Communicating Process Architectures – 2002
James Pascoe, Peter Welch, Roger Loader and Vaidy Sunderam (Eds.)
IOS Press, 2002

Acceptances, Behaviours and Infinite Activity in $\mathcal{CSPP}$

Adrian E. LAWRENCE
Department of Computer Science, Loughborough University, Leicestershire, LE11 3TU UK
A.E.Lawrence@lboro.ac.uk

Abstract. The denotational semantics presented here defines an extension of CSP called $\mathcal{CSPP}$. It includes a full description of infinite behaviour in one simple model using only finite traces. This is true for both finite and infinite alphabets. The structure is a complete lattice, and so also a complete partial order, under refinement. Thus recursion is defined by fixed points in the usual way. It is also a complete restriction metric space so this provides an alternative treatment of recursion for contraction mappings.

Keywords: CSP; CSPP; Denotational semantics; formal methods; concurrency; parallel systems; occam; hardware compilation; priority; p-priority.

CSP is a process algebra which describes processes which engage in events. In the *original* version of CSP presented in [1] and reprinted in [2] processes were sequential, but could run in parallel at an outer syntactic level. They could communicate by unbuffered message passing. Hence the name *Communicating Sequential Processes*. The communication was more like π-calculus than modern CSP: channels did not appear explicitly.

The second version of CSP is described in a very accessible way in Hoare's book [3]. In this version, parallelism is ubiquitous, and the name Communicating *Sequential* Processes is not so obviously appropriate. However, the concurrency is *interleaving*, so events occur in sequence and the historical name can be defended on those grounds.

The language has spawned many variants and even competitors, but the mainstream version has not changed substantially in recent years and is described comprehensively in Roscoe's landmark text [4].

CSP also spawned the concrete language **occam** [5], which was designed by May in consultation with Hoare and other CSP researchers. It is an imperative language and so had to encompass notions which did not appear explicitly in the more abstract CSP. **occam** was one of the first practical concurrent imperative languages with a mathematical foundation. That foundation was CSP extended with algebraic semantics to cover aspects of state. An almost complete denotational semantics is given by Goldsmith, Roscoe and Scott in [6] and [7].

occam, originally closely associated with the transputer, inspired and enthused a world wide community. It is so innovative, simple and elegant that it revolutionised the approach of many practitioners. It must be said that those very same qualities, coupled with a longstanding lack of support for conventional tools and environments, since rectified, alienated others wedded to tradition.

When **occam** was introduced its close connections with CSP were seldom mentioned, and **occam** practioners were typically unfamiliar with the algebra. The **occam** community, who were in effect the main 'applied-CSP' practioners, developed many techniques and insights largely independently from main stream CSP.

occam was regarded from the beginning as merely an initial foundation for more powerful developments, particularly in raising the level of abstraction. The language has indeed evolved from proto-occam to **occam**-3, although the level of abstraction has not changed significantly so far. It also has small extensions for hardware compilation: see [8], [9] and [10]. But so far it has proven to be remarkably resistant to major change or extension: it seems that the core language is so robust, simple and transparent that modification is almost always a backward step.

The work on CSP reported below originated in this context:

- **occam** includes a way to give priority to some CSP events: conventional CSP abstracts from such concerns. This problem became a serious concern when the author attempted to prove that his hardware, software and firmware design in the HARP([11],[12]) codesign circuit boards and Handel-AS compiler was correct.
- Hardware compilation involves true concurrency.
- The semantics in [6] and [7] is rather difficult to use informally: a simpler way to understand imperative state in the same way as CSP events is very attractive.
- The characteristic features of **occam** stem from its mathematical foundations. In seeking successors or extensions, it is essential to have a full mathematical theory on which to build.

This led to two incremental extensions of CSP: $\mathcal{CSPP}$ and $\mathcal{HCSP}$. $\mathcal{CSPP}$ extends CSP by including priority: the trailing P stands for priority. $\mathcal{HCSP}$ is a further extension of $\mathcal{CSPP}$ and includes true concurrency and additional constructors for hardware compilation: the leading H stands for hardware. These ideas were first presented informally at the Twente WoTUG–20 technical meeting.

Readers with only passing acquaintance with CSP may wish to be reminded that there are two elementary processes *Skip* and *Stop*. Neither process does anything, but *Skip* does it successfully! That is *Skip* terminates: it will pass control on to a successor, while *Stop* is a deadlock. The process representing livelock div might also be included as an 'elementary' process: that represents a situation when a broken program goes into an infinite internal loop.

