Thesis Proposal :
A Denotational Framework for Fair Communicating Processes

Susan Older

May 3, 1994

Abstract

Reasoning about (properties of) parallel programs often includes the use of certain *fairness* assumptions: every process sufficiently often enabled is guaranteed to make progress sufficiently often. Different definitions for "sufficiently often" result in different notions of fairness. In languages with synchronous communication, the "enabledness" of a process depends not only on its willingness to communicate but also on the ability of other processes to communicate with it. For this reason, modelling the combination of fairness and deadlock can be complicated. A process can block if it requires synchronization and other processes are not able to synchronize sufficiently often; conversely, a process may be forced to eventually synchronize in order to satisfy the fairness constraints for another process.

We propose a compositional framework for reasoning about fairness in languages with communication. We start by giving a denotational semantics for a language of fair, disjoint, synchronously communicating processes. The denotational semantics corresponds closely to the operational semantics for the language, but provide the means to reason formally and compositionally. The meaning of a term in this semantics is a set of *acceptance traces*; each acceptance trace reflects a possible computation sequence for the term. This semantics is *adequate* with respect to a variety of fair and deadlock-sensitive notions of behavior: that is, the semantics gives identical meanings to different terms only if the terms have the same behaviors in all program contexts. For each of these notions of behavior, we expect to find suitable closure conditions on acceptance trace sets which lead to *full abstraction*. Full abstraction is a stronger notion than adequacy: a fully abstract semantics for a given notion of behavior identifies two terms *iff* they have the same behaviors in all contexts. Fully abstract semantics are desirable, because they make exactly those distinctions which ought to be made.

We believe that acceptance trace semantics capture the information necessary for reasoning about the combination of fairness and communication. To support this belief, we intend to generalize the semantics for a variety of fairness notions. We also plan to extend the approach to other communication-based languages, first to a hybrid language of non-disjoint processes which communicate via synchronous communication and then to a language based on asynchronous message-passing.

1 Introduction

In reasoning about parallel programs, it is common to make use of certain fairness assumptions: every entity that is enabled sufficiently often will make progress sufficiently often. Different definitions of "sufficiently often" then give rise to different notions of fairness, such as weak fairness (justice), strong fairness (fairness), and unconditional fairness (impartiality) [Fra86, LPS81]. In a language of communicating processes such as Hoare's CSP[Hoa78], an extra level of fairness emerges from defining "entity": we can speak of *process* fairness, *channel* fairness, or even *communication*

fairness. Reasoning about certain properties of concurrent programs (such as safety and liveness) often requires making some fairness assumptions. Even proving termination of the simple program

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

requires an assumption of fairness.

Fair semantics are often given operationally, as in [Fra86]. Such semantics have a two-phase approach: the transition rules give all (fair and unfair) computations, and then additional fairness constraints "weed out" the unfair computations. Unfortunately, such constraints are typically easier to describe than to apply. For languages in which processes communicate via synchronized message-passing, there is an additional complication. A process's "enabledness" can depend upon the progress of other processes: synchronization requires the cooperation and active participation of two processes. For this reason, determining whether or not a computation is fair often requires knowledge of the communications which were enabled earlier (and possibly not taken) along the computation path.

Using operational semantics to prove that two programs behave the same in an arbitrary context is often difficult and unwieldy. As a result, rigorous proofs about equivalences among programs (and thus the correctness of program optimizations) are often replaced by arguments based on operational intuition. Although informal reasoning may be better than no reasoning, intuition can easily overlook subtle nuances which affect the correctness of a program. Denotational semantics provide a more rigorous and compositional approach, but it is not without its own difficulties. It is well known that fairness is related to unbounded nondeterminism (see [AP86] or [Par79], for example) which can result in lack of continuity for various semantic functions. For example, it is insufficient to view an infinite computation as an extension of all of its finite prefixes: it may not be fair even though each of its finite prefixes is the prefix of some fair computation.

The gap between rigor and operational intuition can be bridged via the use of *trace semantics*. Roughly speaking, each of a program's possible computations is recorded as a trace, and the meaning of a program is a set of traces. A compositional semantics can be defined by representing each program combinator by an operation on trace sets. The compositional nature of the trace semantics can simplify certain proofs of program (in)equivalence and perhaps offer insight into the relationships among programs. This in turn can facilitate the process of understanding and verifying properties of programs. Because the denotational and operational semantics correspond meaningfully, we can harness the benefits of each: operational intuition can always be made rigorous in a straightforward manner.

