Deconstructing CCS and CSP
Asynchronous Communication, Fairness, and Full Abstraction

Stephen Brookes
Carnegie Mellon University

Paper presented at MFPS 16
In honor of Robin Milner

Abstract

The process algebras CCS and CSP assume that processes interact by means of handshake communication, although it would have been equally reasonable to have adopted asynchronous communication as primitive. Traditional semantics of CCS (based on bisimulation) and CSP (such as the failures model) do not assume fair parallel composition, although fairness is often vital in ensuring liveness properties. It appears to be difficult to adapt these semantic models to incorporate fairness. We consider a process algebra based on asynchronous communication and fair parallel execution. We show that a simple semantic model based on traces suffices for reasoning about safety and liveness, including deadlock analysis. We prove that this semantics is fully abstract, in that processes have the same trace sets if and only if they induce identical safety and liveness properties in all program contexts. Our semantic framework also exposes the underlying similarities between communicating processes and shared-variable parallel programs, despite the disparity between their traditional models. We also propose a form of fair bisimulation, which can be viewed as the finest reasonable notion of equivalence for fair asynchronously communicating processes.

1 Introduction

The process algebras CCS [28] and CSP [10] have been used widely to specify and reason about the behavior of parallel systems. Their semantic foundations have been explored extensively, leading to the development of theories (based on *bisimulation* for CCS, and on *failure equivalence* for CSP) and practical tools for verification of safety and liveness properties (such as the Concurrency Workbench [12] and the model checker FDR [17]). Many extensions, generalizations, and offshoots of the original algebras have been introduced, attesting to their significance and fundamental importance.

CSP and CCS were designed to model *asynchronous processes*, running at indeterminate relative speed; the form of parallel composition typical of CCS and CSP allows independent progress to be made by individual processes. Moreover, processes were assumed to interact by means of *handshake communication*; a process attempting to perform an input action *synchronizes* with another process attempting a matching output, waiting if necessary until a match becomes available.

As Hoare commented in the original CSP paper, it would have been equally reasonable to have assumed *asynchronous communication* instead, since each communication mechanism is implementable using the other. Hoare believed that the synchronizing form would lend itself to a simpler semantics. Whatever the original philosophical justification or pragmatic considerations, the handshake mechanism has become a fundamental building block in the theory of CCS and CSP.

In retrospect it is worth re-examining the rationale for this choice. With the benefit of hindsight it can be argued that the simplicity permitted by the assumption of handshake communication is deceptive, achieved at the expense of ignoring *fairness* [18]. When reasoning about a parallel system it is natural to assume (weakly) fair execution: that every process is eventually scheduled. Weak fairness is a convenient abstraction, allowing us to ignore scheduling details while ensuring that program correctness proofs provide guarantees about program behavior when execution follows a "reasonable" scheduling strategy. For example, any round-robin scheduling strategy ensures weakly fair execution. Many liveness properties of parallel systems cannot be proven without fairness assumptions, so that fairness issues are obviously important.

Park [34] provided a fair semantics for a simple language of shared-variable parallel programs, based on a form of "transition trace", with par-

allel composition modelled by a *fairmerge* operator on trace sets. For this class of programs weak (process) fairness is a natural assumption to make, is relatively easy to formalize, is evidently a sensible abstraction from network implementation details, and there is no pressing need to consider alternative notions of fairness proposed in the literature, such as strong (process) fairness [18]. Park's ideas have been further developed and fine-tuned, yielding fully abstract semantic models for shared-variable programs [3].

For processes in the CCS/CSP style, communicating by handshake, the fairness problem is not as clean-cut, and weak process fairness turns out to be rather ineffective as an assumption in reasoning about liveness [32]. The reason for this is that the ability of a process to make progress typically depends on the simultaneous ability of another process to perform a matching action. This makes fairness for synchronizing processes difficult both to implement and to model mathematically. Indeed, a plethora of distinct notions of fairness exists for handshaking processes (including strong and weak variants of "process fairness" and "channel fairness"), and it is debatable if these constitute sensible abstractions from reasonable scheduling strategies. The difficulties that crop up in trying to devise tractable fair semantic models for CSP are brought out clearly in Older's Ph.D. thesis [32] and related papers such as [9]. It has proven difficult to adapt models of CCS and CSP to incorporate fairness while retaining the structural simplicity of the original models.

Furthermore, there is little apparent structural similarity between the various models and concepts that have been developed for process algebras such as CCS and CSP (including synchronization trees, weak- and strong bisimilarity, and failure sets) and those for shared-variable parallel systems. Roughly speaking, traditional models of shared-variable programs tend to explicitly mention "state" and are clearly "imperative" in nature, whereas traditional accounts of process algebras typically ignore or abstract away from state.

In this paper we introduce a process algebra in the CCS/CSP vein but based explicitly on asynchronous communication and fair parallel composition. We claim that by building our theories on top of *asynchronous* communication instead of handshakes, and by building (weak, process) fairness in directly, we can provide a mathematically simple account of fair communicating processes, robust enough for further generalization (for instance, by adding procedures or a form of "concurrent object").

We begin by recalling the notation, terminology, and semantic concepts

behind CCS and CSP, assuming handshake communication. We then show that the assumption of asynchronous communication leads to a process algebra with a mathematically straightforward semantic model (closely related to Park's semantics for shared-variable programs), in which it is easy to model fair execution. In fact we present two *equivalent* formulations of our semantics: one couched in terms of *communication traces*, obtained by abstracting away from state, and one based on *transition traces* that abstracts away from action labels but retains information about state. Thus we are able to establish a common semantic foundation for state-free and state-ful notions of process. We establish a number of fundamental semantic properties, which confirm the naturality of our approach. In particular, we prove that our semantics is fully abstract [27], in that two processes have the same meaning if and only if they induce the same observable behaviors in all (fair) program contexts. This guarantees that our semantics supports compositional reasoning about safety and liveness properties of parallel systems. We also discuss briefly a more refined semantic equivalence that amounts to a fair version of bisimilarity.

2 Milner's CCS

In 1980 Milner introduced *A Calculus of Communicating Systems*, a notation for specifying systems of parallel computing agents, or processes, which communicate by handshake message-passing [28]. The syntax of CCS processes P, and action labels λ, is based on the following grammar:

$$\begin{array}{rcll} P & ::= & nil \mid & \textit{inaction} \\ & & \lambda.P \mid & \textit{prefix} \\ & & P_1 + P_2 \mid & \textit{sum} \\ & & (P_1|P_2) \mid & \textit{parallel} \\ & & P\backslash a & \textit{restriction} \end{array}$$

$$\lambda \quad ::= \quad a?v \mid a!v \mid \tau$$

Here a ranges over the set Ch of channel names, and v ranges over the set V of communicable values, which we assume for simplicity is the set of integers. We define $ch(\lambda)$, the channel used by action λ, by: $ch(a?v) = ch(a!v) = a$, $ch(\tau) = \epsilon$.

$$\frac{}{\lambda.P \xrightarrow{\lambda} P}$$

$$\frac{P \xrightarrow{\lambda} P'}{P + Q \xrightarrow{\lambda} P'} \qquad \frac{Q \xrightarrow{\lambda} Q'}{P + Q \xrightarrow{\lambda} Q'}$$

$$\frac{P \xrightarrow{\lambda} P'}{P|Q \xrightarrow{\lambda} P'|Q} \qquad \frac{Q \xrightarrow{\lambda} Q'}{P|Q \xrightarrow{\lambda} P|Q'}$$

$$\frac{P \xrightarrow{\lambda} P' \quad Q \xrightarrow{\bar{\lambda}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

$$\frac{P \xrightarrow{\lambda} P' \quad ch(\lambda) \neq a}{P\backslash a \xrightarrow{\lambda} P'\backslash a}$$

Figure 1: Transition rules for CCS

Milner presented an operational semantics for CCS terms, generated by a *labelled transition system*. A transition of form

$$P \xrightarrow{\lambda} Q$$

is taken to represent the ability of process P to perform the action labelled λ and thereafter behave like the process Q. In fact, since communication is intended to require synchronized participation by both sender and receiver, it might be better to view such a transition as representing the *potential* for P to perform its part of the desired communication. The label τ represents a handshake action. The transition rules are summarized in Figure 1. Note that we use the notation $\overline{\lambda}$ for the action *matching* λ: $\overline{a?v} = a!v, \overline{a!v} = a?v$.

