Retracing the semantics of CSP

Stephen Brookes

Carnegie Mellon University

Abstract. CSP was originally introduced as a parallel programming language in which sequential imperative processes execute concurrently and communicate by synchronized input and output. The influence of CSP and the closely related process algebra TCSP is widespread. Over the years CSP has been equipped with a series of denotational semantic models, involving notions such as communication traces, failure sets, and divergence traces, suitable for compositional reasoning about safety properties and deadlock analysis. We revisit these notions (and review some of the underlying philosophy) with the benefit of hindsight, and we introduce a semantic framework based on action traces that permits a unified account of shared memory parallelism, asynchronous communication, and synchronous communication. The framework also allows a relatively straightforward account of (a weak form of) fairness, so that we obtain models suitable for compositional reasoning about liveness properties as well as about safety properties and deadlock. We show how to incorporate race detection into this semantic framework, leading to models more independent of hardware assumptions about the granularity of atomic actions.

1 Introduction

The parallel programming language CSP was introduced in Tony Hoare's classic paper [15]. As originally formulated, CSP is an imperative language of guarded commands [11], extended with primitives for input and output and a form of parallel composition which permits synchronized communication between named processes. The original language derives its full name from the built-in syntactic constraint that processes belong to the sequential subset of the language. The syntax of programs was also constrained to preclude concurrent attempts by one process to write to a variable being used by another process: this may be expressed succinctly as the requirement that processes have "disjoint local states". These design decisions, influenced by Dijkstra's principle of "loose coupling" [10], lead to an elegant programming language in which processes interact solely by message-passing. Ideas from CSP have passed the test of time, having influenced the design of more recent parallel programming languages such as Ada, *occam* [18], and Concurrent ML [26].

Most of the subsequent foundational research has focussed on a process algebra known as *Theoretical CSP* (or *TCSP*) in which the imperative aspects of the original language are suppressed [2]. In TCSP (and in *occam*) processes communicate by message-passing along named channels, again using a synchronized handshake for communication. TCSP permits nested parallelism and recursive process definitions, and includes a form of localization for events known as *hiding*. Instead of Dijkstra-style guarded commands TCSP includes two forms of "choice": internal choice, and external choice. The internal form of choice corresponds to a guarded command with purely boolean guards: if more than one guard is true the selection of branch to execute is non-deterministic and made "internally" without consideration of the surrounding context. An external choice corresponds to a guarded command with input guards, for which the "truth" of a guard depends on the availability of matching output in the surrounding context.

Hoare's early paper on CSP [15] presented an informal sketch of a semantics for processes, expressed in intuitive terms with the help of operational intuition. Plotkin later gave a more formal structured operational semantics for a semantically more natural extension of the language [25]. Plotkin employed a more generous syntax allowing nested parallelism, and a more flexible scoping mechanism for process naming. Over the years CSP has become equipped with a series of semantic models of successively greater sophistication, each designed to support compositional reasoning about a specific class of program property.

In 1980 Hoare introduced a mathematical account of communication traces that developed more rigorously from the intuitions outlined in the original CSP paper [16]. In this model a process is taken to denote a set of communication traces, built from events which represent abstract records of communication. A trace here represents a partial history of the communication sequence occurring when a process interacts with its environment; since communication is synchronized an input or output event really stands for a potential for communication. And since traces record a partial behavior it is natural to work with (non-empty) prefix-closed sets of traces. This semantics is suitable for reasoning about simple safety properties of processes, but is too abstract for many purposes since it ignores the potential for deadlock. For example, the processes

$$\begin{array}{l} \mathbf{if}\ (\mathbf{true} \rightarrow a?x; h!x) \,\square\, (\mathbf{true} \rightarrow b?x; h!x)\ \mathbf{fi} \\ \mathbf{if}\ (a?x \rightarrow h!x) \,\square\, (b?x \rightarrow h!x)\ \mathbf{fi} \end{array}$$

have the same set of communication traces, but only the first one may deadlock when run in parallel with a process such as $b!0$. Moreover, if **stop** is a process incapable of any communication (so that its only communication trace is the empty trace), the processes

$$\begin{array}{l} \mathbf{if}\ (a?x \rightarrow h!x)\ \mathbf{fi} \\ \mathbf{if}\ (a?x \rightarrow h!x) \,\square\, (a?x \rightarrow \mathbf{stop})\ \mathbf{fi} \\ \mathbf{if}\ (a?x \rightarrow h!x) \,\square\, (\mathbf{true} \rightarrow \mathbf{stop})\ \mathbf{fi} \end{array}$$

have the same communication traces (because of prefix-closure) although there are clear operational reasons to distinguish between these processes.

The need to model deadlock led to the *failures* model of Hoare, Brookes and Roscoe [2], in which communication traces were augmented with information about the potential for further communication, represented abstractly (and negatively) as a refusal set. A failure (α, X) consists of a communication trace α and a set X of events, representing the ability to perform the communications in α and then refuse to perform any of the events in X. (Obviously it is equally reasonable to develop a positively formulated notion of acceptance set or ready set rather than refusal [22].) Again operational and observational intuitions suggest that a process should denote a set of failures closed under certain natural rules. The mathematical foundations of the failures model were explored more deeply in the D.Phil. theses of Bill Roscoe and myself [1, 27]. A more readily accessible account, which also discusses a variety of related semantic models, is obtainable in Roscoe's book [28].