We propose a denotational framework for reasoning about the combination of fairness and communication. We first concentrate on a CSP-like language of disjoint, synchronously communicating processes and introduce a denotational semantics based on sets of *acceptance traces*, where each acceptance trace corresponds loosely to an abstract record of a fair computation. This semantics is *adequate* for several fair and deadlock-sensitive behaviors: that is, the semantics identifies terms only if they have the same behaviors in all program contexts. We believe that by finding suitable closure conditions on acceptance trace sets, we can achieve *full abstraction* [Mil73, Mil77] with respect to each of these behaviors as well. Full abstraction is a well-known, objective criterion for judging the utility of a semantics. A semantics is fully abstract with respect to a particular notion of behavior if it gives identical meanings to terms exactly when they exhibit the same behaviors in all contexts. We hope to extend this work then to a language of non-disjoint, synchronously communicating processes and to a language with asynchronous message-passing.

2 Preliminaries

2.1 Communicating Processes

The language we use is a hybrid of Hoare's original CSP [Hoa78] and Milner's CCS [Mil80]. As in CSP, processes have disjoint local states and are constructed from the usual sequential operations, in addition to communication and parallel composition. Unlike CSP, we permit nested parallelism, and our communicating processes are "parallel" in that respect. We do not model the distributed termination convention, and our processes communicate with other processes along independent channels rather than by name. A channel restriction operator (akin to the hiding operator of CCS) forces synchronization along a given channel.

The particular syntax and operational semantics of our language appear in the Appendix. We introduce a small fragment of notation helpful for describing the proposed and completed work. We use λ to range over the set of all possible actions; that is, λ is either $h?v$ (for some channel h and value v) or $h!v$ or ϵ, where $h?v$ represents receipt of the value v on channel h, $h!v$ represents output of v along h, and ϵ represents a silent (internal) action. We write $\langle C, s\rangle \xrightarrow{\lambda} \langle C', s'\rangle$ to indicate that the command C in state s can perform an action labelled λ, leading to C' in state s'. The command C in state s is deadlocked (written $\langle C, s\rangle$dead) iff it has no transitions.

2.2 Fairness

There are many different flavors of fairness, depending on the choice of definitions for "entity" and "sufficiently often". We concentrate primarily on strong and weak process fairness, as both can be motivated easily from a practical standpoint; we direct the interested reader to [Fra86] for more specifics. Weak (process) fairness requires that every *continuously* enabled process eventually proceeds. In contrast, strong (process) fairness requires that every process which is enabled *infinitely often* eventually proceeds. Any strongly fair computation is also weakly fair.

The choice of "entity" becomes especially meaningful for languages with communication. For example, consider the two processes A and B below, when placed in parallel:

$A \equiv$ **while true do** $(a!1 \rightarrow c!1) + (a!1 \rightarrow c!3) + (d!1 \rightarrow c!4)$
$B \equiv$ **while true do** $(b!2 \rightarrow e!2)$

Under strong process fairness, any fair computation of $A \| B$ must output infinitely many 1's and 2's along some combination of channels a, b, c, d and e. There may also be output of 3's or 4's, but fairness does not require it. Under strong channel fairness, every fair computation also outputs infinitely many 4's, since channels a and d are both enabled infinitely often; again, fairness makes no requirement upon the number of 3's output. Lastly, strong communication fairness distinguishes between the two $a!1$ branches, and both must be chosen infinitely often; as a result, each fair computation outputs infinitely many 1's, 2's, 3's and 4's. Unless specified otherwise, for the remainder of this paper we use the term fairness to mean strong process fairness.

In our setting, a process is enabled if its current configuration permits either an internal action or communication on an unrestricted channel. A process is also enabled if its current configuration permits communication on a restricted channel *and* another process is currently able to synchronize with it on that channel. A configuration is *blocked* if it has no currently enabled transitions. Hence it is possible for a given configuration of a process to alternate between being enabled and being blocked: its "enabledness" may be completely dependent upon the behavior of the other processes in parallel with it and their ability to synchronize with it.

Defining fair computations compositionally is complicated by the fact that blocking depends upon the set of restricted channels. Intuitively, one might expect the fair computations of $C\backslash h$ to be the fair computations of C that contain no visible communications along h. Unfortunately, this intuition fails. Consider, for example, the processes A, B and B' below:

$A \equiv$ **while true do** $(a!1 \rightarrow$ **skip**$)$
$B \equiv$ **while true do** $(b!1 \rightarrow$ **skip**$)$
$B' \equiv$ **while true do** $(b!1 \rightarrow$ **skip**$) + (a?x \rightarrow c!1)$

Every fair computation of $A \| B$ must contain infinitely many outputs along both a and b. But what are the fair computations of $(A \| B)\backslash a$? Because A quickly becomes blocked and is never enabled again, B can fairly ignore A in this context. However, B cannot become blocked, and hence the fair computations of $(A \| B)\backslash a$ all have infinitely many outputs along channel b and no communication along channel a. The combination of fairness and restricted channels can also force synchronization which might not have occurred otherwise. For example, there are fair computations of $A \| B'$ which contain no outputs along channel c; because channel a is unrestricted, A can output along channel a without being forced to synchronize with B'. In contrast, every fair computation of $(A \| B')\backslash a$ contains infinitely many outputs along channel c due to forced synchronization along channel a. These examples illustrate the need to speak of blocking with respect to a set of restricted (or dually, unrestricted) channels in order to characterize fairness compositionally.