2.1 Observational equivalence and bisimilarity

Milner's original CCS semantics was based on *observational equivalence*, characterized as the "limit" of a series of increasingly finer equivalence relations $\approx_n$ (for $n \geq 0$) on CCS terms (or, more generally, on labelled transition systems). These relations were defined inductively, with $\approx_0$ being the universal relation and with the inductive clause being:

$P \approx_{n+1} Q$ if and only if

- $\forall \lambda, P'.\ P \stackrel{\lambda}{\Longrightarrow} P'$ *implies* $\exists Q'.\ Q \stackrel{\lambda}{\Longrightarrow} Q'$ & $P' \approx_n Q'$
- $\forall \lambda, Q'.\ Q \stackrel{\lambda}{\Longrightarrow} Q'$ *implies* $\exists P'.\ P \stackrel{\lambda}{\Longrightarrow} P'$ & $P' \approx_n Q'$

where

$$\stackrel{\lambda}{\Longrightarrow} = (\stackrel{\tau}{\longrightarrow})^* \circ \stackrel{\lambda}{\longrightarrow} \circ (\stackrel{\tau}{\longrightarrow})^*$$

Building on work of David Park [35], Milner recast these ideas in a slightly more general and uniform framework by introducing *weak bisimilarity*, the equivalence relation on processes given by:

Definition 1

$P \approx Q$ *if and only if*

- $\forall \lambda, P'.\ P \stackrel{\lambda}{\Longrightarrow} P'$ *implies* $\exists Q'.\ Q \stackrel{\lambda}{\Longrightarrow} Q'$ & $P' \approx Q'$
- $\forall \lambda, Q'.\ Q \stackrel{\lambda}{\Longrightarrow} Q'$ *implies* $\exists P'.\ P \stackrel{\lambda}{\Longrightarrow} P'$ & $P' \approx Q'$

Weak bisimilarity is an equivalence relation, and is respected by prefixing, parallel composition, and restriction. However, the choice operator causes some technical difficulty, since $\tau.Q \approx Q$ always holds but $P + \tau.Q$ may fail to be weakly bisimilar to $P + Q$. As a result, weak bisimilarity is not a *congruence* for CCS. The largest CCS congruence consistent with weak bisimilarity, known as *observation congruence* and denoted $\approx^c$, is characterized as follows.

Definition 2

$$P \approx^c Q \iff \forall R.\ P + R \approx Q + R.$$

Milner's book on *Communication and Concurrency* [29] uses the symbol $=$ (and the name "equality") for this congruence, emphasizing its fundamental role in the theory of CCS.

The *calculus* of CCS involves a large collection of simple laws and rules of process equivalence, including:

- *static laws*, such as

$$\begin{aligned} P|Q &= Q|P \\ P + nil &= P \end{aligned}$$

- *dynamic laws*, such as

$$P + \tau.P = \tau.P$$

- *expansion laws*, such as

$$(\lambda.P)|(\mu.Q) \;=\; \lambda.(P|(\mu.Q)) + \mu.((\lambda.P)|Q) \qquad \textit{if } \bar{\lambda} \neq \mu$$

- the *unique fixed point rule*:

$$\frac{Q = P[Q/p]}{Q = \mathbf{rec}\ p.P} \qquad \textit{if } p \textit{ is guarded in } P$$

3 Hoare's CSP

CSP was introduced as an imperative language of parallel processes, communicating by synchronized message-passing [22]. The CSP process algebra [10, 23] was also designed to model asynchronously executing parallel processes with handshake communication. As in CCS, the process algebra is typically presented in a manner that abstracts away from the imperative nature of process state, focussing instead on the patterns of communication in which processes may engage. CSP process syntax is based on the following grammar:

$$\begin{array}{rcll} P & ::= & \mathit{nil} \mid & \mathit{inaction} \\ & & \lambda.P \mid & \mathit{prefix} \\ & & P_1 \Box P_2 \mid & \mathit{external\ choice} \\ & & P_1 \sqcap P_2 \mid & \mathit{internal\ choice} \\ & & (P_1|P_2) \mid & \mathit{parallel} \\ & & P/a & \mathit{hiding} \end{array}$$

$$\lambda \;::=\; a?v \mid a!v \mid \tau$$

Note that CSP replaces the choice operator of CCS with two distinct choice operators: *internal* choice, written $P \sqcap Q$; and *external* choice, written $P \Box Q$. These two forms of choice correspond to special cases of Dijkstra's *guarded choice* construct, as adapted in Hoare's original CSP paper to permit input and output commands to appear in guards [22]. A guarded choice with purely boolean guards needs no external communication to proceed, whereas

a guarded choice with input or output guards needs to consult its "environment" to determine which of its guards is enabled. Thus for example the processes which would be written as

$$\begin{aligned} P &= \textbf{if } (\textbf{true} \to a?x.P_1) \,\Box\, (\textbf{true} \to b?x.P_2) \textbf{ fi} \\ Q &= \textbf{if } (a?x \to Q_1) \,\Box\, (b?x \to Q_2) \textbf{ fi} \end{aligned}$$

in the notation of Hoare's original paper are represented as

$$\begin{aligned} P &= (a?x.P_1) \sqcap (b?x.P_2) \\ Q &= (a?x.Q_1) \,\Box\, (b?x.Q_2) \end{aligned}$$

In fact the treatment of $\Box$ and $\sqcap$ as fully fledged binary operators on *processes* is much more general than is implied by their origins, allowing for instance such intuitively peculiar constructs as $(P|Q)\Box R$.

CSP also uses a "hiding" operator instead of the CCS restriction operator. We have actually changed the syntax a little to facilitate the contrast with CCS; originally the CSP hiding operator was also written as $P\backslash a$, but we will write P/a instead to avoid confusion with the CCS restriction operator. The intended behavior of the hiding operator is rather different from that of the CCS restriction operator, as will be evident shortly. For this reason it is particularly helpful to use distinct notation for the two constructs.

We also work with a version of CSP in which communication is two-way, rather than allowing multiple synchronization or broadcast events. This reflects the structure of Hoare's original language, in which processes were *named* and each communication specified explicitly the name of its source and its target process. In the CSP process algebra it is common to regard input and output as "symmetric" *events*, to equip processes (either explicitly or implicitly) with *alphabets*, and to permit multiple synchronization whenever an event belongs to the alphabets of several processes. The role of such a general form of communication as *primitive* is rather questionable, as is its implementability. When constrained to permit only two-way synchronization between a sender and a receiver process, the CSP form of parallel composition operates in exactly the same manner as the CCS form of parallel composition, so we will continue to use the notation $P|Q$, rather than $P\|Q$ as in most papers on CSP.

$$\frac{}{P \sqcap Q \xrightarrow{\tau} P} \qquad \frac{}{P \sqcap Q \xrightarrow{\tau} Q}$$

$$\frac{P \xrightarrow{\lambda} P' \quad \lambda \neq \tau}{P \Box Q \xrightarrow{\lambda} P'} \qquad \frac{Q \xrightarrow{\lambda} Q' \quad \lambda \neq \tau}{P \Box Q \xrightarrow{\lambda} Q'}$$

$$\frac{P \xrightarrow{\tau} P'}{P \Box Q \xrightarrow{\tau} P' \Box Q} \qquad \frac{Q \xrightarrow{\tau} Q'}{P \Box Q \xrightarrow{\tau} P \Box Q'}$$

$$\frac{P \xrightarrow{\lambda} P' \quad ch(\lambda) \neq a}{P/a \xrightarrow{\lambda} P'/a} \qquad \frac{P \xrightarrow{\lambda} P' \quad ch(\lambda) = a}{P/a \xrightarrow{\tau} P'/a}$$

$$\frac{P \xrightarrow{\lambda} P'}{P|Q \xrightarrow{\lambda} P'|Q} \qquad \frac{Q \xrightarrow{\lambda} Q'}{P|Q \xrightarrow{\lambda} P|Q'}$$

$$\frac{P \xrightarrow{\lambda} P' \quad Q \xrightarrow{\bar{\lambda}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

Figure 2: Transition rules for CSP

3.1 Operational semantics

Again the operational behavior of CSP processes is specified by a labelled transition system, with transitions of form

$$P \xrightarrow{\lambda} Q$$

interpreted as expressing the potential for process P to perform action λ and evolve to process Q. The transition rules are summarized in Figure 2. Note that hiding a channel causes actions involving that channel to occur "uncontrollably" whenever they are enabled.

3.2 Failure equivalence

Apart from syntactic differences, the major difference between CCS and CSP concerns the choice of semantic foundations. The semantic model for CSP was chosen to be as simple as possible while supporting compositional reasoning about the ability or inability of a process to perform sequences of

communication; in particular, the model was designed to incorporate information about the potential for a process to *deadlock* when run in parallel with other processes. This was the motivation behind the design of the *failures* semantics of CSP [10].