The failures model, like the communication traces model from which it evolved, allows compositional reasoning about safety properties; but failures also permit distinctions based on the potential for deadlock. Revisiting the above examples, the processes

$$\begin{array}{l} \mathbf{if}\ (\mathbf{true} \rightarrow a?x; h!x) \,\square\, (\mathbf{true} \rightarrow b?x; h!x)\ \mathbf{fi} \\ \mathbf{if}\ (a?x \rightarrow h!x) \,\square\, (b?x \rightarrow h!x)\ \mathbf{fi} \end{array}$$

do *not* denote the same set of failures: only the first process can refuse to input on a (or refuse to input on b). Similarly the processes

$$\begin{array}{l}\textbf{if } (a?x \rightarrow h!x) \textbf{ fi}\\ \textbf{if } (a?x \rightarrow h!x) \,\square\, (a?x \rightarrow \textbf{stop}) \textbf{ fi}\\ \textbf{if } (a?x \rightarrow h!x) \,\square\, (\textbf{true} \rightarrow \textbf{stop}) \textbf{ fi}\end{array}$$

do not have the same failures. The behavioral distinctions between these examples, expressible in terms of failures, have a natural operational intuition.

The failures model, although offering good support for safety properties and deadlock analysis, still suffers from a deficiency with respect to the phenomenon of *infinite internal chatter*, or *divergence.* We illustrate the problem with an example. The program

$$\begin{array}{l}\textbf{chan } a \textbf{ in}\\ \qquad (\textbf{while true do } a?x) \parallel (\textbf{while true do } a!0)\end{array}$$

involves two processes which keep communicating "internally" on the hidden channel a. Externally, no visible communication is apparent, and it is natural to ask what responses, if any, the program should be deemed to provide to its environment if the environment offers a potential for communication. Presumably the environment cannot ever discover in a finite amount of time that the program will never become capable of communication, since the program never reaches a "stable" configuration. There is no way to represent this kind of behavior adequately within the confines of the failures model, since divergence (while doing no external communication) could only be represented by failures containing the empty trace (and all possible refusal sets), but this would be tantamount to equating deadlock with divergence.

In response to this problem, Brookes and Roscoe proposed a further augmentation of failures to incorporate *divergence traces* [3]. A divergence trace represents a sequence of communications leading to a possible divergent behavior. For pragmatic reasons, again based on observability criteria and the view that a well behaved process should respond to its environment in a finite amount of time, divergence is treated as a catastrophe in this model. Thus, a process is taken to denote a failure set F, together with a set D of divergence traces, satisfying the following catastrophic closure rule:

- if $\alpha \in D$ then for all traces β and refusal sets X, $(\alpha\beta, X) \in F$, and for all traces β, $\alpha\beta \in D$.

As an example, the simple divergent program listed above has the empty trace as a possible divergence trace, from which it follows from the closure rule that its denotation includes all failures and all divergence traces. In contrast the denotation of a deadlocked process would consist of all failures involving the empty trace, together with the empty set of divergence traces.

The failures/divergences model, despite its rather awkward name, has become the standard semantics for an enormous range of CSP research and implementation [17]. This model underpins the FDR model checker [12], which has been used successfully for the analysis of (and detection of bugs in) parallel systems and protocols [30]. Roscoe and Hoare have also shown how to incorporate state directly into the structure of failure sets, in developing a failure-style semantics for *occam* [29].

2 Reflection

In these early models of CSP the focus is on finite behaviors, with infinite traces either ignored or regarded as being present only by virtue of finite prefixes. Consequently these models did not take fairness into account. Yet fairness assumptions, such as the guarantee that a process waiting for input will eventually be synchronized if another process is simultaneously (and persistently) waiting for a matching output, are vital when trying to reason about liveness properties [24, 23]. As a result it can be argued that these models are, by their very design, less than ideally suited to reasoning about liveness.

Although it is possible to develop straightforward variants of these models that incorporate infinite traces [28], it is not obvious how to augment them in such a way that only *fair* traces get included. Indeed there is a plethora of distinct fairness notions in the literature [13], and it is not clear which (if any) of these notions are simultaneously a reasonable abstraction from network implementation and adaptable to CSP. Susan Older's Ph.D. thesis [21] contains a detailed discussion of the problems that arise as well as a family of models tailored to specific fairness notions. Older's models can be regarded as failures/divergences equipped with infinite traces and book-keeping information about persistently enabled communication [6]. As Older discovered, it can be very difficult to figure out a suitable augmentation regime for extending failures/divergences to match a given notion of fairness, largely because of the fact that enabledness of communication for one process depends on enabledness of matching communication in another process. The difficulties seem less severe when dealing with asynchronous communication [5].

The models described so far were developed specifically with TCSP in mind, and serve this role admirably. However, CSP is not the only paradigmatic parallel programming language and it is natural to compare the semantic framework built for CSP with the models developed over the years for shared memory parallel programs and for networks of asynchronously communicating processes. By the same token, TCSP is not the only process algebra, and the emphasis in CSP on deadlock and divergence is in sharp contrast to the focus on *bisimulation* in calculi based on Milner's CCS [19, 20]. Unfortunately there is frustratingly little similarity in structure between the early semantic models developed for these other paradigms and these CSP models. For instance the *resumptions* of Hennessy and Plotkin [14], and the *transition traces* of Park [24] (later adapted by this author [4]), originally proposed to model shared memory parallel programs, bear no obvious structural relationship with failures.