We include a brief operational description for fairness with respect to a set of channels. Since programs are finite, each program is necessarily restricted to a finite set of channels on which it can communicate; for this reason, we parameterize fairness with respect to a set H of channels on which external communication is possible. Every finite successful or deadlocked transition sequence is fair. An infinite transition sequence of C is fair iff for each parallel component C' of C the subsequence of steps involving C' forms a fair transition sequence for C' or becomes eventually blocked (with respect to H) by the other components of C. In particular, if C has no parallel components, then all of C's computations are considered fair. For instance, in the example above, B' has fair computations which never accept input on channel a. We write

$$\langle C, s_0 \rangle \xrightarrow{\lambda_0} \cdots \xrightarrow{\lambda_{k-1}} \langle C_k, s_k \rangle \xrightarrow{\lambda_k} \cdots \text{fair}$$

if the given infinite computation is fair according to the operational description above.

2.3 Full Abstraction

A semantics is only useful if it provides insight into those properties one wishes to reason about. For this reason, a semantics might be sufficient for reasoning about one type of behavior and yet insufficient for another one. Hence the "usefulness" of a semantics is often parameterized by a certain notion of behavior.

Given a notion of behavior, a semantics is *adequate* with respect to that behavior if whenever two terms have the same meaning, they have the same behaviors in *all* program contexts[1]. That is, an adequate semantics will never identify terms which have observably different behaviors in some context. It may, however, make unnecessary distinctions among programs which have the same behaviors in all contexts. For example, any injective function over the terms of a language is trivially adequate for any behavior: two programs have identical meanings iff they are the same program, and hence they must have the same behaviors in all contexts. Even if it makes too many

[1] Roughly speaking, a program context is a program with a "hole".

distinctions, an adequate semantics can be useful: any programs identified by the semantics can be treated as identical. If a semantics is compositional and if the behavior of a program can be extracted from its meaning, then the semantics is necessarily adequate for that behavior.

A semantics is *fully abstract* with respect to a given behavioral notion if it possesses exactly the right level of detail for reasoning about that behavior: a fully abstract semantics gives two terms the same meaning iff they have the same behaviors in all contexts. In essence, a fully abstract semantics makes exactly the right distinctions for reasoning about the given behavior. Hence, with a fully abstract semantics, one can always determine whether one term can be safely substituted for another term in all contexts.

Although full abstraction is desirable, it is not always possible to find a fully abstract semantics for a given behavior. When reasoning about the behavior of individual programs, an adequate semantics can often be as useful as a fully abstract one.

3 Completed Work

The results to date are from joint work with Steve Brookes. We sketch the work completed so far; specifics may be found in [BO94].

We represent a program by a set of *acceptance traces*, where each trace corresponds to a computation path for that program. Each acceptance trace is comprised of a (finite or infinite) sequence of "steps", with each step indicating the set of possible actions (called an *acceptance set*), the action actually taken, and the corresponding state change. For example, the step $(s, \{a!, b!\}a!1, s)$ indicates that from state s, output was enabled on channels a and b, and that the value 1 was output along channel a, leading back to state s. Letting $\langle C, s\rangle \xrightarrow{A\lambda} \langle C', s'\rangle$ indicate that A is the set of next actions possible for program C and $\langle C, s'\rangle \xrightarrow{\lambda} \langle C', s'\rangle$, the acceptance trace semantics $\mathcal{T}$ can be characterized operationally as follows:

$$\begin{aligned}
\mathcal{T}[\![C]\!] \;=\; & \{(s_0, A_0\lambda_0, s_1)\ldots(s_k, A_k\lambda_k, s_{k+1}) \mid \langle C, s_0\rangle \xrightarrow{A_0\lambda_0} \cdots \xrightarrow{A_{k-1}\lambda_{k-1}} \langle C_k, s_k\rangle \xrightarrow{A_k\lambda_k} s_{k+1}\} \\
\cup\; & \{(s_0, A_0\lambda_0, s_1)\ldots(s_k, A_k\lambda_k, s_{k+1})\delta_{s_{k+1}} \mid \langle C, s_0\rangle \xrightarrow{A_0\lambda_0} \cdots \xrightarrow{A_k\lambda_k} \langle C_{k+1}, s_{k+1}\rangle \mathrm{dead}\} \\
\cup\; & \{(s_0, A_0\lambda_0, s_1)\ldots(s_k, A_k\lambda_k, s_{k+1})\ldots \mid \langle C, s_0\rangle \xrightarrow{A_0\lambda_0} \cdots \xrightarrow{A_{k-1}\lambda_{k-1}} \langle C_k, s_k\rangle \xrightarrow{A_k\lambda_k} \cdots \mathrm{fair}\}.
\end{aligned}$$

The acceptance sets along a trace play a pivotal role in our semantics: only by knowing what actions were possible along a path can we determine whether an execution sequence is fair.

