \mathtt{p}) \rangle$

Without loss of generality, we can assume that $F = E$. We let $\mathsf{x}, \mathsf{y}, \mathsf{flag}, \mathsf{step}$ be fresh identifiers, and we let $h_1, \ldots, h_k$ be the channel names appearing in c. We let a be a fresh channel name not appearing in c or c'.

Let $\langle \alpha, (E_1, E_1, \mathtt{p}) \rangle, \ldots, \langle \alpha, (E_m, E_m, \mathtt{p}) \rangle$ be the finite number of minimal partial α-traces in $\mathcal{T}_s^\dagger[\![c']\!]$, and let $Z = \bigcup_{i=1}^m E_i$. Closure under union ensures that $\langle \alpha, (Z, Z, \mathtt{p}) \rangle$ is in $\mathcal{T}_s^\dagger[\![c']\!]$; by convex closure, it must be that (for each $i \le m$) $\neg(E_i \subseteq E \subseteq Z)$. 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 $d \neq \varepsilon$, and let $P[-]$ be the following context:

$$([-] \parallel \mathsf{Match}_{\mathsf{x},1}(\alpha); \mathsf{flag}{:=}1; g \to \mathsf{flag}{:=}2) \backslash h_1 \backslash \cdots \backslash h_k.$$

(When $d = \varepsilon$, replace the code fragment "$g \to \mathsf{flag}{:=}2$" by "$\mathsf{flag}{:=}2$".) $\mathcal{M}[\![P[c]]\!]$ has a behavior that begins with a correspondence to α, followed by flag being set to 1 and then, exactly two steps later, being set to 2. In contrast, $\mathcal{M}[\![P[c']]\!]$ has no such behavior.

If 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 $d_i \neq \varepsilon$, and let g_i be the guard $\mathsf{a}!0$ when $d_i = \varepsilon$. Let $P[-]$ be the following context:

$$([-] \parallel \mathsf{Match}_{\mathsf{x},1}(\alpha); \mathsf{y}{:=}0; \sum_{i=1}^{m} g_i \to \mathsf{flag}{:=}1) \backslash h_1 \backslash \cdots \backslash h_k \backslash \mathsf{a}.$$

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

Case: $\varphi = \langle \alpha, (F, E, \mathtt{i}) \rangle$

Without loss of generality, assume that φ is minimal in $\mathcal{T}_s^\dagger[\![c]\!]$. We let x, y, $\mathsf{f1}$, $\mathsf{f2}$, synch, value, comm, and count be fresh identifiers, $h_1, \ldots, h_k$ be the channel names appearing in c, and a be a fresh channel name not appearing in c or c'.

Let $\varphi_1 = \langle \alpha, (F_1, E_1, \mathtt{i}) \rangle, \ldots, \varphi_m = \langle \alpha, (F_m, E_m, \mathtt{i}) \rangle$ be the minimal α-traces in $\mathcal{T}_s^\dagger[\![c']\!]$. By Lemma 4.4.5, there is a conflict-free resolution $\mathcal{R}$ of $\mathcal{T}_s^\dagger[\![c']\!]$ for φ. Define sets

```
count:=count + 1;synch:=1;
while true do
      Pick_Int(comm);
      Pick_Int(value);
      case (comm mod (2k + 1)) of
            1: synch:=0; ((h1!value → synch:=1) □ Σ_{g∈G}(g → f1:=1))
            2: synch:=0; ((h1?value → synch:=1) □ Σ_{g∈G}(g → f1:=1))
            ⋮
            2k − 1: synch:=0; ((hk!value → synch:=1) □ Σ_{g∈G}(g → f1:=1))
            2k: synch:=0; ((hk?value → synch:=1) □ Σ_{g∈G}(g → f1:=1))
            0: synch:=0; (((a?value → synch:=1) □ Σ_{g∈G}(g → f1:=1)) || a!0)\a
      endcase;
      count:=count + 1.
```

Figure 4.1: The program $\mathsf{Guess}(H, G_{\mathsf{x}}, \mathsf{f1})$.

$X = \{d_i \mid 1 \le i \le m \;\&\; \mathcal{R}(\varphi_i) = (d_i, \mathtt{F})\}$ and $Y = \{d_i \mid 1 \le i \le m \;\&\; \mathcal{R}(\varphi_i) = (d_i, \mathtt{E})\}$; because $\mathcal{R}$ is conflict-free, it follows that $\neg\mathsf{match}(X,Y)$.

Define sets $G_{\mathsf{x}} = \{h!0 \mid h? \in X\} \cup \{h?\mathsf{x} \mid h! \in X\}$ and $G_{\mathsf{y}} = \{h!0 \mid h? \in Y\} \cup \{h?\mathsf{y} \mid h! \in Y\}$ 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 $\mathsf{Guess}(H, G_{\mathsf{x}}, \mathsf{f1})$ abbreviate the command in Figure 4.1, with the case construct used as syntactic sugar for the corresponding series of nested if-statements. Intuitively, the program $\mathsf{Guess}(H, G_{\mathsf{x}}, \mathsf{f1})$ can synchronize with any computation of any program that uses only the channels $h_1, \ldots, h_k$ for visible communication. For each synchronization, $\mathsf{Guess}(H, G_{\mathsf{x}}, \mathsf{f1})$ "guesses" the particular communication necessary for synchronization[2]. Moreover, in any infinite computation of $\mathsf{Guess}(H, G_{\mathsf{x}}, \mathsf{f1})$, the directions associated with the guards in G_{x} are enabled infinitely often. Consequently, if the program in parallel with $\mathsf{Guess}(H, G_{\mathsf{x}}, \mathsf{f1})$ treats any of the directions in X unfairly, the flag f1 will necessarily be set to 1 eventually.

Let $P[-]$ be the following context:

$$([-] \parallel \mathsf{Guess}(H, G_{\mathsf{x}}, \mathsf{f1}) \parallel \sum_{g \in G_{\mathsf{y}}} g \to \mathsf{f2}{:=}1) \backslash h_1 \backslash \cdots \backslash h_k.$$

$\mathcal{M}[\![P[c]]\!]$ has a behavior corresponding to α in which neither f1 nor f2 is ever set to 1. In contrast, every behavior of $\mathcal{M}[\![P[c']]\!]$ corresponding to α must eventually set at least one of the flags f1 and f2 to 1. ■

[2]The case where comm mod $(2k+1) = 0$ is necessary when α involves only finitely many visible communications (e.g., $(s, \varepsilon, s)^{\omega}$).

$$
\begin{array}{rcl}
c_1\|c_2 & \equiv & c_2\|c_1 \\
(c_1\|c_2) \parallel c_3 & \equiv & c_1 \parallel (c_2\|c_3) \\
(c_1\|c_2)\backslash h & \equiv & c_1 \parallel (c_2\backslash h), \quad \text{provided } h \notin \mathsf{fc}[\![c_1]\!] \\
c\backslash h & \equiv & c, \quad \text{provided } h \notin \mathsf{fc}[\![c]\!] \\
(\mathsf{a!0} \to \mathsf{b!0}) \,\square\, (\mathsf{b!0} \to \mathsf{a!0}) & \equiv & \mathsf{a!0} \parallel \mathsf{b!0} \\
(\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2); c & \equiv & \mathsf{if}\ b\ \mathsf{then}\ c_1; c\ \mathsf{else}\ c_2; c \\
(\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \parallel c & \not\equiv & \mathsf{if}\ b\ \mathsf{then}\ (c_1\|c)\ \mathsf{else}\ (c_2\|c) \\
(\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \parallel (\mathsf{skip}; c) & \equiv & \mathsf{if}\ b\ \mathsf{then}\ (c_1 \parallel \mathsf{skip}; c)\ \mathsf{else}\ (c_2 \parallel \mathsf{skip}; c)
\end{array}
$$

Figure 4.2: Some program equivalences validated by $\mathcal{T}_s^\dagger$.

This full abstraction result show that $\mathcal{T}_s^\dagger$ provides precisely the correct level of abstraction to support compositional reasoning about the program behavior $\mathcal{M}$. As a consequence, the semantics $\mathcal{T}_s^\dagger$ validates several natural program (in)equivalences (with respect to $\mathcal{M}$) that hold under strong fairness. Figure 4.2 lists several of these properties, where we write $c \equiv c'$ to indicate that $\mathcal{T}_s^\dagger[\![c]\!] = \mathcal{T}_s^\dagger[\![c']\!]$. Many of these properties appear obvious, but proving them using purely operational methods is very difficult. Moreover, "obvious" properties may not hold under certain notions of fairness; for example, the equivalence

$$(\mathsf{a!0} \to \mathsf{b!0}) \,\square\, (\mathsf{b!0} \to \mathsf{a!0}) \;\equiv\; \mathsf{a!0} \parallel \mathsf{b!0}$$

does not hold under weak fairness, as we shall see in Chapter 6.

4.5 Other Notions of Program Behavior

The state trace behavior $\mathcal{M}$ introduced in Definition 4.1.1 incorporates the assumptions that external communication is prohibited, that every state change can be detected, and that deadlock can be distinguished from successful termination and infinite chattering. In this section, we consider several other notions of behavior that relax one or more of these assumptions, in each case showing how the semantics can be adapted to yield full abstraction. The changes to the semantics primarily affect 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, remain the same.

The ease with which the semantics can be modified to yield full abstraction for these other notions of behavior reflects the robustness of the framework. In particular, the notion of computational feasibility and the related definitions and lemmas of Section 4.4 are independent of the structure of simple traces and can be revised for other types of simple traces effortlessly. As a result, the proofs of full abstraction for the behaviors in this section all follow the proof of Proposition 4.4.6 very closely.

4.5.1 Simple trace behavior

The state trace behavior $\mathcal{M}$ adopts a view of programs as closed systems that cannot communicate with the external world. According to this view, all communication is internal and synchronous; an observer cannot possibly detect visible (i.e., external) communications, because such communications are not possible in a closed system. However, it is reasonable to relax this assumption and to assume instead that an open system's interactions with its environment are observable. Moreover, to reason about the possible interactions a command may have with its environment, it is essential to assume that these communications can be observed. If we adopt this view, then it is natural to consider the *simple trace* behavior function $\mathcal{S} : \mathsf{Com} \to \mathcal{P}(\Sigma^\infty \cup \Sigma^*\delta)$ defined by:

$$\begin{array}{rcl} \mathcal{S}[\![c]\!] & = & \{\mathsf{trace}(\rho) \mid \rho = \langle c, s_0\rangle \xrightarrow{\lambda_0} \langle c_1, s_1\rangle \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_{k-1}} \langle c_k, s_k\rangle \mathsf{term}\} \\ & \cup & \{\mathsf{trace}(\rho)\delta \mid \rho = \langle c, s_0\rangle \xrightarrow{\lambda_0} \langle c_1, s_1\rangle \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_{k-1}} \langle c_k, s_k\rangle \mathsf{dead}\} \\ & \cup & \{\mathsf{trace}(\rho) \mid \rho = \langle c, s_0\rangle \xrightarrow{\lambda_0} \cdots \xrightarrow{\lambda_{k-1}} \langle c_k, s_k\rangle \xrightarrow{\lambda_k} \cdots \text{ is fair}\}. \end{array}$$

This behavior $\mathcal{S}$ again incorporates the assumption that deadlock can be distinguished from both successful termination and infinite chattering, and that every single transition can be detected.

The behavior $\mathcal{S}$ clearly includes more information about a command's possible computations than $\mathcal{M}$ does: for any command c, the set $\mathcal{S}[\![c]\!]$ is a superset of $\mathcal{M}[\![c]\!]$. 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. The reason for this apparent contradiction is that both behaviors require the same support for compositional reasoning: to reason compositionally about ε-steps along a computation of a parallel command, we need to know the communications that are possible for individual components.

Proposition 4.5.1 *The closed trace semantics $\mathcal{T}_s^\dagger$ is inequationally fully abstract with respect to $\mathcal{S}$: for all commands c and c',*

$$\mathcal{T}_s^\dagger[\![c]\!] \subseteq \mathcal{T}_s^\dagger[\![c']\!] \iff \forall P[-].\mathcal{S}[\![P[c]]\!] \subseteq \mathcal{S}[\![P[c']]\!].$$

Proof: The forward implication follows from the compositionality of $\mathcal{T}_s^\dagger$, the monotonicity of operations on trace sets, and the fact that, for all commands c, $\mathcal{S}[\![c]\!]$ can be extracted from $\mathcal{T}_s^\dagger[\![c]\!]$.

For the reverse implication, assume $\mathcal{T}_s^\dagger[\![c]\!] \not\subseteq \mathcal{T}_s^\dagger[\![c']\!]$. By Proposition 4.4.6, there exists a context $P[-]$ and a behavior β that is in $\mathcal{M}[\![P[c]]\!] - \mathcal{M}[\![P[c']]\!]$. Because

$$\mathcal{M}[\![P[c]]\!] = \{\beta \in \mathcal{S}[\![P[c]]\!] \mid \mathsf{chans}(\beta) = \{\varepsilon\}\}$$

(and likewise for $\mathcal{M}[\![P[c']]\!]$), β must also be in $\mathcal{S}[\![P[c]]\!] - \mathcal{S}[\![P[c']]\!]$. ∎

4.5.2 Stuttering and mumbling

The behaviors $\mathcal{M}$ and $\mathcal{S}$ both assume an "omniscient" observer capable of detecting every state change made during a computation. This assumption corresponds to the use of next-time operators in various temporal logics, whereby (for example) the commands skip and $\mathsf{skip};\mathsf{skip}$ can be distinguished. In many cases, however, an observer cannot be guaranteed to detect each and every state change. Moreover, the concept of "next state" can be ill-defined, because states in the operational semantics do not always correspond to processor states in a meaningful way. For example, suppose a program is distributed across multiple machines with different clock speeds. Even if an observer can look at any (or all) of the machines at any time instant, it is unclear which intervals between those instants correspond to transitions in the operational semantics. When a process on a (relatively) fast processor can perform internal actions, each clock tick may indicate a transition; when that same process is waiting to synchronize with a slower process, intermediate clock ticks may not correspond to transitions in any meaningful way. As a result, it is often appropriate to assume only that an observer is capable of seeing some subsequence of the states encountered during a computation. In doing so, we obtain notions of behavior based on the reflexive, transitive closures of the one-step transition relations.

We first introduce generalized relations $\overset{\lambda}{\Longrightarrow}$ $(\lambda \in \Lambda)$, where $\overset{\varepsilon}{\Longrightarrow}$ is the reflexive, transitive closure of $\overset{\varepsilon}{\longrightarrow}$, and $\overset{\lambda}{\Longrightarrow}$ (for $\lambda \neq \varepsilon$) is defined so that $\langle c, s\rangle \overset{\lambda}{\Longrightarrow} \langle c', s'\rangle$ if and only if there exist c_1, c_2, s_1, s_2 for which $\langle c, s\rangle \overset{\varepsilon}{\Longrightarrow} \langle c_1, s_1\rangle \overset{\lambda}{\longrightarrow} \langle c_2, s_2\rangle \overset{\varepsilon}{\Longrightarrow} \langle c', s'\rangle$. Based on these generalized relations, we define the generalized state transition trace behavior $\mathcal{M}_* : \mathsf{Com} \to \mathcal{P}(S^\infty \cup S^*\delta)$ and the generalized simple trace behavior $\mathcal{S}_* : \mathsf{Com} \to \mathcal{P}(S^\infty \cup S^*\delta)$ as follows:

$$\begin{aligned}
\mathcal{M}_*[\![c]\!] \;&=\; \{s_0 s_1 \ldots s_k \mid \langle c, s_0\rangle \overset{\varepsilon}{\Longrightarrow} \langle c_1, s_1\rangle \overset{\varepsilon}{\Longrightarrow} \cdots \overset{\varepsilon}{\Longrightarrow} \langle c_k, s_k\rangle \mathsf{term}\}\\
&\cup\; \{s_0 s_1 \ldots s_k \delta \mid \langle c_0, s_0\rangle \overset{\varepsilon}{\Longrightarrow} \langle c_1, s_1\rangle \overset{\varepsilon}{\Longrightarrow} \cdots \overset{\varepsilon}{\Longrightarrow} \langle c_k, s_k\rangle \mathsf{dead}\}\\
&\cup\; \{s_0 s_1 \ldots s_k \ldots \mid \langle c_0, s_0\rangle \overset{\varepsilon}{\Longrightarrow} \cdots \overset{\varepsilon}{\Longrightarrow} \langle c_k, s_k\rangle \overset{\varepsilon}{\Longrightarrow} \cdots \text{ is fair}\},
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}_*[\![c]\!] \;&=\; \{\mathsf{trace}(\rho) \mid \rho = \langle c, s_0\rangle \overset{\lambda_0}{\Longrightarrow} \langle c_1, s_1\rangle \overset{\lambda_1}{\Longrightarrow} \cdots \overset{\lambda_{k-1}}{\Longrightarrow} \langle c_k, s_k\rangle \mathsf{term}\}\\
&\cup\; \{\mathsf{trace}(\rho)\delta \mid \rho = \langle c, s_0\rangle \overset{\lambda_0}{\Longrightarrow} \langle c_1, s_1\rangle \overset{\lambda_1}{\Longrightarrow} \cdots \overset{\lambda_{k-1}}{\Longrightarrow} \langle c_k, s_k\rangle \mathsf{dead}\}\\
&\cup\; \{\mathsf{trace}(\rho) \mid \rho = \langle c, s_0\rangle \overset{\lambda_0}{\Longrightarrow} \cdots \overset{\lambda_{k-1}}{\Longrightarrow} \langle c_k, s_k\rangle \overset{\lambda_k}{\Longrightarrow} \cdots \text{ is fair}\}.
\end{aligned}$$

To account properly for the reflexivity and transitivity of the relations $\stackrel{\lambda}{\Longrightarrow}$, we need to impose closure conditions on trace sets corresponding to "stuttering" and "mumbling" [Lam83, Bro96b]. Stuttering captures the reflexivity of $\stackrel{\varepsilon}{\Longrightarrow}$ and has the effect of introducing idle steps into traces. A trace of form $\langle\alpha\beta,\theta\rangle$ stutters to the trace $\langle\alpha(s,\varepsilon,s)\beta,\theta\rangle$ when s is the final state of α and the initial state of β. Each partial trace of form $\langle\alpha,(F,E,\mathrm{p})\rangle$ also stutters to the trace $\langle\alpha,(\{\varepsilon\},\{\varepsilon\},\mathrm{p})\rangle$. Such stuttering steps introduce the relevant partial traces for every possible idle-step introduction: the fairness and enabling sets $\{\varepsilon\}$ reflect the possibility of an idle step immediately following α.

Mumbling has the effect of absorbing ε-steps, just as the $\stackrel{\lambda}{\Longrightarrow}$ relations absorb ε-transitions. A trace with form $\langle\alpha(s,\varepsilon,s')(s',\lambda,s'')\beta,\theta\rangle$ or $\langle\alpha(s,\lambda,s')(s',\varepsilon,s'')\beta,\theta\rangle$ mumbles to the trace $\langle\alpha(s,\lambda,s'')\beta,\theta\rangle$. Each partial trace $\langle\alpha(s,\varepsilon,s'),(F,E,\mathrm{p})\rangle$ also mumbles to the partial trace $\langle\alpha,(F\cup\{\varepsilon\},E\cup\{\varepsilon\},\mathrm{p})\rangle$. Such mumbling steps capture the intuition that, if α represents a transition sequence ending in configuration $\langle c,s\rangle$, then each direction in $E\cup\{\varepsilon\}$ represents some $\stackrel{\lambda}{\Longrightarrow}$-transition possible from the configuration $\langle c,s\rangle$.

We summarize these stuttering and mumbling sets by the relations $stut\subseteq\Phi\times\Phi$ and $mumb\subseteq\Phi\times\Phi$ defined as follows:

$$\begin{array}{rcl}
stut & = & \{(\langle\alpha\varepsilon_s\beta,\theta\rangle,\langle\alpha(s,\varepsilon,s)\beta,\theta\rangle)\mid\alpha\beta\in\Sigma^\infty-\Sigma^0\ \&\ s\in S\}\\
& \cup & \{(\langle\alpha,(F,E,\mathrm{p})\rangle,\langle\alpha,(\{\varepsilon\},\{\varepsilon\},\mathrm{p})\rangle)\mid\alpha\in\Sigma^*\},\\
mumb & = & \{(\langle\alpha(s,\varepsilon,s')(s',\lambda,s'')\beta,\theta\rangle,\langle\alpha(s,\lambda,s'')\beta,\theta\rangle)\mid\alpha(s,\lambda,s'')\beta\in\Sigma^\infty\}\\
& \cup & \{(\langle\alpha(s,\lambda,s')(s',\varepsilon,s'')\beta,\theta\rangle,\langle\alpha(s,\lambda,s'')\beta,\theta\rangle)\mid\alpha(s,\lambda,s'')\beta\in\Sigma^\infty\}\\
& \cup & \{(\langle\alpha(s,\varepsilon,s'),(F,E,\mathrm{p})\rangle,\langle\alpha,(F\cup\{\varepsilon\},E\cup\{\varepsilon\},\mathrm{p})\rangle)\mid\alpha(s,\varepsilon,s')\in\Sigma^*\}.
\end{array}$$

Intuitively, the pair (φ_1,φ_2) is in $stut$ if φ_2 can be obtained from φ_1 by inserting an extra idle step. Similarly, the pair (φ_1,φ_2) is in $mumb$ if φ_2 can be obtained from φ_1 by absorbing an ε-step.

Letting $id=\{(\alpha,\alpha)\mid\alpha\in\Sigma^\infty\}$ be the identity relation on simple traces, we follow the approach of [Bro96a] and define $stut^\infty$ and $mumb^\infty$ to be the (respective) greatest fixed points of the functionals

$$F(R)=stut\cdot R\cup id,\qquad G(R)=mumb\cdot R\cup id.$$

That is, we define

$$stut^\infty\ =\ stut^*\cdot id\ \cup\ stut^\omega,\qquad mumb^\infty\ =\ mumb^*\cdot id\ \cup\ mumb^\omega,$$

with the concatenation operator $(\cdot)$ and the iterative operators ($-^*$ and $-^\omega$) extended to sets of pairs of traces. Intuitively, the pair (φ,φ') is in $stut^\infty$ (respectively, $mumb^\infty$) if φ' can be obtained by inserting an idle step (respectively, eliding an ε-step) at some of the positions along

φ's simple-trace component. In particular, when φ is an infinite trace, the stuttering and mumbling operations can be applied at potentially infinitely many places along φ but not infinitely many times at any particular place along φ. This point is essential for avoiding the accidental introduction of divergence: stuttering should not transform the finite trace $\langle(s,\varepsilon,s),(\emptyset,\emptyset,\mathtt{f})\rangle$ into the infinite trace $\langle(s,\varepsilon,s)^{\omega},(\emptyset,\emptyset,\mathtt{i})\rangle$.

With these definitions in hand, we define closure under stuttering and mumbling on trace sets in the following way.

Definition 4.5.2 Given a trace set T, T_* is the smallest set containing T and closed under stuttering and mumbling:

- If φ is in T_* and $(\varphi,\varphi') \in \mathit{stut}^{\infty}$, then φ' is also in T_*.
- If φ is in T_* and $(\varphi,\varphi') \in \mathit{mumb}^{\infty}$, then φ' is also in T_*. $\diamond$

These closure conditions can be combined with the conditions introduced in Definition 4.2.1. For a trace set T, we define $T_*^{\dagger} = (T_*)^{\dagger}$, so that $T_*^{\dagger}$ is closed under stuttering and mumbling, as well as superset, union, convexity, displacement and contention.

We let $\mathcal{P}_*^{\dagger}\Phi$ be the set of closed sets of traces. Much as before, we can define a denotational semantic function $\mathcal{T}_{s*}^{\dagger} : \mathsf{Com} \to \mathcal{P}_*^{\dagger}\Phi$ such that, for all commands c, $\mathcal{T}_{s*}^{\dagger}[\![c]\!] = (\mathcal{T}_s[\![c]\!])_*^{\dagger}$. The addition of the stuttering and mumbling closure conditions is sufficient to yield full abstraction with respect to the generalized behaviors $\mathcal{M}_*$ and $\mathcal{S}_*$, as shown by the following results.

Proposition 4.5.3 *The semantics $\mathcal{T}_{s*}^{\dagger}$ is inequationally fully abstract with respect to $\mathcal{M}_*$: for all commands c and c',*

$$\mathcal{T}_{s*}^{\dagger}[\![c]\!] \subseteq \mathcal{T}_{s*}^{\dagger}[\![c']\!] \iff \forall P[-].\mathcal{M}_*[\![P[c]]\!] \subseteq \mathcal{M}_*[\![P[c']]\!].$$

Proof: **(Sketch)** The forward implication follows from the compositionality of $\mathcal{T}_{s*}^{\dagger}$, the monotonicity of operations on trace sets, and the fact that, for all commands c, $\mathcal{M}_*[\![c]\!]$ can be extracted from $\mathcal{T}_{s*}^{\dagger}[\![c]\!]$.

The reverse implication follows from a case analysis similar to that used in the proof of Proposition 4.4.6. In fact, the cases for finite and infinite traces are exactly the same; the case for partial traces needs to be modified only slightly, as follows.

Suppose the partial trace $\varphi = \langle\alpha,(F,E,\mathtt{p})\rangle$ is in $\mathcal{T}_{s*}^{\dagger}[\![c]\!] - \mathcal{T}_{s*}^{\dagger}[\![c']\!]$, and without loss of generality assume that $F = E$. Let $h_1,\ldots,h_k$ be the channel names appearing in c, and let x and flag be fresh identifiers, not appearing in c or c'.

Let $\langle\alpha,(E_1,E_1,\mathtt{p})\rangle,\ldots,\langle\alpha,(E_m,E_m,\mathtt{p})\rangle$ be the finite number of minimal partial α-traces in $\mathcal{T}_s^{\dagger}[\![c']\!]$, and let $Z = \bigcup_{i=1}^{m} E_i$. Closure of $\mathcal{T}_s^{\dagger}[\![c']\!]$ under union and convexity again ensures that either $E \not\subseteq Z$ or, for each $i \leq m$, $E_i \not\subseteq E$.

When $E \not\subseteq Z$, the distinguishing context is identical to that used in the proof of Proposition 4.4.6. If (instead) each $E_i \not\subseteq E$, then for each i choose a direction $d_i \in E_i - E$ such that (when possible) $d_i \neq \varepsilon$. Let g_i be a matching guard for d_i whenever $d_i \neq \varepsilon$, and define the set $G = \{d_i \mid d_i \neq \varepsilon \ \& \ 1 \leq i \leq m\}$. Let $P[-]$ be the following context:

$$([-] \parallel \mathsf{Match}_{\mathsf{x},1}(\alpha); \mathsf{y}{:=}0; \sum_{g \in G} g \rightarrow \mathsf{flag}{:=}1) \backslash h_1 \backslash \cdots \backslash h_k.$$

The only difference between this distinguishing context and the one used for the same case in the proof of Proposition 4.4.6 is that we do not include an arbitrary guard $\mathsf{a!0}$ for chosen directions $d_i = \varepsilon$. The cases where $d_i = \varepsilon$ 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.

$\mathcal{M}_*[\![P[c]]\!]$ has a deadlocked behavior corresponding to α in which the value of y in the final state is 0 and c's local portion of the state looks like the final state of α. In contrast, every behavior in $\mathcal{M}_*[\![P[c']]\!]$ with a prefix corresponding to α must do one of the following: set the value of flag to 1; terminate or deadlock in a state in which c's local portion is not the same as the final state of α; or make an infinite number of ε-transitions. ▪

Proposition 4.5.4 *The semantics $\mathcal{T}^{\dagger}_{s*}$ is inequationally fully abstract with respect to $\mathcal{S}_*$: for all commands c and c',*

$$\mathcal{T}^{\dagger}_{s*}[\![c]\!] \subseteq \mathcal{T}^{\dagger}_{s*}[\![c']\!] \iff \forall P[-].\mathcal{S}_*[\![P[c]]\!] \subseteq \mathcal{S}_*[\![P[c']]\!].$$

Proof: By obvious analogy with the proof of Proposition 4.5.1. ▪

4.5.3 Busy waiting

The behaviors $\mathcal{M}$ and $\mathcal{S}$ (as well as their generalized forms $\mathcal{M}_*$ and $\mathcal{S}_*$) assume that deadlock can be distinguished from both successful termination and infinite chattering. The semantics $\mathcal{T}^{\dagger}_{s}$ and $\mathcal{T}^{\dagger}_{s*}$ 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 a priori 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 $\mathcal{W} : \mathsf{Com} \rightarrow \mathcal{P}(S^{\infty})$, in which deadlock is modeled as busy-waiting:

$$\begin{array}{rcl}
\mathcal{W}[\![c]\!] & = & \{s_0 s_1 \ldots s_k \mid \langle c, s_0 \rangle \stackrel{\varepsilon}{\Longrightarrow} \langle c_1, s_1 \rangle \stackrel{\varepsilon}{\Longrightarrow} \cdots \stackrel{\varepsilon}{\Longrightarrow} \langle c_k, s_k \rangle \mathsf{term}\} \\
& \cup & \{s_0 s_1 \ldots s_k (s_k)^{\omega} \mid \langle c_0, s_0 \rangle \stackrel{\varepsilon}{\Longrightarrow} \langle c_1, s_1 \rangle \stackrel{\varepsilon}{\Longrightarrow} \cdots \stackrel{\varepsilon}{\Longrightarrow} \langle c_k, s_k \rangle \mathsf{dead}\} \\
& \cup & \{s_0 s_1 \ldots s_k \ldots \mid \langle c_0, s_0 \rangle \stackrel{\varepsilon}{\Longrightarrow} \cdots \stackrel{\varepsilon}{\Longrightarrow} \langle c_k, s_k \rangle \stackrel{\varepsilon}{\Longrightarrow} \cdots \text{ is strongly fair}\}.
\end{array}$$

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

To reason compositionally about $\mathcal{W}$, we introduce a semantics that is related to $\mathcal{T}_{s*}^\dagger$ 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

$$\langle \alpha(s,\varepsilon,s)^\omega, (F, E, \mathtt{i})\rangle,$$

where α 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 E (and fair mod $F \supseteq E$), and the infinitely enabled directions are the elements of E.

Employing the closure operators defined in Definitions 4.5.2 and 4.2.1 (and ignoring the conditions for partial traces), we introduce closure into our semantics from the beginning. We can give an operational characterization of the trace semantics $\mathcal{T}_{sb} : \mathsf{Com} \to \mathcal{P}_*^\dagger(\Phi)$ as follows:

$$\begin{aligned}
\mathcal{T}_{sb}[\![c]\!] = (\{&\langle \mathsf{trace}(\rho), (F, \mathsf{en}(\rho), \mathtt{f})\rangle \mid \\
&\qquad \rho = \langle c, s_0\rangle \xRightarrow{\lambda_0} \langle c_1, s_1\rangle \xRightarrow{\lambda_1} \cdots \xRightarrow{\lambda_k} \langle c_{k+1}, s_{k+1}\rangle \mathsf{term} \text{ is fair mod } F\} \\
\cup \{&\langle \mathsf{trace}(\rho)(s_k,\varepsilon,s_k)^\omega, (F, E, \mathtt{i})\rangle \mid F \supseteq E = \mathsf{inits}(c_k, s_k)\ \&\ \varepsilon \notin E\ \& \\
&\qquad \rho = \langle c, s_0\rangle \xRightarrow{\lambda_0} \langle c_1, s_1\rangle \xRightarrow{\lambda_1} \cdots \xRightarrow{\lambda_{k-1}} \langle c_k, s_k\rangle\ \&\ \neg\langle c_k, s_k\rangle \mathsf{term}\} \\
\cup \{&\langle \mathsf{trace}(\rho), (F, \mathsf{en}(\rho), \mathtt{i})\rangle \mid \\
&\qquad \rho = \langle c, s_0\rangle \xRightarrow{\lambda_0} \langle c_1, s_1\rangle \xRightarrow{\lambda_1} \cdots \xRightarrow{\lambda_k} \cdots \text{is strongly fair mod } F\})_*^\dagger.
\end{aligned}$$

The denotational characterization of $\mathcal{T}_{sb}$ is very similar to the denotational characterization of $\mathcal{T}_s$ and $\mathcal{T}_s^\dagger$. Once again we define operations on trace sets corresponding to each of the constructs of the language. In general, the operations on trace sets remain the same; the clauses for traces with form $\langle \alpha, (F, E, \mathtt{p})\rangle$ are simply ignored. However, the definition of the guarded-choice operator on trace sets depends critically on partial traces with form $\langle \varepsilon_s, (F, E, \mathtt{p})\rangle$ for generating the correct enabling information for finite traces. We therefore need to adapt the definition to use infinite traces instead of partial traces.

We first introduce a predicate idle on simple traces, such that $\mathsf{idle}(\alpha)$ is true whenever α has the form $(s,\varepsilon,s)^\omega$ for some state s. Because the first true step of any computation of a guarded command necessarily involves a non-ε transition, every idle trace α necessarily represents a partial computation "stuck" in the initial state. Consequently, we can always determine which actions are possible for a given component by examining those directions enabled infinitely often along an idle trace originating in the appropriate state. By replacing each mention of the partial trace $\langle \varepsilon_s, (F, E, \mathtt{p}) \rangle$ in the original definition by the infinite trace $\langle (s, \varepsilon, s)^\omega, (F, E, \mathtt{i}) \rangle$, we define the new guarded-choice operator as follows:

$$
\begin{aligned}
T_1 \,\square\, T_2 = {} & \{ \langle \alpha, (F, E, \mathtt{i}) \rangle \in T_1 \cup T_2 \mid \alpha \in \Sigma^\omega \ \&\ \neg \mathsf{idle}(\alpha) \} \\
& \cup \{ \langle \alpha, (F_1 \cup F_2, E_1 \cup E_2, \mathtt{i}) \rangle \mid \langle \alpha, (F_1, E_1, \mathtt{i}) \rangle \in T_1 \ \&\ \langle \alpha, (F_2, , E_2, \mathtt{i}) \rangle \in T_2 \ \&\ \mathsf{idle}(\alpha) \} \\
& \cup \{ \langle \alpha, (F_1, E_1 \cup E_2, \mathtt{f}) \rangle \mid \langle \varepsilon_s \alpha, (F_1, E_1, \mathtt{f}) \rangle \in T_1 \ \&\ \langle (s, \varepsilon, s)^\omega, (F_2, E_2, \mathtt{i}) \rangle \in T_2 \} \\
& \cup \{ \langle \alpha, (F_2, E_1 \cup E_2, \mathtt{f}) \rangle \mid \langle \varepsilon_s \alpha, (F_2, E_2, \mathtt{f}) \rangle \in T_2 \ \&\ \langle (s, \varepsilon, s)^\omega, (F_1, E_1, \mathtt{i}) \rangle \in T_1 \}.
\end{aligned}
$$

The altered definition of the guarded-choice operator represents the only necessary change to the operations on trace sets. The trace semantics $\mathcal{T}_{sb} : \mathsf{Com} \to \mathcal{P}^\dagger_*(\Phi)$ therefore can be defined in its entirety as follows. Note that the partial traces for skip, assignment, and guards are now represented by infinite, idle traces.