These semantic disparities make it difficult to apply techniques successful in one setting to similar problems occurring in the other settings. For instance, Older's construction of fair models of CSP does not immediately suggest an analogous construction for a language of asynchronously communicating processes. A further disparity is caused by the emphasis (for obvious reasons) on state in traditional models of shared memory parallelism, in contrast to the prevailing tendency in process algebras such as CSP and CCS to abstract away from state [19, 2].

In a paper presented in tribute to the twentieth anniversary of CSP [7] this author proposed a semantic model based on transition traces, suitable for modelling both shared memory parallel programs and networks of processes communicating asynchronously on named channels. At that time it seemed unlikely that similar techniques would prove suitable for modelling synchronized communication, because of the difficulties encountered by Older in adapting failures to fairness in the synchronous setting. Nevertheless the author discovered later that essentially the same framework can also be made to work for synchronously communicating processes, provided a simple enough notion of fairness is adopted [8]. This is a somewhat surprising turn of events given the prior history of separate development. More recently still, we realized that it is possible to modify this semantic framework in a natural way to handle *race conditions*, leading to an improved semantics in which assumptions about the granularity of atomic actions become less significant [9]. We will now summarize the main technical notions behind this semantics. The key turns out to involve the choice of a suitably general notion of trace, which can be presented in a process algebraic formulation and separately instantiated later in a state-dependent setting.

3 Communicating Parallel Processes

We will work with a language combining shared memory parallelism with communicating processes. Thus processes will be allowed to share state and will be permitted to interact by reading and writing to shared variables as well as by sending and receiving messages. We also include resources and conditional critical regions to allow synchronization and mutually exclusive access to critical data.

Let P range over *processes* and G over *guarded processes*, given by the following abstract grammar, in which e ranges over integer-valued expressions, b over boolean expressions, h over the set **Chan** of channel names, x over the set **Ide** of identifiers, and r over resource names. We omit the syntax of expressions, which is conventional.

$$\begin{aligned}
P ::= {} & \mathbf{skip} \mid x{:=}e \mid P_1; P_2 \mid \mathbf{if}\ b\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2 \mid \mathbf{while}\ b\ \mathbf{do}\ P \mid \\
& P_1 \| P_2 \mid \mathbf{with}\ r\ \mathbf{when}\ b\ \mathbf{do}\ P \mid \mathbf{resource}\ r\ \mathbf{in}\ P \mid \\
& h?x \mid h!e \mid \mathbf{if}\ G\ \mathbf{fi} \mid \mathbf{do}\ G\ \mathbf{od} \mid P_1 \sqcap P_2 \mid \mathbf{chan}\ h\ \mathbf{in}\ P \\
G ::= {} & (h?x \to P) \mid G_1 \Box G_2
\end{aligned}$$

As in CSP, $P_1 \sqcap P_2$ is “internal” choice, and $G_1 \Box G_2$ is “external” choice. We distinguish syntactically between guarded and general processes merely to enforce the constraint that the “external choice” construct is only applicable to input-guarded processes. This allows certain simplifications in the semantic definitions but is not crucial. It is straightforward to extend our semantics to allow mixed boolean and input guards.

The construct **chan** h **in** P introduces a local channel named h with scope P. One can also allow locally scoped variable declarations, but we omit the details. We write $\mathit{chans}(P)$ for the set of channel names occurring free in P. In particular, $\mathit{chans}(\textbf{chan } h \textbf{ in } P) = \mathit{chans}(P) - \{h\}$.

A process of form **resource** r **in** P introduces a local resource name r, whose scope is P. A process of form **with** r **when** b **do** P is a *conditional critical region* for resource r, with body P. A process attempting to enter such a region must wait until the resource is available, acquire the resource and evaluate b: if b is **true** the process executes P then releases the resource; if b is **false** the process releases the resource and waits to try again. A resource can only be held by one process at a time. We use the abbreviation **with** r **do** P when b is **true**.

4 Actions

The behavior of a process will be explained in terms of the *actions* that it can perform. An action can be regarded as an atomic step which may or may not be enabled in a given state, and if enabled has an effect on the state. Let V_{int} be the set of integers, with typical member v. An *action* has one of the following forms:

- A *read* $x{=}v$, where x is an identifier and v is an integer.
- A *write* $x{:=}v$, where x is an identifier and v is an integer.
- A *communication* $h?v$ or $h!v$, where h is a channel name and v is an integer. Each communication action has a *direction*: $h!$ for output, $h?$ for input, on a specific channel h.
- A *idling action* of form δ_X, where X is a finite set of directions.
- A *resource action* of one of the forms $\mathit{try}(r)$, $\mathit{acq}(r)$, $\mathit{rel}(r)$, where r is a resource name.
- An error action *abort*.

We will not yet provide formal details concerning states and effects, relying instead for now on the following intuitions.

A read $x{=}v$ is enabled only in a state for which the current value of x is v, and causes no state change. A write $x{:=}v$ is only enabled in states for which x has a current value, and its effect is to change the value of x to v.

An input action $h?v$ or output action $h!v$ represents the *potential* for a process to perform communication, and can only be completed when another process offers a matching communication on the same channel. We write $\mathit{match}(\lambda_1, \lambda_2)$ when λ_1 and λ_2 are matching actions, i.e. when there is a channel h and an integer v such that $\{\lambda_1, \lambda_2\} = \{h?v, h!v\}$. We let $\mathit{chan}(h?v) = \mathit{chan}(h!v) = h$.