A *failure* of a process P is a pair (α, X) consisting of a finite sequence α of communication actions and a set X of actions such that P is capable (if run in a context that offers the required matching actions) of performing α and then "refusing" the actions in X. Thus P can cause deadlock if run in a context which offers only actions from X as its possible next steps. A *Hoare trace* of P is a finite sequence of actions α such that $(\alpha, \{\})$ is a failure of P, i.e. such that α is a potential communication sequence for P. Note that such traces form a non-empty prefix-closed set and each Hoare trace corresponds to a *partial* record of a (not necessarily completed) computation of P.

Definition 3 *The failures of P form the set $f(P)$ characterized operationally by:*

$$f(P) = \{(\alpha, X) \mid P \stackrel{\alpha}{\Longrightarrow} P' \ \& \ P' \ \mathbf{ref} \ X\}$$

where

$$P \ \mathbf{ref} \ X \iff \forall \lambda \in X \cup \{\tau\}.\ \neg(P \stackrel{\lambda}{\longrightarrow})$$

Failures can also be determined in a compositional manner, obtaining a denotational semantic description that supports syntax-directed analysis of deadlock [10]. We define *failure equivalence* for CSP processes by:

Definition 4

$$P \equiv_f Q \iff f(P) = f(Q)$$

Failure equivalence is a congruence for CSP, and can be regarded as the coarsest reasonable congruence, in view of the well known *full abstraction* property: two processes are failure equivalent if and only if they induce identical deadlock behavior in all CSP contexts.

Although the fact seems to be less well known, weak bisimilarity is also a CSP congruence. (This property is implied by some of the results of the author's Ph.D. thesis [2].) It is easy to see that weak bisimilarity implies failure equivalence, but the converse fails. It seems reasonable to regard weak bisimulation as the *finest* reasonable congruence for CSP. Note that there is no need to involve $\approx^c$, since both forms of choice operator in CSP respect weak bisimulation, unlike the CCS choice operator.

4 Limitations

The CCS and CSP frameworks outlined above have proven highly influential, stimulating an enormous body of research concerned with foundational issues as well as practical applications. The original work was tailored to reasoning about specific classes of program property; later work has striven to generalize further. In particular, neither of the above semantic frameworks provides a proper account of infinite computations. We identify two problems: the potential for *divergence*, and the need to assume *fair* parallel composition.

4.1 Divergence and infinite behaviors

Divergence is the potential for an infinite 'internal" computation without any "visible" communication. A process which may diverge might never communicate with its environment. Nevertheless according to the CCS semantics the recursive processes defined by

$$P = a!0.P + \tau.P \qquad Q = a!0.Q$$

are equal, despite the fact that only P has the potential for divergence. Similar problems with the treatment of divergence arose in the CSP framework. Indeed, according to the operational characterization of failures given above, process P has *no* failures whatsoever.

In order to distinguish between processes such as P and Q one can define a form of bisimilarity sensitive to divergence, along lines developed by Walker [41]. In generalizing the failures approach to cope with divergent processes the decision was made to regard even the potential for divergence as "catastrophic". This led to a more refined *failures/divergences* model of CSP, in which all possibly diverging processes are equated with the most non-deterministic process of all.

4.2 Fairness

The early models of CCS and CSP, even including these divergence-sensitive variants, made no distinction between *fair* and *unfair* executions. Nevertheless, when reasoning about the behavior of a parallel system it is natural to assume fairness, in that each process will eventually be scheduled for execution. (Technically, this amounts to assuming *weak process fairness*.) For

example, writing μ^ω for a process capable of repeating the atomic action μ forever, if we assume fair execution we would expect that

$$\lambda|\mu^\omega = \mu^*\lambda\mu^\omega.$$

However, this equivalence fails both in CCS and in CSP, because in neither case is parallel composition interpreted as *fair*.

Recognition of this defect led to work on incorporating (certain forms of) fairness into CCS, notably by Parrow, who augmented the syntax of CCS with *infinitary restriction* operators and introduced a form of weak bisimilarity sensitive to the potential for infinite computation. Attempts to develop fair models of CCS and CSP are hampered by the underlying assumption of *handshake communication*, since this makes it difficult to keep track of the enabledness of a process: one process is only enabled to perform a handshake step when it offers an action λ for which another process simultaneously offers a matching $\bar{\lambda}$ action.

As Hoare commented in his original CSP paper, it would have been equally reasonable to have assumed *asynchronous communication* as the means of process interaction. Hoare rationalized his decision to adopt handshake communication by asserting that this led to a simpler theory, and that asynchronous communication could always be simulated within the synchronized setting by using "buffer" processes. Hoare also chose to ignore fairness. In retrospect the simplicity of the failures model is misleading; it turns out to be extremely difficult to *extend* the failure/divergences approach to incorporate fair parallelism.

We will now examine what happens when we revisit the CCS/CSP designs but assume *fair* process interaction and *asynchronous* communication. We will use the CSP notation as above, except for the hiding operator, which we will replace by a form of local channel declaration similar in intent to the "encapsulation" operators common in ACP [1].

5 Asynchronous communication

The syntax of our language of asynchronously executing, asynchronously communicating processes will again be based on the original CSP design. However, in the asynchronous setting input and output cannot be treated symmetrically: *output* actions can occur *asynchronously*, without requiring the participation of another process; a process wishing to perform an input

action must wait if necessary until the relevant channel is non-empty. For similar reasons it makes no sense to allow a process to perform a choice guarded by *output*. We will therefore only include a form of external choice that involves input-guarded processes. We will also permit *recursive processes*, using the notation **rec** $p.P$. For example, the recursive process

$$\textbf{rec}\ p.\, a?v.b!v.p$$

will behave like a one-place buffer that transmits data from channel a to channel b.

Instead of CCS restriction $P\backslash a$, which prevents all activity on channel a except for synchronizations, or CSP hiding P/a, which renders all actions on channel a "uncontrollable", we need a form of channel localization that properly meshes with asynchronous communication and recognizes that the actions of a process may depend on the contents of "hidden" or "local" channels. At any stage during process execution the contents of a channel will be a finite (possibly empty) sequence of data. Thus we introduce the notation **local** $a = \rho$ **in** P, where $\rho \in V^*$, to represent process P executing with a as a local channel, initially containing the data sequence ρ.

In summary, we will use the following abstract grammar for *processes* (ranged over by P) and *input-guarded processes*, (ranged over by G), in which p ranges over a set of *process identifiers*:

$$\begin{array}{lcll} P & ::= & nil \mid & \textit{inaction} \\ & & a!v.P \mid & \textit{output} \\ & & a?v.P & \textit{input} \\ & & G_1 \,\square\, G_2 & \textit{external choice} \\ & & P_1 \sqcap P_2 \mid & \textit{internal choice} \\ & & (P_1 | P_2) \mid & \textit{parallel} \\ & & \textbf{local}\ a = \rho\ \textbf{in}\ P & \textit{localization} \\ & & p & \textit{process identifier} \\ & & \textbf{rec}\ p.P & \textit{recursion} \\ \\ G & ::= & a?v.P \mid G_1 \,\square\, G_2 & \end{array}$$

We can give a straightforward semantic account of the effects of input and output by modelling each channel as an updateable finite *queue*, and regarding the *state* of a system as the (current) contents of each channel. An input action $a?v$ is enabled if (the queue corresponding to) channel a is non-empty

and v is the first item in the queue; when this action occurs it causes a "side-effect" on the system state, *dequeueing* v from a. A process attempting input on channel a must *wait* if a is empty. An output action $a!v$ is always enabled, and causes the *enqueueing* of value v on a.

Since actions now depend on and affect the state, we need to consider transitions of form

$$\langle P, s\rangle \xrightarrow{\lambda} \langle P', s'\rangle$$

with the interpretation that process P, in state s, can perform action λ, changing the state to s', and then behave like Q. In order to model waiting we introduce a new form of action label: δ_X (where X is a set of channels) represents waiting on (a subset of the) channels in X; the special case when X is empty amounts to an "unconditional" idle step. Thus we use the following grammar for labels:

$$\lambda ::= a?v \mid a!v \mid \delta_X$$

The transition rule for the localization construct shows that local actions are "invisible" (since they become labelled by δ), and that only local output is "uncontrollable". In particular, note that

$$\textbf{local } a = \epsilon \textbf{ in } (a?v.P)$$

has only idling δ-transitions, whereas

$$\textbf{local } a = [v] \textbf{ in } (a?v.P)$$

can perform the local input action "invisibly" and become $\textbf{local } a = \epsilon \textbf{ in } P$.