Definition 4.5.5 The trace semantic function $\mathcal{T}_{sb} : \mathsf{Com} \to \mathcal{P}^\dagger_*(\Phi)$ is defined by:

$$
\begin{aligned}
\mathcal{T}_{sb}[\![\mathsf{skip}]\!] &= \{ \langle (s, \varepsilon, s), (F, \emptyset, \mathtt{f}) \rangle \mid s \in S \ \&\ F \in \mathcal{P}_{\mathrm{fin}}(\Delta) \}^\dagger_* \\
\mathcal{T}_{sb}[\![i{:=}e]\!] &= \{ \langle (s, \varepsilon, [s | i = n]), (F, \emptyset, \mathtt{f}) \rangle \mid \\
& \qquad\qquad \mathsf{fv}[\![i{:=}e]\!] \subseteq \mathsf{dom}(s) \ \&\ F \in \mathcal{P}_{\mathrm{fin}}(\Delta) \ \&\ (s, n) \in \mathcal{E}[\![e]\!] \}^\dagger_* \\
\mathcal{T}_{sb}[\![c_1 ; c_2]\!] &= (\mathcal{T}_{sb}[\![c_1]\!] ; \mathcal{T}_{sb}[\![c_2]\!])^\dagger_* \\
\mathcal{T}_{sb}[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] &= (\mathcal{T}_{sb}[\![b]\!] ; \mathcal{T}_{sb}[\![c_1]\!] \cup \mathcal{T}_{sb}[\![\neg b]\!] ; \mathcal{T}_{sb}[\![C_2]\!])^\dagger_* \\
\mathcal{T}_{sb}[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!] &= ((\mathcal{T}_{sb}[\![b]\!] ; \mathcal{T}_{sb}[\![c]\!])^\omega \cup (\mathcal{T}_{sb}[\![b]\!] ; \mathcal{T}_{sb}[\![c]\!])^* ; \mathcal{T}_{sb}[\![\neg b]\!])^\dagger_* \\
\mathcal{T}_{sb}[\![h?i]\!] &= \{ \langle (s, h?n, [s | i = n]), (F, \{h?\}, \mathtt{f}) \rangle \mid i \in \mathsf{dom}(s) \ \&\ n \in \mathbb{Z} \ \&\ F \in \mathcal{P}_{\mathrm{fin}}(\Delta) \}^\dagger_* \\
& \quad \cup \{ \langle (s, \varepsilon, s)^\omega, (F, \{h?\}, \mathtt{i}) \rangle \mid i \in \mathsf{dom}(s) \ \&\ F \supseteq \{h?\} \}^\dagger_* \\
\mathcal{T}_{sb}[\![h!e]\!] &= \{ \langle (s, h!n, s), (F, \{h!\}, \mathtt{f}) \rangle \mid (s, n) \in \mathcal{E}[\![e]\!] \ \&\ F \in \mathcal{P}_{\mathrm{fin}}(\Delta) \}^\dagger_* \\
& \quad \cup \{ \langle (s, \varepsilon, s)^\omega, (F, \{h!\}, \mathtt{i}) \rangle \mid \mathsf{fv}[\![e]\!] \subseteq \mathsf{dom}(s) \ \&\ F \supseteq \{h!\} \}^\dagger_* \\
\mathcal{T}_{sb}[\![g \to c]\!] &= (\mathcal{T}_{sb}[\![g]\!] ; \mathcal{T}_{sb}[\![c]\!])^\dagger_* \\
\mathcal{T}_{sb}[\![gc_1 \,\square\, gc_2]\!] &= (\mathcal{T}_{sb}[\![gc_1]\!] \,\square\, \mathcal{T}_{sb}[\![gc_2]\!])^\dagger_* \\
\mathcal{T}_{sb}[\![c_1 \| c_2]\!] &= (\mathcal{T}_{sb}[\![c_1]\!] \| \mathcal{T}_{sb}[\![c_2]\!])^\dagger_* \\
\mathcal{T}_{sb}[\![c \backslash h]\!] &= (\mathcal{T}_{sb}[\![c]\!] \backslash h)^\dagger_*.
\end{aligned}
$$

$\diamond$

Proposition 4.5.6 *The semantics $\mathcal{T}_{sb}$ is inequationally fully abstract with respect to $\mathcal{W}$: for all commands c and c',*

$$\mathcal{T}_{sb}[\![c]\!] \subseteq \mathcal{T}_{sb}[\![c']\!] \iff \forall P[-]. \mathcal{W}[\![P[c]]\!] \subseteq \mathcal{W}[\![P[c']]\!].$$

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

The reverse implication uses an abbreviated version of the case analysis in the proof of Proposition 4.4.6. In particular, the cases for finite and infinite traces remain the same, and the case for partial traces disappears. ∎

4.5.4 Communication traces

Each of the behaviors considered so far incorporates the assumption that intermediate states encountered along a computation are observable. However, in many cases, 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. For example, object-oriented programming and abstract data types are built on this tenet: a program's implementation details should be hidden, and only its interface should be accessible. In this subsection, we 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 introduce sets Λ^* and Λ^ω that correspond (respectively) to finite and infinite sequences of visible communications. We redefine

$$\Lambda = \{h!n, h?n \mid h \in \mathsf{Chan} \ \& \ n \in \mathbb{Z}\}$$

to be the set of "interesting" communications, and we let $\Lambda^* = \{\varepsilon\} \cup \Lambda^+$ be the set of finite communication sequences. The set of all communication sequences is

$$\Lambda^\infty = \Lambda^* \ \cup \ \Lambda^*\{\varepsilon\}^\omega \ \cup \ \Lambda^\omega.$$

For each communication sequence $\eta \in \Lambda^\infty$, we define a generalized relation $\stackrel{\eta}{\Longrightarrow}$ as follows:

- When η is finite, $\langle c, s\rangle \stackrel{\eta}{\Longrightarrow} \langle c', s'\rangle$ indicates that the command c in state s can perform the sequence of visible communications η (possibly with some intermediate ε transitions), leading to the command c' in state s'. When η is the single label λ, $\stackrel{\eta}{\Longrightarrow}$ corresponds precisely to the definition of $\stackrel{\lambda}{\Longrightarrow}$ given in Subsection 4.5.2.
- When η is infinite, $\langle c, s\rangle \stackrel{\eta}{\Longrightarrow}$ indicates that there is a strongly fair computation of the command c, originating in state s, with the sequence of communications η. When η has the form $\alpha\varepsilon^\omega$, the computation diverges after α with internal chattering.

Note that the empty sequence ε is distinct from the communication sequence ε^{ω}: the former represents a finite sequence (possibly having length zero) of internal actions, whereas the latter represents an infinite sequence of internal actions.

We can now define the *communication trace behavior* $\mathcal{C} : \mathsf{Com} \to \mathcal{P}(\Lambda^{\infty} \cup \Lambda^{*}\delta)$ as follows:

$$\begin{array}{rcl} \mathcal{C}[\![c]\!] & = & \{\eta \mid \exists s, s', c'. \langle c, s\rangle \stackrel{\eta}{\Longrightarrow} \langle c', s'\rangle \mathsf{term}\} \\ & \cup & \{\eta\delta \mid \exists s, s', c'. \langle c, s\rangle \stackrel{\eta}{\Longrightarrow} \langle c', s'\rangle \mathsf{dead}\} \\ & \cup & \{\eta \mid \exists s. \langle c, s\rangle \stackrel{\eta}{\Longrightarrow} \text{is strongly fair}\}. \end{array}$$

To support compositional reasoning about $\mathcal{C}$, we introduce yet another variant of the semantics $\mathcal{T}_s^{\dagger}$ that records only the initial and terminal states of computations. Even though initial and finial states are not observable in the behavior $\mathcal{C}$, they are necessary for determining which traces can be composed in a meaningful way: in particular, the traces φ_1 of c_1 and φ_2 of c_2 can be used to generate a trace of $c_1; c_2$ only if the computation represented by φ_2 originates in the final state of the computation represented by φ_1.

To this end, we introduce a new style of simple traces. For technical reasons, we need two types of finite traces, one to represent successful computations and one to represent partial computations; thus we define the set of finite simple traces

$$\Sigma_c^{*} = (S \times \Lambda^{*} \times S) \,\cup\, (S \times \Lambda^{*}),$$

with traces (s, η, s') representing successful computations and traces (s, η) representing partial computations. Intuitively, the need for this distinction arises because we can "observe" the final state of a successful computation by transmitting the value of the finite number of variables along some channel; in contrast, there is no reliable way to interrupt a computation to observe intermediate states. Similarly, because there is no final state of an infinite computation, the set of infinite simple traces is

$$\Sigma_c^{\omega} = S \times \Lambda^{\omega}.$$

We then let $\Sigma_c^{\infty} = \Sigma_c^{*} \cup \Sigma_c^{\omega}$ be the set of all finite and infinite traces, and—using the same contextual information as before—we define the set Φ_c of fair communication traces by

$$\begin{array}{rcl} \Phi_c & = & \Sigma_c^{*} \times (\mathcal{P}_{\text{fin}}(\Delta) \times \mathcal{P}_{\text{fin}}(\Delta) \times \{\mathtt{f}\}) \\ & \cup & \Sigma_c^{\omega} \times (\mathcal{P}_{\text{fin}}(\Delta) \times \mathcal{P}_{\text{fin}}(\Delta) \times \{\mathtt{i}\}) \\ & \cup & \Sigma_c^{*} \times (\mathcal{P}_{\text{fin}}(\Delta^{+}) \times \mathcal{P}_{\text{fin}}(\Delta^{+}) \times \{\mathtt{p}\}). \end{array}$$

We now introduce a trace semantic function $\mathcal{T}_{sc} : \mathsf{Com} \to \mathcal{P}(\Phi_c)$ characterized operationally as follows:

$$\begin{array}{rcl} \mathcal{T}_{sc}[\![c]\!] & = & \{\langle (s, \eta, s'), (F, \mathsf{en}(\rho), \mathtt{f})\rangle \mid \rho = \langle c, s\rangle \stackrel{\eta}{\Longrightarrow} \langle c', s'\rangle \mathsf{term} \text{ is fair mod } F\} \\ & \cup & \{\langle (s, \eta), (F, E, \mathtt{p})\rangle \mid \rho = \langle c, s\rangle \stackrel{\eta}{\Longrightarrow} \langle c', s'\rangle \;\&\; \neg\langle c', s'\rangle \mathsf{term} \;\&\; F \supseteq E = \mathsf{inits}(c', s')\} \\ & \cup & \{\langle (s, \eta), (F, \mathsf{en}(\rho), \mathtt{i})\rangle \mid \rho = \langle c, s\rangle \stackrel{\eta}{\Longrightarrow} \text{ is fair mod } F\}. \end{array}$$

As before, two simple traces α and β are composable whenever α is an infinite or partial trace, or when the initial state of β is the final state of α. When α and β are composable, their concatenation $\alpha\beta$ is defined as follows:

$$\alpha\beta = \begin{cases} (s,\eta) & \text{if } \alpha = (s,\eta) \\ (s,\eta_1\eta_2,s'), & \text{if } \alpha = (s,\eta_1,s'') \;\; \& \; \beta = (s'',\eta_2,s'), \text{ for some } s'' \in S, \\ (s,\eta_1\eta_2), & \text{if } \alpha = (s,\eta_1,s'') \;\; \& \; \beta = (s'',\eta_2), \text{ for some } s'' \in S. \end{cases}$$

In turn, two fair traces φ_1 and φ_2 are composable whenever their simple trace components are composable. When $\varphi_1 = \langle \alpha, (F_1,E_1,R_1)\rangle$ and $\varphi_2 = \langle \beta, (F_2,E_2,R_2)\rangle$ are composable, their concatenation $\varphi_1\varphi_2$ is defined by:

$$\varphi_1\varphi_2 = \begin{cases} \langle \alpha, (F_1,E_1,R_1)\rangle, & \text{if } R_1 \in \{\mathtt{i},\mathtt{p}\}, \\ \langle \alpha\beta, (F_2,E_1\cup E_2,\mathtt{f})\rangle, & \text{if } R_1 = R_2 = \mathtt{f}, \\ \langle \alpha\beta, (F_2,E_2,R_2)\rangle, & \text{if } R_1 = \mathtt{f} \text{ and } R_2 \in \{\mathtt{i},\mathtt{p}\}. \end{cases}$$

This definition looks exactly the same as the definition for concatenation given in Section 3.3; the only difference is the interpretation of the simple-trace concatenation $\alpha\beta$. We then define $T_1;T_2 = \{\varphi_1\varphi_2 \mid \varphi_1 \in T_1 \;\&\; \varphi_2 \in T_2 \;\&\; \mathit{composable}(\varphi_1,\varphi_2)\}$. We also define infinite concatenation as before, with the obvious new interpretation of infinite concatenation on simple traces.

The definition of guarded choice on trace sets is very similar to the original definition presented in Section 3.3, the only modification in the structure of partial traces:

$$\begin{aligned} T_1 \,\square\, T_2 = {} & \{\langle \alpha, (F,E,\mathtt{i})\rangle \in T_1 \cup T_2 \mid \alpha \in \Sigma^\omega\} \\ & \cup \{\langle (s,\eta), (F,E,\mathtt{p})\rangle \in T_1 \cup T_2 \mid \eta \neq \varepsilon\} \\ & \cup \{\langle (s,\varepsilon), (F_1\cup F_2, E_1\cup E_2,\mathtt{p})\rangle \mid \langle (s,\varepsilon),(F_1,E_1,\mathtt{p})\rangle \in T_1 \;\&\; \langle (s,\varepsilon),(F_2,E_2,\mathtt{p})\rangle \in T_2\} \\ & \cup \{\langle (s,\eta,s'), (F_1, E_1\cup E_2,\mathtt{f})\rangle \mid \langle (s,\eta,s'),(F_1,E_1,\mathtt{f})\rangle \in T_1 \;\&\; \langle (s,\varepsilon),(F_2,E_2,\mathtt{p})\rangle \in T_2\} \\ & \cup \{\langle (s,\eta,s'), (F_2, E_1\cup E_2,\mathtt{f})\rangle \mid \langle (s,\eta,s'),(F_2,E_2,\mathtt{f})\rangle \in T_2 \;\&\; \langle (s,\varepsilon),(F_1,E_1,\mathtt{p})\rangle \in T_1\}. \end{aligned}$$

The definition for channel restriction is identical to that in Section 3.3, with the obvious change in interpretation for simple traces α. We need to introduce new definitions for parallel composition, but the definitions are natural simplifications of those introduced before.

We define the interleaving of two disjoint, finite simple traces (s_1,η_1,s_1') and (s_2,η_2,s_2') by

$$(s_1,\eta_1,s_1') \,\|\, (s_2,\eta_2,s_2') = (s_1\cup s_2, \eta_1\eta_2, s_1'\cup s_2').$$

The interleaving of a finite simple trace (s_1,η_1,s_1') with either a partial or infinite simple trace (s_2,η_2) is (respectively) a partial or infinite simple trace, and we define

$$(s_1,\eta_1,s_1') \,\|\, (s_2,\eta_2) = (s_1\cup s_2, \eta_1\eta_2)$$

when the traces are disjoint. Finally, the interleaving of a partial or infinite trace (s_1, η) with the empty and disjoint partial trace (s, ε) is the infinite trace defined by

$$(s_1, \eta) \rfloor\!\lfloor (s, \varepsilon) = (s_1 \cup s, \eta).$$

For context triples $\theta_1, \theta_2 \in \Gamma$, the parallel operator $\theta_1 \| \theta_2$ is as defined in Section 3.3, and for fair traces $\varphi_1 = \langle \alpha, \theta_1 \rangle$ and $\varphi_2 = \langle \beta, \theta_2 \rangle$ we again define

$$\varphi_1 \rfloor\!\lfloor \varphi_2 = \{ \langle \alpha \rfloor\!\lfloor \beta, \theta \rangle \mid \theta \in \theta_1 \| \theta_2 \}.$$

Two nonempty, finite traces $\alpha = (s, \lambda_0 \ldots \lambda_k, s')$ and $\beta = (t, \mu_0 \ldots \mu_n, t')$ **match** if $k = n$ and $\mathsf{match}(\lambda_i, \mu_i)$ for each i. For matching, disjoint traces α and β, $\alpha \| \beta$ is the trace in which α and β synchronize at each step: $\alpha \| \beta = (s \cup t, \varepsilon, s' \cup t')$. Likewise, for fair traces $\varphi_1 = \langle \alpha, \theta_1 \rangle$ and $\varphi_2 = \langle \beta, (F_2, E_2, \theta_2) \rangle$,

$$\varphi_1 \| \varphi_2 = \{ \langle \alpha \| \beta, \theta \rangle \mid \theta \in \theta_1 \| \theta_2 \}.$$

Using these new interpretations for $\varphi_1 \rfloor\!\lfloor \varphi_2$ and $\varphi_1 \| \varphi_2$, we can define the relation $\mathit{fairmerge}_c \subseteq \Phi_c \times \Phi_c \times \Phi_c$ in much the same way as before. We define

$$\mathit{fairmerge}_c = \mathit{both}_c^\omega \cup \mathit{both}_c^* \cdot \mathit{one}_c,$$

with the sets both_c and one_c defined as follows:

$$\begin{array}{rcl}
\mathit{both}_c & = & \{(\varphi_1, \varphi_2, \varphi), (\varphi_2, \varphi_1, \varphi) \mid \varphi_1, \varphi_2 \in \Phi_{\mathrm{fin}} \ \&\ \mathsf{disjoint}(\varphi_1, \varphi_2) \ \&\ \varphi \in \varphi_1 \rfloor\!\lfloor \varphi_2 \} \\
& \cup & \{(\varphi_1, \varphi_2, \varphi) \mid \varphi_1, \varphi_2 \in \Phi_{\mathrm{fin}} \ \&\ \mathsf{disjoint}(\varphi_1, \varphi_2) \ \&\ \mathsf{match}(\varphi_1, \varphi_2) \ \&\ \varphi \in \varphi_1 \| \varphi_2 \}, \\
\mathit{one}_c & = & \{(\varphi_1, \varphi_2, \varphi),\ (\varphi_2, \varphi_1, \varphi) \mid \\
& & \qquad \varphi_1 \in \Phi_c \ \&\ \varphi_2 = \langle (s, \varepsilon, s), \theta_2 \rangle \ \&\ \mathsf{disjoint}(\varphi_1, \varphi_2) \ \&\ \varphi \in \varphi_1 \rfloor\!\lfloor \varphi_2 \}, \\
& \cup & \{(\varphi_1, \varphi_2, \varphi),\ (\varphi_2, \varphi_1, \varphi) \mid \\
& & \qquad \varphi_1 \in \Phi_c \ \&\ \varphi_2 = \langle (s, \varepsilon), \theta_2 \rangle \ \&\ \mathsf{disjoint}(\varphi_1, \varphi_2) \ \&\ \varphi \in \varphi_1 \rfloor\!\lfloor \varphi_2 \}.
\end{array}$$

The mergeability criteria remain the same, and we define

$$T_1 \| T_2 = \{ \varphi \mid \varphi_1 \in T_1 \ \&\ \varphi_2 \in T_2 \ \&\ \mathit{mergeable}(\varphi_1, \varphi_2) \ \&\ (\varphi_1, \varphi_2, \varphi) \in \mathit{fairmerge}_c \}.$$

Letting $\mathcal{T}_{sc}[\![b]\!] = \{ \langle (s, \varepsilon, s), (F, \emptyset, \mathtt{f}) \rangle \mid (s, \mathtt{tt}) \in \mathcal{B}[\![b]\!] \ \&\ F \in \mathcal{P}_{\mathrm{fin}}(\Delta) \}$, we can characterize the trace semantics $\mathcal{T}_{sc} : \mathsf{Com} \to \mathcal{P}(\Phi_c)$ denotationally in the following manner. With the new interpretations for the semantic operators, the semantic clauses for $\mathcal{T}_{sc}$ look almost identical to the previous semantics; the only obvious difference is the absence of final states for the partial traces for skip, assignment, and guards.

Definition 4.5.7 The trace semantic function $\mathcal{T}_{sc} : \mathsf{Com} \to \mathcal{P}(\Phi_c)$ is defined by:

$$
\begin{aligned}
\mathcal{T}_{sc}[\![\mathsf{skip}]\!] &= \{\langle (s,\varepsilon,s),(F,\emptyset,\mathtt{f})\rangle \mid s \in S \;\&\; F \in \mathcal{P}_{\mathrm{fin}}(\Delta)\}, \\
&\quad \cup \{\langle (s,\varepsilon),(F,\{\varepsilon\},\mathtt{p})\rangle \mid s \in S \;\&\; F \supseteq \{\varepsilon\}\} \\
\mathcal{T}_{sc}[\![i{:=}e]\!] &= \{\langle (s,\varepsilon,[s|i=n]),(F,\emptyset,\mathtt{f})\rangle \mid \\
&\qquad\qquad \mathsf{fv}[\![i{:=}e]\!] \subseteq \mathsf{dom}(s) \;\&\; F \in \mathcal{P}_{\mathrm{fin}}(\Delta) \;\&\; (s,n) \in \mathcal{E}[\![e]\!]\} \\
&\quad \cup \{\langle (s,\varepsilon),(F,\{\varepsilon\},\mathtt{p})\rangle \mid \mathsf{fv}[\![i{:=}e]\!] \subseteq \mathsf{dom}(s) \;\&\; F \supseteq \{\varepsilon\}\} \\
\mathcal{T}_{sc}[\![c_1;c_2]\!] &= \mathcal{T}_{sc}[\![c_1]\!];\mathcal{T}_{sc}[\![c_2]\!] \\
\mathcal{T}_{sc}[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] &= \mathcal{T}_{sc}[\![b]\!];\mathcal{T}_{sc}[\![c_1]\!] \cup \mathcal{T}_{sc}[\![\neg b]\!];\mathcal{T}_{sc}[\![c_2]\!] \\
\mathcal{T}_{sc}[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!] &= (\mathcal{T}_{sc}[\![b]\!];\mathcal{T}_{sc}[\![c]\!])^{\omega} \cup (\mathcal{T}_{sc}[\![b]\!];\mathcal{T}_{sc}[\![c]\!])^{*};\mathcal{T}_{sc}[\![\neg b]\!] \\
\mathcal{T}_{sc}[\![h?i]\!] &= \{\langle (s,h?n,[s|i=n]),(F,\{h?\},\mathtt{f})\rangle \mid \\
&\qquad\qquad i \in \mathsf{dom}(s) \;\&\; n \in \mathbb{Z} \;\&\; F \in \mathcal{P}_{\mathrm{fin}}(\Delta)\} \\
&\quad \cup \{\langle (s,\varepsilon),(F,\{h?\},\mathtt{p})\rangle \mid i \in \mathsf{dom}(s) \;\&\; F \supseteq \{h?\}\} \\
\mathcal{T}_{sc}[\![h!e]\!] &= \{\langle (s,h!n,s),(F,\{h!\},\mathtt{f})\rangle \mid (s,n) \in \mathcal{E}[\![e]\!] \;\&\; F \in \mathcal{P}_{\mathrm{fin}}(\Delta)\} \\
&\quad \cup \{\langle (s,\varepsilon),(F,\{h!\},\mathtt{p})\rangle \mid \mathsf{fv}[\![e]\!] \subseteq \mathsf{dom}(s) \;\&\; F \supseteq \{h!\} \;\&\; \} \\
\mathcal{T}_{sc}[\![g \to c]\!] &= \mathcal{T}_{sc}[\![g]\!];\mathcal{T}_{sc}[\![c]\!] \\
\mathcal{T}_{sc}[\![gc_1 \,\square\, gc_2]\!] &= \mathcal{T}_{sc}[\![gc_1]\!] \,\square\, \mathcal{T}_{sc}[\![gc_2]\!] \\
\mathcal{T}_{sc}[\![c_1 \| c_2]\!] &= \mathcal{T}_{sc}[\![c_1]\!] \| \mathcal{T}_{sc}[\![c_2]\!] \\
\mathcal{T}_{sc}[\![c \backslash h]\!] &= \mathcal{T}_{sc}[\![c]\!] \backslash h.
\end{aligned}
$$

◇

Not surprisingly, the semantics $\mathcal{T}_{sc}$ is sound with respect to the behavior $\mathcal{C}$, but not fully abstract. To achieve full abstraction, we again need to close trace sets under the closure conditions introduced in Definition 4.2.1. As before, we can then define a closed trace semantic function $\mathcal{T}^{\dagger}_{sc} : \mathsf{Com} \to \mathcal{P}^{\dagger}(\Phi_c)$ denotationally, so that, for each command c, $\mathcal{T}^{\dagger}_{sc}[\![c]\!] = \mathcal{T}_{sc}[\![c]\!]^{\dagger}$.

The proof of full abstraction is similar to the full abstraction proof in Section 4.4. We make the initial and final states of computations "observable" by transmitting the value of state variables along a fresh channel.

Proposition 4.5.8 *The closed trace semantics $\mathcal{T}^{\dagger}_{sc}$ is (inequationally) fully abstract with respect to $\mathcal{C}$: for all commands c and c',*

$$\mathcal{T}^{\dagger}_{sc}[\![c]\!] \subseteq \mathcal{T}^{\dagger}_{sc}[\![c']\!] \iff \forall P[-].\, \mathcal{C}[\![P[c]]\!] \subseteq \mathcal{C}[\![P[c']]\!].$$

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

The reverse implication follows from a case analysis similar to that used in the proof of Proposition 4.4.6. The main difference is that the distinguishing contexts must account for an observable behavior that is communication-based rather than state-based. Whereas the previous contexts signal the occurrence of particular events by setting the values of certain identifiers, these contexts must signal such occurrences with visible communication events.

For example, suppose that c has an infinite trace $\langle (s,\eta),\ (F,E,\mathtt{i})\rangle$ that c' does not. Let $x_1,\dots,x_n$ be the free identifiers of c and c'; without loss of generality, $\mathsf{dom}(s) = \{x_1,\dots,x_n\}$. Let $h_1,\dots,h_k$ be the channel names appearing in c and c', and let a,b and $c_1,\dots,c_k$ be fresh channel names. Finally, let the sets G_{x} and G_{y} be constructed from the minimal α-traces of c' as in previous proofs.

The distinguishing context we construct uses a modification of the command Guess used previously. Roughly speaking, each pair of lines

$$\begin{array}{ll} 2i-1\text{:} & \mathsf{synch}{:=}0;((h_i!\mathsf{value} \to \mathsf{synch}{:=}1) \,\Box\, \sum_{g\in G}(g \to \mathsf{f1}{:=}1)) \\ 2i\text{:} & \mathsf{synch}{:=}0;((h_i?\mathsf{value} \to \mathsf{synch}{:=}1) \,\Box\, \sum_{g\in G}(g \to \mathsf{f1}{:=}1)) \end{array}$$

in $\mathsf{Guess}(H,G,\mathsf{f1})$ of Figure 4.1 can be replaced by the following pair of lines, where $c_1,\dots,c_k$ and b are fresh channels:

$$\begin{array}{ll} 2i-1\text{:} & ((h_i!\mathsf{value} \to c_i!0 \to c_i!\mathsf{value}) \,\Box\, \sum_{g\in G}(g \to \mathsf{b}!0)) \\ 2i\text{:} & ((h_i?\mathsf{value} \to c_i!1 \to c_i!\mathsf{value}) \,\Box\, \sum_{g\in G}(g \to \mathsf{b}!0)) \end{array}$$

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 "transferred value". The guard b!0 serves the same purpose that the variable flag played in the previous proof.

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

$$(\mathsf{a}!x_1 \to \mathsf{a}!x_2 \to \dots \mathsf{a}!x_n \to [-] \parallel \mathsf{Guess}(H,G_{\mathsf{x}},\mathsf{b}!0) \parallel \sum_{g\in G_{\mathsf{y}}} g \to \mathsf{b}!0)\backslash h_1\backslash\cdots\backslash h_k.$$

$\mathcal{C}[\![P[c]]\!]$ contains a behavior corresponding to (s,η) in which the communication b!0 never occurs. In contrast, every behavior of $\mathcal{C}[\![P[c']]\!]$ corresponding to α must eventually perform the action b!0. ∎

Chapter 5

Strong Channel Fairness

In Chapters 3 and 4, we constructed several trace semantics that incorporate assumptions of strong process fairness and yield full abstraction with respect to specific notions of strongly fair behavior. The ease with which we adapted the strongly fair semantics to yield several full abstraction results indicates a certain robustness of the trace framework. In this chapter, we further demonstrate the framework's robustness by constructing a semantics that incorporates assumptions of *strong channel fairness*. The channel-fair semantics retains a lot of the essence of the strongly fair semantics. However, the additional burden of determining when communication is enabled infinitely often on a given channel requires a more complex semantic structure.

We begin by formalizing the concept of channel fairness and introducing a parameterized form of channel fairness. This parameterization of channel fairness admits a compositional characterization and guides the construction of the channel-fair semantics. The need to determine when communication is enabled infinitely often on particular channels makes the resulting channel-fair semantics more complex than the strongly fair semantics of the previous chapter, and it is not fully abstract. We conclude the chapter by discussing this lack of full abstraction: we hint how the semantics might be altered to achieve full abstraction, and we describe why the lack of full abstraction is not an indictment of either the trace framework or the channel-fair semantics.

5.1 Channels, Names, Durations, and Scopes

Informally, a computation is *strongly channel-fair* if it satisfies the following two conditions:

- Every process enabled infinitely often makes progress infinitely often.

- Every channel on which communication is enabled infinitely often is used infinitely often.

That is, strong channel fairness combines strong process fairness with additional constraints on the use of infinitely enabled channels. Thus, for example, every channel-fair computation of the command

$$(\mathsf{while\ true\ do\ } (\mathsf{a!0} \Box \mathsf{b!1}) \parallel \mathsf{while\ true\ do\ } (\mathsf{a?x} \Box \mathsf{b?x}))\backslash \mathsf{a} \backslash \mathsf{b}$$

uses each of the channels a and b infinitely often, thereby changing the value of identifier x from 0 to 1 (and vice versa) infinitely often. Such a computation is also strongly process-fair. However, strong process fairness does not require the infinite use of both channels, as long as both processes make infinite progress: the infinite computation in which x remains set to 0 is also strongly fair.

To formalize strong channel fairness, however, we must make explicit what we mean by the term *channel*. So far, we have used the term in two distinct ways. First, we have used it as a synonym for *channel name*, meaning a member of the syntactic class Chan. Second, we have used *channel* to refer to the abstract (and rather nebulous) concept of a link by which processes communicate with one another and their external environment; in this sense, a channel is a semantic entity. Because channel names provide the only way to refer to particular links, we tend to blur the distinction between names and links, using the phrase "channel a" to mean "the channel designated by name a". This distinction may seem a trifling detail, but it is crucial for defining and understanding channel fairness. Intuitively, the relationship between channel names and channels is analogous to that between a procedure's local variables and their instantiation during procedure activation. We make this connection more explicit in the following discussion.

Let c be a command in which the channel name occurs free (i.e., $h \in \mathsf{fc}[\![c]\!]$). The restriction operator "$\backslash h$" **binds** the free occurrences of h in c, and each occurrence of h in the command $c\backslash h$ is said to be **bound**.[1] For any command c and channel name h, the **(syntactic) scope** of an occurrence of h in c is the smallest subcommand of c in which that occurrence is bound by h; when the occurrence is free in c, its scope is the command c itself. For example, in the command

$$Q \equiv \mathsf{while\ true\ do\ } ((\mathsf{a!0} \Box \mathsf{b!1}) \parallel (\mathsf{a?x} \Box \mathsf{b?x}))\backslash \mathsf{a},$$

the scope of each occurrence of a is the command $((\mathsf{a!0} \Box \mathsf{b!1}) \parallel (\mathsf{a?x} \Box \mathsf{b?x}))\backslash \mathsf{a}$, and the scope of each occurrence of b is the command Q. A single name h may have multiple scopes within a program c, with each scope being the scope of some occurrence of h in c. For example, in the program

$$(\mathsf{a!1} \parallel \mathsf{a?x})\backslash \mathsf{a}; (\mathsf{while\ true\ do\ } (\mathsf{a!0} \parallel \mathsf{a?x}))\backslash \mathsf{a},$$

[1] Indeed, a more suggestive syntax for the command $c\backslash h$ might be "$\mathsf{new\ channel\ in\ } c$", which emphasizes the similarity between channel names and local variables.

the name a has two different scopes: $(\mathsf{a!1} \| \mathsf{a?x})\backslash\mathsf{a}$ and $(\mathsf{while\ true\ do}\ (\mathsf{a!0} \| \mathsf{a?x}))\backslash\mathsf{a}$.

During program execution, each entry into a channel name's scope creates a new channel, and each exit from a name's scope destroys that channel. The **duration** (or **extent**) of a channel is that portion of the execution during which the channel exists. For example, consider an infinite computation of the program

$$P_1 \equiv (\mathsf{while\ true\ do}\ (\mathsf{a!0} \,\Box\, \mathsf{b!1} \;\|\; \mathsf{a?x} \,\Box\, \mathsf{b?x}))\backslash\mathsf{a}\backslash\mathsf{b}.$$

The two names a and b are associated with two different channels, each of which has infinite duration. Because communication is enabled on both channels infinitely often, every strongly channel-fair computation of P_1 must change the value of x from 0 to 1 (and vice versa) infinitely often. In contrast, consider the infinite computations of the program

$$P_2 \equiv \mathsf{while\ true\ do}\ ((\mathsf{a!0} \,\Box\, \mathsf{b!1} \;\|\; \mathsf{a?x} \,\Box\, \mathsf{b?x})\backslash\mathsf{a}\backslash\mathsf{b}).$$

Each iteration through the loop creates (and subsequently destroys) new channels identified by the names a and b; each such channel has only finite duration. No channel ever can be enabled infinitely often in an infinite computation of P_2, because no channel ever has infinite duration. As a result, an infinite computation of P_2 that never sets the value of x to 1 is still strongly channel-fair.

The programs P_1 and P_2 illustrate the difference between channel names and channels, as well as the effect this distinction has on channel fairness: although P_1 and P_2 can match each other step-for-step, P_2 has channel-fair computations that do not correspond to channel-fair computations of P_1. Out of necessity, we shall continue to refer to channels by their names throughout this dissertation. However, it is important to remember that channel fairness involves assumptions about channels, not channel names.

5.2 Parameterized Channel Fairness

As demonstrated in Section 3.1, the fair computations of a command cannot be characterized (in general) by referring only to the fair computations of its subcommands. Synchronous communication requires two active participants, and hence the enabledness of a process (or of a particular communication) depends on the status of other processes. The solution for strong process fairness was to consider "almost strongly fair" computations; we adopt a similar approach here for channel fairness.

A computation can fail to be strongly channel-fair for one of two reasons: (1) some process is enabled infinitely often and yet makes only finite progress, or (2) some channel on which communication is enabled infinitely often is used only finitely often. Similarly, an "almost channel-fair" computation can be characterized by a combination of *process constraints* (representing the infinitely enabled processes that fail to make infinite progress) and *channel constraints* (representing the infinitely enabled channels that do not get used infinitely often).

It is important to separate the process and channel constraints, because they represent different types of assumptions. Intuitively, the process constraints (which we can represent by a set F of directions, as in Chapter 3) correspond to infinitely enabled processes that, when the original command is placed in a larger context, cease to be enabled infinitely often and hence are not treated unfairly. In this sense, process constraints limit the types of communications other processes are allowed to provide. In contrast, the channel constraints (which we can represent by a set H of channels) correspond to infinitely enabled channels that, when the original command is placed in a larger context, cease to be treated unfairly, either because they are no longer enabled infinitely often, or because some other component uses them infinitely often. Thus, in some sense, channel constraints can actually encourage other processes to perform certain types of communications.

Combining process and channel constraints, we parameterize strong channel fairness by pairs (F,H), where F is a finite set of directions and H is a finite set of channels. Informally, a computation ρ is channel-fair mod (F,H) if and only if it is strongly fair mod F and the set H contains exactly[2] those channels that are enabled infinitely often but used only finitely often along ρ. When the sets F and H are both empty, this characterization coincides with the traditional notion of strong channel fairness introduced in Subsection 2.2.2. The formal characterization of parameterized channel fairness follows.

Definition 5.2.1 A computation ρ of command c is **strongly channel-fair modulo** (F,H) (or, **channel-fair mod** (F,H)) provided ρ satisfies one of the following conditions:

- ρ is a finite, successfully terminating computation, and $H = \emptyset$;
- ρ is a partial computation whose final configuration is blocked modulo F, and $H = \emptyset$;
- ρ is an infinite computation, c has form $(c_1;c_2)$, $(\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)$, or $(g \to c_1)$, and the underlying infinite computation of c_1 or c_2 is fair mod (F,H);
- ρ is an infinite computation, c has form $(\mathsf{while}\ b\ \mathsf{do}\ c')$, all underlying computations of c' are fair mod F, and H contains exactly those channels that are enabled infinitely often but used only finitely often along ρ;
- ρ is an infinite computation, c has form $(gc_1 \square gc_2)$, and the underlying computation of the selected gc_i is fair mod (F,H);

[2]There is an asymmetry in this parameterization: the set F is a superset of a computation's process constraints, whereas the set H contains precisely its channel constraints. Having H be a superset of the channel constraints would still permit an accurate compositional characterization; the choice to have H contain exactly the relevant constraints merely simplifies the presentation of the channel-fair trace semantics in the next section.

- ρ is an infinite computation, c has form $c' \backslash h$, the underlying computation ρ' of c' is fair modulo $(F \cup \{h!, h?\}, H \cup \{h\})$, and synchronization on h is not enabled infinitely often along ρ';

- ρ is an infinite computation, c has form $c_1 \| c_2$, and there exist sets F_1, F_2, H_1, H_2, and computations ρ_1 of c_1 and ρ_2 of c_2 such that:
  - ρ_1 is fair mod (F_1, H_1) and ρ_2 is fair mod (F_2, H_2),
  - ρ can be obtained by merging and synchronizing ρ_1 and ρ_2,
  - $F \supseteq F_1 \cup F_2$ and $H = H_1 \cup H_2 - (\mathsf{uchans}(\rho_1) \cup \mathsf{uchans}(\rho_2))$, where $\mathsf{uchans}(\rho_i)$ is the set of channels used infinitely often along ρ_i,
  - neither ρ_i enables infinitely often any direction matching a member of F_j $(i \neq j)$,
  - neither ρ_i uses a direction in F_j infinitely often $(i \neq j)$. $\diamond$

This definition captures the inherent duality between process and channel constraints. Process constraints are verified during parallel composition to guarantee that neither component violates the other's assumptions, and they are discharged through channel restriction. In contrast, channel constraints are discharged either through parallel composition (when one component uses another's unused channels) or through restriction (provided synchronization is not enabled infinitely often), and they are always verified during channel restriction to ensure that no channel with synchronization enabled infinitely often gets ignored.

The following examples illustrate the notion of parameterized channel fairness.