An idling action δ_X represents an unrequited attempt to communicate along the directions in X. When X is a singleton we write $\delta_{h?}$ or $\delta_{h!}$. When X is empty we write δ instead of $\delta_{\{\}}$; the action δ is also used to represent a "silent" local action, such as a synchronized handshake or reading or writing a local variable.

An action of form $\mathit{try}(r)$ represents an unsuccessful attempt to acquire resource r, and $\mathit{acq}(r)$ represents a successful attempt to do so; $\mathit{rel}(r)$ represents the act of releasing the resource. Parallel execution is assumed to be constrained to ensure that at all stages each resource is being held by at most one process. Thus at all stages the sets of resources belonging to each process will be disjoint. Correspondingly, for an action λ and a disjoint pair of resource sets A_1 and A_2 we define a *resource enabling* relation $(A_1, A_2) \xrightarrow{\lambda} (A'_1, A'_2)$, characterized by the following rules:

$$\begin{array}{lll}
(A_1, A_2) \xrightarrow{\mathit{try}(r)} (A_1, A_2) & & \\
(A_1, A_2) \xrightarrow{\mathit{acq}(r)} (A_1 \cup \{r\}, A_2) & & \text{if } r \notin A_1 \cup A_2 \\
(A_1, A_2) \xrightarrow{\mathit{rel}(r)} (A_1 - \{r\}, A_2) & & \text{if } r \in A_1 \\
(A_1, A_2) \xrightarrow{\lambda} (A_1, A_2) & & \text{otherwise}
\end{array}$$

Since A_1 and A_2 are disjoint, if $(A_1, A_2) \xrightarrow{\lambda} (A'_1, A'_2)$ it follows that $A'_2 = A_2$, and A'_1 is disjoint from A_2. Intuitively, when $(A_1, A_2) \xrightarrow{\lambda} (A'_1, A_2)$ holds, a process holding resources A_1 can safely perform action λ in a parallel environment that holds resources A_2, and will hold resources A'_1 afterwards.

The *abort* action represents a runtime error, is always enabled, and leads to an error state, which we denote **abort**.

5 Action traces

An action trace is a non-empty finite or infinite sequence of actions. Let $\mathbf{Tr}$ be the set of action traces; we will use α, β, γ as meta-variables ranging over traces. We write $\alpha\beta$ for the trace obtained by concatenating α and β. We assume that δ behaves as a unit for concatenation, and *abort* behaves as a zero for concatenation, in that $\alpha\delta\beta = \alpha\beta$ and $\alpha\, \mathit{abort}\, \beta = \alpha\, \mathit{abort}$, for all traces α and β.

We assume given the trace semantics for expressions, so that for an integer expression e we have $\llbracket e \rrbracket \subseteq \mathbf{Tr} \times V_{\mathit{int}}$. Similarly for a boolean expression b we have $\llbracket b \rrbracket \subseteq \mathbf{Tr} \times \{\mathbf{true}, \mathbf{false}\}$. We let $\llbracket b \rrbracket_{\mathbf{true}} = \{\rho \mid (\rho, \mathbf{true}) \in \llbracket b \rrbracket\}$ and similarly for $\llbracket b \rrbracket_{\mathbf{false}}$. The only actions occurring in expression traces are δ and reads.

A process denotes a set of traces, denoted $\llbracket P \rrbracket \subseteq \mathbf{Tr}$. The semantics of processes is defined denotationally, by structural induction. We list here some of the key clauses.

$$\begin{array}{l}
\llbracket h!e \rrbracket = {\delta_{\{h!\}}}^{\infty}\{\rho\, h!v \mid (\rho, v) \in \llbracket e \rrbracket\} \\
\llbracket h?x \rrbracket = {\delta_{\{h?\}}}^{\infty}\{h?v\, x{:=}v \mid v \in V_{\mathit{int}}\} \\
\llbracket P_1; P_2 \rrbracket = \llbracket P_1 \rrbracket\, \llbracket P_2 \rrbracket \\
\llbracket P_1 \| P_2 \rrbracket = \llbracket P_1 \rrbracket\ {}_{\{\}}\|_{\{\}}\ \llbracket P_2 \rrbracket \\
\llbracket \mathbf{with}\ r\ \mathbf{when}\ b\ \mathbf{do}\ P \rrbracket = \mathit{wait}^{\infty}\ \mathit{enter} \\
\qquad \text{where } \mathit{wait} = \{\mathit{try}(r)\} \cup \mathit{acq}(r)\, \llbracket b \rrbracket_{\mathbf{false}}\, \mathit{rel}(r) \\
\qquad \text{and } \ \mathit{enter} = \mathit{acq}(r)\, \llbracket b \rrbracket_{\mathbf{true}}\, \llbracket P \rrbracket\, \mathit{rel}(r) \\
\llbracket \mathbf{chan}\ h\ \mathbf{in}\ P \rrbracket = \{\alpha \backslash h \mid \alpha \in \llbracket P \rrbracket\ \&\ h \notin \mathit{chans}(\alpha)\}
\end{array}$$

The semantic clauses for input and output commands include traces that represent infinite waiting for matching output and input, respectively. Sequential composition corresponds to concatenation of traces. The trace set of a conditional critical region reflects the operational behavior discussed earlier: waiting until the resource can be acquired and the test expression evaluates to **true**. In the special case where b is **true** we can derive the following simpler formula:

$$\llbracket \mathbf{with}\ r\ \mathbf{do}\ P \rrbracket = \mathit{try}(r)^{\infty}\ \mathit{acq}(r)\, \llbracket P \rrbracket\, \mathit{rel}(r).$$

The clause for a local channel declaration "forces" synchronization to occur on the local channel. We write $\alpha \backslash h$ for the trace obtained from α by deleting $h!$ and $h?$ from all sets of directions occurring in idling actions along α.