In order to give a compositional characterization for $\mathcal{T}$, we need to define for each language constructor a corresponding operation on trace sets. Defining sequencing, repetition, and conditional and choice operations on acceptance trace sets is straightforward and intuitive. Brookes showed how to define parallel composition by adapting Park's fair merge operator [Par79] for transition trace sets [Bro93] and for communication and synchronization [Bro94]. Utilizing the additional information provided by acceptance sets, we develop a fair merge which accounts for deadlock and blocking as well.

For each finite set H of channels, we define a ternary relation $\mathit{fairmerge}_H$ on acceptance traces. Roughly speaking, (α, β, γ) is in $\mathit{fairmerge}_H$ iff γ is obtainable from a fair merge of α and β, with external communication permitted only along channels in H. Intuitively, whenever (α, β, γ) is in $\mathit{fairmerge}_H$, γ is constructed from α and β by a combination of interleaving and synchronization of matching input and output. Each component proceeds for a finite number of steps, at which point the other component gets its chance. A synchronized communication pair produces an ϵ-step and counts as an atomic action by both processes. If α is finite then as soon as α has been consumed

there is no further fairness requirement to fulfill, and similarly if β is finite. If α and β are both infinite and no blocking occurs, then γ is a merge which results from a consumption of all of α and all of β. The definition also correctly accounts for blocking. For example, if at some point α is only able to communicate along restricted channels (that is, channels not in H) and β never has a synchronizing action to offer, then the rest of α can be fairly ignored.

Although we omit details, the following two examples should help motivate the $fairmerge_H$ definitions.

Example 3.1 Consider the program $((L \| R) \| M) \backslash \ell \backslash r$, where L, R and M are defined as follows:

$L \equiv$ **while true do** $(\ell!0 \rightarrow$ **skip**$)$
$R \equiv$ **while true do** $(r!1 \rightarrow$ **skip**$)$
$M \equiv$ **while true do** $(\ell?x \rightarrow o!x) + (r?x \rightarrow o!x)$

Looking only at visible (that is, input or output) actions and ignoring states, the traces of L have the form $(\{\ell!\}\ell!0)^\omega$ and the traces of R have the form $(\{r!\}r!1)^\omega$. In addition to traces with infinitely many inputs on ℓ and r, M has traces which accept input only finitely often along one channel. Because the traces of M *enable* input on both channels infinitely often, and because any fair merge of L and R contains infinitely many outputs along both channels, every trace of $(L \| R \| M) \backslash \ell \backslash r$ must output infinitely many 1's and infinitely many 0's. •

Example 3.2 Consider the program $(A \| B) \backslash b$, where A and B are defined as follows:

$A \equiv a!0 \rightarrow b!0 \rightarrow c!0$
$B \equiv$ **while true do** $(a!1 \rightarrow$ **skip**$) + (b!1 \rightarrow$ **skip**$)$

The traces of A have the form $\alpha = \{a!\}a!0\{b!\}b!0\{c!\}c!0$. The traces of B include (but are not limited to) traces of the form $(\{a!, b!\}a!1)^\omega$ and $(\{a!, b!\}b!1)^\omega$. Because $\{b!\}$ is blocked along the entire trace $(\{a!, b!\}a!1)^\omega$ (that is, there is no enabled $b?$), B may ignore A after its initial $a!0$ action. In contrast, the trace $\beta = (\{a!, b!\}b!1)^\omega$ contributes nothing to the parallel composition, even though the traces α and β may appear to block each other: although the attempts of β to output along b are blocked, β also has $a!$ actions enabled. •

The acceptance trace semantics, as outlined so far, is *adequate* for reasoning about several notions of behavior, including each of the following.