Example 5.2.2 Consider the commands $Q_1 \equiv \mathsf{while\ true\ do}\ Q_1'$ and $Q_2 \equiv \mathsf{while\ true\ do}\ Q_2'$, where Q_1' and Q_2' are defined as follows:

$$Q_1' \equiv \mathsf{a!0} \rightarrow (\mathsf{b!0} \rightarrow \mathsf{skip} \,\Box\, \mathsf{c!0} \rightarrow \mathsf{skip}), \qquad Q_2' \equiv (\mathsf{a!0} \rightarrow \mathsf{skip} \,\Box\, \mathsf{b?x} \rightarrow \mathsf{skip}).$$

- Let ρ_1 be the following computation of Q_1 in which channel b is never used:

$$\begin{aligned} \langle Q_1, s_1 \rangle &\xrightarrow{\varepsilon} \langle (\mathsf{a!0} \rightarrow (\mathsf{b!0} \rightarrow \mathsf{skip} \,\Box\, \mathsf{c!0} \rightarrow \mathsf{skip})); Q_1, s_1 \rangle \\ &\xrightarrow{\mathsf{a!0}} \langle (\mathsf{b!0} \rightarrow \mathsf{skip} \,\Box\, \mathsf{c!0} \rightarrow \mathsf{skip}); Q_1, s_1 \rangle \\ &\xrightarrow{\mathsf{c!0}} \langle \mathsf{skip}; Q_1, s_1 \rangle \xrightarrow{\varepsilon} \langle Q_1, s_1 \rangle \xrightarrow{\varepsilon} \cdots \end{aligned}$$

  This computation is channel-fair mod $(\emptyset, \{\mathsf{b}\})$: no process blocks, and channel b is the only infinitely enabled channel not used infinitely often.

- Let ρ_2 be the following computation of Q_2 in which the channel b is never used:

$$\begin{aligned}\langle Q_2, s_2\rangle &\xrightarrow{\varepsilon} \langle (\mathsf{a!0} \to \mathsf{skip} \,\Box\, \mathsf{b?x} \to \mathsf{skip}); Q_2, s_2\rangle \\ &\xrightarrow{\mathsf{a!0}} \langle \mathsf{skip}; Q_2, s_2\rangle \xrightarrow{\varepsilon} \langle Q_2, s_2\rangle \xrightarrow{\varepsilon} \cdots\end{aligned}$$

  This computation is also channel-fair mod $(\emptyset, \{\mathsf{b}\})$.

- Let ρ be the following computation, which results from an interleaving of the computations ρ_1 and ρ_2:

$$\begin{aligned}&\langle Q_1 \parallel Q_2, s\rangle \xrightarrow{\varepsilon} \langle Q_1 \parallel ((\mathsf{a!0} \to \mathsf{skip} \,\Box\, \mathsf{b?x} \to \mathsf{skip}); Q_2), s\rangle \\ &\quad\xrightarrow{\varepsilon} \langle (\mathsf{a!0} \to (\mathsf{b!0} \to \mathsf{skip} \,\Box\, \mathsf{c!0} \to \mathsf{skip})); Q_1 \parallel ((\mathsf{a!0} \to \mathsf{skip} \,\Box\, \mathsf{b?x} \to \mathsf{skip}); Q_2), s\rangle \\ &\quad\xrightarrow{\mathsf{a!0}} \langle (\mathsf{a!0} \to (\mathsf{b!0} \to \mathsf{skip} \,\Box\, \mathsf{c!0} \to \mathsf{skip})); Q_1 \parallel (\mathsf{skip}; Q_2), s\rangle \\ &\quad\xrightarrow{\mathsf{a!0}} \langle (\mathsf{b!0} \to \mathsf{skip} \Box \mathsf{c!0} \to \mathsf{skip}); Q_1 \parallel (\mathsf{skip}; Q_2), s\rangle \\ &\quad\xrightarrow{\mathsf{c!0}} \langle (\mathsf{skip}; Q_1) \parallel (\mathsf{skip}; Q_2), s\rangle \\ &\quad\xrightarrow{\varepsilon} \langle Q_1 \parallel (\mathsf{skip}; Q_2), s\rangle \\ &\quad\xrightarrow{\varepsilon} \langle Q_1 \parallel Q_2, s\rangle \\ &\quad\xrightarrow{\varepsilon} \cdots\end{aligned}$$

  This computation is channel-fair mod $(\emptyset, \{\mathsf{b}\})$. Moreover, because synchronization on channel b is never enabled, the corresponding computation of $(Q_1 \| Q_2)\backslash \mathsf{b}$ is channel-fair mod $(\emptyset, \emptyset)$. $\diamond$

The next example illustrates how the channel fairness of a computation can depend on the order in which independent actions occur.

Example 5.2.3 Let ρ_1 and ρ_2 be the computations defined in the previous example, and let ρ' be the following computation, which also arises from an interleaving of ρ_1 and ρ_2:

$$\begin{aligned}\langle Q_1 \parallel Q_2, s\rangle &\xrightarrow{\varepsilon} \langle Q_1 \parallel (\mathsf{a!0} \to \mathsf{skip} \,\Box\, \mathsf{b?x} \to \mathsf{skip}); Q_2, s\rangle \\ &\xrightarrow{\varepsilon} \langle (\mathsf{a!0} \to (\mathsf{b!0} \to \mathsf{skip} \,\Box\, \mathsf{c!0} \to \mathsf{skip})); Q_1 \parallel (\mathsf{a!0} \to \mathsf{skip} \,\Box\, \mathsf{b?x} \to \mathsf{skip}); Q_2, s\rangle \\ &\xrightarrow{\mathsf{a!0}} \langle (\mathsf{b!0} \to \mathsf{skip} \,\Box\, \mathsf{c!0} \to \mathsf{skip}); Q_1 \parallel (\mathsf{a!0} \to \mathsf{skip} \,\Box\, \mathsf{b?x} \to \mathsf{skip}); Q_2, s\rangle \\ &\xrightarrow{\mathsf{a!0}} \langle (\mathsf{b!0} \to \mathsf{skip} \Box \mathsf{c!0} \to \mathsf{skip}); Q_1 \parallel (\mathsf{skip}; Q_2), s\rangle \\ &\xrightarrow{\mathsf{c!0}} \langle (\mathsf{skip}; Q_1) \parallel (\mathsf{skip}; Q_2), s\rangle \\ &\xrightarrow{\varepsilon} \langle Q_1 \parallel (\mathsf{skip}; Q_2), s\rangle \\ &\xrightarrow{\varepsilon} \langle Q_1 \parallel Q_2, s\rangle \\ &\xrightarrow{\varepsilon} \cdots\end{aligned}$$

The computation ρ' is also channel-fair mod $(\emptyset, \{\mathsf{b}\})$. However, synchronization on channel b is enabled in each of the (infinitely many) configurations with form

$$\langle (\mathsf{b!0} \to \mathsf{skip} \;\Box\; \mathsf{c!0} \to \mathsf{skip}); Q_1 \parallel (\mathsf{a!0} \to \mathsf{skip} \;\Box\; \mathsf{b?x} \to \mathsf{skip}); Q_2, s\rangle.$$

As a result, the computation of $(Q_1 \| Q_2)\backslash \mathsf{b}$ that corresponds to ρ' is not channel-fair mod $(\emptyset, \emptyset)$. ◇

Finally, the following example highlights the dual nature of the process and fairness constraints.

Example 5.2.4 Consider the following two commands

$$C_1 \equiv (\mathsf{b!0} \parallel \mathsf{while\ true\ do}\ (\mathsf{a!0} \Box \mathsf{c!0})), \qquad C_2 \equiv \mathsf{while\ true\ do}\ (\mathsf{c?x} \parallel \mathsf{c!0}).$$

Let ρ_1 be an infinite computation of C_1 that repeatedly uses channel a and never uses channels b and c; such a computation is channel-fair mod $(\{\mathsf{b!}\}, \{\mathsf{b}, \mathsf{c}\})$. Additionally, let ρ_2 be an infinite computation of C_2 in which the components $\mathsf{c?x}$ and $\mathsf{c!0}$ repeatedly synchronize with one another; this computation is channel-fair mod $(\emptyset, \emptyset)$.

Finally, let ρ be an infinite computation of $C_1 \| C_2$ that results from some fair interleaving of the computations ρ_1 and ρ_2. The computation ρ is channel-fair mod $(\{\mathsf{b!}\}, \{\mathsf{b}\})$: ρ_2 respects ρ_1's process constraints (that is, it does not enable $\mathsf{b?}$ infinitely often or use $\mathsf{b!}$ infinitely often), and it discharges one of ρ_1's channel constraints by using channel c infinitely often. Because synchronization is not enabled on channel b infinitely often, it follows that the corresponding computation of $(C_1 \| C_2)\backslash \mathsf{b}$ is channel-fair mod $(\emptyset, \emptyset)$. ◇

5.3 Channel-Fair Traces

The definition of parameterized strong fairness is clearly embedded in the definition of parameterized channel fairness. Moreover, the only difference between the two definitions is that the latter also manipulates sets H of channel constraints. This strong connection might lead us to expect that we can construct appropriate channel-fair traces simply by adding to the strongly fair traces an additional component that records the relevant channel constraints: for example, the trace $\langle \alpha, (F, H, E, \mathtt{i})\rangle$ might represent a channel-fair mod (F, H) computation with infinitely enabled directions E.

Unfortunately, the apparent simplicity of the parameterized channel-fairness definition obscures an important fact: determining whether synchronization is enabled on a particular channel requires more information than merely the sets of directions enabled along a transition sequence. For example, recall the commands

$$Q_1' \equiv \mathsf{a!0} \to (\mathsf{b!0} \to \mathsf{skip} \Box \mathsf{c!0} \to \mathsf{skip}), \qquad Q_2' \equiv (\mathsf{a!0} \to \mathsf{skip}) \Box (\mathsf{b?x} \to \mathsf{skip})$$

from Examples 5.2.2 and 5.2.3. The command $Q_1'\|Q_2'$ has two different computations that can each be represented by the simple trace

$$(s,\mathsf{a!0},s)(s,\mathsf{a!0},s)(s,\mathsf{c!0},s)(s,\varepsilon,s)(s,\varepsilon,s),$$

one in which Q_1' makes the first move and one in which Q_2' makes the first move. In each case, input and output are both enabled on channel b, but synchronization on channel b is enabled only when Q_1' makes the first transition. Simply knowing that Q_1' enables $\mathsf{b!}$ and Q_2' enables $\mathsf{b?}$ along their respective computations is insufficient to determine whether synchronization on channel b is enabled: we also need to know whether $\mathsf{b!}$ and $\mathsf{b?}$ are enabled *at the same time*. Generally speaking, even knowing that both directions of a given channel are enabled at the same time may still be inadequate for determining whether synchronization is enabled. For example, consider the following two commands

$$\begin{array}{rcl} Q_3 & \equiv & (\mathsf{a!0}\,\square\,\mathsf{b?x}) \parallel (\mathsf{a!0}\,\square\,\mathsf{b!0}), \\ Q_4 & \equiv & (\mathsf{a!0} \to (\mathsf{a!0}\,\square\,\mathsf{b!0}))\ \square\ (\mathsf{a!0} \to (\mathsf{a!0}\,\square\,\mathsf{b?x})) \\ & \square & (\mathsf{b?x} \to (\mathsf{a!0}\,\square\,\mathsf{b!0}))\ \square\ (\mathsf{b!0} \to (\mathsf{a!0}\,\square\,\mathsf{b?x})), \end{array}$$

both of which have computations that can be represented by the simple trace $(s,\mathsf{a!0},s)(s,\mathsf{a!0},s)$. In each case, both $\mathsf{b!}$ and $\mathsf{b?}$ are enabled in the initial configuration. However, synchronization on channel b is enabled only along the computation of Q_3.

For this reason, we consider sequences of *enabling sets*, which are finite sets of channels and directions such that the channel h appears only in sets that also contain the directions $h?$ and $h!$. Intuitively, the presence of channel h in an enabling set E indicates that synchronization on channel h is enabled,and the absence of h indicates that synchronization on h is not enabled. Given commands c_1 and c_2 with enabling sets E_1 and E_2, respectively, the set

$$E_1\|E_2 = E_1 \cup E_2 \cup \{h \mid \exists d \in E_1.\ \bar{d} \in E_2\ \&\ \mathsf{chan}(d) = h\}$$

represents the enabling set of the parallel command $c_1\|c_2$: the parallel command can perform any action either component can, and it can also synchronize on any channel on which the two components' enabling sets match.

For any configuration $\langle c,s\rangle$, $\mathsf{comms}(c,s)$ is the set of directions and channels that describe the possible communications from the configuration $\langle c,s\rangle$. A structurally inductive definition of $\mathsf{comms}(c,s)$ appears in Figure 5.1. The set $\mathsf{comms}(c,s)$ is related to the set $\mathsf{inits}(c,s)$, except that it may include channels and it never includes ε: in particular, the channel h, rather than the label ε, indicates the possibility of synchronization on channel h. As is clear from the inductive definition, channels appear in $\mathsf{comms}(c,s)$ only through parallel composition. Thus, for example, the programs Q_3 and Q_4 described earlier can be distinguished based on their initial enabling sets: $\mathsf{comms}(Q_3,s) = \{\mathsf{a!},\mathsf{b!},\mathsf{b?},\mathsf{b}\}$, whereas $\mathsf{comms}(Q_4,s) = \{\mathsf{a!},\mathsf{b!},\mathsf{b?}\}$.

$$
\begin{aligned}
\mathsf{comms}(\mathsf{skip}, s) &= \emptyset \\
\mathsf{comms}(i{:=}e, s) &= \emptyset \\
\mathsf{comms}(c_1; c_2, s) &= \mathsf{comms}(c_1, s) \\
\mathsf{comms}(\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, s) &= \emptyset \\
\mathsf{comms}(\mathsf{while}\ b\ \mathsf{do}\ c, s) &= \emptyset \\
\mathsf{comms}(h?i, s) &= \{h?\} \\
\mathsf{comms}(h!e, s) &= \{h!\} \\
\mathsf{comms}(g \to c, s) &= \mathsf{comms}(g, s) \\
\mathsf{comms}(gc_1 \,\square\, gc_2, s) &= \mathsf{comms}(gc_1, s) \cup \mathsf{comms}(gc_2, s) \\
\mathsf{comms}(c_1 \| c_2, s) &= \mathsf{comms}(c_1, s) \parallel \mathsf{comms}(c_2, s) \\
\mathsf{comms}(c \backslash h, s) &= \mathsf{comms}(c, s) - \{h!, h?, h\}.
\end{aligned}
$$

Figure 5.1: Inductive definition of $\mathsf{comms}(c, s)$.

Reusing the set Σ^∞ of simple traces defined in Section 3.3, we can now define the set Φ_{ch} of **channel-fair traces** by

$$\Phi_{ch} = \Sigma^\infty \times (\mathcal{P}_{\text{fin}}(\Delta^+) \times \mathcal{P}_{\text{fin}}(\Delta^+) \times \mathcal{P}_{\text{fin}}(\Delta \cup \mathsf{Chan})^\infty \times \{\mathtt{f}, \mathtt{p}, \mathtt{i}\}).$$

Intuitively, the trace $\langle \alpha, (F, U, \mathcal{E}, \mathtt{f}) \rangle$ represents a (necessarily channel-fair) successfully terminating computation having the finite sequence $\mathcal{E}$ of enabling sets and the set U of enabled but unused channels. The trace $\langle \alpha, (F, U, \mathcal{E}, \mathtt{i}) \rangle$ represents an infinite, fair mod (F, U) computation having the infinite sequence $\mathcal{E}$ of enabling sets. Finally, the trace $\langle \alpha, (F, \emptyset, \mathcal{E}, \mathtt{p}) \rangle$ represents a partial computation having the finite sequence $\mathcal{E}$ of enabling sets; as in strongly fair traces, the set F is a superset of $\mathsf{inits}(c_k, s_k)$, where $\langle c_k, s_k \rangle$ is the final configuration of ρ.

For any computation ρ, $\mathsf{trace}(\rho)$ is (as before) the simple trace that records the transitions made along ρ, and $\mathsf{unused}(\rho)$ is the set of channels that are enabled but not used along ρ. We also define $\mathsf{En}(\rho)$ to be the sequence of enabling sets encountered along the computation ρ. For example, if ρ is the (possibly partial) computation

$$\rho \ = \ \langle c, s_0 \rangle \xrightarrow{\lambda_0} \langle c_1, s_1 \rangle \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_{k-1}} \langle c_k, s_k \rangle,$$

then the sequence $\mathsf{En}(\rho)$ is defined as $\mathsf{En}(\rho) = \langle E_0, E_1, \ldots, E_k \rangle$, where $E_i = \mathsf{comms}(c_i, s_i)$ for each i. Note that, when the configuration $\langle c_k, s_k \rangle$ is terminal, the set $E_k = \mathsf{comms}(c_k, s_k) = \emptyset$. Moreover, for any finite transition sequence ρ of length k, $\mathsf{En}(\rho)$ is a sequence of $k+1$ sets. For technical reasons that will be made explicit in the next section, it is important to record the types of communications enabled in the final configuration of a transition sequence.

Using these definitions, we can give an operational characterization of a channel-fair trace semantics $\mathcal{T}_{ch} : \mathsf{Com} \to \mathcal{P}(\Phi_{ch})$ as follows:

$$\begin{aligned}
\mathcal{T}_{ch}[\![c]\!] = \{ & \langle \mathsf{trace}(\rho), (F, \mathsf{unused}(\rho), \mathsf{En}(\rho), \mathtt{f})\rangle \mid \\
& \qquad \rho = \langle c, s_0\rangle \xrightarrow{\lambda_0} \langle c_1, s_1\rangle \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_{k-1}} \langle c_k, s_k\rangle \mathsf{term} \text{ is channel-fair mod } (F, \emptyset)\} \\
\cup \{ & \langle \mathsf{trace}(\rho), (F, \emptyset, \mathsf{En}(\rho), \mathtt{p})\rangle \mid F \supseteq \mathsf{inits}(c_k, s_k) \ \& \\
& \qquad \rho = \langle c, s_0\rangle \xrightarrow{\lambda_0} \langle c_1, s_1\rangle \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_{k-1}} \langle c_k, s_k\rangle \ \& \ \neg\langle c_k, s_k\rangle \mathsf{term}\} \\
\cup \{ & \langle \mathsf{trace}(\rho), (F, U, \mathsf{En}(\rho), \mathtt{i})\rangle \mid \\
& \qquad \rho = \langle c, s_0\rangle \xrightarrow{\lambda_0} \langle c_1, s_1\rangle \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_k} \cdots \text{ is channel-fair mod } (F, U)\}.
\end{aligned}$$

5.4 Channel-Fair Trace Semantics

To give a denotational characterization of the semantic function $\mathcal{T}_{ch}$, we follow the approach taken in Section 3.3: for each language construct, we introduce an operation on trace sets that reflects the construct's operational behavior. Because the semantic operators reflect operational behavior, they retain the flavor of the operators introduced on strongly fair trace sets. In fact, the manipulation of the simple-trace components and the fairness sets F remains the same. As a result, the explanations of the semantic operators that follow focus on the new aspects of channel-fair traces, namely the sequences of enabling sets and the sets of insufficiently used channels.

We begin with a semantic function $\mathcal{T}_{ch} : \mathsf{BExp} \to \mathcal{P}(\Phi_{ch})$ such that, for each boolean expression b,

$$\begin{aligned}
\mathcal{T}_{ch}[\![b]\!] &= \{\langle (s, \varepsilon, s), (F, \emptyset, \langle \emptyset, \emptyset\rangle, \mathtt{f})\rangle \mid (s, \mathtt{tt}) \in \mathcal{B}[\![b]\!] \ \& \ F \in \mathcal{P}_{\text{fin}}(\Delta)\} \\
&\cup \ \{\langle \varepsilon_s, (F, \emptyset, \langle \emptyset\rangle, \mathtt{p})\rangle \mid (s, \mathtt{tt}) \in \mathcal{B}[\![b]\!] \ \& \ F \supseteq \{\varepsilon\}\}.
\end{aligned}$$

Intuitively, each finite trace $\langle (s, \varepsilon, s), (F, \emptyset, \langle \emptyset, \emptyset\rangle, \mathtt{f})\rangle$ in $\mathcal{T}_w[\![b]\!]$ represents a transition made in the evaluation of the boolean expression b, either to unroll a loop or to select the appropriate component of a conditional. Such a step (taken in isolation) is fair mod F and has no communications enabled along it. Similarly, the partial trace $\langle \varepsilon_s, (F, \emptyset, \langle \emptyset\rangle, \mathtt{p})\rangle$ indicates that, from any initial state s satisfying b, there is exactly one type of transition possible, and it involves an internal action.

Based on the operational characterization of $\mathcal{T}_{ch}$, it is easy to see that

$$\begin{aligned}
\mathcal{T}_{ch}[\![\mathsf{skip}]\!] = \{ & \langle (s, \varepsilon, s), (F, \emptyset, \langle \emptyset, \emptyset\rangle, \mathtt{f})\rangle \mid s \in S \ \& \ F \in \mathcal{P}_{\text{fin}}(\Delta)\} \\
\cup \ \{ & \langle \varepsilon_s, (F, \emptyset, \langle \emptyset\rangle, \mathtt{p})\rangle \mid s \in S \ \& \ F \supseteq \{\varepsilon\}\}
\end{aligned}$$

and

$$\begin{aligned}\mathcal{T}_{ch}[\![i{:=}e]\!] = \{&\langle(s,\varepsilon,[s|i=n]),(F,\emptyset,\langle\emptyset,\emptyset\rangle,\mathtt{f})\rangle \mid i\in\mathsf{dom}(s)\ \&\ F\in\mathcal{P}_{\text{fin}}(\Delta)\ \&\ (s,n)\in\mathcal{E}[\![e]\!]\}\\ &\cup\{\langle\varepsilon_s,(F,\emptyset,\langle\emptyset\rangle,\mathtt{p})\rangle \mid \mathsf{fv}[\![i{:=}e]\!]\subseteq\mathsf{dom}(s)\ \&\ F\supseteq\{\varepsilon\}\}.\end{aligned}$$

Because neither skip nor assignment enables communication along any channels, none of their traces include any insufficiently used channels.

For guards, we obtain the following semantic definitions:

$$\begin{aligned}\mathcal{T}_{ch}[\![h?i]\!] &= \{\langle(s,h?n,[s|i=n]),(F,\emptyset,\langle\{h?\},\emptyset\rangle,\mathtt{f})\rangle \mid i\in\mathsf{dom}(s)\ \&\ n\in\mathbb{Z}\ \&\ F\in\mathcal{P}_{\text{fin}}(\Delta)\}\\ &\cup\ \{\langle\varepsilon_s,(F,\emptyset,\{h?\},\mathtt{p})\rangle \mid i\in\mathsf{dom}(s)\ \&\ F\supseteq\{h?\}\},\\ \mathcal{T}_{ch}[\![h!e]\!] &= \{\langle(s,h!n,s),(F,\emptyset,\langle\{h!\},\emptyset\rangle,\mathtt{f})\rangle \mid (s,n)\in\mathcal{E}[\![e]\!]\ \&\ F\in\mathcal{P}_{\text{fin}}(\Delta)\}\\ &\cup\ \{\langle\varepsilon_s,(F,\emptyset,\{h!\},\mathtt{p})\rangle \mid \mathsf{fv}[\![e]\!]\subseteq\mathsf{dom}(s)\ \&\ F\supseteq\{h!\}\}.\end{aligned}$$

The successful computations of $h?i$ and $h!e$ necessarily use channel h, the only channel on which communication is enabled. As a result, their traces do not include any insufficiently used channels.

Sequential composition

The composability criterion for channel-fair traces is the same as that for strongly fair traces: φ_1 and φ_2 are composable whenever φ_1 is an infinite or partial trace, or when φ_1 is a finite trace and the initial state of φ_2 is the final state of φ_1. When φ_1 is an infinite or partial trace, the concatenation $\varphi_1\varphi_2$ is simply the trace φ_1. When φ_1 is a finite trace, the concatenation $\varphi_1\varphi_2$ must account accurately for the sequences of enabling sets as well as for the unused channels of the resulting trace. We discuss these concerns in turn.

A finite trace $\varphi_1 = \langle\alpha,(F_1,U_1,\mathcal{E}_1,\mathtt{f})\rangle$ represents a successfully terminating computation ρ_1 of some command c_1. However, when ρ_1 is used to generate a computation of the command $(c_1;c_2)$, the final configuration of ρ_1 is skipped: c_1's final action instead leads to the initial configuration of a computation of c_2. Likewise, in combining the finite trace φ_1 with a trace $\varphi_2 = \langle\beta,(F_2,U_2,\mathcal{E}_2,R_2)\rangle$, the final element of $\mathcal{E}_1$ should not appear in the resulting trace's sequence of enabling sets. Therefore, for sequences $\mathcal{E}_1$ and $\mathcal{E}_2$, we let $\mathcal{E}_1\mathcal{E}_2$ indicate the standard notion of sequence concatenation, and we define $\mathcal{E}_1\cdot\mathcal{E}_2$ to be the sequence that looks like $\mathcal{E}_1$ (with its final element removed), followed by $\mathcal{E}_2$. For example, if $\mathcal{E}_1 = \langle A_0,A_1,\ldots,A_{k-1},A_k\rangle$ and $\mathcal{E}_2 = \langle B_0,B_1,\ldots,B_n\rangle$, then $\mathcal{E}_1\mathcal{E}_2$ and $\mathcal{E}_1\cdot\mathcal{E}_2$ are defined as follows:

$$\begin{aligned}\mathcal{E}_1\mathcal{E}_2 &= \langle A_0,A_1,\ldots,A_{k-1},A_k,B_0,B_1,\ldots,B_n\rangle,\\ \mathcal{E}_1\cdot\mathcal{E}_2 &= \langle A_0,A_1,\ldots,A_{k-1},B_0,B_1,\ldots,B_n\rangle.\end{aligned}$$

The sequence $\mathcal{E}_1 \cdot \mathcal{E}_2$ accurately represents the sequence of enabling sets encountered along (the computation represented by) the trace $\varphi_1\varphi_2$.

The definition of concatenation must also account properly for the insufficiently used channels of the resulting trace. When combining the finite trace $\varphi_1 = \langle \alpha, (F_1, U_1, \mathcal{E}_1, \mathtt{f})\rangle$ with a partial or infinite trace $\varphi_2 = \langle \beta, (F_2, U_2, \mathcal{E}_2, R_2)\rangle$, the set U_2 adequately represents the channel constraints of the resulting trace: $\varphi_1\varphi_2$ is either partial (in which case the set $U_2 = \emptyset$ is appropriate) or infinite (in which case φ_1's finitely enabled channels are irrelevant). However, the case where both traces are finite requires more care: a trace's set of unused channels is defined relative to the directions enabled along the trace, and one trace may use some of the other's unused channels. The enabled but unused channels of $\varphi_1\varphi_2$ are those channels in U_1 that are not used along φ_2, combined with those channels in U_2 that are not used along φ_1. For each trace φ_i, the *used* channels of φ_i are those channels that appear along the sequence $\mathcal{E}_i$ but not in the set U_i. Given an enabling set E, we let $\mathsf{chans}(E)$ be the set of channels with directions in E:

$$\mathsf{chans}(E) = \{h \mid \exists d \in E.\mathsf{chan}(d) = h\}.$$

Likewise, $\mathsf{chans}(\mathcal{E})$ is the set of channels with directions occurring along the sequence $\mathcal{E}$: a channel h is in $\mathsf{chans}(\mathcal{E})$ if there is a set E occurring along $\mathcal{E}$ such that h is in $\mathsf{chans}(E)$. It follows that the used channels of the finite trace φ_i can be given by the set $\mathsf{chans}(\mathcal{E}_i) - U_i$, and the *unused* channels of the trace $\varphi_1\varphi_2$ can be given by the set

$$(U_1 - (\mathsf{chans}(\mathcal{E}_2) - U_2)) \cup (U_2 - (\mathsf{chans}(\mathcal{E}_1) - U_1)).$$

We therefore define concatenation on the finite trace $\varphi_1 = \langle \alpha, (F_1, U_1, \mathcal{E}_1, \mathtt{f})\rangle$ and the finite, partial, or infinite trace $\varphi_2 = \langle \beta, (F_2, U_2, \mathcal{E}_2, R_2)\rangle$ by

$$\varphi_1\varphi_2 = \langle \alpha\beta, (F_2, U, \mathcal{E}_1 \cdot \mathcal{E}_2, R_2\rangle,$$

where the set U of insufficiently used channels is in turn defined as

$$U = \begin{cases} (U_1 - (\mathsf{chans}(\mathcal{E}_2) - U_2)) \cup (U_2 - (\mathsf{chans}(\mathcal{E}_1) - U_1)), & \text{if } R_2 = \mathtt{f}, \\ U_2, & \text{if } R_2 \in \{\mathtt{i}, \mathtt{p}\}. \end{cases}$$

As before, we define sequential composition on trace sets T_1 and T_2 by

$$T_1; T_2 = \{\varphi_1\varphi_2 \mid \varphi_1 \in T_1 \;\&\; \varphi_2 \in T_2 \;\&\; \mathit{composable}(\varphi_1, \varphi_2)\},$$

and thus we can define

$$\begin{aligned} \mathcal{T}_{ch}[\![c_1; c_2]\!] &= \mathcal{T}_{ch}[\![c_1]\!]; \mathcal{T}_{ch}[\![c_2]\!], \\ \mathcal{T}_{ch}[\![g \to c]\!] &= \mathcal{T}_{ch}[\![g]\!]; \mathcal{T}_{ch}[\![c]\!], \end{aligned}$$

and

$$\mathcal{T}_{ch}[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] = \mathcal{T}_{ch}[\![b]\!]; \mathcal{T}_{ch}[\![c_1]\!] \cup \mathcal{T}_{ch}[\![\neg b]\!]; \mathcal{T}_{ch}[\![c_2]\!].$$

Iteration

Let $\langle \varphi_i \rangle_{i=0}^{\infty}$ be an infinite sequence of channel-fair traces such that, for each $i \geq 0$, $\varphi_i = \langle \alpha_i, (F_i, U_i, \mathcal{E}_i, R_i) \rangle$. The sequence $\langle \varphi_i \rangle_{i=0}^{\infty}$ is composable if the set $\biguplus_{i=0}^{\infty} F_i$ is finite and (for each i) the traces $\varphi_0 \varphi_1 \ldots \varphi_{i-1}$ and φ_i are composable.

When each φ_i is finite, the insufficiently used channels of the infinite concatenation are those channels that appear in infinitely many sets U_i and in only finitely many sets $\mathsf{chans}(\mathcal{E}_i) - U_i$. Thus we define the infinite concatenation of the infinite sequence $\langle \varphi_i \rangle_{i=0}^{\infty}$ of finite traces to be

$$\varphi_0 \varphi_1 \varphi_2 \ldots = \langle \alpha_0 \alpha_1 \ldots \alpha_n \ldots, (\biguplus_{i=0}^{\infty} F_i, \biguplus_{i=0}^{\infty} U_i \; - \biguplus_{i=0}^{\infty} (\mathsf{chans}(\mathcal{E}_i) - U_i), \mathcal{E}_0 \cdot \mathcal{E}_1 \cdot \mathcal{E}_2 \cdot \ldots, \mathtt{i}) \rangle,$$

where $\mathcal{E}_0 \cdot \mathcal{E}_1 \cdot \mathcal{E}_2 \cdot \ldots$ is the obvious extension of the operation $\mathcal{E}_1 \cdot \mathcal{E}_2$ to the infinite series of finite sequences $\mathcal{E}_i$.

When at least one of the traces φ_i is partial or infinite, then the first such φ_i provides the relevant contextual information for the resulting trace; thus, if φ_k is the first nonfinite trace, then the infinite concatenation of the sequence $\langle \varphi_i \rangle_{i=0}^{\infty}$ is

$$\varphi_0 \varphi_1 \varphi_2 \ldots = \langle \alpha_0 \alpha_1 \ldots \alpha_k, (F_k, U_k, \mathcal{E}_0 \cdot \mathcal{E}_1 \cdot \mathcal{E}_2 \cdot \ldots \cdot \mathcal{E}_k, R_k) \rangle.$$

Finite iteration on the trace set T is again defined by

$$T^* = \bigcup_{i=0}^{\infty} T^i,$$

where we define $T^0 = \{ \langle \varepsilon_s, (\emptyset, \emptyset, \langle \emptyset \rangle, \mathtt{f}) \rangle \mid s \in S \}$ and $T^{n+1} = T^n; T$. Infinite iteration on the trace set T is defined as follows:

$$T^{\omega} = \{ \varphi_0 \varphi_1 \ldots \varphi_k \ldots \mid \; (\forall i \geq 0. \varphi_i \in T) \;\&\; \mathit{composable}(\langle \varphi_i \rangle_{i=0}^{\infty}) \}.$$

Using these definitions, we give the following semantics for loops:

$$\mathcal{T}_{ch}[\![\mathsf{while}\; b \;\mathsf{do}\; c]\!] = (\mathcal{T}_{ch}[\![b]\!]; \mathcal{T}_{ch}[\![c]\!])^{\omega} \cup (\mathcal{T}_{ch}[\![b]\!]; \mathcal{T}_{ch}[\![c]\!])^*; \mathcal{T}_{ch}[\![\neg b]\!].$$

Guarded choice

Every computation ρ of gc_1 or gc_2 induces a computation of $gc_1 \,\square\, gc_2$ that looks like ρ, with the following exception: the actions enabled in its initial configuration are those actions enabled by either component. Intuitively, every channel-fair trace φ of gc_1 or gc_2 should likewise induce a channel-fair trace of $gc_1 \Box gc_2$ that looks like φ, with the following related exception: its initial enabling set should contain those directions initially enabled by either component. For example, if $\varphi_1 = \langle \varepsilon_s \alpha, (F_1, U_1, \langle E_1 \rangle \mathcal{E}_1, R_1) \rangle$ represents a computation of the chosen component and $\varphi_2 = \langle \varepsilon_s, (F_2, \emptyset, \langle E_2 \rangle, \mathtt{p}) \rangle$ is an initial partial trace of the unchosen component, then there is a computation ρ of $gc_1 \Box gc_2$ whose representative trace has simple-trace component α and initial enabling set $E_1 \cup E_2$. When φ_1 is finite, we may also need to update the resulting trace's set of unused channels. The insufficiently used channels of the finite computation ρ are those channels used insufficiently along φ_1 (i.e., U_1) plus those channels in E_2 that are not enabled along φ_1 (i.e., $E_2 - \mathsf{chans}(\langle E_1 \rangle \mathcal{E}_1)$).

We therefore define guarded choice on channel-fair trace sets as follows:

$$
\begin{array}{rcl}
T_1 \Box T_2 & = & \{ \langle \alpha, (F_1, U_1, \langle E_1 \cup E_2 \rangle \mathcal{E}_1, R_1) \rangle \mid R_1 \in \{\mathtt{p}, \mathtt{i}\} \\
& & \qquad \& \ \langle \varepsilon_s \alpha, (F_1, U_1, \langle E_1 \rangle \mathcal{E}_1, R_1) \rangle \in T_1 \ \& \ \langle \varepsilon_s, (F_2, \emptyset, \langle E_2 \rangle, \mathtt{p}) \rangle \in T_2 \} \\
& \cup & \{ \langle \alpha, (F_2, U_2, \langle E_1 \cup E_2 \rangle \mathcal{E}_2, R_2) \rangle \mid R_2 \in \{\mathtt{p}, \mathtt{i}\} \\
& & \qquad \& \ \langle \varepsilon_s \alpha, (F_2, U_2, \langle E_2 \rangle \mathcal{E}_2, R_2) \rangle \in T_2 \ \& \ \langle \varepsilon_s, (F_1, \emptyset, \langle E_1 \rangle, \mathtt{p}) \rangle \in T_1 \} \\
& \cup & \{ \langle \alpha, (F_1,\ U_1 \cup (\mathsf{chans}(E_2) - \mathsf{chans}(\langle E_1 \rangle \mathcal{E}_1)),\ \langle E_1 \cup E_2 \rangle \mathcal{E}_1,\ \mathtt{f}) \rangle \mid \\
& & \qquad \langle \varepsilon_s \alpha, (F_1, U_1, \langle E_1 \rangle \mathcal{E}_1, \mathtt{f}) \rangle \in T_1 \ \& \ \langle \varepsilon_s, (F_2, \emptyset, \langle E_2 \rangle, \mathtt{p}) \rangle \in T_2 \} \\
& \cup & \{ \langle \alpha, (F_2,\ U_2 \cup (\mathsf{chans}(E_1) - \mathsf{chans}(\langle E_2 \rangle \mathcal{E}_2)),\ \langle E_1 \cup E_2 \rangle \mathcal{E}_2,\ \mathtt{f}) \rangle \mid \\
& & \qquad \langle \varepsilon_s \alpha, (F_2, U_2, \langle E_2 \rangle \mathcal{E}_2, \mathtt{f}) \rangle \in T_2 \ \& \ \langle \varepsilon_s, (F_1, \emptyset, \langle E_1 \rangle, \mathtt{p}) \rangle \in T_1 \}.
\end{array}
$$

Unlike the definition of guarded choice for strongly fair trace sets, this definition needs to account accurately for the initial enabling sets and the sets of unused channels. However, the underlying essence of the operation remains the same. We define

$$\mathcal{T}_{ch}[\![gc_1 \Box gc_2]\!] = \mathcal{T}_{ch}[\![gc_1]\!] \Box \mathcal{T}_w[\![gc_2]\!].$$

Channel restriction

For process-fair trace sets, the trace set $T \backslash h$ is constructed from the set T by discarding those traces that use channel h visibly, and then removing $h!$ and $h?$ from the enabling and fairness sets of the traces that remain. We can define a similar operation on channel-fair trace sets, but this operation must also verify that the fairness constraints on channel h are satisfied. In particular, we should discard any infinite trace $\varphi = \langle \alpha, (F, U, \mathcal{E}, \mathtt{i}) \rangle$ that has the channel h both in its set U of insufficiently used channels and in infinitely many of the sets along its sequence $\mathcal{E}$ of enabling sets: such traces correspond to non–channel-fair computations that use channel h only finitely often despite having synchronization on h enabled infinitely often. For a sequence $\mathcal{E}$ of enabling sets, we let $\Cup\mathcal{E}$ be the set of directions that appear in infinitely many of the sets along $\mathcal{E}$, and we discard any trace for which $h \in U$ and $h \in \Cup\mathcal{E}$.