A common way of building more complex programs is to use *prefixing:* if a process performs an event a before stopping, that is written as $a \rightarrow \mathit{Stop}$. And the process that performs a before passing control to a successor is $a \rightarrow \mathit{Skip}$. Notice that a is an *event* while both $a \rightarrow \mathit{Skip}$ and *Skip* are *processes*. Processes just engage in events. Almost: as in the case of *Stop*, they might *refuse* to engage in events.

A slightly more general prefixing is exemplified by $n : \{1,2,3\} \rightarrow \mathit{Skip}$, a process that is willing to engage in the event n where n is drawn from $\{1,2,3\}$. Notice that here we are regarding ordinary integers as CSP events. That is fine: CSP does not specify the nature of events. Here we interpret such an event as the *reception* of a number from some sender. Our process is willing to accept one of the three numbers and then terminate. This is an example of *external choice*: the sender decides on whether to send 1,2 or 3, and our receiver process follows that decision. It is 'driven'—the choice is made—externally: hence 'external choice'.

The more general form of external choice is written $\Box$, so

$$n : \{1,2,3\} \rightarrow \mathit{Skip} = (1 \rightarrow \mathit{Skip}) \Box (2 \rightarrow \mathit{Skip}) \Box (3 \rightarrow \mathit{Skip})\,,$$

for example.

Now consider $(3 \rightarrow 1 \rightarrow \mathit{Skip}) \Box (4 \rightarrow 2 \rightarrow \mathit{Skip})$. A most important aspect of CSP is that it supports abstraction—hiding of detail. Thus

$$\big((3 \rightarrow 1 \rightarrow \mathit{Skip}) \Box (4 \rightarrow 2 \rightarrow \mathit{Skip})\big) \setminus \{3,4\}$$

is our same process, but now with the events 3 and 4 hidden. Because 3 and 4 are hidden, they are unconstrained, and can happen freely as far as the inner process is concerned. The justification for this idea is not difficult, but must be omitted from this brief overview. It is now an *internal* matter whether 3 or 4 occurs, so the external view of the process is non deterministic. We may observe a process that is prepared to accept the number 1 and will refuse 2, or the reverse. In fact

$$\big((3 \to 1 \to \mathit{Skip}) \Box (4 \to 2 \to \mathit{Skip})\big) \setminus \{3,4\} = (1 \to \mathit{Skip}) \sqcap (2 \to \mathit{Skip}) \; .$$

Thus $\sqcap$ is the *internal* or *non deterministic* choice operator.

Concurrency has been implicit in all our examples so far: the 'receiver' $n : \{1,2,3\} \to \mathit{Skip}$ needed to interact with a 'sender'. So the receiver and sender are running in parallel. Thus we may have

$$(1 \to \mathit{Skip}) \parallel (n : \{1,2,3\} \to \mathit{Skip}) = (1 \to \mathit{Skip}) \, ,$$

where $\parallel$ is the parallel operator which requires that every event be a joint event of both partners. A more general version is $\underset{E}{\parallel}$: the parallel partners synchronize on the joint events in E, but are otherwise independent. This is enough to express any sort of parallelism including interleaving $|||$ which is just $\underset{\emptyset}{\parallel}$.

Sequential composition is written as $\fatsemi$, so

$$(1 \to \mathit{Skip}) \fatsemi (2 \to \mathit{Stop}) = 1 \to 2 \to \mathit{Stop} \; .$$

The last of the basic CSP operators is recursion. $\mu P \bullet n : \{1,2,3\} \to P$ is the process that will accept an unending sequence of events drawn from $\{1,2,3\}$.

$\mathcal{CSPP}$ extends CSP by the addition of biased and neutral versions of the external choice and parallel operators. Since prefixing is a form of external choice, that also gets extended as does interleaving which is a particular case of the general parallel operator. Thus $(1 \to \mathit{Skip}) \overleftarrow{\Box} (2 \to \mathit{Skip})$ is a process that will choose to accept a 1 when there is a choice. Frequently, there is no choice: a sender is only prepared to send a single integer on any given occasion. Then $(1 \to \mathit{Skip}) \overleftarrow{\Box} (2 \to \mathit{Skip})$ behaves in just the same way as the ordinary $(1 \to \mathit{Skip}) \Box (2 \to \mathit{Skip})$. All this has been described in previous papers presented in this series of conferences: [13],[14], [15] and [16].