- Single-step transitions

$$\begin{aligned}
\mathcal{M}_1[\![C]\!] &= \{s_0 s_1 \ldots s_{k+1} \mid \langle C, s_0\rangle \xrightarrow{\epsilon} \cdots \xrightarrow{\epsilon} \langle C_k, s_k\rangle \xrightarrow{\epsilon} s_{k+1}\} \\
&\cup \{s_0 s_1 \ldots s_k \ldots \mid \langle C, s_0\rangle \xrightarrow{\epsilon} \cdots \xrightarrow{\epsilon} \langle C_k, s_k\rangle \xrightarrow{\epsilon} \cdots \text{ fair}\} \\
&\cup \{s_0 s_1 \ldots s_k \delta \mid \langle C, s_0\rangle \xrightarrow{\epsilon} \cdots \xrightarrow{\epsilon} \langle C_k, s_k\rangle \text{dead}\}.
\end{aligned}$$

  This behavior records every step along a computation path and therefore can distinguish programs based on the number of steps as well as by the sequence of states encountered. For example, this behavior distinguishes the programs C; **skip** and C.

- Multiple-step transitions

$$\begin{aligned}
\mathcal{M}_2[\![C]\!] &= \{s_0 s_1 \ldots s_{k+1} \mid \langle C, s_0\rangle \longrightarrow^* \langle C_1, s_1\rangle \longrightarrow^* \cdots \longrightarrow^* \langle C_k, s_k\rangle \longrightarrow^* s_{k+1}\} \\
&\cup \{s_0 s_1 \ldots s_k \ldots \mid \langle C, s_0\rangle \longrightarrow^* \langle C_1, s_1\rangle \longrightarrow^* \cdots \longrightarrow^* \langle C_k, s_k\rangle \longrightarrow^* \cdots \text{fair}\} \\
&\cup \{s_0 s_1 \ldots s_k \delta \mid \langle C, s_0\rangle \longrightarrow^* \langle C_1, s_1\rangle \longrightarrow^* \cdots \longrightarrow^* \langle C_k, s_k\rangle \text{dead}\}
\end{aligned}$$

We write $\langle C, s\rangle \longrightarrow^* \langle C', s'\rangle$ to indicate that C' in state s' is reachable from C in state s by a finite number (possibly zero) of ϵ-transitions. For example, if C is the program $y := z; a!z$, then $\langle C, [y = 0, z = 1]\rangle \longrightarrow^* \langle C, [y = 0, z = 1]\rangle$ and $\langle C, [y = 0, z = 1]\rangle \longrightarrow^* \langle a!z, [y = z = 1]\rangle$.

The process of collapsing sequences of steps and "forgetting" intermediate states is hindered by the need to maintain information about which actions were enabled. For example, while we might like to ignore the ϵ step in $(s, A\lambda, s')(s', B\epsilon, s'')$, it is important not to lose the information about enablings that B provides.

- Communication traces

$$\begin{aligned}\mathcal{M}_3[\![C]\!] &= \{(s, \alpha, s') \mid \langle C, s\rangle \stackrel{\alpha}{\Longrightarrow} s'\} \cup \{(s, \alpha, \bot) \mid \langle C, s\rangle \stackrel{\alpha}{\Longrightarrow}\} \\ &\cup \{(s, \alpha\delta, s') \mid \langle C, s\rangle \stackrel{\alpha}{\Longrightarrow} \langle C', s'\rangle \mathrm{dead}\}\end{aligned}$$

Here, α is a member of $\Lambda^\infty = \Lambda^* \cup \Lambda^\omega$, where $\Lambda^* = \{\epsilon\} \cup \{h?n, h!n \mid n \in N, h \in \mathbf{Chan}\}^+$ and $\Lambda^\omega = \{\alpha\epsilon^\omega \mid a \in \Lambda^*\} \cup \{h?n, h!n \mid n \in N, h \in \mathbf{Chan}\}^\omega$, and the generalized transition relations $\stackrel{\alpha}{\Longrightarrow}$ are defined as follows:

  - For finite α, $\langle C, s\rangle \stackrel{\alpha}{\Longrightarrow} \langle C', s'\rangle$ means that C from state s can perform the sequence of communications α, leading to the configuration C' in state s'; finitely many ϵ-transitions are permitted between communications. The same definition holds for terminal configurations.
  - For infinite α, $\langle C, s\rangle \stackrel{\alpha}{\Longrightarrow}$ means that C from state s can perform a *fair* infinite computation in which the action labels form the sequence α.

The communication trace behavior reflects the notion that the only "interesting" behavior of a communicating process is its sequence of communications. With this interpretation, private stores are not only untouchable by other processes, they are considered unobservable as well. For example, the programs **while true do** $x := x + 1$ and **while true do** $x := x + 2$ would be considered equivalent, since they both diverge and their state transition differences are unobservable.

For each of the above behaviors, the acceptance trace semantics identifies two programs only if they have the same behaviors in all program contexts. The semantics is not fully abstract for any of the behaviors, however, since in some respect the acceptance sets provide *too much* information and force distinctions among programs which may have the same observational behaviors. For proving properties (deadlock-freedom, for example) of an individual program, the adequacy of the semantics is sufficient. Still, we believe that by finding suitable closure conditions on acceptance trace sets, we can achieve fully abstract semantics for each of the behaviors.