Our use of busy-waiting to model frustrated input actions means that deadlock manifests itself as the ability to perform only infinite sequences of δ actions. Hence the transition rule for *nil*. It follows from this that we are, effectively, equating deadlock with "livelock" or "divergence"; unlike the failures model we do *not* view divergence as catastrophic, and indeed there would be little point in doing things this way if we were planning to do so. Our approach is, in this respect, closer in spirit to the so-called "chaos-free failures divergences" model of CSP [40], although that model assumes handshake communication and does not deal with fair parallelism.

5.1 Trace semantics

A major consequence of our treatment of deadlock is that we are able to work with a simpler semantic structure than before: *traces* suffice. It is

$$\frac{}{\langle nil,\, s\rangle \xrightarrow{\delta_X} \langle nil,\, s\rangle} \qquad \frac{enq(a)(s) = s'}{\langle a!v.P,\, s\rangle \xrightarrow{a!v} \langle P,\, s'\rangle}$$

$$\frac{}{\langle P \sqcap Q, s\rangle \xrightarrow{\delta_X} \langle P, s\rangle} \qquad \frac{}{\langle P \sqcap Q, s\rangle \xrightarrow{\delta_X} \langle Q, s\rangle}$$

$$\frac{deq(a)(s) = (v,\, s')}{\langle a?v.P,\, s\rangle \xrightarrow{a?v} \langle P,\, s'\rangle} \qquad \frac{null(a)(s) \quad a \in X}{\langle a?v.P,\, s\rangle \xrightarrow{\delta_X} \langle a?v.P,\, s\rangle}$$

$$\frac{\langle G_1,\, s\rangle \xrightarrow{a?v} \langle P',\, s'\rangle}{\langle G_1 \Box G_2,\, s\rangle \xrightarrow{a?v} \langle P',\, s\rangle} \qquad \frac{\langle G_2,\, s\rangle \xrightarrow{a?v} \langle Q',\, s'\rangle}{\langle G_1 \Box G_2,\, s\rangle \xrightarrow{a?v} \langle Q',\, s'\rangle}$$

$$\frac{\langle G_1,\, s\rangle \xrightarrow{\delta_X} \langle G_1,\, s\rangle \quad \langle G_2,\, s\rangle \xrightarrow{\delta_X} \langle G_2,\, s\rangle}{\langle G_1 \Box G_2,\, s\rangle \xrightarrow{\delta_X} \langle G_1 \Box G_2,\, s\rangle}$$

$$\frac{\langle P,\, s\rangle \xrightarrow{\lambda} \langle P',\, s'\rangle}{\langle P|Q,\, s\rangle \xrightarrow{\lambda} \langle P'|Q,\, s'\rangle} \qquad \frac{\langle Q,\, s\rangle \xrightarrow{\lambda} \langle Q',\, s'\rangle}{\langle P|Q,\, s\rangle \xrightarrow{\lambda} \langle P|Q',\, s'\rangle}$$

$$\frac{\langle P, (s, a : \rho)\rangle \xrightarrow{\lambda} \langle P', (s', a : \rho')\rangle \quad \mu \in \lambda/a}{\langle \mathbf{local}\ a = \rho\ \mathbf{in}\ P, s\rangle \xrightarrow{\mu} \langle \mathbf{local}\ a = \rho'\ \mathbf{in}\ P', s'\rangle}$$

where

$$\begin{array}{rcll} \delta_X/a & = & \{\delta_Y \mid X - \{a\} \subseteq Y\} & \\ a?v/a = a!v/a & = & \{\delta_X \mid X \subseteq Ch\} & \\ \lambda/a & = & \{\lambda\} & \textit{otherwise} \end{array}$$

$$\frac{}{\langle \mathbf{rec}\, p.P, s\rangle \xrightarrow{\delta_X} \langle [\mathbf{rec}\, p\, .P/p]P, s\rangle}$$

Figure 3: Transition rules for asynchronous processes

important to notice, however, that we work with traces that record *complete* computations of a process, rather than the *partial* traces used in the failures semantics. This decision is crucial in enabling us to account properly for fair execution. In fact there are two alternative ways to present a trace-theoretic semantics for our language. The first, using what we will call *communication traces*, abstracts away from state and focusses on action labels. The second, using *transition traces*, abstracts away from labels and keeps track of state.

Definition 5 *The set of communication traces $ct(P)$ of a process P is characterized operationally by:*

$$ct(P) = \{\alpha \in \Lambda^\omega \mid P \stackrel{\alpha}{\Longrightarrow} \mathit{fair}\},$$

where $\Lambda = \{a?v, a!v \mid a \in Ch, v \in V\} \cup \{\delta_X \mid X \subseteq Ch\}$ *and we write*

$$P \stackrel{\lambda}{\Longrightarrow} P' \mathit{iff}\ \exists s, s'.\ \langle P, s\rangle \stackrel{\lambda}{\Longrightarrow} \langle P', s'\rangle.$$

For a communication trace $\alpha = \lambda_0\lambda_1 \ldots \lambda_n \ldots$ we write $P \stackrel{\alpha}{\Longrightarrow} \mathit{fair}$ to indicate that P has a fair execution of form

$$P \stackrel{\lambda_0}{\Longrightarrow} P_0 \stackrel{\lambda_1}{\Longrightarrow} P_1 \stackrel{\lambda_2}{\Longrightarrow} P_2 \cdots$$

When P is a parallel composition such an execution is fair if and only if it projects down onto a (complete, infinite) fair execution of each component process. This definition is simple in appearance yet deceptively subtle: since we abstract away from the state, such "executions" actually permit state changes between successive steps. Thus a communication trace should more accurately be viewed as representing the "visible" record of a fair *interactive* computation of the process P with other processes running concurrently and communicating *via* channels with P.

To illustrate the communication trace semantics note that $a?v.P \,\Box\, b?v.Q$ has communication traces $a?v\,\alpha$, where α ranges over the communication traces of P; also $b?v\,\beta$, where β ranges over the communication traces of Q; and δ_{ab}^ω. Also note that $(\delta_a\delta_b)^\omega$ is *not* a trace of this process. In contrast, $a?v.P \sqcap b?v.Q$ has all of these traces, as well as δ_a^ω and δ_b^ω.

Definition 6 *The transition traces $tt(P)$ of a process P are given by*

$$tt(P) = \{\beta \in (S \times S)^\omega \mid P \stackrel{\beta}{\Longrightarrow} \mathit{fair}\},$$

where

$$P \stackrel{(s,s')}{\Longrightarrow} P' \ \mathit{iff}\ \exists\lambda.\langle P, s\rangle \stackrel{\lambda}{\Longrightarrow} \langle P', s'\rangle.$$

Again we characterize fair executions as those built by interleaving (fair) executions of all component processes.

It is easy to show from the above definitions that the communication traces of a process are *closed* under certain natural operations.

Theorem 1 *$ct(P)$ is closed under:*

- *stuttering*
$$\alpha\beta \in ct(P) \;\Rightarrow\; \alpha\delta_X\beta \in ct(P)$$
- *muttering*
$$\begin{aligned} \alpha\delta_\phi\beta \in ct(P) &\;\Rightarrow\; \alpha\beta \in ct(P) \\ \alpha\delta_X\delta_Y\beta \in ct(P) &\;\Rightarrow\; \alpha\delta_{X\cup Y}\beta \in ct(P) \end{aligned}$$

Similarly, it is easy to show that the transition traces of a process are closed under analogous operations, for which we will use the same names. Although the analogy is not as exact as the names might imply, taken together these two operations achieve the same effect as their namesakes above.

Theorem 2 *$tt(P)$ is closed under:*

- *stuttering*
$$\alpha\beta \in tt(P) \;\Rightarrow\; \alpha(s,s)\beta \in tt(P)$$
- *muttering*
$$\begin{aligned} \alpha(s,s)(s,s')\beta \in tt(P) &\;\Rightarrow\; \alpha(s,s')\beta \in tt(P) \\ \alpha(s,s')(s',s')\beta \in tt(P) &\;\Rightarrow\; \alpha(s,s')\beta \in tt(P) \end{aligned}$$

The astute reader may have noticed that this transition trace semantics involves slightly different closure conditions from those employed in the author's earlier transition trace semantics of shared-variable parallel programs, which were referred to as *stuttering* and *mumbling* [3]. (We have, correspondingly, chosen slightly different but suggestively similar names.) Stuttering is exactly the same here, but the muttering condition amounts to mumbling limited to idle steps. The effect is that here each step in a trace corresponds to an operational transition of form $P \stackrel{\lambda}{\Longrightarrow} Q$, i.e. to a single atomic action, possibly with additional idle steps. In contrast in the shared-variable model each trace step corresponds to an arbitrary finite sequence of atomic actions.

The difference in emphasis corresponds with our assumption here that we can observe each and every atomic step *modulo* idling.