The clause for parallel composition involves a form of *mutex fairmerge* for trace sets. When A_1 and A_2 are disjoint sets of resource names and T_1 and T_2 are trace sets, $T_1 \,{}_{A_1}\|_{A_2}\, T_2$ denotes the set of all (synchronizing) interleavings of a trace from T_1 with a trace from T_2, subject to the constraint that the process executing T_1 starts with resources A_1 and the process executing T_2 starts with resources A_2, and at all stages the resources held by the two processes stay disjoint. For each pair of traces α_1 and α_2 we define the set of traces $\alpha_1 \,{}_{A_1}\|_{A_2}\, \alpha_2$, and then we let $T_1 \,{}_{A_1}\|_{A_2}\, T_2 = \bigcup\{\alpha_1 \,{}_{A_1}\|_{A_2}\, \alpha_2 \mid \alpha_1 \in T_1 \ \&\ \alpha_2 \in T_2\}$.

We design this fairmerge operator so that the potential of a *race* (concurrent execution of actions which may interfere in an unpredictable manner) is treated as a catastrophe. We write $\lambda_1 \bowtie \lambda_2$ to indicate a race, given by the following rules:

$$\begin{array}{c} x{=}v \bowtie x{:=}v' \\ x{:=}v \bowtie x{=}v' \\ x{:=}v \bowtie x{:=}v' \\ h!v \bowtie h!v' \\ h?v \bowtie h?v' \end{array}$$

In particular, we regard as a race any concurrent attempt to write to a variable being read or written by another process. And we also treat concurrent attempts to input to the same channel, or to output to the same channel, as a race.

For finite traces (including the empty sequence to allow a simpler base case) the set of mutex fairmerges using any given pair of disjoint resource sets can be characterized inductively from the following clauses:

$$\begin{array}{l}
\alpha \,{}_{A_1}\|_{A_2}\, \epsilon = \{\alpha \mid (A_1, A_2) \xrightarrow{\alpha} \cdot\} \\
\epsilon \,{}_{A_1}\|_{A_2}\, \alpha = \{\alpha \mid (A_2, A_1) \xrightarrow{\alpha} \cdot\} \\
(\lambda_1\alpha_1) \,{}_{A_1}\|_{A_2}\, (\lambda_2\alpha_2) = \{\mathit{abort}\} \quad \text{if } \lambda_1 \bowtie \lambda_2 \\
(\lambda_1\alpha_1) \,{}_{A_1}\|_{A_2}\, (\lambda_2\alpha_2) = \\
\qquad\qquad \{\lambda_1\gamma \mid (A_1, A_2) \xrightarrow{\lambda_1} (A_1', A_2) \ \&\ \gamma \in \alpha_1 \,{}_{A_1'}\|_{A_2}\, (\lambda_2\alpha_2)\} \\
\qquad\quad \cup \{\lambda_2\gamma \mid (A_2, A_1) \xrightarrow{\lambda_2} (A_2', A_1) \ \&\ \gamma \in (\lambda_1\alpha_1) \,{}_{A_1}\|_{A_2'}\, \alpha_2\} \\
\qquad\quad \cup \{\delta\gamma \mid \mathit{match}(\lambda_1, \lambda_2) \ \&\ \gamma \in \alpha_1 \,{}_{A_1}\|_{A_2}\, \alpha_2\} \\
\qquad\qquad\quad \text{if } \neg(\lambda_1 \bowtie \lambda_2)
\end{array}$$

The above clauses actually suffice for all pairs of traces, one of which is finite. We can extend this mutex fairmerge relation to pairs of infinite traces in a natural manner, imposing a fairness constraint that reflects our assumption that a pair of processes waiting for a matching pair of communications will eventually get scheduled to communicate. This is a variant of weak process fairness adapted to take account of the synchronization mechanism used for CSP-style communication. Although we omit the full definition, note the following special case involving two infinite waiting traces:

$$\delta_X{}^{\omega}\;{}_{A_1}\|_{A_2}\;\delta_Y{}^{\omega} = \{\}$$

if $\exists h.\ (h? \in X \ \&\ h! \in Y) \vee (h! \in X \ \&\ h? \in Y)$. This captures formally the fairness assumption from above: there is no fair way to interleave the actions of these two traces because there is a persistent opportunity for synchronization that never gets taken.

6 Examples

We now revisit the examples discussed earlier, previously used to illustrate communication traces, failures, and divergences.