4 Proposed Work

In the previous section, we have described a compositional semantics which is adequate for several notions of behavior under the assumption of strong fairness. There is also a related semantics which is adequate with respect to the analogous behaviors under weak process fairness. Our first goal is to find (for each behavior) suitable closure conditions on acceptance trace sets which achieve full abstraction for that behavior.

We also plan to generalize the results to other notions of fairness as well. In particular, we plan to investigate both strong and weak varieties of channel fairness and communication fairness. Here, as above, we hope to find adequate semantics which can then be made fully abstract.

We would like to investigate a "hybrid" language, in which synchronous communication is permitted between non-disjoint processes. We believe that the issues involved in developing message-based communication traces are orthogonal to those for shared-memory traces, and we expect that the two approaches can be combined in a natural and appealing manner. We would also like to extend the techniques to a treatment for languages with asynchronous message-passing.

Not only can the acceptance trace semantics aid in proving behavioral equivalences among programs, but the semantics can be used directly to prove correctness properties of individual programs. To demonstrate of the utility of acceptance traces for reasoning about the behavior of parallel programs, we plan to tackle a series of examples from the literature, including the following: producer and consumer with a bounded buffer, the sieve of Erastothenes, matrix multiplication, and the dining philosophers [Hoa78]; and the bubble-lattice sort [Bar85]. Networks constructed inductively are natural candidates for this approach. It would be interesting to see if other types of configurations are also well-suited for our semantics.

Lastly, we would like to investigate the relationship between the acceptance trace semantics and various temporal logics. For example, full abstraction for the single-step transitions behavior means that two programs have the same meaning iff they satisfy the same linear time temporal logic formulas. Such full abstraction results might have utility in the area of model checking [CGL94], for example, by providing a safe substitution of one model by an equivalent but smaller one.

5 Related Work

5.1 Denotational Models

In [Bro93], Brookes presented a fair trace semantics for a shared-variable language and showed how it could be adapted for various notions of behavior at various levels of granularity. Independently, Jonsson and Back [JB93] used a similar approach for a UNITY-like language. [Bro94] adapted the trace semantics to a CSP-like language to achieve full abstraction for a notion of fair behavior which ignores the possibility of deadlock. Properly modelling the combination of fairness and deadlock for communicating processes is significantly more difficult than for shared-variable languages, because synchronization requires the cooperation of two processes. This work begins an attempt to understand and capture formally the relationship among fairness, deadlock, and synchronization.

Our acceptance traces are clearly related to the *failures* models of CSP [BHR84] and occam [Ros84] and Hennessy's acceptance trees [Hen85, Hen87, HI93]. Among the closure conditions we need to impose upon trace sets to achieve full abstraction is a generalized convexity closure based on the convex and union closure conditions imposed in these models.

States play a limited role in Roscoe's model [Ros84]: a process's state is passed on to its successors upon termination, but its intermediate states are unobservable. Additionally, divergence is considered catastrophic, and hence the model contains only finite (prefixes of) traces. Consequently, there is no account of fairness.

In [Hen85], Hennessy uses finite acceptance trees to model the behavior of stateless processes whose operations are limited to prefixing, external choice, and internal choice. These acceptance trees correspond loosely to our sets of acceptance traces. Acceptance trees are extended in [HI93] to model a language based on communicating processes with private memories and handshake communication. In this context, only communication is observable (the private memories are indeed private), and no account of fairness is made. Fairness is considered in [Hen87] for a stateless language whose terms are comprised of prefixing, restriction, fair parallel composition, and external and internal choice. The notion of fairness is very limited, however: roughly speaking, an infinite

parallel computation is fair iff both its projected computations are infinite and fair.

5.2 Operational Approaches

We regard our trace semantics as a viable alternative to the formal techniques suggested in Francez [Fra86] and Apt and Olderog [AO91] for reasoning about the termination of communicating processes under communication fairness. The presentation in Francez, based strongly on [GFK84], involves choosing a well-founded, partially-ordered set $(W, \leq)$ and a parameterized invariant. Additionally, one must define for each non-minimal $w \in W$ a matching pair of "directions" which reduces the "distance" from termination and satisfies certain invariant-based proof rules. This approach does not distinguish deadlock from termination; deadlock-freedom must be proven separately via the typical partial correctness rules, such as [AFdR80] or [LG81].