The two alternative forms of trace semantics were deliberately presented in such a way as to emphasize their close connections, differing only in their respective focus: one abstracts away from state, the other abstracts away from labels. The following result establishes formally the fact that the two semantics are *equivalent*: they induce exactly the same notions of semantic approximation and (hence) semantic equivalence.

Theorem 3 *The two alternative forms of trace semantics are equivalent, in that for all processes P and Q,*

$$ct(P) \subseteq ct(Q) \quad \textit{iff} \quad tt(P) \subseteq tt(Q).$$

Proof
This can be shown as follows. Let $[\![-]\!] : \Lambda \to \mathcal{P}(S \times S)$ be given by:

$$\begin{array}{lcl} [\![a?v]\!] & = & \{(s,s') \mid (v,s') = deq(a)(s)\} \\ [\![a!v]\!] & = & \{(s,s') \mid s' = enq(a,v)(s)\} \\ [\![\delta_X]\!] & = & \{(s,s) \mid \forall a \in X.\ null(a)(s)\} \end{array}$$

This function extends to $[\![-]\!] : \Lambda^\omega \to \mathcal{P}((S \times S)^\omega)$ in the obvious way. It is then easy to prove that

- $tt(P) \;=\; \bigcup\{[\![\alpha]\!] \mid \alpha \in ct(P)\}$
- $ct(P) \;=\; \{\alpha \mid [\![\alpha]\!] \subseteq tt(P)\}$

Hence,

$$\begin{array}{rcl} ct(P) \subseteq ct(Q) & \iff & \forall \alpha.\ ([\![\alpha]\!] \subseteq tt(P) \;\Rightarrow\; [\![\alpha]\!] \subseteq tt(Q)) \\ & \iff & tt(P) \subseteq tt(Q). \end{array}$$

•

In view of the isomorphism of the two kinds of trace semantics we no longer need to keep working separately with definitions and results pertaining to communication traces and definitions and results couched in terms of transition traces. Nevertheless we will continue to do so, in order to facilitate some of the definitions.

All CSP constructs are *monotone* with respect to trace inclusion:

Theorem 4 *For all programs contexts $C[-]$,*

$$\begin{array}{rcl} ct(P) \subseteq ct(Q) & \Rightarrow & ct(C[P]) \subseteq ct(C[Q]) \\ tt(P) \subseteq tt(Q) & \Rightarrow & tt(C[P]) \subseteq tt(C[Q]) \end{array}$$

5.2 Denotational semantics

The trace semantics ct and tt may also be defined denotationally, by structural induction, so that one can reason about traces in a compositional manner. Our denotational account of the behavior of recursive processes relies on Tarski's fixed point theorem. The collection of trace sets (either communication traces, or transition traces) forms a complete lattice under set inclusion. (The closed sets of traces form a complete sub-lattice.) When P is a process containing free occurrences of the process variable p, we can view the function

$$\lambda t.ct(P)[p \mapsto t]$$

as a monotone function on the complete lattice of trace sets. The (closure of the) *greatest fixed point* of this function coincides with the set of communication traces that would be obtained for $\mathbf{rec}\, p.P$ according to the operational recipe. A similar approach works for transition traces.

We give the denotational semantic equations below (in Figure 4) for the communication trace semantics. Some of the notation used in the semantic clauses warrants explanation. Let $\Delta_a = \{\delta_Y^\omega \mid a \in Y\}$ and $\Delta = \{\delta_Y^\omega \mid Y \subseteq Ch\}$. We write $\nu t.T$ for the greatest fixed point of the function $\lambda t.T$, and we let $T^\dagger$ be the smallest closed set of traces containing T. We say that a communication trace α is int-free for a from ρ if, assuming that the initial contents of channel a is ρ, and assuming that no other process is permitted to interfere with the contents of a, all actions involving this channel along α are enabled. For example, the trace $a?0\,(a!0\,a?0)^\omega$ is int-free for a from 0 but not from ϵ. We write α/a for the obvious extension of the corresponding operator λ/a on actions, as given earlier with the transition rules.

Theorem 5 *The communication trace semantic function determined by the clauses in Figure 4 coincides with the function characterized operationally in Definition 5.*

5.3 Recovering failures

Since outputs are always executable whenever enabled, it no longer makes sense to include output actions in refusal sets; moreover a deadlock can only occur when a system is *stable*, in that no output action can occur. Thus we may characterize the *asynchronous failures* of a process operationally as follows.

$$
\begin{array}{l}
ct(nil) \;=\; \Delta^\dagger \\
ct(a!v.P) \;=\; \{a!v\,\alpha \mid \alpha \in ct(P)\}^\dagger \\
ct(a?v.P) \;=\; \Delta_a^\dagger \;\cup\; \{a?v\,\alpha \mid \alpha \in ct(P)\}^\dagger \\
ct(P \sqcap Q) \;=\; ct(P) \cup ct(Q) \\
ct(P \Box Q) \;=\; ((ct(P) \cap ct(Q)) \cap \Delta)^\dagger \;\cup\; ((ct(P) \cup ct(Q)) - \Delta)^\dagger \\
ct(P|Q) \;=\; \{\gamma \mid \exists \alpha \in ct(P), \beta \in ct(Q).\ (\alpha,\beta,\gamma) \in \mathit{fairmerge}\} \\
ct(\mathbf{local}\ a = \rho\ \mathbf{in}\ P) \;=\; \{\gamma \mid \exists \alpha \in ct(P).\ \gamma \in \alpha/a\ \&\ \\
\qquad\qquad\qquad\qquad\qquad\qquad \alpha\ \mathit{int\text{-}free\ for\ a\ from}\ \rho\} \\
ct(\mathbf{rec}\,p.P) = (\nu t.ct(P)[p \mapsto t])^\dagger
\end{array}
$$

Figure 4: Denotational description of trace semantics

Definition 7 *The asynchronous failures $af(P)$ of a process P are given by:*

$$af(P) = \{(\alpha, X) \mid P \stackrel{\alpha}{\Longrightarrow} P' \ \&\ P' \,\mathbf{ref}\, X\},$$

where

$$P \,\mathbf{ref}\, X \ \textit{iff}\ P\ \textit{stable}\ \&\ \forall a \in X.\, \neg(P \stackrel{a?}{\longrightarrow})$$

and P is stable if for all output actions λ, $\neg(P \stackrel{\lambda}{\longrightarrow})$.

A process attempting input on one or more channels is stable, according to this definition, and its refusals provide information about its deadlock potential. Note also that a process capable of performing an infinite sequence of "hidden" actions may have an empty failure set. It is also possible to define a modified notion of failure that treats the potential for such divergence as catastrophic, along the lines of the failures-divergences model of CSP.

As an example, note that

$$
\begin{array}{rcl}
af(a?v.P \,\Box\, b?v.Q) & = & \{(\epsilon, X) \mid a \in X\ \&\ b \in X\} \\
& \cup & \{(a?v\,\alpha, X) \mid (\alpha, X) \in af(P)\} \\
& \cup & \{(b?v\,\beta, X) \mid (\beta, X) \in af(Q)\} \\
af(a?v.P \sqcap b?v.Q) & = & \{(\epsilon, X) \mid a \in X\ \vee\ b \in X\} \\
& \cup & \{(a?v\,\alpha, X) \mid (\alpha, X) \in af(P)\} \\
& \cup & \{(b?v\,\beta, X) \mid (\beta, X) \in af(Q)\}
\end{array}
$$

Trace semantics is sufficient to determine the failures of a process, as shown by the following elementary result:

Theorem 6 *The asynchronous failures of a process can be recovered from its communication traces:*

$$af(P) = \{(\alpha, X) \mid \alpha\delta_X^\omega \in ct(P)\}.$$

Hence, trace equivalence implies failure equivalence. The converse obviously fails, since failures take no account of infinite non-deadlocking behaviors. In fact we regard traces as the natural *generalization* of asynchronous failures to incorporate fair infinite behaviors. Looking back over the history of the development of CSP semantic models this might seem perverse, since the original *failures* model arose as an attempt to generalize Hoare's *finite (communication) trace* model of CSP, and it is well known that Hoare-style traces are insufficient to distinguish between processes with different deadlocking potential. However, once we assume asynchronous communication and choose to interpret deadlock as infinite idling it becomes possible to use (finite and infinite) traces in a deadlock-sensitive manner, and to build fairness directly in.

5.4 Full abstraction

Trace semantics is at exactly the right level of abstraction to support reasoning about the deadlock, safety and liveness properties of processes. In fact, trace semantics is *fully abstract* with respect to two equivalent notions of observable behavior, as we will now demonstrate.