For any enabling set E, $E\backslash h$ is the set that results from removing all references to channel h: $E\backslash h = E - \{h!, h?, h\}$. This operation extends to sequences of sets in the obvious way: for example, $\langle E_0, E_1, E_2, \ldots, E_k\rangle\backslash h = \langle E_0\backslash h, E_1\backslash h, E_2\backslash h, \ldots, E_k\backslash h\rangle$. Given a trace set T and a channel h, we then define the channel restriction of h on T by

$$\begin{array}{rcl} T\backslash h & = & \{\langle \alpha, (F', U\backslash h, \mathcal{E}\backslash h, \mathtt{f})\rangle \mid \langle \alpha, (F, U, \mathcal{E}, \mathtt{f})\rangle \in T \ \&\ F' \supseteq F\backslash h \ \&\ h \notin \mathsf{chans}(\alpha)\} \\ & \cup & \{\langle \alpha, (F', \emptyset, \mathcal{E}\backslash h, \mathtt{p})\rangle \mid \langle \alpha, (F, \emptyset, \mathcal{E}, \mathtt{p})\rangle \in T \ \&\ F' \supseteq F\backslash h \ \&\ h \notin \mathsf{chans}(\alpha)\} \\ & \cup & \{\langle \alpha, (F', U\backslash h, \mathcal{E}\backslash h, \mathtt{i})\rangle \mid \langle \alpha, (F, U, \mathcal{E}, \mathtt{i})\rangle \in T \\ & & \qquad \&\ (h \in U \implies h \notin \Cup\mathcal{E}) \ \&\ F' \supseteq F\backslash h \ \&\ h \notin \mathsf{chans}(\alpha)\}, \end{array}$$

and we define $\mathcal{T}_{ch}[\![c\backslash h]\!] = \mathcal{T}_{ch}[\![c]\!]\backslash h$.

Parallel composition

Parameterized channel fairness relies on two types of fairness constraints: *process constraints*, which place limits on which computations can be combined through parallel composition, and *channel constraints*, which place limits on which computations can be "restricted" (in the sense of channel restriction). Because only the process constraints affect which computations can be combined meaningfully, the mergeability requirements (and the *mergeable* predicate) remain the same as for process fairness, modulo the need to extract the set of infinitely enabled directions from the sequence of enabling sets. For channel-fair traces $\varphi_1 = \langle \alpha_1, (F_1, U_1, \mathcal{E}_1, R_1)\rangle$ and $\varphi_2 = \langle \alpha_2, (F_2, U_2, \mathcal{E}_2, R_2)\rangle$, we define the predicate $mergeable(\varphi_1, \varphi_2)$ as follows:

$$\begin{array}{l} mergeable(\varphi_1, \varphi_2) \iff (R_1 = \mathtt{f}) \text{ or } (R_2 = \mathtt{f}) \text{ or } (R_1 = R_2 = \mathtt{p}) \text{ or } (\varepsilon \notin F_1 \cup F_2 \\ \qquad \&\ \neg\mathsf{match}(F_1, \Cup\mathcal{E}_2) \ \&\ \neg\mathsf{match}(F_2, \Cup\mathcal{E}_1) \ \&\ F_1 \cap \mathsf{vis}(\alpha_2) = \emptyset \ \&\ F_2 \cap \mathsf{vis}(\alpha_1) = \emptyset). \end{array}$$

This predicate makes no mention of the sets U_1 and U_2: channel constraints are orthogonal to the issue of mergeability.

To define a *fairmerge* operation on channel-fair traces, we employ an approach similar to that taken in Section 3.3, defining new sets *both* and *one* that account for traces' unused channels and sequences of enabling sets. However, before constructing these sets, we need to define several auxiliary operations.

We begin by introducing a concatenation-like operator $\cdot$ on channel-fair traces that allows us to combine traces that represent *segments* of computations (rather than complete computations) while maintaining accurate enabling information. The idea is that, given two traces φ_1 and φ_2, their fair merges are defined by interleaving and synchronizing finite portions of each (at least until one or both traces "run out") and then combining all of the partial results. To record the sequences of enabling sets accurately, we need a way to split each φ_i into the appropriate finite portions without losing any of the relevant enabling information. For example, consider the transition sequence

$$\rho = \langle c, s_0\rangle \xrightarrow{\lambda_0} \langle c_1, s_1\rangle \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_{n-1}} \langle c_n, s_n\rangle,$$

which can be separated into the following two transition sequences (for any k):

$$\begin{aligned}\rho_1 &= \langle c, s_0\rangle \xrightarrow{\lambda_0} \langle c_1, s_1\rangle \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_{k-1}} \langle c_k, s_k\rangle,\\ \rho_2 &= \langle c_k, s_k\rangle \xrightarrow{\lambda_k} \langle c_{k+1}, s_{k+1}\rangle \xrightarrow{\lambda_{k+1}} \cdots \xrightarrow{\lambda_{n-1}} \langle c_n, s_n\rangle.\end{aligned}$$

For each ρ, there are as many such decompositions as there are configurations occurring along ρ; in each case, the final configuration of ρ_1 is the initial configuration of ρ_2. If we let channel-fair traces φ_1 and φ_2 represent ρ_1 and ρ_2, respectively—ignoring for the moment that ρ_1 is not a successfully terminating computation—then we should be able to define an operation $\varphi_1 \cdot \varphi_2$ that represents the computation ρ. The trace $\varphi_1 \cdot \varphi_2$ needs to be defined only when φ_1 is a finite trace with form $\langle \alpha_1, (F_1, \mathcal{E}_1\langle X\rangle, \mathtt{f})\rangle$ and φ_2 has form $\langle \alpha_2, (F_2, \langle X\rangle\mathcal{E}_2, R)\rangle$—that is, when the final enabling set of φ_1 is the first enabling set of φ_2. In such cases, we define

$$\varphi_1 \cdot \varphi_2 = \langle \alpha_1\alpha_2, (F_2, \mathcal{E}_1\langle X\rangle\mathcal{E}_2, R)\rangle,$$

which indeed is a valid representation for the computation ρ. (Note that, when defined, the trace $\varphi_1 \cdot \varphi_2$ is precisely the more general concatenation $\varphi_1\varphi_2$. We can also extend this operation to infinite sequences of traces in the obvious way, basing it on infinite concatenation.) We then extend this operation to sets of triples of traces in the obvious way: for sets Y_1 and Y_2,

$$\begin{aligned}Y_1 \cdot Y_2 = \{(\varphi_1\cdot\varphi_1', \varphi_2\cdot\varphi_2', \varphi_3\cdot\varphi_3') \mid (\varphi_1,\varphi_2,\varphi_3) \in Y_1 \;\&\; (\varphi_1',\varphi_2',\varphi_3') \in Y_2 \\ \&\; \varphi_1\cdot\varphi_1,\;\; \varphi_2\cdot\varphi_2',\;\; \varphi_3\cdot\varphi_3' \text{ are all defined}\}.\end{aligned}$$

We also define the obvious iterative extensions to the dot operator. For a set Y of triples of traces, the finite iteration of Y is defined by

$$Y^{*\bullet} = \bigcup_{i=0}^{\infty} Y^i,$$

where $Y^0 = \{(\varphi,\varphi,\varphi) \mid \exists s \in S, X \in \mathcal{P}_{\mathrm{fin}}(\Delta).\varphi = \langle \varepsilon_s, (\emptyset,\emptyset,\langle X\rangle,\mathtt{f})\rangle\}$ and $Y^{n+1} = Y^n \cdot Y$. The infinite iteration of Y is defined by

$$\begin{aligned}Y^{\omega\bullet} = \{(\varphi_0\cdot\varphi_1\cdot\ \ldots\ \cdot\varphi_k\ \cdot\ldots, \varphi_0'\cdot\varphi_1'\cdot\ \ldots\ \cdot\varphi_k'\ \cdot\ldots, \varphi_0''\cdot\varphi_1''\cdot\ \ldots\ \cdot\varphi_k''\ \cdot\ldots) \mid \\ \forall i \geq 0.\ (\varphi_i,\varphi_i',\varphi_i'') \in Y \;\&\; \varphi_i\cdot\varphi_{i+1},\;\; \varphi_i'\cdot\varphi_{i+1}',\;\; \varphi_i''\cdot\varphi_{i+1}'' \text{ are all defined}\}.\end{aligned}$$

We again make use of the interleaving ($\alpha \,\rfloor\!\lfloor\, \beta$) and merging ($\alpha \| \beta$) operators on simple traces, and we introduce corresponding operators on sequences of enabling sets. For a (finite or infinite) sequence $\mathcal{E}$ and the enabling set E, $\mathcal{E} \,\rfloor\!\lfloor\, E$ is the sequence $\mathcal{E}$ with the set E propagated: for example,

$$\langle E_0, E_1, E_2, \ldots, E_k\rangle \,\rfloor\!\lfloor\, E = \langle E_0\|E,\ E_1\|E,\ E_2\|E, \ldots, E_k\|E\rangle.$$

For finite sequences $\mathcal{E}_1 = \langle A_0, A_1, \ldots, A_k \rangle$ and $\mathcal{E}_2 = \langle B_0, B_1, \ldots, B_n \rangle$, the sequence $\mathcal{E}_1 \rfloor\!\rfloor \mathcal{E}_2$ is the sequence $\mathcal{E}_1$ (with the set B_0 propagated) followed by the sequence $\mathcal{E}_2$ (with the set A_k propagated). That is,

$$\mathcal{E}_1 \rfloor\!\rfloor \mathcal{E}_2 = (\mathcal{E}_1 \rfloor\!\rfloor B_0) \cdot (\mathcal{E}_2 \rfloor\!\rfloor A_k) = \langle A_0 \| B_0,\ A_1 \| B_0,\ \ldots,\ A_{k-1} \| B_0,\ A_k \| B_0,\ A_k \| B_1,\ \ldots,\ A_k \| B_n \rangle.$$

Intuitively, if $\mathcal{E}_1$ is the sequence of enabling sets occurring along the transition sequence represented by α and $\mathcal{E}_2$ is the sequence of enabling sets for β, then $\mathcal{E}_1 \rfloor\!\rfloor \mathcal{E}_2$ is the sequence of enabling sets that occurs along the transition sequence represented by $\alpha \rfloor\!\rfloor \beta$. This definition requires knowing which directions are enabled in the final configuration of a transition sequence (and, indeed, it is precisely for this reason that we include the final enabling set in the channel-fair traces). After one component performs its transitions α, the directions enabled in its final configuration remain enabled as the other component makes its transitions β. For example, consider the transition sequences

$$\rho_1 = \langle \mathsf{a!0} \to \mathsf{b!0} \to \mathsf{a?x}, s_1 \rangle \xrightarrow{\mathsf{a!0}} \langle \mathsf{b!0} \to \mathsf{a?x}, s_1 \rangle \xrightarrow{\mathsf{b!0}} \langle \mathsf{a?x}, s_1 \rangle$$

and

$$\rho_2 = \langle \mathsf{a!1} \to \mathsf{b!1}, s_2 \rangle \xrightarrow{\mathsf{a!1}} \langle \mathsf{b!1}, s_2 \rangle,$$

which can be interleaved to yield the following transition sequence:

$$\begin{aligned} \rho = \langle (\mathsf{a!0} \to \mathsf{b!0} \to \mathsf{a?x}) \parallel (\mathsf{a!1} \to \mathsf{b!1}), s \rangle &\xrightarrow{\mathsf{a!0}} \langle (\mathsf{b!0} \to \mathsf{a?x}) \parallel (\mathsf{a!1} \to \mathsf{b!1}), s \rangle \\ &\xrightarrow{\mathsf{b!0}} \langle \mathsf{a?x} \parallel (\mathsf{a!1} \to \mathsf{b!1}), s \rangle \\ &\xrightarrow{\mathsf{a!1}} \langle \mathsf{a?x} \parallel \mathsf{b!1}, s \rangle. \end{aligned}$$

Just before the right component makes its $\mathsf{a!1}$ transition, the direction $\mathsf{a?}$—which is enabled in the final configuration of ρ_1—is also enabled for the parallel command. The transition sequences ρ_1 and ρ_2 can be represented by the channel-fair traces

$$\varphi_1 = \langle (s_1, \mathsf{a!0}, s_1)(s_1, \mathsf{b!0}, s_1), (\emptyset, \emptyset, \langle \{\mathsf{a!}\}, \{\mathsf{b!}\}, \{\mathsf{a?}\} \rangle, \mathtt{f}) \rangle$$

and

$$\varphi_2 = \langle (s_2, \mathsf{a!1}, s_2), (\emptyset, \emptyset, \langle \{\mathsf{a!}\}, \{\mathsf{b!}\} \rangle, \mathtt{f}) \rangle.$$

The sequence of enabled sets along ρ can therefore be defined by

$$\langle \{\mathsf{a!}\}, \{\mathsf{b!}\}, \{\mathsf{a?}\} \rangle \rfloor\!\rfloor \langle \{\mathsf{a!}\}, \{\mathsf{b!}\} \rangle = \langle \{\mathsf{a!}\}, \{\mathsf{b!}, \mathsf{a!}\}, \{\mathsf{a?}, \mathsf{a!}, \mathsf{a}\}, \{\mathsf{a?}, \mathsf{b!}\} \rangle.$$

Finally, analogous to the definition of $\alpha \| \beta$ for matching simple traces α and β, we define the operation $\mathcal{E}_1 \| \mathcal{E}_2$ when $\mathcal{E}_1$ and $\mathcal{E}_2$ are sequences of enabled sets with equal length. That is, if $\mathcal{E}_1 = \langle A_0, A_1, \ldots, A_k \rangle$, $\mathcal{E}_2 = \langle B_0, B_1, \ldots, B_n \rangle$, and $k = n$, we define

$$\mathcal{E}_1 \| \mathcal{E}_2 = \langle A_0 \| B_0,\ A_1 \| B_1,\ \ldots,\ A_k \| B_k \rangle.$$

Intuitively, $\mathcal{E}_1 \| \mathcal{E}_2$ represents the sequence of enabling sets encountered along a transition sequence in which the two components of a parallel command repeatedly synchronize with one another.

With these operations in hand, we can define the related operations $\varphi_1 \rfloor\!\lfloor \varphi_2$ and $\varphi_1 \| \varphi_2$ on finite channel-fair traces. For finite traces $\varphi_1 = \langle \alpha_1, (F_1, U_1, \mathcal{E}_1, \mathtt{f})\rangle$ and $\varphi_2 = \langle \alpha_2, (F_2, U_2, \mathcal{E}_2, \mathtt{f})\rangle$ such that $\alpha_1 \rfloor\!\lfloor \alpha_2$ is defined, we define

$$\varphi_1 \rfloor\!\lfloor \varphi_2 = \{\langle \alpha_1 \rfloor\!\lfloor \alpha_2, (F, (U_1 - M_2) \cup (U_2 - M_1), \mathcal{E}_1 \rfloor\!\lfloor \mathcal{E}_2, \mathtt{f})\rangle \mid F \supseteq (F_1 \cup F_2)\},$$

where we again let $M_1 = \mathsf{chans}(\mathcal{E}_1) - U_1$ and $M_2 = \mathsf{chans}(\mathcal{E}_2) - U_2$ be the sets of used channels for φ_1 and φ_2. Intuitively, each trace $\varphi \in \varphi_1 \rfloor\!\lfloor \varphi_2$ represents a transition sequence of a parallel command in which one component performs actions corresponding to φ_1, followed by the other component performing actions corresponding to φ_2. Likewise, for matching finite traces $\varphi_1 = \langle \alpha_1, (F_1, U_1, \mathcal{E}_1, \mathtt{f})\rangle$ and $\varphi_2 = \langle \alpha_2, (F_2, U_2, \mathcal{E}_2, \mathtt{f})\rangle$, $\varphi_1 \| \varphi_2$ is the set of traces corresponding to their synchronization at each step:

$$\varphi_1 \| \varphi_2 = \{\langle \alpha_1 \| \alpha_2, (F, U_1 \cup U_2, \mathcal{E}_1 \| \mathcal{E}_2, \mathtt{f})\rangle \mid F \supseteq (F_1 \cup F_2)\}.$$

In the case of synchronization, the two traces necessarily use the same channels; as a result, the set of insufficiently used channels is simply $U_1 \cup U_2$. We can now define the set $both \subseteq \Phi_{ch} \times \Phi_{ch} \times \Phi_{ch}$, whose triples represent transition sequences made while both components are active, as follows:

$$\begin{array}{rcl} both & = & \{(\varphi_1, \varphi_2, \varphi), (\varphi_2, \varphi_1, \varphi) \mid \varphi_1 = \langle \alpha, (F_1, U_1, \mathcal{E}_1, \mathtt{f})\rangle \ \&\ \varphi_2 = \langle \beta, (F_2, U_2, \mathcal{E}_2, \mathtt{f})\rangle \ \& \\ & & \qquad \mathsf{disjoint}(\alpha, \beta) \ \&\ \varphi \in \varphi_1 \rfloor\!\lfloor \varphi_2\} \\ & \cup & \{(\varphi_1, \varphi_2, \varphi) \mid \varphi_1 = \langle \alpha, (F_1, U_1, \mathcal{E}_1, \mathtt{f})\rangle \ \&\ \varphi_2 = \langle \beta, (F_2, U_2, \mathcal{E}_2, \mathtt{f})\rangle \ \& \\ & & \qquad \mathsf{disjoint}(\alpha, \beta) \ \&\ \mathsf{match}(\alpha, \beta) \ \&\ \varphi \in \varphi_1 \| \varphi_2\}. \end{array}$$

Once one component of a parallel command has either terminated successfully or become permanently blocked, the remaining component may proceed uninterrupted. Of course, the remaining component may itself eventually terminate, or it may become blocked (modulo some set F), or it may proceed indefinitely. We extend the operator $\rfloor\!\lfloor$ on channel-fair traces to account for each of these cases as well. Suppose that we have a parallel command $c_1 \| c_2$ in which c_2 has terminated in its local state s; the future execution of c_2 can be represented by the empty trace $\varphi_2 = \langle \varepsilon_s, (F_2, \emptyset, \langle \emptyset \rangle, \mathtt{f})\rangle$, for any set F_2. If the future execution of c_1 is represented by the trace $\varphi_1 = \langle \alpha, (F_1, U_1, \mathcal{E}_1, R_1)\rangle$, then the parallel command's future execution can be represented by any of the fair traces in the set

$$\varphi_1 \rfloor\!\lfloor \varphi_2 = \{\langle \alpha \rfloor\!\lfloor \varepsilon_s, (F, U_1, \mathcal{E} \rfloor\!\lfloor \emptyset, R_1)\rangle \mid F \supseteq F_1 \cup F_2\}.$$

If, instead of terminating successfully, c_2 becomes blocked mod F_2 in local state s, then its future execution can be represented by a partial trace $\varphi_2 = \langle \varepsilon_s, (F_2, \emptyset, \langle E \rangle, \mathtt{p})\rangle$. In this case, again letting $\varphi_1 = \langle \alpha, (F_1, U_1, \mathcal{E}_1, R_1)\rangle$ represent the future execution of c_1, the future execution of the parallel command can be defined as follows:

$$\varphi_1 \rfloor\!\rfloor \varphi_2 = \begin{cases} \{\langle \alpha \rfloor\!\rfloor \varepsilon_s, (F, U_1, \mathcal{E}_1 \rfloor\!\rfloor E, \mathtt{p})\rangle \mid F \supseteq F_1 \cup F_2\}, & \text{if } R_1 \in \{\mathtt{f}, \mathtt{p}\}, \\ \{(\alpha \rfloor\!\rfloor \varepsilon_s, (F, U_1, \mathcal{E}_1 \rfloor\!\rfloor E, \mathtt{i})) \mid F \supseteq (F_1 \cup F_2)\}, & \text{if } R_1 = \mathtt{i}. \end{cases}$$

We therefore define the set $one \subseteq \Phi_{ch} \times \Phi_{ch} \times \Phi_{ch}$, whose triples reflect transition sequences made when only one component remains active, as follows:

$$\begin{aligned} one \quad = \quad & \{(\varphi_1, \varphi_2, \varphi), (\varphi_2, \varphi_1, \varphi) \mid \varphi_1 = \langle \alpha, (F_1, U_1, \mathcal{E}_1, R)\rangle \ \&\ \varphi_2 = \langle \varepsilon_s, (F_2, U_2, \mathcal{E}_2, R)\rangle \ \& \\ & \qquad \mathsf{disjoint}(\alpha, s) \ \&\ \varphi \in \varphi_1 \rfloor\!\rfloor \varphi_2\}. \end{aligned}$$

We then define

$$fairmerge = both^{\omega\bullet} \cup both^{*\bullet} \cdot one.$$

The triple $(\varphi, \varphi', \psi)$ is in $both^{\omega\bullet}$ if and only if the traces φ, φ', and ψ can be written as

$$\varphi = \varphi_0 \cdot \varphi_1 \cdot \varphi_2 \cdot \varphi_3 \cdot \ldots, \qquad \varphi' = \varphi'_0 \cdot \varphi'_1 \cdot \varphi'_2 \cdot \varphi'_3 \cdot \ldots, \qquad \psi = \psi_0 \cdot \psi_1 \cdot \psi_2 \cdot \psi_3 \cdot \ldots,$$

such that each φ_i, φ'_i and ψ_i is finite, and each ψ_i is in the set $(\varphi_i \rfloor\!\rfloor \varphi'_i \ \cup\ \varphi'_i \rfloor\!\rfloor \varphi_i \ \cup\ \varphi_i \| \varphi_i)$. Such triples represent the merging of two infinite traces. Likewise, the triple $(\varphi, \varphi', \psi)$ is in $both^{*\bullet} \cdot one$ if and only if the traces φ, φ', and ψ can be written as

$$\varphi = \varphi_0 \cdot \varphi_1 \cdot \varphi_2 \cdot \varphi_3 \cdot \ldots\ \varphi_n, \quad \varphi' = \varphi'_0 \cdot \varphi'_1 \cdot \varphi'_2 \cdot \varphi'_3 \cdot \ldots\ \varphi'_n, \quad \psi = \psi_0 \cdot \psi_1 \cdot \psi_2 \cdot \psi_3 \cdot \ldots \cdot \psi_n,$$

such that each φ_i, φ'_i and ψ_i (for $i < n$) is a nonempty finite trace, each ψ_i (for $i < n$) is a member of the set $(\varphi_i \rfloor\!\rfloor \varphi'_i \ \cup\ \varphi'_i \rfloor\!\rfloor \varphi_i \ \cup\ \varphi_i \| \varphi_i)$, at least one of φ_n and φ'_n has form $\langle \varepsilon_s, \theta\rangle$, and ψ_n is a member of the set $(\varphi_n \rfloor\!\rfloor \varphi'_n \ \cup\ \varphi'_n \rfloor\!\rfloor \varphi_n)$. These triples represent the merging of traces when at least one of them is finite or partial.

Finally, we define channel-fair parallel composition on trace sets as

$$T_1 \| T_2 = \{\varphi \mid \varphi_1 \in T_1 \ \&\ \varphi_2 \in T_2 \ \&\ mergeable(\varphi_1, \varphi_2) \ \&\ (\varphi_1, \varphi_2, \varphi) \in fairmerge\},$$

so that $\mathcal{T}_{ch}[\![c_1 \| c_2]\!] = \mathcal{T}_{ch}[\![c_1]\!] \| \mathcal{T}_{ch}[\![c_2]\!]$.

We summarize the preceding discussion by giving the following complete denotational characterization of the trace semantics $\mathcal{T}_{ch}$. This characterization of the semantics $\mathcal{T}_{ch}$ looks essentially the same as the denotational characterizations of the various strongly fair semantics introduced previously. The only obvious difference is the inclusion of the (trivial) sets of unused channels and the sequences of enabling sets for skip, assignment, and the input and output guards. The real differences in the semantics lie in the new interpretations of the various semantic operators, and these differences reflect only the more complicated bookkeeping necessary for modeling channel fairness.

Definition 5.4.1 The channel-fair trace semantic function $\mathcal{T}_{ch} : \mathsf{Com} \to \mathcal{P}(\Phi_{ch})$ is defined by:

$$
\begin{array}{rcl}
\mathcal{T}_{ch}[\![\mathsf{skip}]\!] & = & \{\langle (s,\varepsilon,s),(F,\emptyset,\langle\emptyset,\emptyset\rangle,\mathtt{f})\rangle \mid s\in S \;\&\; F\in\mathcal{P}_{\mathrm{fin}}(\Delta)\} \\
& \cup & \{\langle \varepsilon_s,(F,\emptyset,\langle\emptyset\rangle,\mathtt{p})\rangle \mid s\in S \;\&\; F\supseteq\{\varepsilon\}\} \\
\mathcal{T}_{ch}[\![i{:=}e]\!] & = & \{\langle (s,\varepsilon,[s|i=n]),(F,\emptyset,\langle\emptyset,\emptyset\rangle,\mathtt{f})\rangle \mid \\
& & \qquad i\in\mathsf{dom}(s) \;\&\; F\in\mathcal{P}_{\mathrm{fin}}(\Delta) \;\&\; (s,n)\in\mathcal{E}[\![e]\!]\} \\
& \cup & \{\langle \varepsilon_s,(F,\emptyset,\langle\emptyset\rangle,\mathtt{p})\rangle \mid \mathsf{dom}(s)\supseteq\{i\}\cup\mathsf{fv}[\![e]\!] \;\&\; F\supseteq\{\varepsilon\}\} \\
\mathcal{T}_{ch}[\![c_1;c_2]\!] & = & \mathcal{T}_{ch}[\![c_1]\!];\mathcal{T}_{ch}[\![c_2]\!], \\
\mathcal{T}_{ch}[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] & = & \mathcal{T}_{ch}[\![b]\!];\mathcal{T}_{ch}[\![c_1]\!]\cup\mathcal{T}_{ch}[\![\neg b]\!];\mathcal{T}_{ch}[\![c_2]\!] \\
\mathcal{T}_{ch}[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!] & = & (\mathcal{T}_{ch}[\![b]\!];\mathcal{T}_{ch}[\![c]\!])^{\omega}\cup(\mathcal{T}_{ch}[\![b]\!];\mathcal{T}_{ch}[\![c]\!])^{*};\mathcal{T}_{ch}[\![\neg b]\!] \\
\mathcal{T}_{ch}[\![h?i]\!] & = & \{\langle (s,h?n,[s|i=n]),(F,\emptyset,\langle\{h?\},\emptyset\rangle,\mathtt{f})\rangle \mid \\
& & \qquad i\in\mathsf{dom}(s) \;\&\; n\in\mathbb{Z} \;\&\; F\in\mathcal{P}_{\mathrm{fin}}(\Delta)\} \\
& \cup & \{\langle \varepsilon_s,(F,\emptyset,\langle\{h?\}\rangle,\mathtt{p})\rangle \mid i\in\mathsf{dom}(s) \;\&\; F\supseteq\{h?\}\} \\
\mathcal{T}_{ch}[\![h!e]\!] & = & \{\langle (s,h!n,s),(F,\emptyset,\langle\{h!\},\emptyset\rangle,\mathtt{f})\rangle \mid (s,n)\in\mathcal{E}[\![e]\!] \;\&\; F\in\mathcal{P}_{\mathrm{fin}}(\Delta)\} \\
& \cup & \{\langle \varepsilon_s,(F,\emptyset,\langle\{h!\}\rangle,\mathtt{p})\rangle \mid \mathsf{fv}[\![e]\!]\subseteq\mathsf{dom}(s) \;\&\; F\supseteq\{h!\}\} \\
\mathcal{T}_{ch}[\![g\to c]\!] & = & \mathcal{T}_{ch}[\![g]\!];\mathcal{T}_{ch}[\![c]\!], \\
\mathcal{T}_{ch}[\![gc_1\,\Box\,gc_2]\!] & = & \mathcal{T}_{ch}[\![gc_1]\!]\,\Box\,\mathcal{T}_{ch}[\![gc_2]\!] \\
\mathcal{T}_{ch}[\![c\backslash h]\!] & = & \mathcal{T}_{ch}[\![c]\!]\backslash h \\
\mathcal{T}_{ch}[\![c_1\|c_2]\!] & = & \mathcal{T}_{ch}[\![c_1]\!]\|\mathcal{T}_{ch}[\![c_2]\!].
\end{array}
$$

$\diamond$

The following two examples illustrate how the strongly channel-fair semantics $\mathcal{T}_{ch}$ can be used to reason about the channel-fair behavior of programs.

Example 5.4.2 Recall the following processes introduced in Figure 2.9, where we assumed that communication occurred only when all processes were inside their loops:

$$
\begin{array}{rcl}
P & \equiv & \mathsf{while}\ (\mathsf{x}\neq 0)\ \mathsf{do}\ (\mathsf{a!0}\to\mathsf{x{:=}0}\,\Box\,\mathsf{b!1}\to\mathsf{skip}), \\
Q & \equiv & \mathsf{n{:=}1};\ \mathsf{while}\ (\mathsf{w}\neq 0)\ \mathsf{do}\ (\mathsf{a?w}\to\mathsf{c!w}\,\Box\,\mathsf{c!n}\to\mathsf{n{:=}n+1}), \\
R & \equiv & \mathsf{while}\ (\mathsf{v}\neq 0)\ \mathsf{do}\ (\mathsf{c?v}\to\mathsf{skip}\,\Box\,\mathsf{b?v}\to\mathsf{skip}).
\end{array}
$$

Using the trace semantics $\mathcal{T}_{ch}$, we illustrate why this assumption was necessary for proving termination of the program $(P\|Q\|R)\backslash\mathsf{a}\backslash\mathsf{b}\backslash\mathsf{c}$ under strong channel fairness. In particular, we now show that the program cannot be guaranteed to terminate under strong channel fairness without this assumption.

$\mathcal{T}_{ch}[\![P]\!]$ contains an infinite trace of form

$$\langle\alpha,(\emptyset,\{\mathsf{a}\},\langle\emptyset,\{\mathsf{a!},\mathsf{b!}\},\emptyset\rangle^{\omega},\mathtt{i})\rangle,$$

where α involves only ε-transitions and output actions on channel b. Similarly, $\mathcal{T}_{ch}[\![Q]\!]$ contains an infinite trace of form

$$\langle \beta, (\emptyset, \{\mathtt{a}\}, \langle \emptyset \rangle \langle \emptyset, \{\mathtt{a!}, \mathtt{c!}\}, \emptyset \rangle^{\omega}, \mathtt{i}) \rangle,$$

where β involves only ε-transitions and output actions on channel c. As a result, $\mathcal{T}_{ch}[\![P \| Q]\!]$ has an infinite trace with form

$$\langle \gamma, (\emptyset, \{\mathtt{a}\}, \langle \emptyset \rangle \mathcal{E}, \mathtt{i}) \rangle,$$

where γ is an interleaving merge of α and β and $\mathcal{E} = \langle \emptyset, \{\mathtt{a!}, \mathtt{b!}\}, \emptyset, \emptyset, \{\mathtt{a!}, \mathtt{c!}\}, \emptyset \rangle^{\omega}$ is an interleaving of the sequences $\langle \emptyset, \{\mathtt{a!}, \mathtt{b!}\}, \emptyset \rangle^{\omega}$ and $\langle \emptyset, \{\mathtt{a!}, \mathtt{c!}\}, \emptyset \rangle^{\omega}$. In this trace, synchronization on channel a is never enabled, because the commands P and Q are never inside their loops at the same time; despite the infinite occurrences of a! and a? along $\mathcal{E}$, the channel a does not occur along $\mathcal{E}$.

To wrap up the details, $\mathcal{T}_{ch}[\![R]\!]$ has a trace with form $\langle \gamma', (\emptyset, \emptyset, \mathcal{E}', \mathtt{i}) \rangle$ in which γ' alternates b?1 actions with c?n actions. Therefore, there is a trace of $\mathcal{T}_{ch}[\![P \| Q \| R]\!]$ with form

$$\langle \zeta, (\emptyset, \{\mathtt{a}\}, \mathcal{E}'', \mathtt{i}) \rangle,$$

where $\mathcal{E}''$ is an interleaving of $\mathcal{E}$ and $\mathcal{E}'$ and ζ is a trace in which each communication of γ is synchronized with a communication of γ'. Because neither a! nor a? can possibly occur along $\mathcal{E}'$, the channel a does not appear along $\mathcal{E}''$. It follows that there is an infinite trace with form $\langle \zeta, (\emptyset, \emptyset, \langle \emptyset \rangle^{\omega}, \mathtt{i}) \rangle$ in $\mathcal{T}_{ch}[\![(P \| Q \| R) \backslash \mathtt{a} \backslash \mathtt{b} \backslash \mathtt{c}]\!]$, corresponding to a nonterminating, channel-fair computation of the program $(P \| Q \| R) \backslash \mathtt{a} \backslash \mathtt{b} \backslash \mathtt{c}$. $\diamond$

In the following example, we modify the previous example to ensure termination under strong channel fairness. Essential to proving termination is the introduction of additional communications that keep the processes synchronized with one another: communications on channels a, b and c can occur only when all three processes are inside their loops.

Example 5.4.3 Consider the following processes P', Q' and R', which are revised versions of the processes P, Q, and R (respectively) of the previous example:

$$\begin{aligned}
P' \;&\equiv\; \mathsf{while}\ (\mathsf{x} \neq 0)\ \mathsf{do} \\
&\qquad \mathsf{sync!1} \rightarrow (\mathsf{a!0} \rightarrow \mathsf{x{:=}0} \;\square\; \mathsf{b!1} \rightarrow \mathsf{e!1} \rightarrow \mathsf{skip} \;\square\; \mathsf{e?x} \rightarrow \mathsf{skip}), \\
Q' \;&\equiv\; \mathsf{n{:=}1}; \\
&\quad\ \mathsf{while}\ (\mathsf{w} \neq 0)\ \mathsf{do} \\
&\qquad \mathsf{sync?w} \rightarrow \mathsf{sync?w} \rightarrow (\mathsf{a?w} \rightarrow \mathsf{c!w} \;\square\; \mathsf{c!n} \rightarrow \mathsf{e!1} \rightarrow \mathsf{n{:=}n+1} \;\square\; \mathsf{e?w} \rightarrow \mathsf{skip}), \\
R' \;&\equiv\; \mathsf{while}\ (\mathsf{v} \neq 0)\ \mathsf{do}\ \mathsf{sync!1} \rightarrow (\mathsf{c?v} \rightarrow \mathsf{skip} \;\square\; \mathsf{b?v} \rightarrow \mathsf{skip}).
\end{aligned}$$

We use the trace semantics $\mathcal{T}_{ch}$ to prove that the program $(P' \| Q' \| R') \backslash \mathtt{sync} \backslash \mathtt{e} \backslash \mathtt{a} \backslash \mathtt{b} \backslash \mathtt{c}$ always terminates under strong channel fairness.

Let C abbreviate the program $(P' \| Q' \| R')\backslash\mathsf{sync}\backslash\mathsf{e}$. The only infinite computations of C are those in which each of P', Q' and R' make infinite progress: the need to synchronize on channel sync prevents two processes from conspiring against the third. Moreover, in any such infinite computation, synchronization on the channel a is enabled infinitely often. Therefore, every infinite trace of C has form $\langle \alpha, (F, U, \mathcal{E}, \mathtt{i}) \rangle$, where the channel a is in the set $\Cup\mathcal{E}$. (That is, the channel a, in addition to the directions $\mathsf{a}!$ and $\mathsf{a}?$, appears in infinitely many sets along the sequence $\mathcal{E}$.) As a result, the only possible infinite traces in $C\backslash\mathsf{a}$ have form

$$\langle \alpha, (F', U\backslash\mathsf{a}, \mathcal{E}\backslash\mathsf{a}, \mathtt{i}) \rangle,$$

where $F' \supseteq F\backslash\mathsf{a}$.