First, we contrast the action traces of an internal choice with those of the corresponding external choice.

$$\begin{aligned}[\![(a?x \to h!x) \sqcap (b?x \to h!x)]\!] = {} & \delta_{\{a?\}}{}^{\infty}\{a?v\; x{:=}v\; h!v \mid v \in V_{\mathit{int}}\} \\ & \cup\ \delta_{\{b?\}}{}^{\infty}\{b?v\; x{:=}v\; h!v \mid v \in V_{\mathit{int}}\}\end{aligned}$$

$$\begin{aligned}[\![\mathbf{if}\ (a?x \to h!x) \,\square\, (b?x \to h!x)\ \mathbf{fi}]\!] = {} & \delta_{\{a?,b?\}}{}^{\infty}\{a?v\; x{:=}v\; h!v \mid v \in V_{\mathit{int}}\} \\ & \cup\ \delta_{\{a?,b?\}}{}^{\infty}\{b?v\; x{:=}v\; h!v \mid v \in V_{\mathit{int}}\}\end{aligned}$$

As with the failures model this semantics distinguishes properly between these processes. There is an obvious sense in which the ability to "refuse" input on a is represented here by the presence of a trace involving infinite waiting for input on b.

Similarly we have

$$[\![a?x \to h!x]\!] = \delta_{\{a?\}}{}^{\infty}\{a?v\; x{:=}v\; h!v \mid v \in V_{\mathit{int}}\}$$

$$\begin{aligned}& [\![\mathbf{if}\ (a?x \to h!x) \,\square\, (a?x \to \mathbf{stop})\ \mathbf{fi}]\!] = \\ & \qquad \delta_{\{a?\}}{}^{\infty}\{a?v\; x{:=}v\; h!v \mid v \in V_{\mathit{int}}\} \\ & \quad\ \cup\ \delta_{\{a?\}}{}^{\infty}\{a?v\; x{:=}v\; \alpha \mid \alpha \in [\![\mathbf{stop}]\!]\ \&\ \ \mid v \in V_{\mathit{int}}\}\end{aligned}$$

so that again we distinguish correctly between these processes. (We note in passing here that every process, even **stop**, denotes a non-empty set of traces.)

The divergent program discussed earlier,

$$\begin{array}{l}\textbf{chan } a \textbf{ in} \\ \quad (\textbf{while true do } a?x) \parallel (\textbf{while true do } a!0),\end{array}$$

denotes the action trace set $\{\delta^\omega\}$. We see no compelling reason to distinguish this program from a deadlocked process such as **stop** or the process

$$\textbf{chan } a \textbf{ in } (a?x; h!x),$$

which will never be able to engage in external communication on channel h because the local channel a is not in scope for any external process. This process waits forever for input on the local channel, a phenomenon that gives rise to the trace ${\delta_{\{\}}}^\omega$. Indeed our semantics gives this program the corresponding trace set $\{\delta^\omega\}$. We also choose *not* to interpret divergence as catastrophic, although it would be possible to derive a model along those lines by imposing suitable closure conditions on action trace sets.

The following semantic equivalences illustrate how our model supports reasoning about process behavior.

Theorem 1 (Synchronous Laws)
The following laws hold in the synchronous trace semantics:

1. $[\![\textbf{chan } h \textbf{ in } (h?x; P) \| (h!v; Q)]\!] = [\![\textbf{chan } h \textbf{ in } (x{:=}v;\ (P\|Q))]\!]$
2. $[\![\textbf{chan } h \textbf{ in } (h?x; P) \| (Q_1; Q_2)]\!] = [\![Q_1; \textbf{chan } h \textbf{ in } (h?x; P) \| Q_2]\!]$ *provided* $h \notin \texttt{chans}(Q_1)$
3. $[\![\textbf{chan } h \textbf{ in } (h!v; P) \| (Q_1; Q_2)]\!] = [\![Q_1; \textbf{chan } h \textbf{ in } (h!v; P) \| Q_2]\!]$ *provided* $h \notin \texttt{chans}(Q_1)$.

These laws reflect our assumption of fairness, and can be particularly helpful in proving liveness properties. They are not valid in an unfair semantics: if execution is unfair there is no guarantee in the first law that the synchronization will eventually occur, and there is no guarantee in the second or third laws that the right-hand process will ever execute its initial (non-local) code.

7 Granularity

Our semantics involves actions which represent rather high-level operations, such as assignments of an entire integer value to a variable, and communication of an entire integer value along a channel. Rather than assuming that such actions are implemented at the hardware or network level as indivisible atomic operations, we have designed our parallel composition so that any concurrent attempt to perform actions whose combined effect is sensitive to granularity is treated as a catastrophe. As a consequence, our semantics can be shown to be independent of granularity in a precise sense.

Specifically, we can give a "low-level" semantics for processes, based on "low-level" actions that represent fine-grained atomic steps. At low-level we assume that integers are represented as lists of words, with some fixed word size W, and that messages are transmitted as sequences of packets, of some fixed packet size M. A high-level read of the form $x{=}v$ then corresponds to a sequence of low-level reads of the form $x.0{=}w_0 \ldots x.k{:=}w_k$, where $x.0, \ldots, x.k$ represent the various components of x and the sequence of word values $w_0 \ldots w_k$ represents the integer v (for word size W). Similarly a high-level write corresponds to a sequence of low-level writes. A high-level input action $h?v$ corresponds to a sequence of low-level input actions terminated by an end-of-transmission signal, and similarly for an output action. Let us write $v = [w_0, \ldots, w_k]_W$ to indicate that the given word sequence represents v, i.e. that $v = w_0 + 2^W w_1 + \cdots + 2^{kW} w_k$, with a similar notation $v = [m_0, \ldots, m_n]_M$ for messages. In the low-level semantics, which we will denote $[\![P]\!]_{low}$, we would thus have:

$$\begin{aligned}
[\![h!e]\!]_{low} &= \{\rho\, h!m_0 \ldots h!m_n\, h!\mathtt{EOT} \mid (\rho, v) \in [\![e]\!] \;\&\; [m_0, \ldots, m_n]_M = v\} \\
[\![h?x]\!]_{low} &= \{h?m_0 \ldots h?m_n\, h?\mathtt{EOT}\, x.0{:=}w_0 \ldots x.k{:=}w_k \mid \\
&\qquad\qquad v = [m_0, \ldots, m_n]_M = [w_0, \ldots, w_k]_W\}
\end{aligned}$$