We assume that a *program* is a network of processes, possibly with free channel names, and we assume that we can observe the actions performed by the system. Initially all channels are empty, and the system evolves without interference from outside, the process in the network communicating with each other *via* the channels. There are two obvious choices of observable.

Definition 8 (Observing communication labels)
For a process P we define $c(P)$ to be the set of label sequences observable along fair, interference-free computations of P:

$$c(P) = \{\alpha \in \Lambda^\omega \mid P \stackrel{\alpha}{\Longrightarrow} \mathit{fair}, \mathit{int\text{-}free}\}.$$

Definition 9 (Observing state change)
For a process P we define $t(P)$ to be the set of state sequences observable along fair, interference-free computations of P:

$$t(P) = \{\beta \in (S \times S)^\omega \mid P \stackrel{\beta}{\Longrightarrow} \mathit{fair}, \mathit{int\text{-}free}\}.$$

Just as we have already seen that the two forms of trace semantics are equivalent, observers watching state can make exactly the same distinctions between processes as can observers watching labels: in this sense c and t are *equivalent* as notions of observable. This is shown formally by (either of) the following full abstraction results.

Theorem 7

- *ct is fully abstract for c*

$$ct(P) \subseteq ct(Q) \quad \textit{iff} \quad \forall C.\ c(C[P]) \subseteq c(C[Q])$$

- *tt is fully abstract for t*

$$tt(P) \subseteq tt(Q) \quad \textit{iff} \quad \forall C.\ t(C[P]) \subseteq t(C[Q])$$

Proof
We focus on communication traces, noting that the result for transition traces will then follow automatically.

The forward implication is obvious, by compositionality and monotonicity of trace semantics.

For the reverse direction, suppose $ct(P) \not\subseteq ct(Q)$. Let A be a finite set of channels containing all of the channels used by P and Q. Choose a "fresh" channel $z \notin A$. Define a *testing process*:

$$\begin{array}{rcl} RUN_A & = & \Box_{a\in A, v\in V}(a?v.z!0.RUN_A) \sqcap \\ & & \Box_{a\in A, v\in V}(a!v.z!1.RUN_A) \sqcap \\ & & \Box_{X\subseteq A} \mathit{fill}_X\, \mathit{empty}_X\, RUN_A \end{array}$$

where for each subset $X = \{a_1, \ldots, a_k\}$ we write

$$\mathit{fill}_X = a_1!0. \ldots a_k!0.z!1, \qquad \mathit{empty}_X = a_1?0. \ldots a_k?0.z!0$$

Let

$$\begin{array}{rcl} \widehat{a!v} & = & a!v\, a?v\, z!0 \\ \widehat{a?v} & = & a!v\, a?v\, z!1 \\ \widehat{\delta_X} & = & \mathit{fill}_{A-X}\, \delta_X\, \mathit{empty}_{A-X} \end{array}$$

For each action λ, $\widehat{\lambda}$ is a finite interference-free trace. Extend this operator to traces in the obvious way, so that when $\alpha = \lambda_0\lambda_1 \ldots$, we obtain another trace $\widehat{\alpha} = \widehat{\lambda_0}\,\widehat{\lambda_1} \ldots$. Note that $\widehat{\alpha}$ is always an interference-free trace.

We then show that a process P has trace α if and only if $P \,|\, RUN_A$ has the interference-free trace $\widehat{\alpha}$. Thus we can use the context $[-] \,|\, RUN_A$ to distinguish between P and Q, since by assumption there exists a trace α of P that does not belong to the trace set of Q. •

In view of the equivalence of the two forms of trace semantics we may paraphrase these results by saying that $\{ct, tt\}$ is fully abstract for $\{c, t\}$. These results justify our claim that *trace equivalence* is the coarsest reasonable congruence for fair asynchronous CSP.

It is perhaps worth examining an example at this point, to help confirm that this full abstraction result is valid. The processes $a?v|b?v$ and $(a?v.b?v)\Box(b?v.a?v)$ have almost exactly the same trace sets: the only difference (modulo the closure conditions) is that $a?v|b?v$ has the trace $(\delta_a\delta_b)^\omega$, which is not possible for the other process. Intuitively this corresponds to the fact that we can find a parallel context in which channels a and b are alternately filled and emptied (by the context), so that if $a?v|b?v$ is placed into this context there will be a fair execution in which every time the left-hand process is running only channel b is non-empty, and every time the right-hand process is running only channel a is non-empty. Thus there will be a fair execution in which $a?v|b?v$ idles forever. If instead we were to place $(a?v.b?v)\Box(b?v.a?v)$ into this context we would find that this process only idles when *both* channels a and b are empty, so that no such execution will exist.

5.5 Fair bisimilarity

The above full abstraction result, based as it is on observing the sequence of actions that occur during program execution, shows that trace semantics is well suited to reasoning about *linear-time* temporal properties of programs. If we want to reason about *branching-time* properties, which require analysis of the alternative behaviors that *might* have been possible during execution, we would need a finer notion of semantic equivalence that takes account of branching. For example, $nil \sqcap (a?x.P \Box b?x.Q)$ and $nil \sqcap a?x.P \sqcap b?x.Q$ have identical trace sets and satisfy the same sets of linear-time formulas. But only the second of these processes satisfies a (branching-time) formula to the effect that "there is a computation in which it is possible to do $a?x$ and not do $b?x$". If we wish to make distinctions based on branching-time properties we would need a more refined notion of semantic equivalence. To this end, we now introduce a notion of *fair (weak) bisimilarity*.

Definition 10 *$P \approx_{fair} Q$ if and only if*

- $ct(P) = ct(Q)$
- $\forall \lambda, P'.\ P \stackrel{\lambda}{\Longrightarrow} P'$ *implies*
  $\exists Q'.\ Q \stackrel{\lambda}{\Longrightarrow} Q'\ \&\ P' \approx_{fair} Q'$
- $\forall \lambda, Q'.\ Q \stackrel{\lambda}{\Longrightarrow} Q'$ *implies*
  $\exists P'.\ P \stackrel{\lambda}{\Longrightarrow} P'\ \&\ P' \approx_{fair} Q'$

This definition is reminiscent of Parrow's notion of weak ω-bisimulation[36], which can be obtained from ours by replacing $ct(P)$, $ct(Q)$ by the corresponding sets of infinite (not necessarily fair) traces. (To be more rigorous here we should really regard the above "definition" of $\approx_{fair}$ as a greatest fixed-point property characterizing the relation: we define $\approx_{fair}$ to be the greatest relation on processes satisfying this property.)

It is easy to see that

$$P \approx_{fair} Q \ \Rightarrow\ P \approx Q \ \Rightarrow\ ct(P) = ct(Q) \ \Rightarrow P \approx_1 Q$$

and each implication is proper. Moreover, $\approx_{fair}$ is a *congruence* for asynchronous CSP, just as $\approx$ was a congruence for the synchronous language. Although we have not made this notion precise and the very interpretation of this property depends on how we characterize "reasonableness", we believe that fair bisimilarity is the finest reasonable congruence for fair asynchronous CSP. Indeed we conjecture that $P \approx_{fair} Q$ if and only if P and Q induce the same branching-time properties in all fair asynchronous contexts, so that fair bisimilarity gives rise to a semantics that is fully abstract for branching-time properties.

Again, given the isomorphism between communication traces and transition traces, we can equally well characterize fair bisimilarity in terms of transition traces:

Theorem 8 *$P \approx_{fair} Q$ if and only if*

- $tt(P) = tt(Q)$
- $\forall s, s', P'.\ P \stackrel{(s,s')}{\Longrightarrow} P'$ *implies*
  $\exists Q'.\ Q \stackrel{(s,s')}{\Longrightarrow} Q'\ \&\ P' \approx_{fair} Q'$

- $\forall s, s', Q'.\ Q \stackrel{(s,s')}{\Longrightarrow} Q'$ *implies*
  $\exists P'.\ P \stackrel{(s,s')}{\Longrightarrow} P'\ \&\ P' \approx_{fair} Q'$

Note that fair bisimilarity does distinguish between the example processes mentioned above:

$$nil \sqcap (a?x.P \,\Box\, b?x.Q) \not\approx_{fair} nil \sqcap a?x.P \sqcap b?x.Q$$

6 Calculus for Asynchronous Communication

Returning again to trace semantics, with trace equivalence as the relevant notion of "equality", we obtain a calculus for reasoning about processes. Trace semantics validates a number of useful laws of process equivalence, including:

- static laws, such as
$$P|Q = Q|P$$
$$P|(Q|R) = (P|Q)|R$$
$$P \sqcap Q \supseteq P \Box Q$$

- dynamic laws, such as:
$$(a?x.P \,\Box\, a?x.Q) = a?x.(P \sqcap Q)$$
$$nil \sqcap (a?x.P \,\Box\, b?x.Q) = nil \sqcap a?x.P \sqcap b?x.Q$$

- expansion laws, such as:
$$(a?x.P)|(b?y.Q) \supseteq a?x.(P|(b?y.Q)) \,\Box\, b?y.((a?x.P)|Q)$$

- recursion
$$\mathbf{rec}\ p.P \ = \ P[\mathbf{rec}\ p.P/p]$$

Note in particular that the expansion law given here is an *inequation*; we have already seen the reason for this. The left-hand process has trace $(\delta_a\delta_b)^\omega$, whereas the right-hand process does not.