However, every infinite trace of P' has form $\langle \beta, (F_p, U_p, \mathcal{E}_p, \mathtt{i}) \rangle$, where β involves no communications on channel a and a is in U_p. Therefore, a must also be in U, and hence the traces of $\mathcal{T}_{ch}[\![C]\!]$ with form $\langle \alpha, (F, U, \mathcal{E}, \mathtt{i}) \rangle$ (that is, all the infinite traces) must be discarded in creating the set $\mathcal{T}_{ch}[\![C\backslash\mathsf{a}]\!]$. It follows that there are no infinite traces in the set $\mathcal{T}_{ch}[\![C\backslash\mathsf{a}]\!]$, and therefore no infinite traces in the set $\mathcal{T}_{ch}[\![C\backslash\mathsf{a}\backslash\mathsf{b}\backslash\mathsf{c}]\!] = \mathcal{T}_{ch}[\![(P' \| Q' \| R')\backslash\mathsf{sync}\backslash\mathsf{e}\backslash\mathsf{a}\backslash\mathsf{b}\backslash\mathsf{c}]\!]$.

A similar analysis shows that deadlock of the program is impossible, and hence the program must always terminate successfully. $\diamond$

5.5 Lack of Full Abstraction

The semantics $\mathcal{T}_{ch}$ is sound with respect to all of the (channel-fair equivalents of the) behaviors introduced in Chapter 4, but it is fully abstract with respect to none of them. Of course, this is not surprising: the strongly fair semantics $\mathcal{T}_s$ required the addition of closure conditions to yield full abstraction.

Some of the inappropriate distinctions made by $\mathcal{T}_{ch}$ can indeed be eliminated by the simple introduction of closure conditions. For example, recall the commands C_1 and C_2 that led us to introduce union and superset closure conditions for strong fairness:

$$\begin{aligned} C_1 &\equiv (\mathsf{a}!0 \to \mathsf{b}!0) \,\square\, (\mathsf{a}!0 \to \mathsf{c}!0), \\ C_2 &\equiv (\mathsf{a}!0 \to \mathsf{b}!0) \,\square\, (\mathsf{a}!0 \to \mathsf{c}!0) \,\square\, (\mathsf{a}!0 \to (\mathsf{b}!0 \,\square\, \mathsf{c}!0)). \end{aligned}$$

These commands have different trace sets and yet are indistinguishable in all program contexts, even under channel fairness. Introducing union and superset conditions suited to channel-fair traces can eliminate the distinction between C_1 and C_2.

However, other inappropriate distinctions cannot be remedied so easily. For instance, consider the following two commands:

$$
\begin{aligned}
C_3 \;&\equiv\; \mathsf{a!0} \to ((\mathsf{b!0} \to \ \mathsf{while\ true\ do\ a!0}) \,\Box\, (\mathsf{c!0} \to \mathsf{skip})) \\
&\Box\;\; \mathsf{a!0} \to (\mathsf{b!0} \to \mathsf{skip} \ \Box\ \mathsf{d!0} \to \mathsf{skip}), \\
C_4 \;&\equiv\; C_3 \,\Box\, \mathsf{a!0} \to ((\mathsf{b!0} \to \mathsf{while\ true\ do\ a!0}) \ \Box\ (\mathsf{d!0} \to \mathsf{skip})).
\end{aligned}
$$

Let α be the simple trace $(s, \mathsf{a!0}, s)(s, \mathsf{b!0}, s)[(s, \varepsilon, s)(s, \mathsf{a!0}, s)]^{\omega}$, and let $\mathcal{E}$ be the infinite sequence $\langle \emptyset, \{\mathtt{a!}\}\rangle^{\omega}$. The infinite trace $\varphi_3 = \langle \alpha, (\emptyset, \langle\{\mathtt{a!}\}, \{\mathtt{b!}, \mathtt{c!}\}\rangle\mathcal{E}, \mathtt{i})\rangle$ is possible for both C_3 and C_4, whereas the infinite trace $\varphi_4 = \langle \alpha, (\emptyset, \langle\{\mathtt{a!}\}, \{\mathtt{b!}, \mathtt{d!}\}\rangle\mathcal{E}, \mathtt{i})\rangle$ is possible only for C_4. These two traces differ in the sets of directions enabled on their second steps. Despite this difference, the commands C_3 and C_4 exhibit the same behaviors in all program contexts. In essence, the traces φ_3 and φ_4 are indistinguishable from the standpoint of strong-channel fairness, because they share the same infinite suffix of enabling sets: after some finite period of time, both enable the same communications on precisely the same steps.

In general, eliminating this type of distinction requires a more direct approach than closure conditions provide: every pair of congruent (i.e., sharing the same simple trace component and fairness sets F and U) traces that also share an infinite suffix of enabling sequences must be considered equivalent. Formalizing such relationships requires the introduction of an equivalence relation on traces that identifies exactly such pairs, followed by the imposition of a quotient structure on trace sets based on this equivalence relation. It seems likely that such an approach would yield a fully abstract channel-fair semantics. However, it is unclear that these technical contortions would would provide significantly (if any) more insight than the semantics $\mathcal{T}_{ch}$ already provides.

The difficulty in achieving fully abstraction (and the expected complexity of such a model) should not be construed automatically as an indictment of the general trace framework. Rather, they reflect the inherent complexity that underlies the notion of strong channel fairness. Strong channel fairness is not *equivalence robust* [AFK88], in that the specific order in which independent actions occur affects the fairness of a given computation. For example, recall Examples 5.2.2 and 5.2.3: the order in which the computations ρ_1 and ρ_2 are interleaved affects the channel fairness of the resulting computation. Because channel-fairness depends on the order in which actions are enabled and occur, any semantics that incorporates assumptions of channel-fairness must account for this dependence in some way. It should not be surprising that the resulting semantics is complex when the underlying notion of fairness is as well.

The lack of full abstraction should also not be interpreted as a condemnation of the semantics $\mathcal{T}_{ch}$. Full abstraction is an ideal that is not always easily achievable, and it is well known that certain notions of behavior for certain languages do not admit fully abstract models [Mil77, AP86, Sto88]. Moreover, the semantics $\mathcal{T}_{ch}$ still supports compositional reasoning about strongly channel-fair behavior, and its soundness for several behavioral notions still provides useful, if incomplete, information about program equivalence and substitutability: two

$$
\begin{array}{rcl}
c_1 \| c_2 & \equiv & c_2 \| c_1 \\
(c_1 \| c_2) \parallel c_3 & \equiv & c_1 \parallel (c_2 \| c_3) \\
(c_1 \| c_2) \backslash h & \equiv & c_1 \parallel (c_2 \backslash h), \quad \text{provided } h \notin \mathsf{fc}[\![c_1]\!] \\
c \backslash h & \equiv & c, \quad \text{provided } h \notin \mathsf{fc}[\![c]\!] \\
(\mathsf{a!0} \to \mathsf{b!0}) \;\square\; (\mathsf{b!0} \to \mathsf{a!0}) & \equiv & \mathsf{a!0} \parallel \mathsf{b!0}
\end{array}
$$

Figure 5.2: Some program equivalences validated by $\mathcal{T}_{ch}^{\dagger}$.

program terms are guaranteed to behave equivalently whenever $\mathcal{T}_{ch}$ gives them identical meanings. For example, the soundness of $\mathcal{T}_{ch}$ is sufficient for validating the program equivalences (with respect to any of the channel-fair equivalents of the behaviors $\mathcal{M}$, $\mathcal{S}$, $\mathcal{W}$ or $\mathcal{C}$) of Figure 5.2, properties which also hold under strong-fairness assumptions.

Chapter 6

Weak Process Fairness

This chapter focuses on weak process fairness, which requires every continuously enabled process to make progress. The assumption of weak fairness is weaker (and therefore more general) than strong fairness. Perhaps ironically, then, incorporating weak-fairness assumptions into a semantics for communicating processes is more complicated than incorporating strong-fairness assumptions. In particular, the task of determining which processes are enabled *continuously* requires significantly more structure than determining which processes are enabled infinitely often does: not only can a process be enabled continuously along a computation of $c_1 \| c_2$ without being enabled continuously along either component's subcomputation, but it can be enabled continuously without any one of its possible actions being enabled continuously.

In this chapter, we show how to adapt the trace framework to incorporate assumptions of weak process fairness. We discuss why weak fairness is harder to model than strong fairness, and we indicate what type of additional semantic structure weak fairness requires. Based on these observations, we introduce a parameterized form of weak fairness that is based on parameterized strong fairness but tailored for reasoning about the *continuous* enabling of processes. This parameterization guides our construction of a weakly fair trace semantics that is strikingly similar to the channel-fair semantics of Chapter 5.

6.1 Parameterized Weak Fairness

In Section 3.1, we introduced the notion of parameterized strong process fairness to permit a compositional characterization of strongly fair computation. Roughly speaking, we tag "almost strongly fair" computations with sets of directions that represent the actions possible for those processes that are treated unfairly. These sets do not distinguish between the actions possible for a single process and the actions possible for a collection of processes, because such distinctions are irrelevant for strong fairness. A process P_i having the set of enabled directions E_i is enabled infinitely along a given computation if and only if some element of E_i is enabled infinitely often. Similarly, some member of the collection of processes $\{P_1, \dots, P_k\}$ is enabled for communication infinitely often along a given computation if and only if some direction in one of the sets in $\{E_1, \dots, E_k\}$ is enabled infinitely often. Thus, for example, the single process

$$Q_1 \equiv \mathsf{a!0} \to \mathsf{b!0} \,\Box\, \mathsf{b!0} \to \mathsf{a!0}$$

has precisely the same set of fairness constraints as the parallel command

$$Q_2 \equiv (\mathsf{a!0} \parallel \mathsf{b!0});$$

each Q_i is enabled for synchronization infinitely often along a computation of $Q_i \| C$ (for any command C) if and only if C enables input on channel a or b infinitely often.

The situation changes, however, when we consider the *continuous* enabling of directions and processes. The process P_i can be enabled continuously along a computation without any particular element of E_i being enabled for synchronization continuously. For example, consider the command

$$C \equiv (\mathsf{while\ true\ do}\ (\mathsf{a?x} \Box \mathsf{c!1})) \parallel (\mathsf{while\ true\ do}\ (\mathsf{b?y} \Box \mathsf{c!2})),$$

and let ρ be a computation of C such that (1) both parallel subcomponents repeatedly perform output on channel c; and (2) at any time after the initial step, at least one of the components is inside its loop. Along this computation ρ, the directions a? and b? are each enabled infinitely often and disabled infinitely often; moreover, at any time after the first step, at least one of the directions a? and b? is enabled.

Because the single process Q_1 can perform output on either channel a or channel b, it is enabled continuously in any computation of $(Q_1 \| C)\backslash \mathsf{a} \backslash \mathsf{b}$ in which C performs the transition sequence ρ. As a result, in any weakly fair computation of $(Q_1 \| C)\backslash \mathsf{a} \backslash \mathsf{b}$, C must eventually deviate from the transition sequence ρ. In contrast, there are weakly fair computations of $(Q_2 \| C)\backslash \mathsf{a} \backslash \mathsf{b}$ in which C performs the transitions ρ: Q_2 contains two processes, neither of which is enabled continuously by ρ. Whereas the commands Q_1 and Q_2 have identical behaviors under strong fairness, they can exhibit different behaviors under weak fairness. For this reason, parameterized weak fairness—unlike parameterized strong fairness—must distinguish between the actions possible for a single process and the actions possible for a collection of processes.

To this end, we tag "almost weakly fair" computations with a set $\mathcal{F}$ of *sets of directions*, each set $F \in \mathcal{F}$ intuitively indicating that one or more subprocesses are blocked modulo F. For example, we use the set

$$\mathcal{F}_1 = \{\{\mathsf{a!}, \mathsf{b!}\}\}$$

$$
\begin{array}{rcl}
\mathsf{initsets}(\mathsf{skip}, s) & = & \{\{\varepsilon\}\} \\
\mathsf{initsets}(i{:=}e, s) & = & \{\{\varepsilon\}\} \\
\mathsf{initsets}(\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, s) & = & \{\{\varepsilon\}\} \\
\mathsf{initsets}(\mathsf{while}\ b\ \mathsf{do}\ c, s) & = & \{\{\varepsilon\}\} \\
\mathsf{initsets}(c_1; c_2, s) & = & \mathsf{initsets}(c_1, s) \\
\mathsf{initsets}(h?i, s) & = & \{\{h?\}\} \\
\mathsf{initsets}(h!e, s) & = & \{\{h!\}\} \\
\mathsf{initsets}(g \rightarrow c, s) & = & \mathsf{initsets}(g, s) \\
\mathsf{initsets}(gc_1 \square gc_2, s) & = & \{X_1 \cup X_2 \mid X_1 \in \mathsf{initsets}(gc_1, s)\ \&\ X_2 \in \mathsf{initsets}(gc_2, s)\} \\
\mathsf{initsets}(c_1 \| c_2, s) & = & \mathsf{initsets}(c_1, s) \cup \mathsf{initsets}(c_2, s) \\
 & \cup & \{\{\varepsilon\} \mid \mathsf{match}(\mathsf{initsets}(c_1, s), \mathsf{initsets}(c_2, s))\} \\
\mathsf{initsets}(c \backslash h, s) & = & \{F - \{h!, h?\} \mid F \in \mathsf{initsets}(c, s)\}
\end{array}
$$

Figure 6.1: The definition $\mathsf{initsets}(c, s)$.

to tag computations in which one or more subprocesses are blocked modulo $\{\mathsf{a}!, \mathsf{b}!\}$; the partial computation $\langle Q_1, s\rangle$ can be tagged by $\mathcal{F}_1$. In contrast, we use the set

$$\mathcal{F}_2 = \{\{\mathsf{a}!\}, \{\mathsf{b}!\}\}$$

to tag computations in which one or more processes are blocked modulo $\{\mathsf{a}!\}$ and one or more processes are blocked modulo $\{\mathsf{b}!\}$; the partial computation $\langle Q_2, s\rangle$ can be tagged by $\mathcal{F}_2$.

The set $\mathsf{inits}(c, s)$, introduced in Section 2.1, is the set of directions (possibly including ε) corresponding to the possible transitions from configuration $\langle c, s\rangle$. We can likewise define a set $\mathsf{initsets}(c, s)$ that contains *sets* of directions (possibly including $\{\varepsilon\}$), with the intuition that each set reflects the transitions possible for one (or more) or c's subprocesses from the configuration $\langle c, s\rangle$. A structurally inductive definition of the set $\mathsf{initsets}(c, s)$ appears in Figure 6.1.[1] When the command c has only one associated process, $\mathsf{initsets}(c, s)$ is necessarily a singleton set; in particular, the set $\mathsf{initsets}(gc_1 \square gc_2, s)$ is a singleton set whose only element may contain several directions. This definition provides a way to distinguish the commands Q_1 and Q_2 as required: for all states s, $\mathsf{initsets}(Q_1, s) = \{\{\mathsf{a}!, \mathsf{b}!\}\}$, whereas $\mathsf{initsets}(Q_2, s) = \{\{\mathsf{a}!\}, \{\mathsf{b}!\}\}$.

Finally, we note that different sets may represent the same weak-fairness constraints. For example, consider the sets $\mathcal{F}_1 = \{\{\mathsf{a}!, \mathsf{b}!\}, \{\mathsf{a}!\}\}$ and $\mathcal{F}_2 = \{\{\mathsf{a}!, \mathsf{b}!\}\}$. Both sets represent

[1]This inductive definition relies on the obvious extension of the predicate match to sets of sets of directions: for such sets $\mathcal{X}_1$ and $\mathcal{X}_2$, the predicate $\mathsf{match}(\mathcal{X}_1, \mathcal{X}_2)$ is true if and only if there exists sets $X_1 \in \mathcal{X}_1$ and $X_2 \in \mathcal{X}_2$ such that $\mathsf{match}(X_1, X_2)$.

identical constraints: each $\mathcal{F}_i$ will be enabled for synchronization continuously along any computation that enables the set $\{\mathsf{a}!, \mathsf{b}!\}$ continuously. In effect, the possibilities inherent in the set $\{\mathsf{a}!\}$ are subsumed by the set $\{\mathsf{a}!, \mathsf{b}!\}$: any computation that provides the set $\{\mathsf{a}!\}$ with continuous synchronization opportunities necessarily provides the set $\{\mathsf{a}!, \mathsf{b}!\}$ with continuous synchronization opportunities. We use downwards closure to yield canonical representations of the fairness constraints.

Definition 6.1.1 Let $\mathcal{F}$ be a member of $\mathcal{P}_{\text{fin}}(\mathcal{P}_{\text{fin}}(\Delta))$. The **downwards closure** of $\mathcal{F}$, written $\mathcal{F}\!\downarrow$, is the set of all subsets of members of $\mathcal{F}$: $\quad \mathcal{F}\!\downarrow = \{F' \mid \exists F \in \mathcal{F}. F' \subseteq F\}$. $\diamond$

Intuitively, the sets $\mathcal{F}_1$ and $\mathcal{F}_2$ represent identical weak-fairness constraints whenever $\mathcal{F}_1\!\downarrow = \mathcal{F}_2\!\downarrow$.

Definition 6.1.2 Let $\mathcal{F}$ be a member of $\mathcal{P}_{\text{fin}}(\mathcal{P}_{\text{fin}}(\Delta))$. A configuration $\langle c, s\rangle$ is **blocked modulo** $\mathcal{F}$ if $\mathsf{initsets}(c,s) - \mathcal{F}\!\downarrow = \emptyset$, and it is **enabled modulo** $\mathcal{F}$ otherwise. $\diamond$

Thus a configuration is blocked modulo $\mathcal{F}$ if each of its subprocesses is blocked mod F for some $F \in \mathcal{F}\!\downarrow$.

Example 6.1.3 Recall the commands $Q_1 \equiv \mathsf{a}!0 \to \mathsf{b}!0 \,\Box\, \mathsf{b}!0 \to \mathsf{a}!0$ and $Q_2 \equiv (\mathsf{a}!0 \parallel \mathsf{b}!0)$, with the sets of enabled communications

$$\mathsf{initsets}(Q_1, s) = \{\{\mathsf{a}!, \mathsf{b}!\}\}, \qquad \mathsf{initsets}(Q_2, s) = \{\{\mathsf{a}!\}, \{\mathsf{b}!\}\}.$$

The configurations $\langle Q_1, s\rangle$ and $\langle Q_2, s\rangle$ are both blocked modulo $\{\{\mathsf{a}!, \mathsf{b}!\}\}$, because (for each i)

$$\mathsf{initsets}(Q_i, s) \subseteq \{\{\mathsf{a}!, \mathsf{b}!\}\}\!\downarrow = \{\emptyset, \{\mathsf{a}!\}, \{\mathsf{b}!\}, \{\mathsf{a}!, \mathsf{b}!\}\}.$$

However, only the configuration $\langle Q_2, s\rangle$ is blocked modulo $\{\{\mathsf{a}!\}, \{\mathsf{b}!\}\}$:

$$\mathsf{initsets}(Q_2, s) - \{\{\mathsf{a}!\}, \{\mathsf{b}!\}\}\!\downarrow = \emptyset, \text{ whereas } \{\mathsf{a}!, \mathsf{b}!\} \in \mathsf{initsets}(Q_1, s) - \{\{\mathsf{a}!\}, \{\mathsf{b}!\}\}\!\downarrow.$$

$\diamond$

We can now give a parameterized notion of weak fairness that mimics the parameterization of strong fairness in Section 3.1 but also accounts for the additional structure of the fairness sets $\mathcal{F}$. A computation is weakly fair (in the standard sense) if and only if it is weakly fair modulo $\emptyset$.

Definition 6.1.4 Let $\mathcal{F}$ be a member of $\mathcal{P}_{\text{fin}}(\mathcal{P}_{\text{fin}}(\Delta))$. A computation ρ of the command c is **weakly fair modulo** $\mathcal{F}$ provided ρ satisfies one of the following conditions:

- ρ is a finite, successfully terminating computation;

- ρ is a partial computation whose final configuration is blocked modulo $\mathcal{F}$;
- ρ is an infinite computation, c has form $(c_1;c_2)$ or $(\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)$, and the underlying infinite computation of c_1 or c_2 is weakly fair mod $\mathcal{F}$;
- ρ is an infinite computation, c has form $(\mathsf{while}\ b\ \mathsf{do}\ c')$ or $(g \to c')$, and all underlying computations of c' are weakly fair mod $\mathcal{F}$;
- ρ is an infinite computation, c has form $(gc_1 \Box gc_2)$, and the underlying computation of the selected gc_i is weakly fair mod $\mathcal{F}$;
- ρ is an infinite computation, c has form $c' \backslash h$, and ρ's underlying computation of c' is weakly fair modulo $\{F \cup \{h!, h?\} \mid F \in \mathcal{F}\}$;
- ρ is an infinite computation, c has form $c_1 \| c_2$, and there exist sets $\mathcal{F}_1$ and $\mathcal{F}_2$, and computations ρ_1 of c_1 and ρ_2 of c_2, such that:
  - ρ_1 is weakly fair mod $\mathcal{F}_1$ and ρ_2 is weakly fair mod $\mathcal{F}_2$,
  - ρ can be obtained by merging and synchronizing ρ_1 and ρ_2,
  - $\mathcal{F}{\downarrow} \supseteq (\mathcal{F}_1 \cup \mathcal{F}_2){\downarrow}$, and
  - no subcomponent of c_1 or c_2 that fails to make infinite progress is enabled for synchronization almost everywhere along ρ. $\diamond$

The final condition in the parallel-composition clause ensures that no process that becomes blocked modulo $\mathcal{F}$ continuously has some opportunity to synchronize. Unlike the parallel-composition clause for parameterized strong fairness, c_1's constraints do not depend solely on ρ_2 (and likewise for c_2 and ρ_1): a (sub)process can be enabled for synchronization continuously along the computation ρ without being enabled for synchronization continuously along either ρ_1 or ρ_2. For example, consider the commands

$$C_1 \equiv \mathsf{a?x} \,\|\, \mathsf{while\ true\ do}\ (\mathsf{a!1} \Box \mathsf{b!1}), \qquad C_2 \equiv \mathsf{while\ true\ do}\ (\mathsf{a!2} \Box \mathsf{b!2}).$$

Suppose that ρ_1 is an infinite, weakly fair mod $\{\{\mathsf{a?}\}\}$ computation of C_1 in which the process $\mathsf{a?x}$ makes no progress, and let ρ_2 be a weakly fair computation of C_2. The process $\mathsf{a?x}$ is enabled for synchronization infinitely often—but not continuously—along each of the computations ρ_1 and ρ_2. However, the computations ρ_1 and ρ_2 can be interleaved to yield a computation ρ of $C_1 \| C_2$ in such a way that the process $\mathsf{a?x}$ is enabled for synchronization continuously along ρ. As a result, it is often necessary to look at the resulting computation of the parallel command to determine whether any blocked processes are actually enabled continuously. We explore this situation in more detail in the following examples.

Example 6.1.5 Let C_1 and C_2 be the commands of the preceding discussion:

$$C_1 \equiv \mathsf{a?x} \parallel \mathsf{while\ true\ do}\ (\mathsf{a!1} \Box \mathsf{b!1}), \qquad C_2 \equiv \mathsf{while\ true\ do}\ (\mathsf{a!2} \Box \mathsf{b!2}).$$

For notational expediency, we let C abbreviate the command $\mathsf{while\ true\ do}\ (\mathsf{a!1} \Box \mathsf{b!1})$, so that $C_1 \equiv \mathsf{a?x} \parallel C$.

1. Let ρ_1 be the following infinite computation

$$\begin{aligned} \rho_1 = \ \langle \mathsf{a?x} \parallel C,\ s\rangle &\xrightarrow{\varepsilon} \langle \mathsf{a?x} \parallel (\mathsf{a!1} \Box \mathsf{b!1}); C, s\rangle \\ &\xrightarrow{\mathsf{b!1}} \langle \mathsf{a?x} \parallel C,\ s\rangle \xrightarrow{\varepsilon} \cdots \end{aligned}$$

   in which the process $\mathsf{a?x}$ never makes a transmission and the value 1 is repeatedly transmitted along channel b.

   The computation ρ_1 is weakly fair modulo $\{\{\mathsf{a?}\}\}$: the only continually enabled process that does not make progress is blocked modulo $\{\{\mathsf{a?}\}\}$, and it is not enabled for synchronization continuously.

2. Let ρ_2 be the infinite computation

$$\begin{aligned} \rho_2 = \ \langle C_2,\ t\rangle &\xrightarrow{\varepsilon} \langle (\mathsf{a!2} \Box \mathsf{b!2}); C_2,\ t\rangle \\ &\xrightarrow{\mathsf{b!2}} \langle C_2,\ t\rangle \xrightarrow{\varepsilon} \cdots \end{aligned}$$

   that repeatedly transmits the value 2 along channel b. The computation ρ_2 is weakly fair modulo $\emptyset$.

3. Let ρ be the following interleaving of ρ_1 and ρ_2 in which every transition of C_1 is followed by a transition of C_2 and vice versa:

$$\begin{aligned} \rho = \ \langle \mathsf{a?x} \parallel C \| C_2,\ s \cup t\rangle &\xrightarrow{\varepsilon} \langle \mathsf{a?x} \parallel (\mathsf{a!1} \Box \mathsf{b!1}); C \parallel C_2, s \cup t\rangle \\ &\xrightarrow{\varepsilon} \langle \mathsf{a?x} \parallel (\mathsf{a!1} \Box \mathsf{b!1}); C \parallel (\mathsf{a!2} \Box \mathsf{b!2}); C_2, s \cup t\rangle \\ &\xrightarrow{\mathsf{b!1}} \langle \mathsf{a?x} \parallel C \parallel (\mathsf{a!2} \Box \mathsf{b!2}); C_2, s \cup t\rangle \\ &\xrightarrow{\mathsf{b!2}} \langle \mathsf{a?x} \parallel C \parallel C_2, s \cup t\rangle \xrightarrow{\varepsilon} \cdots \end{aligned}$$

   The computation ρ is weakly fair mod $\{\{\mathsf{a?}\}\}$, because the process $\mathsf{a?x}$ never becomes enabled for synchronization continuously along ρ. In particular, $\mathsf{a?x}$ is disabled for synchronization at every configuration $\langle \mathsf{a?x} \parallel C \| C_2,\ s \cup t\rangle$.

4. The corresponding computation of $(C_1\|C_2)\backslash\mathsf{a}$ in which the process $\mathsf{a?x}$ never makes a transition is weakly fair modulo $\emptyset$. ◇

The following example, taken together with the preceding one, shows how the order in which independent actions occur can affect the weak fairness of a computation.

Example 6.1.6 Let the commands C_1 and C_2, and the computations ρ_1 and ρ_2, be as defined in the preceding example, and let ρ' be the following interleaving of ρ_1 and ρ_2:

$$\begin{aligned}
\rho' = \langle \mathsf{a?x} \parallel C \| C_2,\ s\cup t\rangle &\xrightarrow{\varepsilon} \langle \mathsf{a?x} \parallel (\mathsf{a!1}\Box\mathsf{b!1});C \parallel C_2, s\cup t\rangle \\
&\xrightarrow{\varepsilon} \langle \mathsf{a?x} \parallel (\mathsf{a!1}\Box\mathsf{b!1});C \parallel (\mathsf{a!2}\Box\mathsf{b!2});C_2, s\cup t\rangle \\
&\xrightarrow{\mathsf{b!1}} \langle \mathsf{a?x} \parallel C \parallel (\mathsf{a!2}\Box\mathsf{b!2});C_2, s\cup t\rangle \\
&\xrightarrow{\varepsilon} \langle \mathsf{a?x} \parallel (\mathsf{a!1}\Box\mathsf{b!1});C \parallel (\mathsf{a!2}\Box\mathsf{b!2});C_2, s\cup t\rangle \\
&\xrightarrow{\mathsf{b!2}} \langle \mathsf{a?x} \parallel (\mathsf{a!1}\Box\mathsf{b!1});C \parallel C_2, s\cup t\rangle \\
&\xrightarrow{\varepsilon} \langle \mathsf{a?x} \parallel (\mathsf{a!1}\Box\mathsf{b!1});C \parallel (\mathsf{a!2}\Box\mathsf{b!2});C_2, s\cup t\rangle \\
&\xrightarrow{\varepsilon} \cdots
\end{aligned}$$

In this computation, from the second configuration onward, at least one of C and C_2 is always inside its loop. As a result, the process $\mathsf{a?x}$ is enabled for synchronization continuously, and the computation ρ' is not weakly fair modulo $\{\{\mathsf{a?}\}\}$. As a result, the corresponding computation of $(C_1\|C_2)\backslash\mathsf{a}$ is not weakly fair. ◇

The following example shows that, under weak fairness, a process can block on a communication, even though that same channel is used for synchronization infinitely often by other processes.

Example 6.1.7 Let P_1 and P_2 be the following processes:

$$P_1 \equiv \mathsf{while\ true\ do\ }(\mathsf{b?x}\Box\mathsf{a!1}), \qquad P_2 \equiv \mathsf{while\ true\ do\ }(\mathsf{b?y}\Box\mathsf{a!2}\Box\mathsf{b!2}).$$

1. The infinite computation

$$\begin{aligned}
\rho_1 = \langle(\mathsf{b!0} \parallel P_1), [\mathsf{x}=2]\rangle &\xrightarrow{\varepsilon} \langle(\mathsf{b!0} \parallel (\mathsf{b?x}\Box\mathsf{a!1});P_1), [\mathsf{x}=2]\rangle \\
&\xrightarrow{\mathsf{b?2}} \langle(\mathsf{b!0} \parallel P_1), [\mathsf{x}=2]\rangle \\
&\xrightarrow{\varepsilon} \cdots
\end{aligned}$$

that repeatedly receives the value 2 on channel b and never performs the action $\mathsf{b!0}$ is weakly fair modulo $\{\{\mathsf{b!}\}\}$.

2. The infinite computation

$$\begin{array}{rcl} \rho_2 = \langle P_2, [\mathsf{y}=1]\rangle & \xrightarrow{\varepsilon} & \langle (\mathsf{b?y} \Box \mathsf{a!2} \Box \mathsf{b!2}); P_2, [\mathsf{y}=1]\rangle \\ & \xrightarrow{\mathsf{b!2}} & \langle P_2, [\mathsf{y}=1]\rangle \\ & \xrightarrow{\varepsilon} & \cdots \end{array}$$

that repeatedly transmits the value 2 on channel b is weakly fair modulo $\emptyset$.

3. Let s represent the state $[\mathsf{x}=2, \mathsf{y}=1]$, and let ρ be the following computation, which can be obtained by interleaving and merging ρ_1 and ρ_2:

$$\begin{array}{rcl} \langle (\mathsf{b!0} \parallel P_1) \parallel P_2, s\rangle & \xrightarrow{\varepsilon} & \langle (\mathsf{b!0} \parallel (\mathsf{b?x} \Box \mathsf{a!1}); P_1) \parallel P_2, s\rangle \\ & \xrightarrow{\varepsilon} & \langle (\mathsf{b!0} \parallel (\mathsf{b?x} \Box \mathsf{a!1}); P_1) \parallel (\mathsf{b?y} \Box \mathsf{a!2} \Box \mathsf{b!2}); P_2, s\rangle \\ & \xrightarrow{\varepsilon} & \langle (\mathsf{b!0} \parallel P_1) \parallel P_2, s\rangle \\ & \xrightarrow{\varepsilon} & \cdots \end{array}$$

The computation ρ is weakly fair modulo $\{\{\mathsf{b!}\}\}$: although the process $\mathsf{b!0}$ is enabled for synchronization infinitely often, it is not enabled for synchronization continuously. In particular, the computation is weakly fair modulo $\{\{\mathsf{b!}\}\}$ despite the infinite use of channel b for synchronization between P_1 and P_2.

4. It follows that the corresponding computation of $((\mathsf{b!0} \parallel P_1) \parallel P_2)\backslash\mathsf{b}$ is weakly fair. $\diamond$

6.2 Weakly Fair Traces

The definition of parameterized weak fairness, combined with the experience of defining strongly fair and channel-fair traces, guides us in the construction of appropriate weakly fair traces. First, we need sets $\mathcal{F}$ of sets of directions to represent the process constraints, because a process can be enabled continuously without any particular action being enabled continuously. Second, we need to record the directions enabled *at each step* along a computation, because directions can be enabled continuously along a computation of a parallel command without being enabled continuously by any individual component.

We therefore define the set Φ_w of **weakly fair traces** by

$$\Phi_w \quad = \quad \Sigma^\infty \times \mathcal{P}_{\text{fin}}(\mathcal{P}_{\text{fin}}(\Delta^+)) \times (\mathcal{P}_{\text{fin}}(\Delta \cup \mathsf{Chan}))^\infty \times \{\mathtt{f}, \mathtt{i}, \mathtt{p}\}.$$

Intuitively, the weakly fair trace $\langle \alpha, (\mathcal{F}, \mathcal{E}, \mathtt{f})\rangle$ represents a (necessarily weakly fair) successfully terminating computation having the finite sequence $\mathcal{E}$ of enabling sets. Similarly, the weakly fair trace $\langle \alpha, (\mathcal{F}, \mathcal{E}, \mathtt{i}) \rangle$ represents an infinite, weakly fair mod $\mathcal{F}$ computation having the infinite sequence $\mathcal{E}$ of enabling sets. Finally, the weakly fair trace $\langle \alpha, (\mathcal{F}, \mathcal{E}, \mathtt{p}) \rangle$ represents a partial computation such that $\mathcal{F} \supseteq \mathsf{initsets}(c_k, s_k)$, where $\langle c_k, s_k \rangle$ is the final configuration of ρ; $\mathcal{E}$ again represents the sequence of enabling sets encountered along the computation.

We characterize a weakly fair trace semantics $\mathcal{T}_w : \mathsf{Com} \to \mathcal{P}(\Phi_w)$ operationally as follows:

$$\begin{aligned}
\mathcal{T}_w[\![c]\!] \;=\; & \{\langle \mathsf{trace}(\rho), (\mathcal{F}, \mathsf{En}(\rho), \mathtt{f}) \rangle \mid \\
& \qquad \rho = \langle c, s_0 \rangle \xrightarrow{\lambda_0} \langle c_1, s_1 \rangle \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_{k-1}} \langle c_k, s_k \rangle \mathsf{term} \text{ is weakly fair mod } \mathcal{F}\} \\
\cup\; & \{\langle \mathsf{trace}(\rho), (\mathcal{F}, \mathsf{En}(\rho), \mathtt{p}) \rangle \mid \mathcal{F}{\downarrow}\supseteq \mathsf{initsets}(c_k, s_k)\ \& \\
& \qquad \rho = \langle c, s_0 \rangle \xrightarrow{\lambda_0} \langle c_1, s_1 \rangle \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_{k-1}} \langle c_k, s_k \rangle\ \&\ \neg\langle c_k, s_k \rangle \mathsf{term}\} \\
\cup\; & \{\langle \mathsf{trace}(\rho), (\mathcal{F}, \mathsf{En}(\rho), \mathtt{i}) \rangle \mid \\
& \qquad \rho = \langle c, s_0 \rangle \xrightarrow{\lambda_0} \langle c_1, s_1 \rangle \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_k} \cdots \text{is weakly fair mod } \mathcal{F}\}.
\end{aligned}$$

6.3 Weakly Fair Trace Semantics

The denotational characterization of the weakly fair trace semantics $\mathcal{T}_w$ is very similar to that for the channel-fair trace semantics $\mathcal{T}_{ch}$. In fact, many of the semantic operators are simpler for weakly fair trace sets: we no longer have to keep track of insufficiently used channels, and the sets $\mathcal{F}$ of sets can be manipulated in pretty much the same way as sets F of directions. As a result, almost all the explanations that accompany the semantic definitions in this section are abbreviated forms of those encountered in Chapter 5.

We first introduce a semantic function $\mathcal{T}_w : \mathsf{BExp} \to \mathcal{P}(\Phi_w)$ such that

$$\begin{aligned}
\mathcal{T}_w[\![b]\!] = & \{\langle (s, \varepsilon, s), (\mathcal{F}, \langle \emptyset, \emptyset \rangle, \mathtt{f}) \rangle \mid (s, \mathtt{tt}) \in \mathcal{B}[\![b]\!]\ \&\ \mathcal{F} \in \mathcal{P}_{\mathrm{fin}}(\mathcal{P}_{\mathrm{fin}}(\Delta))\} \\
& \cup \{\langle \varepsilon_s, (\mathcal{F}, \langle \emptyset \rangle, \mathtt{p}) \rangle \mid (s, \mathtt{tt}) \in \mathcal{B}[\![b]\!]\ \&\ \mathcal{F}{\downarrow}\supseteq \{\{\varepsilon\}\}\}.
\end{aligned}$$

As in the earlier semantics, each finite trace in $\mathcal{T}_w[\![b]\!]$ represents a transition made in the evaluation of the boolean expression b.