A high-level state σ describes the values of a finite collection of variables, so that σ is a finite partial function from identifiers to integers. A low-level state can be regarded as a finite partial function τ from identifiers to lists of words. The effect of a high-level action λ can be formalized as a partial function $\xRightarrow{\lambda}$ between high-level states, and similarly for low-level actions and their effect on low-level states. In both cases we use **abort** for an error state. There is an obvious way to define an appropriate notion of correspondence between high- and low-level states: $\sigma \approx \tau$ if $\mathtt{dom}(\sigma) = \mathtt{dom}(\tau)$ and, for each $x \in \mathtt{dom}(\sigma)$, $\sigma(x) = [\tau(x)]_W$. It can then be shown that for all processes P:

- for all high-level traces α of P, if $\sigma \approx \tau$ and $\sigma \overset{\alpha}{\Longrightarrow} \sigma' \neq$ **abort** then there is a low-level trace β of P and a low-level state τ' such that $\tau \overset{\beta}{\Longrightarrow} \tau'$ and $\sigma' \approx \tau'$;
- for all low-level traces β of P, if $\sigma \approx \tau$ and $\tau \overset{\beta}{\Longrightarrow} \tau'$ then there is a high-level trace α of P such that either $\sigma \overset{\alpha}{\Longrightarrow}$ **abort** or $\sigma \overset{\alpha}{\Longrightarrow} \sigma'$ for some high-level state σ' such that $\sigma' \approx \tau'$.

This formalizes the sense in which our high-level action trace semantics expresses behavioral properties of programs in a manner independent of assumptions about details such as word size or packet size. The role played in this result by the race-detecting clause in our definition of parallel composition is crucial.

8 Asynchrony

To model a language of asynchronously communicating processes, with the same syntax, we need only to make a few modifications in the key semantic definitions concerning communication. Specifically, the clauses for input, sequential composition, and conditional critical regions remain as before but we alter the clauses for input (since waiting is no longer required), parallel composition (since synchronization is no longer needed), and local channel declaration (since an output to a local channel can occur autonomously but we still require the inputs to obey the queueing discipline). The new clauses are:

$$\begin{aligned}
&[\![h!e]\!] = \{\rho\, h!v \mid (\rho, v) \in [\![e]\!]\} \\
&[\![P_1 \| P_2]\!] = [\![P_1]\!]\ {}_{\{\}}\|_{\{\}}\ [\![P_2]\!] \\
&[\![\textbf{chan}\ h\ \textbf{in}\ P]\!] = \{\alpha \backslash h \mid \alpha \in [\![P]\!]_h\}
\end{aligned}$$

We adjust the definition of interleaving, to delete the synchronization case:

$$\begin{aligned}
&(\lambda_1\alpha_1)\ {}_{A_1}\|_{A_2}\ (\lambda_2\alpha_2) = \{abort\} \quad \text{if } \lambda_1 \bowtie \lambda_2 \\
&(\lambda_1\alpha_1)\ {}_{A_1}\|_{A_2}\ (\lambda_2\alpha_2) = \\
&\qquad\qquad \{\lambda_1\gamma \mid (A_1, A_2) \xrightarrow{\lambda_1} (A_1', A_2)\ \&\ \gamma \in \alpha_1\ {}_{A_1'}\|_{A_2}\ (\lambda_2\alpha_2)\} \\
&\qquad\quad\ \cup \{\lambda_2\gamma \mid (A_2, A_1) \xrightarrow{\lambda_2} (A_2', A_1)\ \&\ \gamma \in (\lambda_1\alpha_1)\ {}_{A_1}\|_{A_2'}\ \alpha_2\} \\
&\qquad\qquad \text{otherwise}
\end{aligned}$$

We also need to adjust the definition of fairmerge for pairs of infinite traces to fit with the asynchronous interpretation, along the lines of [8] but modified to handle race conditions as above.

Given a trace set T and channel h, we let T_h be the set of traces α in T along which the actions involving h obey the queue discipline. We write $\alpha\backslash h$ for the trace obtained from α by replacing each action that mentions channel h by δ.

Theorem 2 (Asynchronous Laws)
The following laws hold in the asynchronous trace semantics:

1. $[\![\textbf{chan } h \textbf{ in } (h?x; P) \| (h!v; Q)]\!] = [\![\textbf{chan } h \textbf{ in } (x{:=}v; P) \| Q]\!]$
2. $[\![\textbf{chan } h \textbf{ in } (h?x; P) \| (Q_1; Q_2)]\!] = [\![Q_1; \textbf{chan } h \textbf{ in } (h?x; P) \| Q_2]\!]$ *provided* $h \notin \texttt{chans}(Q_1)$.

Again these laws reflect our assumption of fair execution. There is an obvious similarity with the corresponding pair of laws from the synchronous setting, but note the subtly different positioning of the assignment to x in the first law.