In marked contrast to CCS and the failures semantics of CSP, a guarded recursion may have more than one fixed point. This is a consequence of our incorporation of fairness. For example, let p and q be defined recursively by:

$$p = a!0.p \qquad q = b!1.q$$

Then $p|q$ is guaranteed, because of fair execution, to perform infinitely many $a!0$ actions and infinitely many $b!1$ actions. The process r defined recursively by

$$r = a!0.r \,\Box\, b!1.r$$

also performs an infinite sequence of outputs, but is capable of forever outputting on a and avoiding b (or vice versa). Nevertheless $p|q$ satisfies the defining equation for r:

$$p|q = a!0.(p|q) \,\Box\, b!1.(p|q)$$

Since $ct(p|q) \neq ct(r)$ this shows that more than one solution exists for this recursion equation.

The uniqueness property for guarded recursion plays a fundamental and prominent role in Milner's book on CCS and in Hoare's book on CSP. By embracing fairness we are forced to abandon this technique and look instead for laws that reflect fair parallel composition. For example, much as in our earlier work on shared-variable programs and on Idealized CSP, trace semantics validates a number of "expansion laws" which *fail* to hold in unfair models. Of particular importance is a family of *fairness laws*, such as:

$$\begin{aligned} &\textbf{local } a{=}\epsilon \textbf{ in } (a?x.P)|(Q_1;Q_2) \\ &\quad = \; Q_1; \textbf{local } a{=}\epsilon \textbf{ in } (a?x.P)|Q_2 \end{aligned}$$

if $a \notin ch(Q_1)$.

It is also possible to perform algebraic reasoning about processes by manipulating a generalized form of *expansion* [7]. If A_i (for $i = 1, \ldots, n$) are expressions denoting sets of (non-empty) finite traces we say that a formula of form

$$P = \sum_{i=1}^{n} A_i.P_i$$

is *valid* if $ct(P) = \bigcup_{i=1}^{n} \{\alpha\beta \mid \alpha \in A_i \;\&\; \beta \in ct(P_i)\}^{\dagger}$. This notation recalls Milner's use of what we will call "atomic" expansions, of form $P = \sum_{i=1}^{n} \lambda_i.P_i$, but is markedly different from Milner-style expansion in that the terms describe finite prefixes of process behaviors rather than just the initial atomic steps.

We say that such an expansion is *fair* if, for every trace $\gamma \in ct(P)$, and every finite prefix α of γ, there exists a trace β and index i such that $\gamma = \alpha\beta$ and $\alpha \in A_i$ and $\beta \in ct(P_i)$. Intuitively this means that the A_i are

"deep" enough to describe the sequences of actions possible when P is run for an arbitrary finite amount of time. Every finite-state process has such an expansion, and each of the A_i may be represented as a regular expression over an alphabet of atomic actions.

We can then show soundness of the following *fair parallel expansion law*: if $P = \sum_i A_i.P_i$ and $Q = \sum_j B_j.Q_j$ are fair expansions, then

$$P|Q = \sum_{i,j} (A_i \| B_j).(P_i|Q_j)$$

is a valid expansion. Here $A\|B$ denotes the *shuffle* of the (languages denoted by) A and B.

Further details on the interactions between fair parallelism, local channel declarations, and recursion, are provided in [7].

7 What about +?

Our asynchronous language did not include a CCS-style sum. It is worth examining what goes wrong if we attempt to work with the CCS sum operator. Recall the operational rules for CCS sum, adapted (trivially) to the state-ful setting:

$$\frac{\langle P, s\rangle \xrightarrow{\lambda} \langle P', s'\rangle}{\langle P + Q, s\rangle \xrightarrow{\lambda} \langle P, s'\rangle} \qquad \frac{\langle Q, s\rangle \xrightarrow{\lambda} \langle Q', s'\rangle}{\langle P + Q, s\rangle \xrightarrow{\lambda} \langle Q', s'\rangle}$$

It follows immediately from the operational characterization of ct that

$$ct(P + Q) = ct(P) \cup ct(Q),$$

so that, despite the difference in transition rules for $+$ and $\sqcap$, if we only care about traces these differences are masked. If this were the whole story one might be tempted to argue that there is no need to introduce $+$, since $P+Q = P \sqcap Q$, and it would seem that $\sqcap$ usurps the role intended for $+$. Yet this analysis fails to take into account the *combinational* role of the summation operator. Clearly $a?x.P + b?x.Q$ should (according to the transition rules and the operational characterization of ct) behave like $a?x.P \,\Box\, b?x.Q$ and *not* like $a?x.P \sqcap b?x.Q$. Thus it can be argued that the CCS operational rule for $+$ ceases to reflect the intended behavior of a choice operator once we move to the asynchronous setting; instead, the CSP-style choice operators seem more natural in this setting and the "external choice" operator $\Box$ plays the combinational role intended for Milner-style summation.

8 Related work

Much of the foundational research on the semantics of process algebras has assumed handshake communication. Milner [31] explored both "synchronous" and "asynchronous" versions of CCS, differing primarily in the nature of the parallel composition operator but still based on handshake communication. Milner's dichotomy between synchrony and asynchrony refers to the nature of parallel composition, not to the mechanism behind input and output. The asynchronous form of CCS is essentially as described above, with a parallel composition operator that permits steps by each component process and also allows synchronized input and output, which becomes a τ-step. The synchronous form of CCS (known as SCCS) uses a form of parallel composition in which every step is taken by all component processes together, and labels are combined by a form of "multiplication" with the property that $\lambda \times \bar{\lambda} = \tau$.

As already mentioned, none of the early models of CCS or CSP included a fair parallel composition operator. It is by no means straightforward to generalize these models to incorporate fairness. Milner[30] explored the semantic consequences of adding a *finite delay* operator to synchronous CCS, but continued to employ the standard parallel operator. Since this synchronous parallel operator executes processes in lock-step it enforces a rather strong kind of fair execution. Costa and Stirling[13, 14] proposed an alternative semantics for a fair version of CCS.

Parrow[36] augmented CCS with *infinitary restriction* operators of form $P\langle\langle\phi\rangle\rangle$, where ϕ is a temporal logic formula expressing constraints on the set of "allowed" infinite executions of P. Again this was done on top of a language based on unfair parallel composition. Parrow's semantics involves *infinitary charts* and an (unfair) notion of *weak ω-bisimulation* which our notion of *fair bisimilarity* resembles *modulo* our explicit inclusion of fairness.

Hennessy[19] described a CCS-like process calculus in which divergence is regarded as catastrophic and a rather strong form of fairness is assumed (much stronger than the weak process fairness built into our model). Hennessy's semantics involves *acceptance trees* augmented by a set of *fair paths*. This model fails to validate certain natural laws of process equivalence, since (for example)

$$\lambda^\omega | nil \neq \lambda^\omega.$$

Considerable work has also been done on the foundations of process al-

gebras based on asynchronous communication, beginning with Kahn's early model of deterministic dataflow networks, continuing with asynchronous variants of CSP[25] and ACP[1], and culminating more recently with concurrent logic programming languages and concurrent constraint programming. However, apart from the implict role played by fairness in Kahn's model, these investigations have typically ignored fairness. For example, de Boer, Klop, Palamidessi, and Rutten [16] introduced a model for concurrent constraint programs, using (finite) transition traces – subsets of $\mathcal{P}((S \times S)^*)$ – and similar closure properties to our notions of stuttering and muttering this model does not deal with infinite behaviors or fair parallel composition. Later de Boer, Klop, and Palamidessi [15] described an asynchronous process calculus and a semantics that distinguishes between *intended* and *complete* actions, with a model based on "queue failure sets". Fairness was not addressed in this work, and infinite behaviors were ignored.

The most prominent early foundational work on the semantics of fair parallelism was performed by David Park [34]. He showed how to model shared-variable parallel programs running under a weakly fair scheduling assumption, with a semantics based on transition traces. Park's model did not include any closure conditions on trace sets, and each step in a Park-style trace represents the occurrence of *exactly one* atomic action; consequently Park's semantics made too many distinctions between programs, distinguishing for example between $\mathbf{skip}; P$ and $P; \mathbf{skip}$. Park formulated the *fairmerge* relation on traces using a nested combination of greatest- and least fixed point operators.