Based on the operational characterization of $\mathcal{T}_w$, it should be easy to see that

$$\begin{aligned}
\mathcal{T}_w[\![\mathsf{skip}]\!] = & \{\langle (s, \varepsilon, s), (\mathcal{F}, \langle \emptyset, \emptyset \rangle, \mathtt{f}) \rangle \mid s \in S\ \&\ \mathcal{F} \in \mathcal{P}_{\mathrm{fin}}(\mathcal{P}_{\mathrm{fin}}(\Delta))\} \\
\cup\ & \{\langle \varepsilon_s, (\mathcal{F}, \langle \emptyset \rangle, \mathtt{p}) \rangle \mid s \in S\ \&\ \mathcal{F}{\downarrow}\supseteq \{\{\varepsilon\}\}\}
\end{aligned}$$

and

$$\begin{aligned}
\mathcal{T}_w[\![i{:=}e]\!] = & \{\langle (s, \varepsilon, [s \mid i = n]), (\mathcal{F}, \langle \emptyset, \emptyset \rangle, \mathtt{f}) \rangle \mid i \in \mathsf{dom}(s)\ \&\ \mathcal{F} \in \mathcal{P}_{\mathrm{fin}}(\mathcal{P}_{\mathrm{fin}}(\Delta))\ \&\ (s, n) \in \mathcal{E}[\![e]\!]\} \\
\cup\ & \{\langle \varepsilon_s, (\mathcal{F}, \langle \emptyset \rangle, \mathtt{p}) \rangle \mid \mathsf{fv}[\![i{:=}e]\!] \subseteq \mathsf{dom}(s)\ \&\ \mathcal{F}{\downarrow}\supseteq \{\{\varepsilon\}\}\}.
\end{aligned}$$

Similarly, for guards we obtain

$$\begin{aligned}\mathcal{T}_w[\![h?i]\!] = \{\langle (s, h?n, [s|i=n]), (\mathcal{F}, \langle \{h?\}, \emptyset\rangle, \mathtt{f})\rangle \mid i \in \mathsf{dom}(s)\ \&\ n \in \mathbb{Z}\ \&\ \mathcal{F} \in \mathcal{P}_{\mathrm{fin}}(\mathcal{P}_{\mathrm{fin}}(\Delta))\}\\ \cup\ \{\langle \varepsilon_s, (\mathcal{F}, \langle \{h?\}\rangle, \mathtt{p})\rangle \mid i \in \mathsf{dom}(s)\ \&\ \mathcal{F}{\downarrow} \supseteq \{\{h?\}\}\}\end{aligned}$$

and

$$\begin{aligned}\mathcal{T}_w[\![h!e]\!] = \{\langle (s, h!n, s), (\mathcal{F}, \langle \{h!\}, \emptyset\rangle, \mathtt{f})\rangle \mid (s,n) \in \mathcal{E}[\![e]\!]\ \&\ \mathcal{F} \in \mathcal{P}_{\mathrm{fin}}(\mathcal{P}_{\mathrm{fin}}(\Delta))\}\\ \cup\ \{\langle \varepsilon_s, (\mathcal{F}, \langle \{h!\}\rangle, \mathtt{p})\rangle \mid \mathsf{fv}[\![e]\!] \subseteq \mathsf{dom}(s)\ \&\ \mathcal{F}{\downarrow} \supseteq \{\{h!\}\}\}.\end{aligned}$$

Note the use of downwards closure in the partial traces of the communication guards: for all suitable states s, the configuration $\langle h?i, s\rangle$ is blocked modulo $\mathcal{F}$ for all sets $\mathcal{F}{\downarrow} \supseteq \{\{h?\}\}$, and similarly for $\langle h!e, s\rangle$.

Sequential composition

Two weakly fair traces φ_1 and φ_2 are composable whenever φ_1 is an infinite or partial trace, or when φ_1 is a finite trace and the initial state of φ_2 is the final state of φ_1. Moreover, their concatenation $\varphi_1\varphi_2$ is defined almost identically to the concatenation of channel-fair traces, except that we no longer need to keep track of the unused channels. For composable traces $\varphi_1 = \langle \alpha, (\mathcal{F}_1, \mathcal{E}_1, \mathtt{f})\rangle$ and $\varphi_2 = \langle \beta, (\mathcal{F}_2, \mathcal{E}_2, R_2)\rangle$, we define

$$\varphi_1\varphi_2 = \begin{cases} \varphi_1 & \text{if } R_1 \in \{\mathtt{p}, \mathtt{i}\},\\ \langle \alpha\beta, (\mathcal{F}_2, \mathcal{E}_1 \cdot \mathcal{E}_2, R_2)\rangle, & \text{if } R_1 = \mathtt{f}.\end{cases}$$

We then define sequential composition on weakly fair trace sets T_1 and T_2 in the familiar way:

$$T_1; T_2 \quad = \quad \{\varphi_1\varphi_2 \mid \varphi_1 \in T_1\ \&\ \varphi_2 \in T_2\ \&\ \mathit{composable}(\varphi_1, \varphi_2)\}.$$

Finally, we define

$$\begin{aligned}\mathcal{T}_w[\![c_1; c_2]\!] = \mathcal{T}_w[\![c_1]\!]; \mathcal{T}_w[\![c_2]\!],\\ \mathcal{T}_w[\![g \to c]\!] = \mathcal{T}_w[\![g]\!]; \mathcal{T}_w[\![c]\!],\end{aligned}$$

and

$$\mathcal{T}_w[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] = \mathcal{T}_w[\![b]\!]; \mathcal{T}_w[\![c_1]\!] \cup \mathcal{T}_w[\![\neg b]\!]; \mathcal{T}_w[\![c_2]\!].$$

Iteration

Let $\langle\varphi_i\rangle_{i=0}^{\infty}$ be an infinite sequence of weakly fair traces such that, for each $i \geq 0$, $\varphi_i = \langle\alpha_i, (\mathcal{F}_i, \mathcal{E}_i, R_i)\rangle$. The sequence $\langle\varphi_i\rangle_{i=0}^{\infty}$ is composable if the set $\biguplus_{i=0}^{\infty} \mathcal{F}_i$ is finite and (for each i) the traces $\varphi_0\varphi_1\ldots\varphi_{i-1}$ and φ_i are composable. When each φ_i is finite, the infinite concatenation of the infinite sequence $\langle\varphi_i\rangle_{i=0}^{\infty}$ of finite traces is

$$\varphi_0\varphi_1\varphi_2\ldots = \langle\alpha_0\alpha_1\ldots\alpha_n\ldots, (\biguplus_{i=0}^{\infty}\mathcal{F}_i, \mathcal{E}_0\cdot\mathcal{E}_1\cdot\mathcal{E}_2\cdot\;\ldots, \mathtt{i})\rangle.$$

When at least one of the traces φ_i is a partial or infinite trace, then the first such φ_i provides the relevant contextual information for the resulting trace; thus, if φ_k is the first nonfinite trace, then we define the infinite concatenation of the sequence $\langle\varphi_i\rangle_{i=0}^{\infty}$ to be

$$\varphi_0\varphi_1\varphi_2\ldots = \langle\alpha_0\alpha_1\ldots\alpha_k, (\mathcal{F}_k, \mathcal{E}_0\cdot\mathcal{E}_1\cdot\;\ldots\;\cdot\mathcal{E}_k, R_k)\rangle.$$

Once again, the definitions for finite and infinite iteration on trace sets follow directly from the definitions of concatenation and sequential composition. We define finite iteration on the trace set T by

$$T^* = \bigcup_{i=0}^{\infty} T^i,$$

where $T^0 = \{\langle\varepsilon_s, (\emptyset, \langle\emptyset\rangle, \mathtt{f})\rangle \mid s \in S\}$ and $T^{n+1} = T^n; T$. We define infinite iteration on trace set T as follows:

$$T^\omega = \{\varphi_0\varphi_1\ldots\varphi_k\ldots \mid (\forall i \geq 0.\varphi_i \in T)\ \&\ \mathit{composable}(\langle\varphi_i\rangle_{i=0}^{\infty})\}.$$

The semantics of loops again relies on the definitions of iteration:

$$\mathcal{T}_w[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!] = (\mathcal{T}_w[\![b]\!];\mathcal{T}_w[\![c]\!])^\omega \cup (\mathcal{T}_w[\![b]\!];\mathcal{T}_w[\![c]\!])^*;\mathcal{T}_w[\![\neg b]\!].$$

Guarded choice

The definition of guarded choice on weakly fair trace sets is a simple generalization of that for channel-fair traces: there is no need to keep track of the unused channels. For weakly fair trace sets T_1 and T_2, we define:

$$\begin{aligned} T_1 \,\Box\, T_2 \;=\; & \{\langle\alpha, (\mathcal{F}_1, \langle E_0 \cup E\rangle\mathcal{E}, R)\rangle \mid \langle\varepsilon_s\alpha, (\mathcal{F}_1, \langle E_0\rangle\mathcal{E}, R)\rangle \in T_1\ \&\ \langle\varepsilon_s, (\mathcal{F}_2, \langle E\rangle, \mathtt{p})\rangle \in T_2\} \\ \cup\; & \{\langle\alpha, (\mathcal{F}_2, \langle E_0 \cup E\rangle\mathcal{E}, R)\rangle \mid \langle\varepsilon_s\alpha, (\mathcal{F}_2, \langle E_0\rangle\mathcal{E}, R)\rangle \in T_2\ \&\ \langle\varepsilon_s, (\mathcal{F}_1, \langle E\rangle, \mathtt{p})\rangle \in T_1\}. \end{aligned}$$

We then define $\mathcal{T}_w[\![gc_1 \,\Box\, gc_2]\!] = \mathcal{T}_w[\![gc_1]\!] \,\Box\, \mathcal{T}_w[\![gc_2]\!]$.

Channel restriction

The weakly fair trace set $T\backslash h$ can be obtained from T by first removing those traces in which h is visible and then deleting all mentions of h from enabling sequences and fairness sets. For a set $\mathcal{F}$ of sets of directions, we define $\mathcal{F}\backslash h$ in the obvious way: $\mathcal{F}\backslash h = \{F\backslash h \mid F \in \mathcal{F}\}$. We then define

$$T\backslash h = \{\langle \alpha, (\mathcal{F}', \mathcal{E}\backslash h, R)\rangle \mid \langle \alpha, (\mathcal{F}, \mathcal{E}, R)\rangle \in T \ \&\ \mathcal{F}'\!\downarrow \supseteq (\mathcal{F}\backslash h)\!\downarrow \ \&\ h \notin \mathsf{chans}(\alpha)\},$$

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

Parallel composition

To define parallel composition for sets of weakly fair traces, we follow the same general approach taken in Chapter 3 for strongly fair traces. In particular, we define a relation *fairmerge* as the greatest fixed point of a functional

$$F(Y) = both \cdot Y \cup one,$$

and we introduce a predicate *mergeable* that indicates which mergings of computations are meaningful. Because the weak fairness of a computation can depend on the particular order in which independent actions occur, the *mergeable* predicate depends not only on the traces to be merged but also on the resulting trace. We therefore begin by defining *fairmerge*, deferring for now the question of which fair merges correspond to weakly fair computations.

The *fairmerge* relation for weakly fair traces is a simple generalization of the *fairmerge* relation for channel-fair traces: we need only omit the sets of unused channels and use fairness sets $\mathcal{F}$ of sets rather than fairness sets F of directions. For completeness, the definitions are included here, but with very few accompanying explanations.

For finite traces $\varphi_1 = \langle \alpha_1, (\mathcal{F}_1, \mathcal{E}_1, \mathtt{f})\rangle$ and $\varphi_2 = \langle \alpha_2, (\mathcal{F}_2, \mathcal{E}_2, \mathtt{f})\rangle$ such that $\alpha_1 \rfloor\!\lfloor \alpha_2$ is defined, we define

$$\varphi_1 \rfloor\!\lfloor \varphi_2 = \{\langle \alpha_1 \rfloor\!\lfloor \alpha_2, (\mathcal{F}, \mathcal{E}_1 \rfloor\!\lfloor \mathcal{E}_2, \mathtt{f})\rangle \mid \mathcal{F}\!\downarrow \supseteq (\mathcal{F}_1 \cup \mathcal{F}_2)\!\downarrow\}.$$

Each trace $\varphi \in \varphi_1 \rfloor\!\lfloor \varphi_2$ represents a transition sequence of a parallel command in which one component performs actions corresponding to φ_1, followed by the other component performing actions corresponding to φ_2. Likewise, for matching finite traces $\varphi_1 = \langle \alpha_1, (\mathcal{F}_1, \mathcal{E}_1, \mathtt{f})\rangle$ and $\varphi_2 = \langle \alpha_2, (\mathcal{F}_2, \mathcal{E}_2, \mathtt{f})\rangle$, $\varphi_1 \| \varphi_2$ is the set of traces corresponding to their synchronization at each step:

$$\varphi_1 \| \varphi_2 = \{\langle \alpha_1 \| \alpha_2, (\mathcal{F}, \mathcal{E}_1 \| \mathcal{E}_2, \mathtt{f})\rangle \mid \mathcal{F}\!\downarrow \supseteq (\mathcal{F}_1 \cup \mathcal{F}_2)\!\downarrow\}.$$

These two operations on traces form the basis for the set *both* $\subseteq \Phi_w \times \Phi_w \times \Phi_w$, whose triples reflect finite transition sequences that occur while both components remain active:

$$\begin{array}{rcl} both & = & \{(\varphi_1,\varphi_2,\varphi),(\varphi_2,\varphi_1,\varphi) \mid \varphi_1 = \langle \alpha,(\mathcal{F}_1,\mathcal{E}_1,\mathtt{f})\rangle \ \&\ \varphi_2 = \langle \beta,(\mathcal{F}_2,\mathcal{E}_2,\mathtt{f})\rangle \ \& \\ & & \qquad\qquad\qquad \mathsf{disjoint}(\alpha,\beta) \ \&\ \varphi \in \varphi_1 \rfloor\!\rfloor \varphi_2\} \\ & \cup & \{(\varphi_1,\varphi_2,\varphi) \mid \varphi_1 = \langle \alpha,(\mathcal{F}_1,\mathcal{E}_1,\mathtt{f})\rangle \ \&\ \varphi_2 = \langle \beta,(\mathcal{F}_2,\mathcal{E}_2,\mathtt{f})\rangle \ \& \\ & & \qquad\qquad \mathsf{disjoint}(\alpha,\beta) \ \&\ \mathsf{match}(\alpha,\beta) \ \&\ \varphi \in \varphi_1 \| \varphi_2\}. \end{array}$$

Once a component terminates successfully or becomes permanently blocked (modulo some set $\mathcal{F}$), the other component may proceed uninterrupted. Such situations are reflect by traces in the set $\varphi_1 \rfloor\!\rfloor \varphi_2$, where $\varphi_1 = \langle \alpha,(\mathcal{F}_1,\mathcal{E}_1,R)\rangle$ represents the active component and φ_2 is an empty finite trace $\langle \varepsilon_s,(\mathcal{F}_2,\langle\emptyset\rangle,\mathtt{f})\rangle$ or an empty partial trace $\langle \varepsilon_s,(\mathcal{F}_2,\langle E\rangle,\mathtt{p})\rangle$. When $\varphi_2 = \langle \varepsilon_s,(\mathcal{F}_2,\langle\emptyset\rangle,\mathtt{f})\rangle$, we define

$$\varphi_1 \rfloor\!\rfloor \varphi_2 = \{\langle \alpha \rfloor\!\rfloor \varepsilon_s,(\mathcal{F},\mathcal{E}\rfloor\!\rfloor\emptyset,R_1)\rangle \mid \mathcal{F}\!\downarrow \supseteq (\mathcal{F}_1 \cup \mathcal{F}_2)\!\downarrow\}.$$

When $\varphi_2 = \langle \varepsilon_s,(\mathcal{F}_2,\langle E\rangle,\mathtt{p})\rangle$, we define

$$\varphi_1 \rfloor\!\rfloor \varphi_2 = \begin{cases} \{\langle \alpha \rfloor\!\rfloor \varepsilon_s,(\mathcal{F},\mathcal{E}_1\rfloor\!\rfloor E,\mathtt{p})\rangle \mid \mathcal{F}\!\downarrow \supseteq (\mathcal{F}_1 \cup \mathcal{F}_2)\!\downarrow\}, & \text{if } R_1 \in \{\mathtt{f},\mathtt{p}\}, \\ \{(\alpha \rfloor\!\rfloor \varepsilon_s,(\mathcal{F},\mathcal{E}_1\rfloor\!\rfloor E,\mathtt{i})) \mid \mathcal{F}\!\downarrow \supseteq (\mathcal{F}_1 \cup \mathcal{F}_2)\!\downarrow\}, & \text{if } R_1 = \mathtt{i}. \end{cases}$$

These definitions provide the basis for the set *one* $\subseteq \Phi_w \times \Phi_w \times \Phi_w$, whose triples reflect transition sequences in which only one component remains active:

$$\begin{array}{rcl} one & = & \{(\varphi_1,\varphi_2,\varphi) \mid \varphi_1 = \langle \alpha,(\mathcal{F},\mathcal{E},R)\rangle \ \&\ \varphi_2 = \langle \varepsilon_s,(\mathcal{F},E',R)\rangle \ \& \\ & & \qquad\qquad\qquad \mathsf{disjoint}(\alpha,s) \ \&\ \varphi \in \varphi_1 \rfloor\!\rfloor \varphi_2\}. \end{array}$$

We can now also define *fairmerge* $= both^{\omega\bullet} \cup both^{*\bullet} \cdot one$, again with the intuition that the triple $(\varphi_1,\varphi_2,\varphi)$ is in *fairmerge* if and only if the trace φ is a fair merging and interleaving of the traces φ_1 and φ_2. In particular, just as for strong channel fairness, the triple (φ,φ',ψ) is in $both^{\omega\bullet}$ if and only if the traces φ, φ', and ψ can be written as

$$\varphi = \varphi_0 \cdot \varphi_1 \cdot \varphi_2 \cdot \varphi_3 \cdot \ldots, \qquad \varphi' = \varphi'_0 \cdot \varphi'_1 \cdot \varphi'_2 \cdot \varphi'_3 \cdot \ldots, \qquad \psi = \psi_0 \cdot \psi_1 \cdot \psi_2 \cdot \psi_3 \cdot \ldots,$$

such that each φ_i, φ'_i and ψ_i is finite, and each ψ_i is in the set $(\varphi_i \rfloor\!\rfloor \varphi'_i \ \cup\ \varphi'_i \rfloor\!\rfloor \varphi_i \ \cup\ \varphi_i \| \varphi_i)$. Likewise, the triple (φ,φ',ψ) is in $both^{*\bullet} \cdot one$ if and only if the traces φ, φ', and ψ can be written as

$$\varphi = \varphi_0 \cdot \varphi_1 \cdot \varphi_2 \cdot \varphi_3 \cdot \ldots\ \varphi_n, \quad \varphi' = \varphi'_0 \cdot \varphi'_1 \cdot \varphi'_2 \cdot \varphi'_3 \cdot \ldots\ \varphi'_n, \quad \psi = \psi_0 \cdot \psi_1 \cdot \psi_2 \cdot \psi_3 \cdot \ldots \cdot \psi_n,$$

such that each φ_i, φ_i' and ψ_i (for $i < n$) is a nonempty finite trace, each ψ_i (for $i < n$) is a member of the set $(\varphi_i \rfloor\!\lfloor \varphi_i' \ \cup\ \varphi_i' \rfloor\!\lfloor \varphi_i \ \cup\ \varphi_i \| \varphi_i)$, at least one of φ_n and φ_n' has form $\langle \varepsilon_s, \theta \rangle$, and ψ_n is a member of the set $(\varphi_n \rfloor\!\lfloor \varphi_n' \ \cup\ \varphi_n' \rfloor\!\lfloor \varphi_n)$.

However, defining the triples $(\varphi_1, \varphi_2, \varphi)$ of *fairmerge* is not enough for defining parallel composition on weakly fair trace sets. Despite being a fair merging of the two traces, φ may not represent a weakly fair computation: it is important to verify that the subprocesses that are blocked fairly along φ_1 and φ_2 are not enabled continuously along φ. Thus, we define a ternary predicate *mergeable* $\subseteq \Phi_w \times \Phi_w \times \Phi_w$ that not only takes into account the properties of φ_1 and φ_2, but also ensures that the resulting parallel trace φ satisfies all necessary process constraints.

A set F of directions is enabled for synchronization with the enabling set E—written $\mathsf{enabled}(F, E)$—if there exists a direction $d \in F$ such that $\mathsf{chan}(d) \in E$. Intuitively, the set F represents the set of directions enabled by some subprocess Q of a command c_1. If E is the enabling set of the parallel command $c_1 \| c_2$, then the process Q is enabled for synchronization with another process if and only if there is some direction $d \in Q$ such that the channel $\mathsf{chan}(d)$ appears in E.

A set F of directions is **blocked along** $\mathcal{E}$—written $\mathsf{blocked}(F, \mathcal{E})$—if $\mathcal{E}$ is finite or if there are infinitely many sets E along $\mathcal{E}$ such that F is not enabled for synchronization with E. That is, letting E_i represent the i^{th} element of the sequence $\mathcal{E}$, the predicate $\mathsf{blocked}(F, \mathcal{E})$ is defined as follows:

$$\mathsf{blocked}(F, \mathcal{E}) \iff \forall i \geq 0. \exists j > i. \neg \mathsf{enabled}(F, E_i).$$

We extend this notion of blocking to sets $\mathcal{F}$ of sets of directions as well: the set $\mathcal{F}$ is blocked along $\mathcal{E}$ if every member of $\mathcal{F}$ is blocked along $\mathcal{E}$. That is,

$$\mathsf{blocked}(\mathcal{F}, \mathcal{E}) \iff \forall F \in \mathcal{F}. \mathsf{blocked}(F, \mathcal{E}).$$

This notion of blocking forms the basis of the ternary predicate *mergeable*: for traces $\varphi_1 = \langle \alpha_1, (\mathcal{F}_1, \mathcal{E}_1, R_1) \rangle$, $\varphi_2 = \langle \alpha_2, (\mathcal{F}_2, \mathcal{E}_2, R_2) \rangle$, and $\varphi = \langle \alpha, (\mathcal{F}, \mathcal{E}, R) \rangle$,

$$\mathit{mergeable}(\varphi_1, \varphi_2, \varphi) \iff (R = \mathtt{f}) \text{ or } (R = \mathtt{p}) \text{ or } (\{\varepsilon\} \notin \mathcal{F}_1 \cup \mathcal{F}_2 \ \&\ \mathsf{blocked}(\mathcal{F}_1 \cup \mathcal{F}_2, \alpha)).$$

Thus the predicate $\mathit{mergeable}(\varphi_1, \varphi_2, \varphi)$ is true whenever φ is a finite or partial trace, or if no member of $\mathcal{F}_1 \cup \mathcal{F}_2$ is enabled for synchronization continuously along the infinite trace φ.

Finally, we define fair parallel composition on trace sets as follows:

$$\begin{aligned} T_1 \| T_2 \quad = \quad & \{\varphi \mid \varphi_1 = \langle \alpha, (\mathcal{F}_1, \mathcal{E}_1, R_1) \rangle \in T_1 \ \&\ \varphi_2 = \langle \beta, (\mathcal{F}_2, \mathcal{E}_2, R_2) \rangle \in T_2 \ \& \\ & (\varphi_1, \varphi_2, \varphi) \in \mathit{fairmerge} \ \&\ \mathit{mergeable}(\varphi_1, \varphi_2, \varphi)\}. \end{aligned}$$

It follows that $\mathcal{T}_w[\![c_1 \| c_2]\!] = \mathcal{T}_w[\![c_1]\!] \| \mathcal{T}_w[\![c_2]\!]$.

We can now give the denotational characterization of the weakly fair trace semantics $\mathcal{T}_w$ in its entirety. Once again, this characterization looks essentially the same as the denotational characterizations of the strongly fair and the strongly channel-fair semantics of previous chapters. The only differences are the sets of process constraints and the bookkeeping operations that underlie the new interpretations of the semantic operators.

Definition 6.3.1 The trace semantic function $\mathcal{T}_w : \mathsf{Com} \to \mathcal{P}(\Phi_w)$ is defined by:

$$
\begin{aligned}
\mathcal{T}_w[\![\mathsf{skip}]\!] &= \{\langle (s,\varepsilon,s),(\mathcal{F},\langle\emptyset,\emptyset\rangle,\mathtt{f})\rangle \mid s \in S \;\&\; \mathcal{F} \in \mathcal{P}_{\mathrm{fin}}(\mathcal{P}_{\mathrm{fin}}(\Delta))\} \\
&\cup\; \{\langle \varepsilon_s,(\mathcal{F},\langle\emptyset\rangle,\mathtt{p})\rangle \mid s \in S \;\&\; \mathcal{F}{\downarrow} \supseteq \{\{\varepsilon\}\}\} \\
\mathcal{T}_w[\![i{:=}e]\!] &= \{\langle (s,\varepsilon,[s|i=n]),(\mathcal{F},\langle\emptyset,\emptyset\rangle,\mathtt{f})\rangle \mid \\
&\qquad\qquad i \in \mathsf{dom}(s) \;\&\; \mathcal{F} \in \mathcal{P}_{\mathrm{fin}}(\mathcal{P}_{\mathrm{fin}}(\Delta)) \;\&\; (s,n) \in \mathcal{E}[\![e]\!]\} \\
&\cup\; \{\langle \varepsilon_s,(\mathcal{F},\langle\emptyset\rangle,\mathtt{p})\rangle \mid \mathsf{fv}[\![i{:=}e]\!] \subseteq \mathsf{dom}(s) \;\&\; \mathcal{F}{\downarrow} \supseteq \{\{\varepsilon\}\}\} \\
\mathcal{T}_w[\![c_1;c_2]\!] &= \mathcal{T}_w[\![c_1]\!];\mathcal{T}_w[\![c_2]\!] \\
\mathcal{T}_w[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] &= \mathcal{T}_w[\![b]\!];\mathcal{T}_w[\![c_1]\!] \cup \mathcal{T}_w[\![\neg b]\!];\mathcal{T}_w[\![c_2]\!] \\
\mathcal{T}_w[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!] &= (\mathcal{T}_w[\![b]\!];\mathcal{T}_w[\![c]\!])^\omega \cup (\mathcal{T}_w[\![b]\!];\mathcal{T}_w[\![c]\!])^*;\mathcal{T}_w[\![\neg b]\!] \\
\mathcal{T}_w[\![h?i]\!] &= \{\langle (s,h?n,[s|i=n]),(\mathcal{F},\langle\{h?\},\emptyset\rangle,\mathtt{f})\rangle \mid \\
&\qquad\qquad i \in \mathsf{dom}(s) \;\&\; n \in \mathbb{Z} \;\&\; \mathcal{F} \in \mathcal{P}_{\mathrm{fin}}(\mathcal{P}_{\mathrm{fin}}(\Delta))\} \\
&\cup\; \{\langle \varepsilon_s,(\mathcal{F},\langle\{h?\}\rangle,\mathtt{p})\rangle \mid i \in \mathsf{dom}(s) \;\&\; \mathcal{F}{\downarrow} \supseteq \{\{h?\}\}\} \\
\mathcal{T}_w[\![h!e]\!] &= \{\langle (s,h!n,s),(\mathcal{F},\langle\{h!\},\emptyset\rangle,\mathtt{f})\rangle \mid (s,n) \in \mathcal{E}[\![e]\!] \;\&\; \mathcal{F} \in \mathcal{P}_{\mathrm{fin}}(\mathcal{P}_{\mathrm{fin}}(\Delta))\} \\
&\cup\; \{\langle \varepsilon_s,(\mathcal{F},\langle\{h!\}\rangle,\mathtt{p})\rangle \mid \mathsf{fv}[\![e]\!] \subseteq \mathsf{dom}(s) \;\&\; \mathcal{F}{\downarrow} \supseteq \{\{h!\}\}\} \\
\mathcal{T}_w[\![g \to c]\!] &= \mathcal{T}_w[\![g]\!];\mathcal{T}_w[\![c]\!] \\
\mathcal{T}_w[\![gc_1 \,\Box\, gc_2]\!] &= \mathcal{T}_w[\![gc_1]\!] \,\Box\, \mathcal{T}_w[\![gc_2]\!] \\
\mathcal{T}_w[\![c_1 \| c_2]\!] &= \mathcal{T}_w[\![c_1]\!] \| \mathcal{T}_w[\![c_2]\!] \\
\mathcal{T}_w[\![c \backslash h]\!] &= \mathcal{T}_w[\![c]\!] \backslash h.
\end{aligned}
$$

$\diamond$

6.4 Final Comments on $\mathcal{T}_w$

The semantics $\mathcal{T}_w$ is sound with respect to all the (weakly fair equivalents of the) behaviors introduced in Chapter 4. However, it is not fully abstract with respect to any of them, for many of the same reasons that the channel-fair semantics fails to be fully abstract.

Despite the problems with full abstraction, the semantics $\mathcal{T}_w$ still sheds light on the problem of incorporating fairness assumptions into denotational semantics. It demonstrates the further applicability and robustness of the trace framework. Simply by replacing simple sets of actions by sets of sets of actions, we can parameterize and model weak process constraints instead of strong process constraints. Perhaps surprisingly, the weakly fair semantics retains a significant portion of the structure necessary for the channel-fair semantics, despite the underlying differences in the notions of fairness. In particular, both the channel-fair and the weakly fair semantics require sequences of enabling sets to account for the effect that the ordering of independent actions can have on the perceived fairness of a computation. Such sequences seem a natural consequence of fairness notions that are not equivalence robust.

Chapter 7

Hybrid Communicating Processes

Both Brookes' fair transition traces for shared-variable programs [Bro96b] and the fair traces for communicating processes in this dissertation play the same role in their respective semantics: they serve as abstract representations of fair computations. In each semantics, the meaning of a command is the set of traces corresponding to its fair computations (or, more accurately, corresponding to its fair transition sequences), and the structure of the traces reflects the communication features of the underlying paradigm. Transition traces represent transition sequences in which the external environment may alter the state between successive transitions. In contrast, the fair traces we developed for communicating processes represent transition sequences in which the environment never makes a state change and may interact with processes only by message passing. Because a process's external environment cannot alter its private state, state changes between steps of a fair trace are disallowed. The fair traces also require an additional contextual component that chronicles the relevant information for modeling fairness.

These two different kinds of trace structure are intuitively orthogonal, representing distinct but compatible aspects of computation. In particular, the two structures can be combined in a very intuitive way to yield a semantics for a hybrid language of processes that communicate through both message passing and shared memory. In this chapter, we introduce such a hybrid language, and we construct for it a semantics that incorporates assumptions of strong fairness. Horita, de Bakker, and Rutten define a fully abstract semantics for a similar hybrid language [HdBR94]; the semantics of this chapter generalizes their semantics by incorporating fairness assumptions.

The addition of shared-variable parallelism requires a generalization of parameterized strong fairness that accounts for state interruptions. By combining the shared-variable transition traces with the communicating processes' strongly fair traces in a natural way, we construct a hybrid trace semantics suitable for reasoning about the behavior of these hybrid processes. This semantics is also fully abstract, and the full-abstraction proof is a natural amalgam of the full-abstraction proofs of the original two semantics. The full-abstraction result indicates that the hybrid traces accurately capture the type of information necessary for reasoning about systems in which communication occurs both through message passing and through changes to shared memory.

The ease with which these two different semantics can be combined demonstrates the modularity of the semantic features and provides further evidence that the transition traces and the strongly fair traces accurately capture the important essence of fair computation for their underlying paradigms. The resulting hybrid semantics requires the same closure conditions for full abstraction as the original two semantics did, and the full-abstraction proof relies on the same observations and subsidiary lemmas that underly the full-abstraction proofs of the original semantics. Indeed, part of the value of the hybrid semantics' full-abstraction result is the ease with which we obtain it.

7.1 A Language of Hybrid Processes

The language of communicating processes that we have considered so far allows processes to communicate only through synchronous message passing. In this section, we add shared-variable parallelism and conditional critical regions to yield a hybrid language of processes that can communicate with one another both by message passing and by changes to shared memory. The resulting language captures the following abstract view of systems.

Intuitively, a system is a (possibly dynamic) collection of *realms*, with potentially multiple *threads of control* in each realm. Each realm has its own local state, and communication between threads in the same realm occurs via this shared local memory. In contrast, communication between threads in different realms occurs via message passing along named channels. For example, one can imagine several clusters of workstations connected to one another by high-speed networks, with processes on same-cluster workstations communicating across distributed shared memory and distant-cluster workstations communicating by messages across the network; each cluster is a realm, and the processes on the individual workstations are the threads of that realm. This view of systems encompasses (and generalizes) both the shared-memory and the communicating-process models. Shared-variable programs correspond to a single realm containing multiple threads; communicating processes correspond to multiple-realm systems in which each realm has precisely one thread of control.

This type of hybrid language supports the modeling of systems such as distributed databases, automated banking systems (i.e., ATMs), airline-reservation systems, and so on. These applications all share three common features: (1) various nodes can be physically distant from one another, making message passing the only viable communication mechanism; (2) "local" processes may require fine-grained sharing, making shared memory the most efficient mechanism; and (3) clients (either software or human) cannot or will not tolerate being ignored forever, making fairness an essential feature of the system.

The language's syntax and operational semantics are very similar to those described in Section 2.1 for the simple communicating processes.

7.1.1 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;
- Com, the set of *commands*, ranged over by c.

We again 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:

$$\begin{array}{rcl} g & ::= & h?i \mid h!e \\ gc & ::= & g \rightarrow c \mid gc_1 \,\Box\, gc_2 \\ c & ::= & \mathsf{skip} \mid i{:=}e \mid c_1; c_2 \mid \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ b\ \mathsf{do}\ c \mid gc \\ & & \quad \mid \mathsf{await}\ b\ \mathsf{then}\ c \mid c_1 \,|||\, c_2 \mid c_1 \| c_2 \mid c \backslash h \end{array}$$

We impose two additional syntactic constraints. First, in commands of the form

$$\mathsf{await}\ b\ \mathsf{then}\ c,$$

we require that the command c contains only assignments and skips. This requirement ensures that the command c terminates, and it represents a reasonable expectation of the scheduler: it is straightforward for a scheduler to disable all other processes to allow a single process to perform a finite series of assignments uninterrupted, but it is unreasonable for the scheduler to disable other processes permanently to allow a process to enter what may turn out to be an infinite loop. Moreover, this syntactic restriction does not restrict the expressive power of the language. Second, for commands of form $c_1 \| c_2$, we require that c_1 and c_2 have disjoint free identifiers. This restriction ensures that the processes associated with c_1 and c_2 maintain their own private state spaces: the only way that either component can affect the other's execution is through handshake communications.

$$\langle \bullet, s\rangle \mathsf{term} \qquad \frac{\langle c_1, s\rangle \mathsf{term} \quad \langle c_2, s\rangle \mathsf{term}}{\langle c_1 \,|||\, c_2, s\rangle \mathsf{term}}$$

$$\frac{\langle c, s\rangle \mathsf{term}}{\langle c\backslash h, s\rangle \mathsf{term}} \qquad \frac{\langle c_1, s_1\rangle \mathsf{term} \quad \langle c_2, s_2\rangle \mathsf{term}}{\langle c_1 \| c_2, s_1 \cup s_2\rangle \mathsf{term}} \text{ if } \mathsf{disjoint}(s_1, s_2)$$

Figure 7.1: The predicate term for hybrid processes.

$$\langle c, s\rangle \stackrel{\varepsilon}{\Longrightarrow} \langle c, s\rangle \qquad \frac{\langle c, s\rangle \stackrel{\varepsilon}{\Longrightarrow} \langle c_1, s_1\rangle \quad \langle c_1, s_1\rangle \stackrel{\lambda}{\longrightarrow} \langle c_2, s_2\rangle \quad \langle c_2, s_2\rangle \stackrel{\varepsilon}{\Longrightarrow} \langle c', s'\rangle}{\langle c, s\rangle \stackrel{\lambda}{\Longrightarrow} \langle c', s'\rangle}$$

Figure 7.2: Axiom and inference rule for the generalized relations $\stackrel{\lambda}{\Longrightarrow}$.

7.1.2 Operational semantics

The operational semantics makes use of a labeled transition system very similar to that used in Section 2.1. A configuration is a pair $\langle c, s\rangle$, where s is a state defined at least on the free identifiers of c. We use the place-holder $\bullet$ to represent termination, allowing (for example) configurations with form $\langle \bullet \| c, s\rangle$ or $\langle \bullet \,|||\, c, s\rangle$. A configuration $\langle c, s\rangle$ is *terminal* is the predicate $\langle c, s\rangle \mathsf{term}$ can be proved from the axiom and inference rules in Figure 7.1.

For simplicity, we assume that an evaluation semantics for boolean and arithmetic expressions is already known, and that expression evaluation is atomic, always terminates, and produces no side effects.

We again write $\langle c, s\rangle \stackrel{\lambda}{\longrightarrow} \langle c', s'\rangle$ to indicate that the command c is state s can perform an action labeled λ, leading to command c' in state s'. The transition relations $\stackrel{\lambda}{\longrightarrow}$ (and their generalized forms $\stackrel{\lambda}{\Longrightarrow}$) are characterized by a collection of axioms and inference rules. The transitions for sequential constructs, guards and guarded commands are identical to those introduced for communicating processes in Figures 2.4 and 2.5. The appropriate inference rules for the various parallel constructs appear in Figures 7.3. The inference rules for the generalized transition relations $\stackrel{\lambda}{\Longrightarrow}$ appear in Figure 7.2.