Apt and Olderog suggest a program transformation technique, which transforms the original program into a non-deterministic program without communication and then embeds a fair scheduler into the new program using random assignment. Proof rules are then applied to the resulting program. Because the transformation has already incorporated knowledge about possible matching communications into boolean guards, the proof rules are slightly easier than those in Francez. Again, the proof depends on a choice of a partially-ordered well-founded set and verifying that the choice of a helpful direction satisfies certain necessary conditions.

These approaches seem unsatisfactory for several reasons. First of all, these techniques reason only about termination. They provide no insight into programs which are designed not to terminate. The program transformation of Apt and Olderog results in a program which is significantly more complicated than the original one, both syntactically and semantically. Finally, these approaches can be very sensitive to small changes in the original program, thus disallowing significant "re-use" of proof fragments. Because program changes can affect how the fair scheduler is embedded or the interaction of possible matching communications, proving the correctness of relatively simple program optimizations may require significant effort.

6 Expected Contributions and Timetable

To conclude, we have proposed a semantic framework for compositional reasoning about fairness in languages with communication. We summarize the expected contributions of this work:

- A family of fully abstract compositional semantics which satisfactorily and appropriately account for a range of behaviors and fairness notions. In addition, we hope that the construction of these semantics will provide insight into constructing new models for other related behaviors.
- Extension of the approach to "hybrid" concurrent languages, including a language with asynchronous message-based communication.
- Possible identification of program configurations whose analysis benefits from using acceptance traces.
- Techniques for (and examples of) trace-based reasoning.

The basic plan of research follows, with (necessarily) rough time estimates:

- Full abstraction for single-step, multiple-step and communication trace behaviors, with respect to both weak and strong fairness. (3.5 months)
- Fully abstract models for various generalizations of fairness, including channel and communication fairness. (2 months)
- Extension to hybrid and asynchronous message-passing languages. (4 months)
- Techniques for analysis and extended examples. (3 months)
- Investigation of relationship between acceptance traces and temporal logics. (2 months)
- Writing. (4 months)

Overall, an expected completion date of December 1995 seems reasonable.

References

[AFdR80] K. R. Apt, N. Francez, and W. P. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2(3):359–385, 1980.

[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, 1986.

[Bar85] H. Barringer. *A Survery of Verification Techniques for Parallel Programs*. Springer-Verlag, 1985.

[BHR84] S. Brookes, C.A.R. Hoare, and A.W Roscoe. A theory of communicating sequential processes. *JACM*, 31(3):560–599, July 1984.

[BO94] S. Brookes and S. Older. Full abstraction for fair communicating processes, February 1994. Draft.

[Bro93] S. Brookes. Full abstraction for a shared variable parallel language. In *Proc. 8^{th} Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, June 1993.

[Bro94] S. Brookes. Fair communicating processes. In *A Classical Mind: Essays in Honour of C.A.R. Hoare*, chapter 4. Prentice-Hall, January 1994.

[CGL94] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems, January 1994.

[Fra86] N. Francez. *Fairness*. Springer-Verlag, 1986.

[GFK84] O. Grumberg, N. Francez, and S. Katz. Fair termination of communicating processes. In *Proc. 3^{rd} ACM-PODC*, August 1984.

[Hen85] M. Hennessy. Acceptance trees. *JACM*, 32(4), 1985.

[Hen87] M. Hennessy. An algebraic theory of fair asynchronous communicating processes. *Theoretical Computer Science*, 49, 1987.

[HI93] M. Hennessy and A. Ingólfsdóttir. Communicating processes with value-passing and assignments. *Formal Aspects of Computing*, 3, 1993.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *CACM*, 21(8):666–677, August 1978.

[JB93] B. Jonsson and R. J. R. Back. Fully abstract semantic orderings for shared-variable concurrent programs, December 1993. Draft.

[LG81] G.M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15:281–302, 1981.

[LPS81] D. Lehmann, A. Pneuli, and J. Stavi. Impartiality, justice and fairness: the ethis of concurrent termination. In *Proc. 8^{th} ICALP*, Lecture Notes in Computer Science, pages 264–277. Springer-Verlag, 1981.

[Mil73] R. Milner. Processes: A mathematical model of computing agents. In *Logic Colloquium 73*, pages 157–173. North Holland, 1973.

[Mil77] R. Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[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.

[Plo82] G. D. Plotkin. An operational semantics for CSP. In D. Björner, editor, *Formal Description of Programming Concepts II*, pages 199–225. North-Holland, 1982.

[Ros84] A. W. Roscoe. Denotational semantics for occam. In *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*. Springer-Verlag, 1984.

A Appendix

This appendix, adapted from [BO94], contains the syntax and operational semantics for our language. Details of the acceptance trace semantics, as well as motivating examples, can be found in [BO94].