9 Conclusion

We have introduced a semantic framework based on action traces and a form of resource-sensitive, race-detecting, parallel composition. This can be used to provide models for a language combining shared memory parallelism with communicating processes. (We explore the use of such a semantics for shared memory programs, and connections with *separation logic*, in [9].) This language can be viewed as a generalization of CSP that retains and expands on the imperative essence of original CSP yet possesses a semantic model that reflects the elegance of the design principles behind the original language. Action traces allow the expression of concepts such as failures and divergences, familiar from the traditional models of CSP, without the need to commit to a catastrophic treatment of divergence.

The syntactic constraints built into the original version of CSP – disjoint local states, no nested parallelism, and restricted patterns of communication between processes because of the naming discipline – are sufficient to rule out race conditions, so that for programs in the original CSP we could adapt our semantics in the obvious manner, by deleting the race-detection clauses. Our language – and semantics – allow a more generous syntax within which one can reason about program behavior in a manner independent of hardware assumptions.

References

1. Brookes, S., *A model for communicating sequential processes*, D. Phil. thesis, Oxford University (1983).
2. Brookes, S.D. and Hoare, C.A.R., and Roscoe, A.W., *A theory of communicating sequential processes*, JACM 31(3):560–599 (1984).
3. Brookes, S. and Roscoe, A.W., *An improved failures model for CSP*, Proc. Seminar on Concurrency, Springer-Verlag LNCS 197, 1985.
4. Brookes, S., *Full abstraction for a shared-variable parallel language*, Proc. 8th IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press (1993), 98–109.
5. Brookes, S., *Fair communicating processes*, in A.W. Roscoe (ed.), **A Classical Mind: Essays in Honour of C.A.R. Hoare**, Prentice-Hall International (1994), 59–74.
6. Brookes, S., and Older, S., *Full abstraction for strongly fair communicating processes*, Proc. 11th Conference on Mathematical Foundations of Programming Semantics (MFPS'95), ENTCS vol. 1, Elsevier Science B. V. (1995).
7. Brookes, S., *Communicating Parallel Processes*, Symposium in Celebration of the work of C.A.R. Hoare, Oxford University, MacMillan (2000).
8. Brookes, S., *Traces, Pomsets, Fairness and Full Abstraction for Communicating Processes*, Proc. CONCUR'02, Springer-Verlag, 2002.
9. Brookes, S., *A semantics for concurrent separation logic*, to appear in: Proc. CONCUR'04, Springer-Verlag, September 2004.
10. Dijsktra, E. W., *Cooperating sequential processes*, in: **Programming Languages**, in F. Genuys (ed.), Academic Press (1968), 43–112.
11. Dijkstra, E. W., *Guarded Commands, Nondeterminacy, and Formal Derivation of Programs*, Comm. ACM 18(8):453–457 (1975).
12. Formal Systems (Europe) Ltd, *Failures-Divergence Refinement: FDR2 Manual*, 1997.
13. Francez, N., **Fairness**, Springer-Verlag (1986).
14. Hennessy, M. and Plotkin, G.D., *Full abstraction for a simple parallel programming language*, Proc. 8th MFCS, Springer-Verlag LNCS vol. 74, pages 108-120 (1979).
15. Hoare, C. A. R., *Communicating Sequential Processes*, Comm. ACM, 21(8):666–677 (1978).
16. Hoare, C.A.R., *A model for communicating sequential processes*, in: **On the construction of programs**, McKeag and McNaughton (eds.), Cambridge University Press (1980).
17. Hoare, C.A.R., **Communicating Sequential Processes**, Prentice-Hall (1985).
18. Inmos Ltd., *occam2* reference manual, Prentice-Hall (1988).
19. R. Milner, *A Calculus of Communicating Systems*, Springer LNCS 92, 1980.
20. R. Milner, **Communication and Concurrency**, Prentice-Hall, London, 1989.
21. Older, S., *A Denotational Framework for Fair Communicating Processes*, Ph.D. thesis, Carnegie Mellon University, (1997).

22. Olderog, E-R., and Hoare, C.A.R., *Specification-oriented semantics for communicating processes*, Acta Informatica 23, 9-66, 1986.
23. S. Owicki and L. Lamport, *Proving liveness properties of concurrent programs*, ACM TOPLAS, 4(3): 455-495, July 1982.
24. Park, D., *On the semantics of fair parallelism.* In D. Bjørner, editor, **Abstract Software Specifications**, Springer-Verlag LNCS vol. 86 (1979), 504–526.
25. Plotkin, G. D., *An operational semantics for CSP*, In D. Bjørner, editor, **Formal Description of Programming Concepts II**, Proc. IFIP Working Conference, North-Holland (1983), 199-225.
26. Reppy, J., *Concurrent ML: Design, Application and Semantics*, in: **Functional Programming, Concurrency, Simulation and Automated Reasoning**, P. Lauer (ed.), Springer-Verlag LNCS 693, 165–198 (1993).
27. Roscoe, A.W., *A mathematical theory of communicating processes*, D. Phil. thesis, Oxford University (1982).
28. Roscoe, A.W., **The Theory and Practice of Concurrency**, Prentice-Hall (1998).
29. Roscoe, A.W. and Hoare, C.A.R., *The laws of occam programming*, Theoretical Computer Science, 60:177–229 (1988).
30. Roscoe, A.W., *Model checking CSP*, in **A Classical Mind: Essays in Honour of C.A.R. Hoare**, Prentice-Hall (1994).