The transition rules for parallel composition highlight the distinction between the two different types of parallel commands, $c_1 \,|||\, c_2$ and $c_1 \| c_2$. The command $c_1 \,|||\, c_2$ represents the parallel composition of components that share a common state and communicate with one another only by changes to this shared state: transitions made by either component affect the global state, and handshakes between the two components are impossible. In contrast, the command $c_1 \| c_2$ represents the parallel composition of components with disjoint local states that communicate with one another only by message passing: transitions made independently by one

$$\frac{\langle b,s\rangle \longrightarrow^* \mathtt{tt} \quad \langle c,s\rangle \stackrel{\varepsilon}{\Longrightarrow} \langle c',s'\rangle\mathsf{term}}{\langle \mathsf{await}\ b\ \mathsf{then}\ c,s\rangle \xrightarrow{\varepsilon} \langle c',s'\rangle} \qquad \frac{\langle b,s\rangle \longrightarrow^* \mathtt{ff}}{\langle \mathsf{await}\ b\ \mathsf{then}\ c,s\rangle \xrightarrow{\varepsilon} \langle \mathsf{await}\ b\ \mathsf{then}\ c,s\rangle}$$

$$\frac{\langle c_1,s\rangle \xrightarrow{\lambda} \langle c_1',s'\rangle}{\langle c_1 \,|||\, c_2,s\rangle \xrightarrow{\lambda} \langle c_1' \,|||\, c_2,s'\rangle} \qquad \frac{\langle c_2,s\rangle \xrightarrow{\lambda} \langle c_2',s'\rangle}{\langle c_1 \,|||\, c_2,s\rangle \xrightarrow{\lambda} \langle c_1 \,|||\, c_2',s'\rangle}$$

$$\frac{\langle c_1,s_1\rangle \xrightarrow{\lambda} \langle c_1',s_1'\rangle}{\langle c_1\|c_2,s_1\cup s_2\rangle \xrightarrow{\lambda} \langle c_1'\|c_2,s_1'\cup s_2\rangle}\ \text{if } \mathsf{disjoint}(s_1,s_2)$$

$$\frac{\langle c_2,s_2\rangle \xrightarrow{\lambda} \langle c_2',s_2'\rangle}{\langle c_1\|c_2,s_1\cup s_2\rangle \xrightarrow{\lambda} \langle c_1\|c_2',s_1\cup s_2'\rangle}\ \text{if } \mathsf{disjoint}(s_1,s_2)$$

$$\frac{\langle c_1,s_1\rangle \xrightarrow{\lambda_1} \langle c_1',s_1'\rangle \quad \langle c_2,s_2\rangle \xrightarrow{\lambda_2} \langle c_2',s_2'\rangle}{\langle c_1\|c_2,s_1\cup s_2\rangle \xrightarrow{\varepsilon} \langle c_1'\|c_2',s_1'\cup s_2'\rangle}\ \text{if } \mathsf{match}(\lambda_1,\lambda_2)\ \ \&\ \mathsf{disjoint}(s_1,s_2)$$

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

Figure 7.3: Inference rules for the parallel constructs.

component affect only its local portion of the state, and the components may also handshake along a given channel.

The set of enabled directions for a configuration $\langle c,s\rangle$ is again given by the set

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

Note that, given the inference rules of Figure 7.3, the configuration $\langle \mathsf{await}\ b\ \mathsf{then}\ c,s\rangle$ always has an ε-transition enabled, regardless of the value of the expression of b in state s. Therefore the only configurations that can be blocked are those that are trying to communicate along restricted channels.

A **quasi-computation** of a command c from state s is a maximal sequence of transitions starting in $\langle c,s\rangle$ in which the state may be changed between successive transitions. For example, the following sequence of transitions is a quasi-computation of the command $(\mathsf{x}{:=}1;\mathsf{a!x})$ from state $[\mathsf{x}=0]$:

$$\langle \mathsf{x}{:=}1;\mathsf{a!x},\ [\mathsf{x}=0]\rangle \xrightarrow{\varepsilon} \langle \mathsf{a!x},\ [\mathsf{x}=1]\rangle\ \&\ \langle \mathsf{a!x},\ [\mathsf{x}=3]\rangle \xrightarrow{\mathsf{a!3}} \langle \bullet,\ [\mathsf{x}=3]\rangle.$$

We use the notation

$$\Big[\langle c_i, s_i\rangle \xrightarrow{\lambda_i} \langle c_{i+1}, s_i'\rangle\Big]_{i=0}^{k}, \qquad \Big[\langle c_i, s_i\rangle \xrightarrow{\lambda_i} \langle c_{i+1}, s_i'\rangle\Big]_{i=0}^{\infty}$$

to abbreviate (respectively) the finite quasi-computation

$$\langle c, s_0\rangle \xrightarrow{\lambda_0} \langle c_1, s_0'\rangle \;\&\; \langle c_1, s_1\rangle \xrightarrow{\lambda_1} \langle c_2, s_1'\rangle \;\&\; \cdots \;\&\; \langle c_k, s_k\rangle \xrightarrow{\lambda_k} \langle c_{k+1}, s_k'\rangle\mathsf{term},$$

and the infinite quasi-computation

$$\langle c, s_0\rangle \xrightarrow{\lambda_0} \langle c_1, s_0'\rangle \;\&\; \langle c_1, s_1\rangle \xrightarrow{\lambda_1} \langle c_2, s_1'\rangle \;\&\; \cdots \;\&\; \langle c_k, s_k\rangle \xrightarrow{\lambda_k} \langle c_{k+1}, s_k'\rangle \;\&\; \cdots.$$

A computation of c is a quasi-computation in which the state is never changed between successive transitions; that is, a computation is an *interference-free quasi-computation*.

Quasi-computations capture the intuition that a process's execution can be interrupted—and the state altered—by an external force (namely, the process's environment). In general, the computations of $c_1 \,|||\, c_2$ cannot be defined solely in terms of the computations of c_1 and c_2, precisely because of this interference. For example, consider the following two commands:

$$\begin{aligned} c_1 &\equiv \mathsf{x}{:=}0; \mathsf{if}\ \mathsf{x}=1\ \mathsf{then}\ \mathsf{y}{:=}0\ \mathsf{else}\ \mathsf{y}{:=}1, \\ c_2 &\equiv \mathsf{x}{:=}1. \end{aligned}$$

The parallel command $c_1 \,|||\, c_2$ has a computation that sets the value of y to 0, but there is no way to generate this computation by considering only computations of c_1 and c_2: the command c_2 does not access y, and every computation of c_1 sets y to 1.

However, the quasi-computations of $c_1 \,|||\, c_2$ can be defined in terms of the quasi-computations of c_1 and c_2. For example, combining the quasi-computations

$$\begin{aligned} \rho_1 \;=\; & \langle c_1,\ [\mathsf{x}=2\,\mathsf{y}=2]\rangle \xrightarrow{\varepsilon} \langle \mathsf{if}\ \mathsf{x}=1\ \mathsf{then}\ \mathsf{y}{:=}0\ \mathsf{else}\ \mathsf{y}{:=}1,\ [\mathsf{x}=0,\mathsf{y}=2]\rangle \\ \& \; & \langle \mathsf{if}\ \mathsf{x}=1\ \mathsf{then}\ \mathsf{y}{:=}0\ \mathsf{else}\ \mathsf{y}{:=}1,\ [\mathsf{x}=1,\mathsf{y}=2]\rangle \xrightarrow{\varepsilon} \langle \mathsf{y}{:=}0,\ [\mathsf{x}=1,\mathsf{y}=2]\rangle \\ \& \; & \langle \mathsf{y}{:=}0,\ [\mathsf{x}=1\,\mathsf{y}=2]\rangle \xrightarrow{\varepsilon} \langle \bullet,\ [\mathsf{x}=1\,\mathsf{y}=0]\rangle, \end{aligned}$$

and

$$\rho_2 \;=\; \langle c_2,\ [\mathsf{x}=2\,\mathsf{y}=2]\rangle \xrightarrow{\varepsilon} \langle \bullet, [\mathsf{x}=1\,\mathsf{y}=2]\rangle$$

yields a (quasi-)computation of $c_1 \| c_2$. It is this insight that drives the use of transition traces to model shared-variable programs.

7.2 Fairness for Hybrid Processes

All the notions of fairness for communicating processes introduced in Section 2.2 can be adapted for hybrid communicating processes. In this chapter, we shall consider the following version of strong fairness:

> Every process that is enabled infinitely often makes progress infinitely often.

To be precise, this notion of fairness constitutes strong fairness only because the operational semantics models blocking of await-statements by busy-waiting (i.e., by idle steps). As a result, the only "true" blocking of a process arises from unsatisfiable communication attempts. If, instead, the operational semantics represented blocking of await-statements by true blocking (that is, if we omitted the idle-step transition rule for await-statements), then the intended notion of fairness might be described more accurately as follows:

> Every continuously enabled process, and every process infinitely able to communicate, eventually makes progress.

That is, a process can block fairly on await-statements whose conditionals are not enabled continuously and on communications that are not enabled infinitely often. Imposing different fairness requirements on different types of transitions is not a new idea: Manna and Pnueli discuss the abstract construction of temporal proof systems predicated on identifying both strongly fair and weakly fair transition sets [MP83].

We introduce a parameterized form of strong fairness that is based on the parameterization of strong fairness given in Definition 3.1.2. This parameterization includes clauses for the shared-variable constructs await b then c and $c_1 \,|||\, c_2$. Moreover, because in general the computations of $c_1 \,|||\, c_2$ cannot be defined solely in terms of the computations of c_1 and c_2, we base this definition on quasi-computations.

Every infinite quasi-computation of await b then c involves the repeated evaluation of the boolean expression b in states that do not satisfy b. In every such quasi-computation, the single process repeatedly makes progress, and hence it is treated fairly. (Equivalently, an infinite quasi-computation indicates that the await-statement is infinitely often disabled and hence can block fairly under weak fairness.)

The requirements for fairness of the state-based parallel command $c_1 \,|||\, c_2$ are similar to (but simpler than) those for the message-based parallel command $c_1 \| c_2$. Intuitively, every quasi-computation ρ of $c_1 \,|||\, c_2$ arises from interleaving a quasi-computation ρ_1 of c_1 with a quasi-computation ρ_2 of c_2, and ρ inherits its fairness constraints from both ρ_1 and ρ_2. In particular, if ρ_1 is fair mod F_1 and ρ_2 is fair mod F_2, then ρ is fair mod $F_1 \cup F_2$, provided that the two quasi-computations respect the fairness constraints of one another. As with message-based parallel commands, neither component can use directions that appear in the other's fairness set, for the following reason. Intuitively, the fairness set F_1 represents the assumption that the command c_1 (and hence $c_1 \,|||\, c_2$) will appear in a context that restricts communication on the channels of F_1 without providing synchronization opportunities for them. If c_2 used a direction in F_1 infinitely often, then the eventual context would have to provide c_2 infinitely many synchronization opportunities for that direction, thereby offering c_1 those same opportunities as well. However, it is legitimate for one component to enable (and perhaps even use) infinitely often directions whose matching counterparts appear in the other's fairness set: for example, ρ_1 may enable the direction $\mathsf{a}!$ infinitely often even if $\mathsf{a}?$ is in F_2. Because there is no possibility of handshaking between c_1 and c_2, the directions enabled by one component do not affect the other's fairness constraints.

Definition 7.2.1 A quasi-computation ρ of command c is **fair modulo** F provided ρ satisfies one of the following conditions:

- ρ is a finite, successfully terminating quasi-computation;
- ρ is a partial quasi-computation whose final configuration is blocked modulo F;
- ρ is an infinite quasi-computation, c has form $(c_1; c_2)$ or $(\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)$, and the underlying infinite quasi-computation of c_1 or c_2 is fair mod F;
- ρ is an infinite quasi-computation, c has form $(\mathsf{while}\ b\ \mathsf{do}\ c)$ or $(g \to c)$, and all underlying quasi-computations of c are fair mod F;
- ρ is an infinite quasi-computation, c has form $(gc_1 \square gc_2)$, and the underlying quasi-computation of the selected gc_i is fair mod F;
- ρ is an infinite quasi-computation, and c has form $\mathsf{await}\ b\ \mathsf{then}\ c$;
- ρ is an infinite quasi-computation, c has form $c_1 \,|||\, c_2$, and there exists sets F_1 and F_2 and quasi-computations ρ_1 of c_1 and ρ_2 of c_2 such that ρ_1 is fair mod F_1, ρ_2 is fair mod F_2, $F \supseteq F_1 \cup F_2$, and neither ρ_i uses a direction in F_j $(i \neq j)$ infinitely often;
- ρ is an infinite quasi-computation, c has form $c' \backslash h$, and the underlying quasi-computation of c' is fair modulo $F \cup \{h!, h?\}$;
- ρ is an infinite quasi-computation, c has form $c_1 \| c_2$, and there exist sets F_1 and F_2 and quasi-computations ρ_1 of c_1 and ρ_2 of c_2 such that ρ_1 is fair mod F_1, ρ_2 is fair mod F_2, $F \supseteq F_1 \cup F_2$, ρ can be obtained by merging and synchronizing ρ_1 and ρ_2, neither ρ_i enables infinitely often any direction matching a member of F_j $(i \neq j)$, and neither ρ_i uses a direction in F_j infinitely often. $\diamond$

The following two examples, taken together, illustrate the difference in how fairness constraints are combined for the two different types of parallel composition. In particular, there are unfair computations of the command $c_1 \| c_2$ that, step for step, behave like fair computations of $c_1 \,|||\, c_2$.

Example 7.2.2 Consider the command $((R_1 \,|||\, R_2) \| R_3) \backslash \mathsf{a}$, where R_1, R_2 and R_3 are defined as follows:

$$R_1 \equiv \mathsf{a!0}, \qquad R_2 \equiv \mathsf{while\ true\ do\ a?x}, \qquad R_3 \equiv \mathsf{while\ true\ do\ a!1}.$$

1. The command R_1 has the partial (quasi-)computation
$$\rho_1 = \langle \mathsf{a!0}, [\mathsf{x} = 1] \rangle,$$
which is fair modulo $\{\mathsf{a!}\}$.

2. Let ρ_2 be the following infinite (quasi-)computation of R_2, which repeatedly receives the value 1 along channel a:
$$\rho_2 = \langle R_2, [\mathsf{x} = 1] \rangle \xrightarrow{\varepsilon} \langle \mathsf{a?x}; R_2, [\mathsf{x} = 1] \rangle \xrightarrow{\mathsf{a?1}} \langle R_2, [\mathsf{x} = 1] \rangle \xrightarrow{\varepsilon} \cdots$$
This computation is fair mod $\emptyset$.

3. Let ρ_3 be the following infinite (quasi-)computation in which R_3 repeatedly transmits the value 1 along channel a:
$$\rho_3 = \langle R_3, [\mathsf{y} = 1] \rangle \xrightarrow{\varepsilon} \langle \mathsf{a!1}; R_3, [\mathsf{y} = 1] \rangle \xrightarrow{\mathsf{a?1}} \langle R_3, [\mathsf{y} = 1] \rangle \xrightarrow{\varepsilon} \cdots .$$
This computation is also fair mod $\emptyset$.

4. Let ρ be the following infinite (quasi-)computation of $R_1 \,|||\, R_2$:
$$\rho = \langle R_1 \,|||\, R_2, [\mathsf{x} = 1] \rangle \xrightarrow{\varepsilon} \langle R_1 \,|||\, (\mathsf{a?x}; R_2), [\mathsf{x} = 1] \rangle \xrightarrow{\mathsf{a?1}} \langle R_1 \,|||\, R_2, [\mathsf{x} = 1] \rangle \xrightarrow{\varepsilon} \cdots$$
The computation ρ can be obtained by a (trivial) interleaving of ρ_1 and ρ_2. Because neither ρ_1 nor ρ_2 uses a direction in the other computation's fairness set, ρ inherits the fairness constraints of its underlying quasi-computations and is fair mod $\{\mathsf{a!}\}$.

   In particular, ρ is fair mod $\{\mathsf{a!}\}$ despite the fact that R_2 enables (and uses) the direction $\mathsf{a?}$ infinitely often: R_1 and R_2 are processes that can communicate with one another only through state changes.

5. The following computation of $((R_1 \,|||\, R_2) \parallel R_3)$, in which R_2 and R_3 repeatedly handshake on channel a, can be obtained by merging and synchronizing ρ and ρ_3:

$$\begin{array}{rcl} \langle (R_1 \,|||\, R_2) \parallel R_3, s\rangle & \xrightarrow{\varepsilon} & \langle (R_1 \,|||\, (\mathsf{a?x}; R_2)) \parallel R_3, s\rangle \\ & \xrightarrow{\varepsilon} & \langle (R_1 \,|||\, (\mathsf{a?x}; R_2)) \parallel (\mathsf{a!1}; R_3), s\rangle \\ & \xrightarrow{\varepsilon} & \langle (R_1 \,|||\, R_2) \parallel R_3, s\rangle \\ & \xrightarrow{\varepsilon} & \cdots \end{array}$$

   This computation is also fair modulo $\{\mathsf{a!}\}$.

6. As an immediate consequence, the following computation of $((R_1 \,|||\, R_2) \parallel R_3)\backslash\mathsf{a}$ is strongly fair:

$$\begin{array}{rcl} \langle ((R_1 \,|||\, R_2) \parallel R_3)\backslash\mathsf{a}, s\rangle & \xrightarrow{\varepsilon} & \langle ((R_1 \,|||\, (\mathsf{a?x}; R_2)) \parallel R_3)\backslash\mathsf{a}, s\rangle \\ & \xrightarrow{\varepsilon} & \langle ((R_1 \,|||\, (\mathsf{a?x}; R_2)) \parallel (\mathsf{a!1}; R_3))\backslash\mathsf{a}, s\rangle \\ & \xrightarrow{\varepsilon} & \langle ((R_1 \,|||\, R_2) \parallel R_3)\backslash\mathsf{a}, s\rangle \\ & \xrightarrow{\varepsilon} & \cdots \end{array}$$

◇

The following example, when compared with the previous example, illustrates how the type of communication possible between two components placed in parallel can affect the fairness of a given computation. In particular, the command $((R_1 \,|||\, R_2) \| R_2)\backslash\mathsf{a}$ has fair computations in which R_1 never makes a transition, but the command $((R_1 \| R_2) \| R_2)\backslash\mathsf{a}$ does not.

Example 7.2.3 Let R_1, R_2 and R_3 be defined as in the previous example, and consider the program $((R_1 \parallel R_2) \parallel R_3)\backslash\mathsf{a}$. That is, let R_1 and R_2 now represent processes that communicate with one another by message passing rather than by changes to the state.

Let ρ_2 be the computation of R_2 defined previously, and let $\rho_1' = \langle R_1, s\rangle$ be a trivial partial computation of R_1 with $\mathsf{x} \notin \mathsf{dom}(s)$. The following computation of $R_1 \| R_2$ that looks almost identical to the computation ρ of $R_1 \,|||\, R_2$, with state-based communication replaced by message-based communication:

$$\langle R_1 \| R_2, [s|\mathsf{x} = 1]\rangle \xrightarrow{\varepsilon} \langle R_1 \| (\mathsf{a?x}; R_2), [s|\mathsf{x} = 1]\rangle \xrightarrow{\mathsf{a?1}} \langle R_1 \| R_2, [s|\mathsf{x} = 1]\rangle \xrightarrow{\varepsilon} \cdots .$$

Unlike ρ, this computation is not fair modulo $\{\mathsf{a!}\}$: R_1 is enabled for synchronization with R_2 on channel a infinitely often and yet never makes progress. ◇

7.3 Strongly Fair, Hybrid-Trace Semantics

As hinted previously, we can define hybrid traces that combine the features of both the fair transition traces for shared-variable programs and the strongly fair traces for communicating processes. These traces provide the foundation for a trace semantics for the language of hybrid communicating processes introduced in Section 7.1.

The development of the hybrid traces and the hybrid-trace semantics is very similar to the development of the strongly fair trace semantics in Chapters 3 and 4. However, the order of presentation differs, in part because the previous chapters provide a useful foundation for concepts. For example, we can introduce the necessary closure conditions earlier, because the previous chapters make their purpose clearer. Additionally, the desire to retain as much structural similarity to both the transition traces and the strongly fair traces affects certain semantic decisions; it makes sense to explain these choices at the point of occurrence. For example, rather than constructing a semantics and then introducing a notion of behavior for which it can be made fully abstract, we begin by introducing a notion of behavior for which we will then construct a fully abstract semantics.

7.3.1 A busy-waiting behavior

We considered several different notions of strongly fair program behavior in Chapter 4. In this chapter, we consider a single notion of program behavior, namely the following busy-waiting behavior $\mathcal{W}$.

Definition 7.3.1 The *busy-waiting state trace* behavior $\mathcal{W} : \mathsf{Com} \to \mathcal{P}(S^\infty)$ is defined by:

$$\begin{array}{rcl} \mathcal{W}[\![c]\!] & = & \{s_0 s_1 \dots s_k \mid \langle c, s_0\rangle \overset{\varepsilon}{\Longrightarrow} \langle c_1, s_1\rangle \overset{\varepsilon}{\Longrightarrow} \cdots \overset{\varepsilon}{\Longrightarrow} \langle c_k, s_k\rangle \mathsf{term}\} \\ & \cup & \{s_0 s_1 \dots s_k (s_k)^\omega \mid \langle c_0, s_0\rangle \overset{\varepsilon}{\Longrightarrow} \langle c_1, s_1\rangle \overset{\varepsilon}{\Longrightarrow} \cdots \overset{\varepsilon}{\Longrightarrow} \langle c_k, s_k\rangle \mathsf{dead}\} \\ & \cup & \{s_0 s_1 \dots s_k \dots \mid \langle c_0, s_0\rangle \overset{\varepsilon}{\Longrightarrow} \cdots \overset{\varepsilon}{\Longrightarrow} \langle c_k, s_k\rangle \overset{\varepsilon}{\Longrightarrow} \cdots \text{ is strongly fair}\}. \end{array}$$

◇

The choice of this behavior is a pragmatic one: $\mathcal{W}$ corresponds both to the busy-waiting behavior $\mathcal{W}$ considered in Subsection 4.5.3 for communicating processes and to the behavior considered in [Bro96b] for shared-variable programs. As a result, constructing a semantics for reasoning about this notion of behavior should require minimal changes from the other two semantics.

As before, this behavior does not distinguish between deadlock and infinite idle chattering. Thus, for example, $\mathcal{W}[\![\mathsf{a!0\backslash a}]\!] = \mathcal{W}[\![\mathsf{while\ true\ do\ skip}]\!] = \{s^\omega \mid s \in S\}$. Of course, this identification is consistent with the interpretation of deadlock as busy-waiting.

7.3.2 Hybrid traces

We again employ the set of steps

$$\Sigma = S \times \Lambda \times S,$$

with the intuition that the step $(s, \lambda, s') \in \Sigma$ represents a transition of form $\langle c, s\rangle \stackrel{\lambda}{\Longrightarrow} \langle c', s'\rangle$. We define the set Σ^+ of finite traces by

$$\Sigma^+ = \{(s_0, \lambda_0, s'_0)(s_1, \lambda_1, s'_1)\dots(s_k, \lambda_k, s'_k) \mid k \geq 0 \;\&\; \forall i \leq k.(s_i, \lambda_i, s'_i) \in \Sigma\},$$

so that state changes between successive steps are permitted. Likewise, we define the set Σ^ω of infinite traces by

$$\Sigma^\omega = \{\sigma_0\sigma_1\dots\sigma_k\dots \mid \forall i \geq 0.\ \sigma_i \in \Sigma\},$$

and we let $\Sigma^\infty = \Sigma^+ \cup \Sigma^\omega$ be the set of all simple traces. These traces are an obvious combination of the shared-variable transition traces (which allow intermediate state changes) and the communicating-process traces (which include transition labels). Each simple trace $\alpha \in \Sigma^\infty$ now represents a quasi-computation, which allows us to relax the composability criteria for simple traces: every combination of traces α and β is composable, as is every infinite collection of simple traces.

Because we are interested in a behavior that models blocking by busy-waiting, we need only finite and infinite traces, with the latter representing both partial (i.e., blocking) computations and "true" infinite computations. To reason about strongly fair quasi-computations, we again need to augment infinite traces with fairness sets (representing process constraints) and sets of infinitely enabled directions. Similarly, because finite quasi-computations can be used to generate infinite quasi-computations, we augment finite traces with sets of enabled directions. Thus we again make use of the set

$$\Gamma = \mathcal{P}_{\mathrm{fin}}(\Delta) \times \mathcal{P}_{\mathrm{fin}}(\Delta) \times \{\mathtt{f}, \mathtt{i}\}$$

to provide the relevant contextual information for traces, and we define the set Φ of **fair hybrid traces** as

$$\Phi \;=\; \Sigma^\infty \times (\mathcal{P}_{\mathrm{fin}}(\Delta) \times \mathcal{P}_{\mathrm{fin}}(\Delta) \times \{\mathtt{f}, \mathtt{i}\}).$$

The finite trace $\langle \alpha, (F, E, \mathtt{f})\rangle$ represents a (necessarily fair mod F) successfully terminating quasi-computation with enabled directions E. Likewise, the infinite trace $\langle \alpha, (F, E, \mathtt{i})\rangle$ represents an infinite (or blocked), fair mod F quasi-computation with infinitely enabled directions E.

7.3.3 Closure conditions

Because the behavior $\mathcal{W}$ relies on the generalized transitions $\stackrel{\lambda}{\Longrightarrow}$, the semantics we develop must be able to introduce and absorb ε-transitions; that is, the semantics must be closed under stuttering and mumbling. However, we now need a more general notion of stuttering that permits the introduction of idle steps involving arbitrary states. We define the relation *stut* $\subseteq \Phi \times \Phi$ as follows:

$$stut \;=\; \{(\langle \alpha\beta, \theta\rangle,\ \langle \alpha(s, \varepsilon, s)\beta, \theta\rangle) \mid \alpha\beta \in \Sigma^{\infty} - \Sigma^{0}\ \&\ s \in S\}.$$

We also define a relation *mumb* $\subseteq \Phi \times \Phi$ that reflects the "absorption" of $\stackrel{\varepsilon}{\Longrightarrow}$ transitions:

$$\begin{array}{rcl} mumb & = & \{(\langle \alpha(s,\varepsilon,s')(s',\lambda,s'')\beta,\theta\rangle, \langle \alpha(s,\lambda,s'')\beta,\theta\rangle) \mid \alpha(s,\lambda,s'')\beta \in \Sigma^{\infty}\} \\ & \cup & \{(\langle \alpha(s,\lambda,s')(s',\varepsilon,s'')\beta,\theta\rangle, \langle \alpha(s,\lambda,s'')\beta,\theta\rangle) \mid \alpha(s,\lambda,s'')\beta \in \Sigma^{\infty}\}. \end{array}$$

These definitions are simplifications of the stuttering and mumbling relations introduced in Subsection 4.5.2.

Again letting $id = \{(\alpha,\alpha) \mid \alpha \in \Sigma^{\infty}\}$ be the identity relation on simple traces, we define $stut^{\infty}$ and $mumb^{\infty}$ to be the (respective) greatest fixed points of the functionals

$$F(R) = stut \cdot R \cup id, \qquad G(R) = mumb \cdot R \cup id,$$

so that $stut^{\infty} = stut^{\omega} \cup stut^{*} \cdot id$ and $mumb^{\infty} = mumb^{\omega} \cup mumb^{*} \cdot id$. Intuitively, the pair (φ, φ') is in $stut^{\infty}$ (respectively, $mumb^{\infty}$) if φ' can be obtained by inserting an idle step (respectively, eliding an ε-step) at some of the positions along φ's simple-trace component. Although the stuttering and mumbling steps can be applied at potentially infinitely many positions along a trace, they cannot be applied infinitely many times at any single position along a trace. Once again, this point is essential for preventing the accidental introduction of divergent traces.

To achieve full abstraction, we will also need the closure conditions superset, displacement, and contention as introduced in Chapter 4. Because these conditions act only on the contextual components of traces and not the simple-trace components, they translate directly to hybrid trace sets.

Definition 7.3.2 For a set T of hybrid traces, $T^{\dagger}_{*}$ is the smallest set containing T and satisfying the following closure conditions:

- *Superset*: If $\langle \alpha, (F, E, R)\rangle$ is in $T^{\dagger}_{*}$, $R \in \{\mathtt{f}, \mathtt{i}\}$, $F \subseteq F'$, and $E \subseteq E'$, then $\langle \alpha, (F', E', R)\rangle$ is in $T^{\dagger}_{*}$.
- *Displacement*: If $\langle \alpha, (F, E \cup X, R)\rangle$ is in $T^{\dagger}_{*}$, $R \in \{\mathtt{f}, \mathtt{i}\}$, $X \cap \mathsf{vis}(\alpha) = \emptyset$, and $\overline{X} \subseteq \mathsf{vis}(\alpha)$, then $\langle \alpha, (F, E, R)\rangle$ is in $T^{\dagger}_{*}$.

- *Contention*: If $\langle \alpha, (F \cup \{d\}, E, \mathtt{i}) \rangle$ and $\langle \alpha, (F, E \cup \{\bar{d}\}, \mathtt{i}) \rangle$ are both in $T^{\dagger}_{*}$, then $\langle \alpha, (F, E, \mathtt{i}) \rangle$ is also in $T^{\dagger}_{*}$.

- *Stuttering*: If φ is in $T^{\dagger}_{*}$ and $(\varphi, \varphi') \in \mathit{stut}^{\infty}$, then φ' is also in $T^{\dagger}_{*}$.

- *Mumbling*: If φ is in $T^{\dagger}_{*}$ and $(\varphi, \varphi') \in \mathit{mumb}^{\infty}$, then φ' is also in $T^{\dagger}_{*}$. $\diamond$

7.3.4 Hybrid trace semantics

We characterize a closed trace semantics $\mathcal{T}_h : \mathsf{Com} \to \mathcal{P}^{\dagger}_{*}(\Phi)$ as follows, building closure into the semantics from the beginning:

$$
\begin{aligned}
\mathcal{T}_h[\![c]\!] = (\{ \langle \mathsf{trace}(\rho),\ (F, \mathsf{en}(\rho), \mathtt{f}) \rangle \mid \\
& F \in \mathcal{P}_{\mathrm{fin}}(\Delta)\ \&\ \rho = \left[\langle c_i, s_i \rangle \xrightarrow{\lambda_i} \langle c_{i+1}, s'_i \rangle \right]_{i=0}^{k}\ \&\ \langle c_{k+1}, s'_k \rangle \mathsf{term}\ \} \\
\cup \{ \langle \mathsf{trace}(\rho)\alpha, (F, E, \mathtt{i}) \rangle \mid F \supseteq E = \mathsf{inits}(c_k, s_k)\ \&\ \varepsilon \notin E\ \&\ \neg \langle c_{k+1}, s'_k \rangle \mathsf{term} \\
& \rho = \left[\langle c_i, s_i \rangle \xrightarrow{\lambda_i} \langle c_{i+1}, s'_i \rangle \right]_{i=0}^{k}\ \&\ \alpha \in \{ (s, \varepsilon, s) \mid \mathsf{fv}[\![c_{k+1}]\!] \subseteq \mathsf{dom}(s) \}^{\omega}\ \} \\
\cup \{ \langle \mathsf{trace}(\rho),\ (F, \mathsf{en}(\rho), \mathtt{i}) \rangle \mid \\
& \rho = \left[\langle c_i, s_i \rangle \xrightarrow{\lambda_i} \langle c_{i+1}, s'_i \rangle \right]_{i=0}^{\infty}\ \text{is strongly fair mod } F \})^{\dagger}_{*}.
\end{aligned}
$$

The denotational characterization of this semantic function proceeds in the same manner as in previous chapters. In particular, most of the semantic operators can be defined as in in Section 3.3 and Subsection 4.5.3, with the only difference being the more liberal interpretation of the predicate *composable* (and the subsequent effects on traces).

For boolean expressions b, we define

$$\mathcal{T}_h[\![b]\!] = \{ \langle (s, \varepsilon, s),\ (F, \emptyset, \mathtt{f}) \rangle \mid (s, \mathtt{tt}) \in \mathcal{B}[\![b]\!]\ \&\ F \in \mathcal{P}_{\mathrm{fin}}(\Delta) \}^{\dagger}_{*},$$

so that each trace in $\mathcal{T}_h[\![b]\!]$ represents a sequence of idle steps, at least one of which occurs in a state that satisfies b.

The infinite quasi-computations of $\mathsf{await}\ b\ \mathtt{then}\ c$ are simply infinite sequences of idle transitions from states that fail to satisfy the boolean expression b. Thus the closed set of infinite traces of $\mathsf{await}\ b\ \mathsf{then}\ c$ can be given by $(\mathcal{T}_h[\![b]\!]^{\omega})^{\dagger}_{*}$. The command's finite quasi-computations reflect the intended atomicity of the command c: after some finite sequence of idle steps in which b is not satisfied, the command c is executed atomically from a state satisfying b. Thus the command's closed set of finite traces can be defined by

$$(\mathcal{T}_h[\![\neg b]\!]^*; \{\langle (s,\varepsilon,s'),\ (F,E,\mathtt{f})\rangle \in \mathcal{T}_h[\![c]\!] \mid (s,\mathtt{tt}) \in \mathcal{B}[\![b]\!]\})^\dagger_*,$$

which (due to stuttering) is equivalent simply to

$$\{\langle (s,\varepsilon,s'),\ (F,E,\mathtt{f})\rangle \in \mathcal{T}_h[\![c]\!] \mid (s,\mathtt{tt}) \in \mathcal{B}[\![b]\!]\}^\dagger_*.$$

It follows that

$$\mathcal{T}_h[\![\mathsf{await}\ b\ \mathsf{then}\ c]\!] = (\mathcal{T}_h[\![b]\!]^\omega)^\dagger_* \cup \{\langle (s,\varepsilon,s'),\ (F,E,\mathtt{f})\rangle \in \mathcal{T}_h[\![c]\!] \mid (s,\mathtt{tt}) \in \mathcal{B}[\![b]\!]\}^\dagger_*.$$

There are two types of parallel composition for the hybrid communicating processes: the state-based composition $c_1 \,|||\, c_2$, whereby the components communicate with one another via shared memory; and the message-based composition $c_1 \| c_2$, whereby the components communicate with one another via synchronous message passing. Both types of fair parallel composition can be defined on traces (and trace sets) through the introduction of fair-merge relations on triples of traces. We have already seen the *fairmerge* relation for the message-based communication in Chapters 3 and 4, which we again use[1] to define message-based parallel composition on hybrid trace sets T_1 and T_2:

$$T_1 \| T_2 = \{\varphi \mid \varphi_1 \in T_1\ \&\ \varphi_2 \in T_2\ \&\ \mathit{mergeable}(\varphi_1,\varphi_2)\ \&\ (\varphi_1,\varphi_2,\varphi) \in \mathit{fairmerge}\}.$$

We can likewise define a relation $\mathit{fairmerge}_{sv} \subseteq \Phi \times \Phi \times \Phi$, whose triples represent fair interleavings of steps made by processes that share a common state. Because each process alters the shared state, these triples do not need to propagate states in the way that the triples for message-based *fairmerge* do. Instead, we can define these triples using only trace concatenation, which performs the necessary bookkeeping operations on the contextual components of traces.

The set both_{sv} represents the interleavings of steps that occur while both components remain active. Intuitively, if the command c_1 can perform a finite transition sequence represented by φ_1 and the command c_2 can perform a finite transition sequence represented by φ_2, then the

[1] To be precise, we need to extend the underlying operations $\alpha \rfloor\!\rfloor \beta$ and $\alpha \| \beta$ to traces with intermediate state changes. However, these changes are straightforward: for example, if $\alpha = (s_0,\lambda_0,s_0')(s_1,\lambda_1,s_1')\ldots(s_k,\lambda_k,s_k')$ and the state s is disjoint from α, then we define

$$\alpha \rfloor\!\rfloor \varepsilon_s = (s_0 \cup s,\lambda_0,s_0' \cup s)(s_1 \cup s,\lambda_1,s_1' \cup s)\ldots(s_k \cup s,\lambda_k,s_k' \cup s).$$

Similarly, we define $\alpha \rfloor\!\rfloor \beta = (\alpha \rfloor\!\rfloor \varepsilon_t)(\beta \rfloor\!\rfloor \varepsilon_s)$, where s and t are the final state of α and initial state of β, respectively. The trace $\alpha \| \beta$ again represents the stepwise synchronization of α and β.

parallel command $c_1 \parallel\!\mid c_2$ can perform the corresponding finite transition sequences represented by $\varphi_1\varphi_2$ and $\varphi_2\varphi_1$. We therefore define the set $both_{sv}$ as follows:

$$both_{sv} = \{(\varphi_1, \varphi_2, \varphi_1\varphi_2),\ (\varphi_1, \varphi_2, \varphi_2\varphi_1) \mid \varphi_1, \varphi_2 \in \Phi_{\text{fin}}\}.$$

Once one component has terminated successfully, the remaining component may proceed uninterrupted. Such situations are captured by the set one_{sv}, whose triples correspond to the steps taken by one component after the other component has terminated. Letting ε represent the null trace, we define:

$$one_{sv} = \{(\varphi_1, \varepsilon, \varphi_1),\ (\varepsilon, \varphi_1, \varphi_1) \mid \varphi_1 \in \Phi\}.$$

We then define $fairmerge_{sv} = both_{sv}^{\omega} \cup both_{sv}^{*} \cdot one_{sv}$. The triple $(\varphi, \varphi', \psi)$ is in $both_{sv}^{\omega}$ if and only if the traces φ, φ', and ψ can be written as infinite concatenations of finite nonempty traces

$$\varphi = \varphi_0\ \varphi_1\ \varphi_2\ \varphi_3\ \ldots, \qquad \varphi' = \varphi_0'\ \varphi_1'\ \varphi_2'\ \varphi_3'\ \ldots, \qquad \psi = \psi_0\ \psi_1\ \psi_2\ \psi_3\ \ldots,$$

such that each ψ_i is either $\varphi_i\varphi_i'$ or $\varphi_i'\varphi_i$. Such triples represent the interleaving of two infinite traces. Likewise, the triple $(\varphi, \varphi', \psi)$ is in $both_{sv}^{*} \cdot one_{sv}$ if and only if the traces φ, φ', and ψ can be written as finite concatenations

$$\varphi = \varphi_0\ \varphi_1\ \varphi_2\ \varphi_3\ \ldots\ \varphi_n, \quad \varphi' = \varphi_0'\ \varphi_1'\ \varphi_2'\ \varphi_3'\ \ldots\ \varphi_n', \quad \psi = \psi_0\ \psi_1\ \psi_2\ \psi_3\ \ldots\ \psi_n,$$

such that each φ_i, φ_i' and ψ_i (for $i < n$) is a nonempty finite trace, at least one of φ_n and φ_n' is the null trace, and each ψ_i is either $\varphi_i\varphi_i'$ or $\varphi_i'\varphi_i$.