A.1 Syntax

We define the abstract syntax of our language as follows. There are seven syntactic sets: **Ide**, the set of identifiers, ranged over by I; **Exp**, the set of expressions, ranged over by E; **BExp**, the set of boolean expressions (or conditions), ranged over by B; **Chan**, the set of channel names, ranged over by h; **Gua**, the set of communication guards, ranged over by G; **GCom**, the set of guarded commands, ranged over by GC; and **Com**, the set of commands, ranged over by C.

The abstract syntax for identifiers, channel names, expressions and conditions will be taken for granted; we assume that identifiers and expressions denote integers, boolean expressions denote truth values, and the language contains the usual arithmetic and boolean operators and constants. For commands, guarded commands, and guards we specify the following grammar:

$$\begin{array}{rcl} C & ::= & \textbf{skip} \mid I{:=}E \mid C_1;C_2 \mid \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \mid \textbf{while } B \textbf{ do } C \mid \\ & & GC \mid C_1 \| C_2 \mid C \backslash h \\ GC & ::= & (G \rightarrow C) \mid GC_1 + GC_2 \\ G & ::= & h?I \mid h!E \end{array}$$

We impose the syntactic constraint that in all parallel commands $C_1 \| C_2$ the components C_1 and C_2 must have disjoint sets of identifiers.

A.2 Operational Semantics

A state is a finite partial function from identifiers to integer values. We use N for the set of integers, and we let $S = [\textbf{Ide} \rightarrow_p N]$ denote the set of states. We use s as a meta-variable ranging over S, and we write $[s \mid I = n]$ for the state which agrees with s except that it gives identifier I the value n. The *domain* of a state, denoted $\mathrm{dom}(s)$, is the set of identifiers for which the state has a value. We say that two states s_1 and s_2 are *disjoint* when their domains are disjoint.

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

For commands, guarded commands and guards we use a *labelled transition system*, much as in [Plo82]. Non-terminal configurations have the form $\langle C, s\rangle$ (respectively, $\langle GC, s\rangle$ and $\langle G, s\rangle$), where s is a state defined at least on the free identifiers of C (respectively, GC and G). A terminal configuration is simply a state. We decorate transitions with a label indicating the type of atomic action involved: ϵ represents an internal action, $h?n$ represents receiving value n on channel h, and $h!n$ represents sending value n along channel h. A label λ is a member of the set $\Lambda = \{\epsilon\} \cup \{h?n, h!n \mid n \in N, h \in \textbf{Chan}\}$.

For a label or direction λ, $\mathrm{chan}(\lambda)$ is the channel associated with λ; by convention, we define $\mathrm{chan}(\epsilon) = \epsilon$. We write $\langle C, s\rangle \xrightarrow{\lambda} \langle C', s'\rangle$ to indicate that command C in state s can perform an action labelled λ, leading to C' in state s'. Two labels λ_1 and λ_2 *match* iff one has form $h?n$ and the other $h!n$ for some channel name h and value n; when this holds we write $\mathrm{match}(\lambda_1, \lambda_2)$.

The transition relations $\xrightarrow{\lambda}$ $(\lambda \in \Lambda)$ are characterized by the axioms and inference rules in Figure 1 and Figure 2.

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

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

$$\frac{\langle B, s\rangle \to^* \texttt{tt}}{\langle \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2, s\rangle \xrightarrow{\epsilon} \langle C_1, s\rangle} \qquad \frac{\langle B, s\rangle \to^* \texttt{ff}}{\langle \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2, s\rangle \xrightarrow{\epsilon} \langle C_2, s\rangle}$$

$$\frac{\langle B, s\rangle \to^* \texttt{tt}}{\langle \mathbf{while}\ B\ \mathbf{do}\ C, s\rangle \xrightarrow{\epsilon} \langle C; \mathbf{while}\ B\ \mathbf{do}\ C, s\rangle} \qquad \frac{\langle B, s\rangle \to^* \texttt{ff}}{\langle \mathbf{while}\ B\ \mathbf{do}\ C, s\rangle \xrightarrow{\epsilon} s}$$

Figure 1: Transition rules for sequential constructs

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

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

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

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

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

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

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

$$\frac{\langle C, s\rangle \xrightarrow{\lambda} \langle C', s'\rangle}{\langle C \backslash h, s\rangle \xrightarrow{\lambda} \langle C' \backslash h, s'\rangle}\,(**) \qquad \frac{\langle C, s\rangle \xrightarrow{\lambda} s'}{\langle C \backslash h, s\rangle \xrightarrow{\lambda} s'}\,(**)$$

$(*)$ provided $\text{match}(\lambda_1, \lambda_2)$ and $\text{disjoint}(s_1, s_2)$
$(**)$ provided $\text{chan}(\lambda) \neq h$

Figure 2: Transition rules for parallel constructs