Building on Park's foundations, we later showed how to achieve full abstraction for shared-variable programs, by imposing closure conditions known as stuttering and mumbling [3]. We generalized this approach still further to obtain transition trace models for a language (Parallel Algol) combining procedures with shared-variable parallelism [4], and for Idealized CSP, a language combining procedures with asynchronous CSP [5]. We also applied these ideas to derive a trace-theoretic model of non-deterministic Kahn networks [6].

9 Conclusions

We have gone back to the origins of the process calculi CCS and CSP and re-examined the rationale behind their design, summarizing the main ideas

involved in setting up their semantic foundations. Both Milner and Hoare were aware of the dichotomy between handshake communication and asynchronous communication, and both chose to assume handshake communication as primitive. Indeed, Hoare explicitly argued that each assumption would be equally tenable as the basis for the development of a theory and calculus of parallel processes, but felt that the handshake assumption would lead to a simpler theory.

We have examined the semantic consequences of assuming *asynchronous communication* and (weakly) fair parallel composition as primitive. In contrast to the handshake case, we no longer need to deal with (more or less complicated variants of) synchronization trees or failure sets, and fairness is easy to incorporate: *traces* suffice. We show that a very simple trace-theoretic semantics is fully abstract with respect to a notion of behavior that subsumes safety and liveness properties. We present two equivalent but ostensibly different forms of trace semantics: communication traces and transition traces. The equivalence of these two forms of trace models enables us to bring out the essential underlying similarities between two paradigms of concurrency which traditionally have been provided with mathematically incompatible semantic models: shared variable programs, and asynchronous communicating processes, including non-deterministic Kahn-style networks. The advantages of our approach are explored in [3, 4, 5, 6, 7, 8]. Both paradigms can be interpreted in the *same* semantic framework – transition traces – and this unification of paradigms makes it easier to show how to augment parallel programming with features such as a procedure mechanism or an object-oriented style.

References

[1] J. A. Bergstra, J.W. Klop, and J. V. Tucker, *Process algebra with asynchronous communication*, Proc. Seminar on Concurrency, Springer LNCS 197, pp. 76-95 (1985).

[2] S. Brookes, *A model for communicating sequential processes*, D. Phil. thesis, Oxford University (1983).

[3] S. Brookes, *Full abstraction for a shared-variable parallel language*, Information and Computation, vol 127, No. 2, Academic Press (June 1996).

[4] S. Brookes, *The essence of Parallel Algol*, Proc. 11th IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press (1996) 164–173. To appear in *Information and Computation*.

[5] S. Brookes, *Idealized CSP: Combining Procedures with Communicating Processes*, 13^{th} Conference on Mathematical Foundations of Programming Semantics (MFPS'97), Pittsburgh, March 1997. Electronic Notes in Theoretical Computer Science 6, Elsevier Science (1997). URL: `http://www.elsevier.nl/locate/entcs/volume6.html`.

[6] S. Brookes, *On the Kahn Principle and Fair Networks*, 14^{th} Conference on Mathematical Foundations of Programming Semantics, Queen Mary Westfield College, University of London, May 1998.

[7] S. Brookes, *Reasoning about Recursive Processes: Expansion is not always fair*, Proc. 15^{th} Conference on Mathematical Foundations of Programming Semantics, Tulane University, New Orleans, April 1999, Electronic Notes in Theoretical Computer Science 20, Elsevier (1999), URL: http://www.elsevier.nl/locate/entcs/volume20.html.

[8] S. Brookes, *Communicating Parallel Processes*, Symposium in Celebration of the work of C.A.R.Hoare, Oxford University, September 1999. To appear, MacMillan Publishers (2000).

[9] S. Brookes and S. Older, *Full abstraction for strongly fair communicating processes*, 11^{th} Conference on Mathematical Foundations of Programming Semantics, New Orleans, March 1995. Electronic Notes in Theoretical Computer Science 1, Elsevier Science. URL: `http://www.elsevier.nl/locate/entcs/volume1.html`

[10] S. Brookes, C. A. R. Hoare, and A. W. Roscoe, *A Theory of Communicating Sequential Processes*, JACM (July 1984).

[11] S. Brookes, and A. W. Roscoe, *An improved failures model for CSP*, Seminar on concurrency, Springer-Verlag, LNCS 197, 1984.

[12] R. Cleaveland, J. Parrow, and B. Steffen, *The Concurrency Workbench: A Semantics-based tool for the Verification of Concurrent Systems*, ACM TOPLAS, vol. 15, no. 1, pp. 36-72 (1993).

[13] G. Costa and C. Stirling, *A fair calculus of communicating systems*, ACTA Informatica 21:417-441 (1984).

[14] G. Costa and C. Stirling, *Weak and strong fairness in CCS*, Technical Report CSR-16-85, University of Edinburgh, January 1985.

[15] F. de Boer, J.W. Klop, and C. Palamidessi, *Asynchronous Communication in Process Algebra*, Proc. LICS'92, IEEE Computer Society Press (1992).

[16] F. de Boer, J.W. Klop, C. Palamidessi, and J. Rutten, *The failure of failures: Towards a paradigm for asynchronous communication*, Proc. CONCUR'91, Springer LNCS 527, pp. 111-126 (1991).

[17] Formal Systems (Europe), Ltd., *Failures-Divergence Refinement: FDR2 Manual*, 1997.

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

[19] M. Hennessy, *An algebraic theory of fair asynchronous communicating processes*, Theoretical Computer Science, 49:121-143 (1987).

[20] M. Hennessy and G. Plotkin, *Full Abstraction for a Simple Parallel Language*, Proceedings of Mathematical Foundations of Computer Science, Springer-Verlag, LNCS vol. 74, 1979.

[21] M. Hennessy and G. Plotkin, *A term model for CCS*, Springer LNCS vol. 88 (1980).

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

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

[24] M. Josephs, C. A. R. Hoare, and He Jifeng, *A theory of asynchronous processes*, Technical Report, Oxford University Computaing Laboratory (1990).

[25] M. Josephs, *Receptive Process Theory*, Acta Informatica, vol. 29, no. 1, pp. 17-31, Springer-Verlag (1992).

[26] G. Kahn and D. MacQueen, *Coroutines and Networks of Parallel Processes*, Information Processing '77, North Holland, 1977.

[27] R. Milner, *Fully abstract models of typed λ-calculi*, Theoretical Computer Science, vol. 4, pp 1–22 (1977).

[28] R. Milner, **A Calculus of Communicating Systems**, Springer LNCS vol. 92 (1980).

[29] R. Milner, **Communication and Concurrency**, Springer-Verlag (1989).

[30] R. Milner, *A finite delay operator in CCS*, Technical Report CSR-116-82, University of Edinburgh (1982).

[31] R. Milner, *Calculi for Synchrony and Asynchrony*, Theoretical Computer Science, vol. 25, pp. 267-310 (1983).

[32] S. Older, *A Denotational Framework for Fair Communicating Processes*, Ph.D. thesis, Carnegie Mellon University (December 1996). Technical report CMU-CS-96-204.

[33] S. Older, *A Framework for Fair Communicating Processes*, Proc. 13th Conference on Mathematical Foundations of Programming Semantics, Electronic Notes in Computer Science 6, Elsevier Science (1997). URL: `http://www.elsevier.nl/locate/entcs/volume6.html`

[34] D. Park, *On the semantics of fair parallelism.* In D. Bjørner, editor, **Abstract Software Specifications**, Springer-Verlag LNCS vol. 86 (1979), 504–526.

[35] D. Park, *Concurrency and automata on infinite sequences*, Springer LNCS vol.l 104 (1981).

[36] J. Parrow, *Fairness Properties in Process Algebras*, Ph. D. thesis, University of Uppsala (1985).

[37] A. W. Roscoe, *A mathematical theory of communicating processes*, D. Phil. thesis, Oxford University (1982).

[38] A. W. Roscoe, **The Theory and Practice of Concurrency**, Prentice-Hall, 1998.

[39] A. Stoughton, *Fully Abstract Models of Programming Languages*, Research Notes in Theoretical Computer Science, Pitman (1988).

[40] A. Valmari and A. Tienari, *Compositional Failure-Based Semantics for Basic LOTOS*, Formal Aspects of Computing, 7:440-468 (1995).

[41] D. Walker, *Bisimulations and divergence*, Proc. LICS'88, IEEE Computer Society Press, pp. 186-192 (1988).