Finally, before defining state-based parallel composition on trace sets, we introduce a binary predicate $interleavable(\varphi_1, \varphi_2)$ that indicates when the traces φ_1 and φ_2 can be interleaved meaningfully (i.e., when they respect each other's fairness constraints). Following the criteria specified in Definition 7.2.1, we define the predicate $interleavable(\varphi_1, \varphi_2)$ for hybrid traces $\varphi_1 = \langle \alpha, (F_1, E_1, R_1) \rangle$ and $\varphi_2 = \langle \beta, (F_2, E_2, R_2) \rangle$ as follows:

$$interleavable(\varphi_1, \varphi_2) \iff (R_1 = \mathtt{f}) \text{ or } (R_2 = \mathtt{f}) \text{ or } (F_1 \cap \mathsf{vis}(\alpha_2) = \emptyset \ \&\ F_2 \cap \mathsf{vis}(\alpha_1) = \emptyset).$$

A finite trace can always be interleaved with any other trace. Moreover, two infinite traces φ_1 and φ_2 can be interleaved as long as neither trace uses infinitely often a direction that appears in the other's fairness set. We then define

$$T_1 \parallel\!\mid T_2 = \{\varphi \mid \varphi_1 \in T_1 \ \&\ \varphi_2 \in T_2 \ \&\ interleavable(\varphi_1, \varphi_2) \ \&\ (\varphi_1, \varphi_2, \varphi) \in fairmerge_{sv}\},$$

so that $\mathcal{T}_h[\![c_1 \parallel\!\mid c_2]\!] = (\mathcal{T}_h[\![c_1]\!] \parallel\!\mid \mathcal{T}_h[\![c_2]\!])_{*}^{\dagger}$.

In summary, we present the following complete denotational characterization of the semantic function $\mathcal{T}_h$. Other than the newly introduced clauses for the shared-variable constructs $\mathsf{await}\ b\ \mathsf{then}\ c$ and $c_1 \parallel\!\mid c_2$, this characterization looks identical to that given for the busy-waiting semantics $\mathcal{T}_{sb}$ in Subsection 4.5.3. Once again, the true differences are the underlying interpretations of the semantic operators: in particular, the semantic operators have been extended to operate on sets whose traces may contain intermediate state changes.

Definition 7.3.3 The trace semantic function $\mathcal{T}_h : \mathsf{Com} \to \mathcal{P}^{\dagger}_{*}(\Phi)$ is defined by:

$$
\begin{aligned}
\mathcal{T}_h[\![\mathsf{skip}]\!] &= \{\langle (s,\varepsilon,s),(F,\emptyset,\mathtt{f})\rangle \mid s\in S \;\&\; F\in \mathcal{P}_{\mathrm{fin}}(\Delta)\}^{\dagger}_{*}\\
\mathcal{T}_h[\![i{:=}e]\!] &= \{\langle (s,\varepsilon,[s|i=n]),(F,\emptyset,\mathtt{f})\rangle \mid \\
&\qquad \mathsf{fv}[\![i{:=}e]\!]\subseteq \mathsf{dom}(s) \;\&\; F\in \mathcal{P}_{\mathrm{fin}}(\Delta) \;\&\; (s,n)\in \mathcal{E}[\![e]\!]\}^{\dagger}_{*}\\
\mathcal{T}_h[\![c_1;c_2]\!] &= (\mathcal{T}_h[\![c_1]\!];\mathcal{T}_h[\![c_2]\!])^{\dagger}_{*}\\
\mathcal{T}_h[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] &= (\mathcal{T}_h[\![b]\!];\mathcal{T}_h[\![c_1]\!]\cup \mathcal{T}_h[\![\neg b]\!];\mathcal{T}_h[\![C_2]\!])^{\dagger}_{*}\\
\mathcal{T}_h[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!] &= ((\mathcal{T}_h[\![b]\!];\mathcal{T}_h[\![c]\!])^{\omega}\cup(\mathcal{T}_h[\![b]\!];\mathcal{T}_h[\![c]\!])^{*};\mathcal{T}_h[\![\neg b]\!])^{\dagger}_{*}\\
\mathcal{T}_h[\![\mathsf{await}\ b\ \mathsf{then}\ c]\!] &= (\mathcal{T}_h[\![b]\!]^{\omega})^{\dagger}_{*}\cup\{\langle (s,\varepsilon,s'),\ (F,E,\mathtt{f})\rangle\in \mathcal{T}_h[\![c]\!] \mid (s,\mathtt{tt})\in \mathcal{B}[\![b]\!]\}^{\dagger}_{*},\\
\mathcal{T}_h[\![h?i]\!] &= \{\langle (s,h?n,[s|i=n]),(F,\{h?\},\mathtt{f})\rangle \mid i\in \mathsf{dom}(s) \;\&\; n\in\mathbb{Z} \;\&\; F\in \mathcal{P}_{\mathrm{fin}}(\Delta)\}^{\dagger}_{*}\\
&\quad \cup\{\langle \alpha,(F,\{h?\},\mathtt{i})\rangle \mid \alpha\in\{(s,\varepsilon,s)^{\omega}\mid i\in\mathsf{dom}(s)\}^{\omega} \;\&\; F\supseteq\{h?\}\}^{\dagger}_{*}\\
\mathcal{T}_h[\![h!e]\!] &= \{\langle (s,h!n,s),(F,\{h!\},\mathtt{f})\rangle \mid (s,n)\in \mathcal{E}[\![e]\!] \;\&\; F\in \mathcal{P}_{\mathrm{fin}}(\Delta)\}^{\dagger}_{*}\\
&\quad \cup\{\langle \alpha,(F,\{h!\},\mathtt{i})\rangle \mid \alpha\in\{(s,\varepsilon,s)^{\omega}\mid \mathsf{fv}[\![e]\!]\subseteq\mathsf{dom}(s)\}^{\omega} \;\&\; F\supseteq\{h?\}\}^{\dagger}_{*}\\
\mathcal{T}_h[\![g\to c]\!] &= (\mathcal{T}_h[\![g]\!];\mathcal{T}_h[\![c]\!])^{\dagger}_{*}\\
\mathcal{T}_h[\![gc_1\,\square\, gc_2]\!] &= (\mathcal{T}_h[\![gc_1]\!]\,\square\,\mathcal{T}_h[\![gc_2]\!])^{\dagger}_{*}\\
\mathcal{T}_h[\![c_1 \,|||\, c_2]\!] &= (\mathcal{T}_h[\![c_1]\!]\,|||\,\mathcal{T}_h[\![c_2]\!])^{\dagger}_{*}\\
\mathcal{T}_h[\![c_1\|c_2]\!] &= (\mathcal{T}_h[\![c_1]\!]\|\mathcal{T}_h[\![c_2]\!])^{\dagger}_{*}\\
\mathcal{T}_h[\![c\backslash h]\!] &= (\mathcal{T}_h[\![c]\!]\backslash h)^{\dagger}_{*}.
\end{aligned}
$$

◇

7.4 Full Abstraction for the Behavior $\mathcal{W}$

The semantic $\mathcal{T}_h$ is fully abstract with respect to the busy-waiting trace behavior $\mathcal{W}$ introduced in Definition 7.3.1. Indeed, the full abstraction proof captures the flavor of the full abstraction proofs of both the transition trace semantics for shared-variable programs and the strongly fair trace semantics for communicating processes.

Proposition 7.4.1 *The closed trace semantics $\mathcal{T}_h$ is inequationally fully abstract with respect to $\mathcal{W}$: for all commands c and c',*

$$\mathcal{T}_h[\![c]\!]\subseteq \mathcal{T}_h[\![c']\!] \iff \forall P[-].\,\mathcal{W}[\![P[c]]\!]\subseteq \mathcal{M}[\![W[c']]\!].$$

Proof: The forward implication follows from the compositionality of $\mathcal{T}_h$, the monotonicity of operations on trace sets, and the fact that, when $\mathcal{T}_h[\![c]\!] \subseteq \mathcal{T}_h[\![c']\!]$,

$$\begin{array}{rcl}
\mathcal{W}[\![P[c]]\!] & = & \{\mathsf{states}(\alpha) \mid \alpha \in \mathcal{T}_h[\![P[c]]\!] \ \&\ \mathsf{chans}(\alpha) = \{\varepsilon\}\} \\
& \cup & \{\mathsf{states}(\alpha) \mid \exists \langle \alpha, (\emptyset, E, \mathtt{i})\rangle \in \mathcal{T}_h[\![P[c]]\!] \,.\, \mathsf{chans}(\alpha) = \{\varepsilon\} \ \&\ \mathsf{intfree}(\alpha)\} \\
& \subseteq & \{\mathsf{states}(\alpha) \mid \alpha \in \mathcal{T}_h[\![P[c']]\!] \ \&\ \mathsf{chans}(\alpha) = \{\varepsilon\}\} \\
& \cup & \{\mathsf{states}(\alpha) \mid \exists \langle \alpha, (\emptyset, E, \mathtt{i})\rangle \in \mathcal{T}_h[\![P[c']]\!] \,.\, \mathsf{chans}(\alpha) = \{\varepsilon\} \ \&\ \mathsf{intfree}(\alpha)\} \\
& = & \mathcal{W}[\![P[c']]\!].
\end{array}$$

For the reverse implication, consider $\varphi = \langle \alpha, (F, E, R)\rangle$ in $\mathcal{T}_h[\![c]\!] - \mathcal{T}_h[\![c']\!]$. Because the analysis differs only slightly depending on whether φ is finite or infinite, we consider both cases together. The distinguishing context we construct combines features from the full abstraction proofs for both strongly fair communicating processes and weakly fair shared-variable programs.

Let $\langle \alpha, (F_1, E_1, R)\rangle, \ldots, \langle \alpha, (F_m, E_m, R)\rangle$ be the (necessarily finite number of) minimal α-traces in $\mathcal{T}_s^{\dagger}[\![c']\!]$. We define sets X and Y of directions, and a simple context $Q[-]$, as follows:

- If $R = \mathtt{f}$, then we can assume without loss of generality that $F_i = \emptyset$ for each i. Closure under superset ensures that $E_i \not\subseteq E$ for each $i \leq m$; thus for each i we can choose a direction $d_i \in E_i - E$. We let $X = \emptyset$ and $Y = \{d_i \mid 1 \leq i \leq m\}$, and we let $Q[-]$ be the context $Q[-] \equiv \mathsf{while}\ \mathsf{true}\ \mathsf{do}\ [-]$.
- If $R = \mathtt{i}$, then Lemma 4.4.5 ensures a conflict-free resolution $\mathcal{R}$ of $\mathcal{T}_h[\![c']\!]$ for φ. We define

  $$X = \{d_i \mid 1 \leq i \leq m \ \&\ \mathcal{R}(\varphi_i) = (d_i, \mathtt{F})\},$$
  $$Y = \{d_i \mid 1 \leq i \leq m \ \&\ \mathcal{R}(\varphi_i) = (d_i, \mathtt{E})\}.$$

  Because $\mathcal{R}$ is conflict-free, it follows that $\neg\mathsf{match}(X, Y)$. We let $Q[-]$ be the simple context $[-]$.

Intuitively, the context Q is the minimal context necessary for generating an infinite computation from the trace α. Every direction in X represents a direction that is enabled by a permanently blocked process along some quasi-computation of $Q[c']$. Every direction in Y is a direction enabled infinitely along some quasi-computation of $Q[c']$ and yet enabled only finitely often along φ (or φ^{ω}, if α is finite). Moreover, every computation of $Q[c']$ with the simple trace α (or α^{ω}) must have an infinitely enabled direction of Y or a blocked process with an enabled direction in X.

Let x and y be fresh identifiers, and define sets of "matching guards" for X and Y as follows:

$$G_{\mathsf{x}} = \{h!0 \mid h? \in X\} \cup \{h?\mathsf{x} \mid h! \in X\}, \quad G_{\mathsf{y}} = \{h!0 \mid h? \in Y\} \cup \{h?\mathsf{y} \mid h! \in Y\}.$$

In the full abstraction proofs of Chapter 4, this analysis sufficed for constructing the distinguishing context: we placed $Q[-]$ in parallel with commands $\mathsf{Guess}(H, G_{\mathsf{x}}, \mathsf{f1})$ and $\sum_{g \in G_{\mathsf{y}}} g \to \mathsf{f2}{:=}1$, and only $Q[c]$ could perform the transitions of α without setting either flag $\mathsf{f1}$ or $\mathsf{f2}$ to 1. However, we now also have to consider the the possibility that α may not be interference-free. For example, if

$$\alpha = (s_0, \lambda_0, s_0')(s_1, \lambda_1, s_1') \ldots (s_k, \lambda_k, s_k'),$$

our distinguishing context must provide a way to "fill in the gaps" and convert each state s_i' into s_{i+1}.

Let $x_1, \ldots, x_n$ be the free identifiers of c and c', and let $h_1, \ldots, h_k$ be the channel names appearing in c. Without loss of generality, we can assume that each state appearing along α is defined on precisely the identifiers $x_1, x_2, \ldots, x_n$. Let $\mathsf{f1}, \mathsf{f2}, \mathsf{ct}, \mathsf{t}, y_1, \ldots, y_n, z_1, \ldots, z_n$ be fresh identifiers.

Let $\tilde{x}{:=}\tilde{y}$ abbreviate the command $x_1{:=}y_1; x_2{:=}y_2; \cdots; x_n{:=}y_n$, let $\tilde{x}{:=}\tilde{0}$ abbreviate the command $x_1{:=}0; x_2{:=}0; \cdots; x_n{:=}0$, and let $\tilde{x} = \tilde{y}$ represent the boolean expression

$$(x_1 = y_1)\ \&\ (x_2 = y_2)\ \&\ \cdots\ \&\ (x_n = y_n).$$

Let $\mathsf{Choose}(\tilde{y}, \mathsf{t})$ be the following command:

$$\begin{array}{ll} \tilde{y}{:=}\tilde{0}; \mathsf{t}{:=}0;\ (& \mathsf{t}{:=}1 \\ \quad\quad\quad\quad\quad \Vert\!| & \mathsf{while}\ \mathsf{t} = 0\ \mathsf{do}\ y_1{:=}y_1 + 1 \\ \quad\quad\quad\quad\quad \Vert\!| & \mathsf{while}\ \mathsf{t} = 0\ \mathsf{do}\ y_2{:=}y_1 + 2 \\ \quad\quad\quad\quad\quad \Vert\!| & \cdots \\ \quad\quad\quad\quad\quad \Vert\!| & \mathsf{while}\ \mathsf{t} = 0\ \mathsf{do}\ y_n{:=}y_n + 2 \\ \quad\quad\quad\quad\quad) & \end{array}$$

Intuitively, the command $\mathsf{Choose}(\tilde{y}, \mathsf{t})$ can "guess" states: for every state s with domain $\{x_1, \ldots, x_n\}$, $\mathsf{Choose}(\tilde{y}, \mathsf{t})$ has a successfully terminating computation whose final state assigns to variable y_i the value of x_i in state s.

Finally, we construct the following command $\mathsf{CloseGap}(\tilde{x}, \tilde{y}, \tilde{z}, \mathsf{t}, \mathsf{ct})$, which provides the mechanism to close α's state gaps:

$$\begin{array}{l} \mathsf{while\ \ true\ do} \\ \quad\quad (\ \mathsf{Choose}(\tilde{y}, \mathsf{t}); \mathsf{Choose}(\tilde{z}, \mathsf{t}); \\ \quad\quad\quad \mathsf{ct}{:=}\mathsf{ct} + 1; \\ \quad\quad\quad \mathsf{await}\ (\tilde{\mathsf{x}} = \tilde{\mathsf{y}})\ \mathsf{then}\ \tilde{\mathsf{x}}{:=}\tilde{\mathsf{y}} \\ \quad\quad) \end{array}$$

Intuitively, this command has a computation that, on its i^{th} iteration through the loop, guesses the values of $\{x_1, \dots, x_n\}$ in state s'_i (storing them in $\{y_1, \dots, y_n\}$) and in state s_{i+1} (storing them in $\{z_1, \dots, z_n\}$), waits until state s'_i is reached, and then changes the state from s'_i to s_{i+1} atomically. The identifier ct indicates which state gap is being closed: ct changes value from i to $i+1$ on the iteration that closes the gap between states s'_i and s_{i+1}.

We can now define the distinguishing context $P[-]$ as follows:

$$\left[(Q[-] \mathbin{|||} \mathsf{CloseGap}(\tilde{x}, \tilde{y}, \tilde{z}, \mathsf{t}, \mathsf{ct})) \parallel \mathsf{Guess}(H, G_{\mathsf{x}}, \mathsf{f1}) \;\parallel\; \sum_{g \in G_{\mathsf{y}}} g \to \mathsf{f2}{:=}1) \right] \backslash h_1 \backslash \cdots \backslash h_k.$$

$\mathcal{M}[\![P[c]]\!]$ has a behavior corresponding to α in which neither $\mathsf{f1}$ nor $\mathsf{f2}$ is ever set to 1. In contrast, every behavior of $\mathcal{M}[\![P[c']]\!]$ corresponding to α must eventually set at least one of the flags $\mathsf{f1}$ and $\mathsf{f2}$ to 1. ∎

This full abstraction result is meaningful not only for what it says about the utility of the semantics $\mathcal{T}_h$ but also for what it says about the robustness and applicability of the general trace framework. By combining two fully abstract semantics for different languages in a natural way, we construct a third semantics that is fully abstract for a hybrid language based on the original two languages. Moreover, the full-abstraction proof for the hybrid semantics arises as a natural combination of the two original full-abstraction proofs.

Chapter 8

Conclusions

In this dissertation, I have described a general, trace-based, denotational framework for modeling fair communicating processes. In this chapter, I discuss some connections between this framework and related work, as well as some directions for future work. I conclude with a summary of the contributions of this thesis and some final thoughts.

8.1 Related Work

The framework that I have described builds on a long history of trace models for concurrency [Par79, Bro96b, Hoa81, BHR84, BR84, Hen85, Jon87, Rus90, Jos92, JJH90]. In fact, my fair trace semantics can be viewed as extensions to both the CSP failures model and the CCS acceptance-tree model for dealing with fair, infinite computations. Of course, I am not the first to provide extensions for modeling fairness.

For dataflow and asynchronous networks, Jonsson provides a fully abstract trace model that incorporates assumptions of weak fairness [Jon94]. By modeling channels as transition systems with their own fairness constraints and limiting use of each channel, he ensures that every process makes progress if enabled infinitely often. Essential for modeling weak fairness are the assumptions that each channel is used for input by at most one node, that each channel is used for output by at most one node, and that no channel is used for both input and output by any node.

In [Hen87], Hennessy extends acceptance trees with limit points that indicated which infinite paths were fair. The notion of fairness incorporated into this semantics is a form of unconditional fairness: an infinite computation is considered fair if *every* process makes infinitely many transitions along that computation. In particular, certain commands—such as $(\mathsf{skip} \parallel \mathsf{while\ true\ do\ skip})$—do not have any fair computations: skip cannot make infinitely many transitions and `while true do skip` can never terminate. Brookes adds infinite traces to Hoare's trace semantics [Hoa81] to model fair, infinite computations [Bro94], adapting Park's fairmerge operator [Par79] to handle the potential of synchronization between parallel components. The result of these modifications is a semantics suited for reasoning about a slightly more liberal notion of fairness: an infinite computation is considered fair if every process either makes infinitely many transitions or terminates successfully.

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. As a result, to support reasoning about these types of fairness, it is essential to augment traces with additional information about the types of communications possible along the computation.

This observation, which underlies my framework, also provides a foundation for Darondeau's fully abstract, strongly fair semantics for a stateless, CCS-like language [Dar85]. In this semantics, the meaning of a term is a set of *histories*, each having form $\langle \delta, \rho, d \rangle$: ρ is a (finite or infinite) trace of a program's interactions with its environment, δ is a set containing the actions on which processes are blocked, and d (which is disjoint from δ) is a set containing the actions enabled infinitely often (but not involved in blocking) along the trace ρ. Generally speaking, an infinite trace $\langle \alpha, (F, E, \mathtt{i}) \rangle$ in my framework corresponds to a history $\langle F, \alpha, (E - F) \rangle$.

I discovered Darondeau's work late in the process of writing this dissertation, two years after first developing the strongly fair semantics of Chapter 3. Although developed independently, my framework places Darondeau's work in a more general light. In addition to its statelessness, the language he considers has no notion of sequential composition and only a very limited form of recursion based on iteration: the iterative constructs generate only infinite computations, and no other language constructs can appear in the context of these iterative constructs. Moreover, my 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. In contrast, Darondeau provides hints of the source of the fairness-related sets δ and d, but he never presents an exact explanation of what these sets represent. As a result, it is unclear how he would extend his approach to other notions of fairness.

8.2 Directions for Future Work

Throughout this dissertation, we have focused on a simple language of communicating processes. However, we have omitted several common language features, including recursion and more general message types. We now consider these features briefly in turn.

To give semantics to loops, we introduced finite and infinite iteration on trace sets. Although we did not state so explicitly, the meaning of the loop $(\mathsf{while}\ b\ \mathsf{do}\ c)$ could be formulated equivalently as the greatest fixed point of the functional

$$F(X) = ([\![b]\!];[\![c]\!]);X \cup [\![\neg b]\!],$$

as in [Bro96b]. Likewise, for general recursive constructs such as **rec** $x.t$, we should again be able to use greatest fixed points, using functionals of the general form

$$F'(X) = [\![t]\!]\ \rho[X/x],$$

where ρ is a fixed environment for the free variables of t other than the recursion variable X. However, this type of characterization not only requires the introduction of environments (as in "standard" denotational semantics) but also obscures the understanding of the role of fairness constraints. For example, how do the fair computations of $(\mathbf{rec}\, x.\mathsf{a}.x)\|(\mathbf{rec}\, x.\mathsf{b}.x)$ compare with the fair computations of $\mathbf{rec}\ x.(\mathsf{a}.x\|\mathsf{b}.x)$? They share the same finite prefixes, and yet the latter generates significantly more processes dynamically (each with its own fairness constraints).

The only types of messages allowed in the simple language we have considered are integer values, but it is easy to imagine more general message types. For example, the π-calculus [MPW92] allows the transmission of *names*, which may refer to links (i.e., channels) between processes; similarly, there are higher-order calculi that allow processes themselves to be sent as messages [Tho89]. In these situations, messages can alter the communication topology dynamically. Related to this situation is the potential of procedures that accept channel names or processes as parameters. In both cases, an accurate semantics must account for the dual role of channel names: they not only refer to the communication links between processes, they also provide necessary information about fairness constraints. I expect that environments that reflect this dual role of channels can be introduced in a straightforward way.

There are several results that classify the relative expressive power of various fair-merge [PS88b, PS88a] and fair-choice [MPS88] operators for dataflow networks, as well as the power of different delay operators for SCCS [CP91]. A similar question arises in the setting of communicating processes. As mentioned in Chapter 2, there are programs that terminate under one notion of fairness that do not necessarily terminate under other notions of fairness. However, is the hierarchy also one of implementability? For example, can a weakly fair scheduler be used to implement a strongly fair scheduler, and (if so) what type of language features are necessary?

Throughout this dissertation, I have hinted how the trace framework might support reasoning about fair behavior, but the question remains: how does this framework help the practitioner? Fairness is an abstraction introduced to support reasoning about program behavior. While the framework provides a way to model fairness compositionally, the model is useful only if it helps the task of reasoning about programs. An important open question is: what type of insight does this framework provide the programmer, either directly or indirectly? Are there particular structuring techniques that facilitate reasoning about programs under fairness assumptions? For example, we saw that modeling weak fairness required a significant amount of program semantic structure: are there certain classes of programs for which modeling weak fairness become simpler?

8.3 Thesis Contributions

In this dissertation, I have presented a general framework for constructing denotational semantics that incorporate fairness assumptions for communicating processes. The primary units of this framework are *fair traces*, which are abstract representations of program computations. The meaning of a program is the set of fair traces that correspond to its fair computations; the semantic operators on trace sets correspond intuitively to the operational behavior of the program constructs. The use of traces provides an intuitive connection between a program's operational behavior and its semantic meaning.

This framework is the primary contribution of this thesis: it provides a general, extendible, modular approach for constructing semantics that support reasoning about fair program behavior. To demonstrate the robustness of the framework, I have focused on a single language and constructed for it several semantics that incorporate different types of fairness assumptions. In the process, I developed:

- Several fully abstract, strongly fair denotational semantics for state-based communicating processes.
- A sound channel-fair denotational semantics for communicating processes.
- A sound weakly fair denotational semantics for communicating processes.

I also constructed a fully abstract, strongly fair semantics for a language that combines both synchronous message passing and shared-variable parallelism. Figure 8.1 summarizes these semantics, highlighting the structure of the fair traces for each semantics.

Through these semantics, the framework also provides the following secondary contributions:

- The introduction and formalization of *parameterized fairness*, which provides a compositional characterization of fairness.

  The definition of parameterized strong process fairness, and the related parameterized definitions for channel fairness and weak process fairness, were introduced to permit a denotational characterization of fairness. However, they are also suitable for purely operational reasoning, allowing syntax-directed reasoning about program behavior.

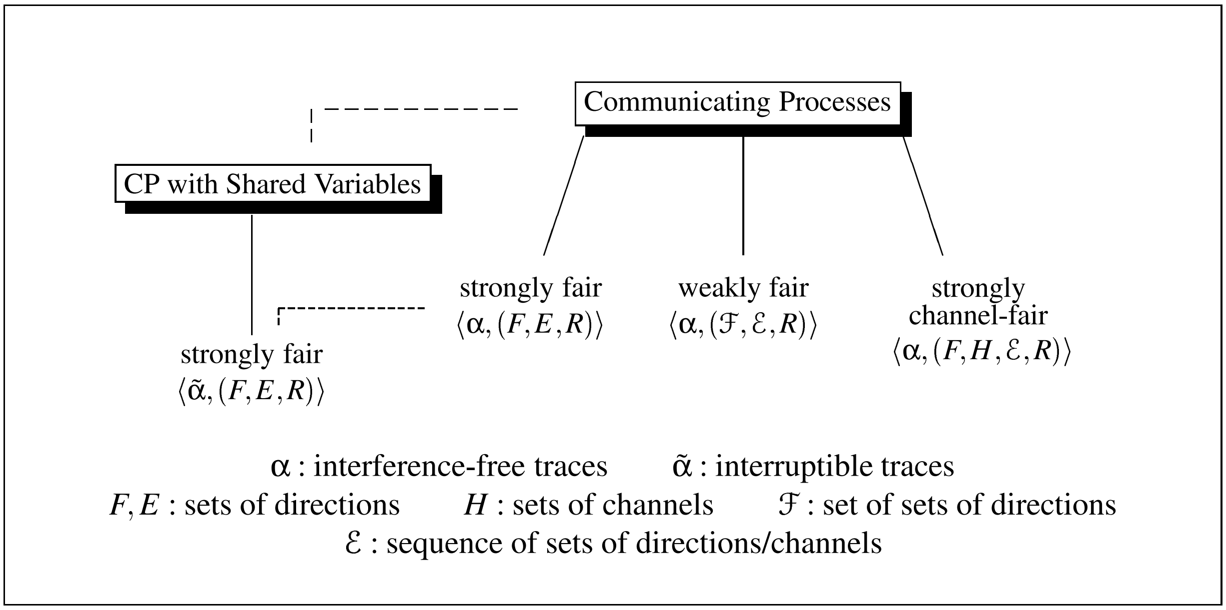

Figure 8.1: Summary of semantics in the fair-trace framework.

- Implicit comparison of several different fairness assumptions.

  When taken together, the strongly process-fair semantics, the strongly channel-fair semantics, and the weakly process-fair semantics provide an interesting side-by-side comparison of some of the notions of fairness commonly considered for communicating processes. Because these semantics have all been constructed for the same language, they highlight both the differences in semantic structure that these various assumptions require and the effects that these fairness assumptions have on program behavior.

  In particular, the channel-fair and weakly fair semantics require significantly more structure than the strongly fair semantics does, reflecting their lack of equivalence robustness [AFK88]. The need to keep track of the communications enabled at each step not only complicates the semantic models but also suggests that perhaps these notions fairness do not provide useful and practical abstractions to the programmer.

- Fully abstract semantics for strong process fairness.

  The full-abstraction results validate the suitability of the strongly fair traces for reasoning about strongly fair behavior. In particular, they indicate that the strongly fair traces provide precisely the necessary information for reasoning about strongly fair program behavior in a compositional, syntax-directed way.

  The fully abstract semantics also provide interesting technical results, indicating that fairness can be modeled accurately in spite of the expected difficulties.

8.4 Final Comments

Fairness provides an important abstraction to the programmer, but the problems inherent in modeling fairness have prevented its widespread use in reasoning formally about program behavior. The introduction of fair traces helps bridge this gap: they permit operational intuition to guide formal reasoning. Moreover, the notion of parameterized fairness provides an accessible way to reason about fair behavior in a systematic, syntax-directed way.

Bibliography

[AFK88] Krzysztof R. Apt, Nissim Francez, and Shmuel Katz. Appraising fairness in languages for distributed programming. *Distributed Computing*, 2(4):226–241, 1988.

[AO91] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 1991.

[AP86] K. R. Apt and G. D. Plotkin. Countable nondeterminism and random assignment. *JACM*, 33(4):724–767, October 1986.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *JACM*, 31(3):560–599, July 1984.

[BO95] Stephen Brookes and Susan Older. Full abstraction for strongly fair communicating processes. In S. Brookes, M. Main, A. Melton, and M. Mislove, editors, *Proceedings of the* 11^{th} *Annual Conference on Mathematical Foundations of Programming Semantics*, volume 1 of *Electronic Notes in Computer Science*. Elsevier, June 1995.

[BR84] S. D. Brookes and A. W. Roscoe. An improved failures model for communicating processes. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 281–305, Pittsburgh, PA, July 1984. Springer-Verlag.

[Bro94] Stephen Brookes. Fair communicating processes. In *A Classical Mind: Essays in Honour of C.A.R. Hoare*, chapter 4. Prentice-Hall, January 1994.

[Bro96a] Stephen Brookes. The essence of Parallel Algol. In *Proceedings of the* 11^{th} *Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, July 1996.

[Bro96b] Stephen Brookes. Full abstraction for a shared-variable parallel language. *Information and Computation*, 127(2):145–163, June 15, 1996.

[Cha78] Ashok K. Chandra. Computable nondeterministic functions. In *Proceedings of the 19^{th} Annual Symposium on Foundations of Computer Science*, pages 724–767. IEEE, 1978.

[CP91] Carol Critchlow and Prakash Panangaden. The expressive power of delay operators in sccs. *Acta Informatica*, 28:447–452, 1991.

[Dar85] Philippe Darondeau. About fair asynchrony. *Theoretical Computer Science*, 37(3):305–336, 1985.

[Dij88] Edsger W. Dijkstra. Position paper on "fairness". *Software Engineering Notes*, 13(2):18–20, April 1988.

[EC80] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In J.W. de Bakker and J. van Leeuwen, editors, *Proceedings 7^{th} ICALP*, number 85 in LNCS, pages 264–277. Springer-Verlag, 1980.

[Fra86] Nissim Francez. *Fairness*. Texts and Monographs in Computer Science. Springer-Verlag, 1986.

[HdBR94] E. Horita, J. W. de Bakker, and J. J. M. M. Rutten. Fully abstract models for nonuniform concurrent languages. *Information and Computation*, 115(1):125–178, November 15, 1994.

[Hen85] M. Hennessy. Acceptance trees. *JACM*, 32(4):896–928, 1985.

[Hen87] Matthew Hennessy. An algebraic theory of fair asynchronous communicating processes. *Theoretical Computer Science*, 49:121–143, 1987.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *CACM*, 21(8):666–677, August 1978.

[Hoa81] C. A. R. Hoare. A model for communicating sequential processes. Technical Report PRG-22, Oxford University Programming Research Group, Oxford, England, 1981.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice Hall, 1985.

[INM84] INMOS Limited. *The Occam Programming Manual*. Prentice-Hall International, 1984.

[JJH90] He Jifeng, M. B. Josephs, and C. A. R. Hoare. A theory of synchrony and asynchrony. In *Proceedings of the IFIP Working Conference on Programming Concepts and Methods*, pages 459–78. North-Holland, 1990.

[Jon87] Bengt Jonsson. *Compositional Verification of Distributed Systems*. PhD thesis, Uppsala University, 1987.

[Jon94] Bengt Jonsson. A fully abstract model for dataflow and asynchronous networks. *Distributed Computing*, 7(4):197–212, 1994.

[Jos92] Mark B. Josephs. Receptive process theory. *Acta Informatica*, 29(1):17–31, 1992.

[KdR83] R. Kuiper and W. P. de Roever. Fairness assumptions for CSP in a temporal logic framework. In D. Bjørner, editor, *Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts - II*, pages 159–167. North-Holland, 1983.

[Kwi89] M. Z. Kwiatkowska. Survey of fairness notions. *Information and Software Technology*, 31(7):371–386, September 1989.

[Lam83] Leslie Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 657–668. IFIP, North Holland, September 1983.

[LPS81] D. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In O. Kariv and S. Even, editors, *Proceedings 8th ICALP*, number 115 in LNCS, pages 264–277. Springer-Verlag, 1981.

[Mes94] Message Passing Interface Forum. MPI: a message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4):169–416, 1994.

[Mil75] Robin Milner. Processes: A mathematical model of computing agents. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73*, volume 80, pages 157–173. North-Holland/American Elsevier, 1975.

[Mil77] Robin Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92. Springer-Verlag, 1980.

[Mor68] James H. Morris. *Lambda-Calculus Models of Programming Languages*. PhD thesis, Massachusetts Institute of Technology, 1968.

[MP83] Zohar Manna and Amir Pnueli. How to cook a temporal proof system for your pet language. In *Proceedings of Tenth ACM Symposium on Principles of Programming Languages*, pages 141–154, 1983.

[MPS88] David McAllester, Prakash Panangaden, and Vasant Shanbhogue. Nonexpressibility of fairness and signaling. In 29^{th} *Annual Symposium on Foundations of Computer Science*, pages 377–86, 1988.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, September 1992.

[OL82] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[Par79] D. Park. On the semantics of fair parallelism. In D. Bjørner, editor, *Abstract Software Specifications*, volume 86 of *Lecture Notes in Computer Science*, pages 504–526. Springer-Verlag, 1979.

[Plo83] G. D. Plotkin. An operational semantics for CSP. In D. Bjørner, editor, *Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts - II*, pages 199–225. North-Holland, 1983.

[PS88a] Prakash Panangaden and Vasant Shanbhogue. Mccarthy's amb cannot implement fair merge. In *Proceedings of the* 8^{th} *Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 348–63. Springer-Verlag, 1988.

[PS88b] Prakash Panangaden and Eugene W. Stark. Computations, residuals, and the power of indeterminacy. pages 439–54, 1988.

[Rus90] James R. Russell. *Full Abstraction and Fixed-Point Principles for Indeterminate Computation*. PhD thesis, Cornell University, April 1990. Available as TR 90-1120.

[Sto88] Allen Stoughton. *Fully Abstract Models of Programming Languages*. Research Notes in Theoretical Computer Science. Pitman Publishing, London, 1988.

[Tho89] Bent Thomsen. A calculus of higher order communicating systems. In *Proceedings of Sixteenth ACM Symposium on Principles of Programming Languages*, pages 143–54, 1989.

[Uni80] United States Department of Defense. *Reference Manual for the Ada Programming Language*, 1980.