Merely adding extra syntax to CSP does not achieve very much in its own right. It is necessary to ensure that a consistent and meaningful structure emerges. Since we are building an algebra, we must be able to identify what algebraic laws are obeyed, at the very least. The original and traditional way to do that for CSP is by way of a denotational semantics. That is a mapping from syntax into a mathematical structure embedded in a known theory typically incorporating some sort of fixed point property.

The standard denotational semantics of CSP is based on Failures and has a long history some of which is recounted in [4]. It took a while for the right concepts to emerge, and various difficulties to be recognised and sidestepped. But there have always been severe problems in handling certain sorts of infinite behaviour connected with unbounded non determinism. These have been largely overcome mainly by outstanding work by Roscoe, see [17]. Yet it has to be said that when the *full* theory is included, the whole semantics is somewhat convoluted. And the treatment of termination is a little contrived. It is believed that the semantics presented in this paper completely overcomes all of those problems as well as having additional merits.

The design of $\mathcal{CSPP}$ was driven by the acceptance denotational semantics below. This too evolved, and various versions were explored and presented in this conference series. A number of difficulties were encountered but have now been resolved. The version in this paper is pleasingly simple and appears to have solved all the outstanding problems with a single theory. Time will tell whether it will become the standard semantics for CSP and its extensions. This semantics defines $\mathcal{HCSP}$ including true concurrency with almost no change: that is the subject of the companion paper [18].

1 Introduction

The theory introduced here is a significant advance over earlier denotational semantics for CSP:

- It captures infinite behaviour in the simplest and only model.
- It is a complete partial order, indeed a complete lattice, under refinement in all cases.
- ✓ is a token rather than a first class event; a new token ✗ is introduced to represent divergence.
- It deals with certain 'awkward' processes like $Skip \,\square\, P$ in a simple and natural way.
- It is expressive enough to capture extensions to CSP, in particular priority.

The main family of denotational semantics for CSP is based on Failures, see [3], [4],[19], [20] and the references therein. The principal idea is to characterise a process by what it can *refuse* to do after it has performed some trace, that is a sequence of events. The idea is elegant and economical, if somewhat anti-intuitive. However there are some difficulties, especially when infinite behaviour is included. In particular, refinement then fails to be a complete partial order.

In contrast to Failures, the Acceptance semantics below is based on the idea of simultaneously offering a process a number of mutually exclusive events, and observing which event or events can be chosen or *accepted*.

A problem which seems to be inherent in a semantics based on Failures is an inability to describe *conditional refusal*. So if a process prefers to engage in an event a when it is available, but otherwise will perform b, then b will sometimes be refused and sometimes accepted. The best that a simple Failures model can do is to model that non deterministically.

$\mathcal{CSPP}$ extends CSP to include priority, so the last observation is very pertinent: acceptance semantics was devised as a means of defining $\mathcal{CSPP}$ precisely. Traditionally CSP abstracts from priority, modelling it with nondeterminism, presumably arising from the underlying Failures semantics. And sometimes it is said that CSP is not appropriate for modelling 'fine detail' like priority.

Yet the correctness of some systems, including those built directly in hardware, depend upon priority. So there is every reason to capture them in a formal and precise way in a single unified language. This is a prime motivation for extending CSP and giving it a rigorous definition.

Nondeterminism is fundamental and is modelled here in a rather direct way by allowing a process to have multiple *behaviours*. That is a type of internal choice: a parallel partner cannot control which is selected.

Nondeterminism also arises when a particular behaviour does not make a unique choice of an event from those offered to it. This is a extension of external choice: such a response indicates a behaviour which is prepared to be flexible or *compliant*: it is prepared to negotiate or be driven by a parallel partner into the refinement of the choice down to a particular event. If there is no such partner, or the response of the partner is also compliant, then the final selection of a particular event is nondeterministic.

The entity 'offering' the choice of events to a process is abstracted as the *environment*. This is generally the set of parallel partners of the process. But even if our process represents a whole system, it can at least be observed, perhaps passively, from the outside: one expects it to engage in at least one event if it is of any use. In this sense, there is always an outermost environment.

Those familiar with the standard semantics of CSP will recognise much of this picture, but the interpretation in Acceptances is enriched. Events remain atomic and require the joint participation of all partners. It is useful to think of some such partners as ‘always ready and compliant’ as in a passive observer.

2 Extended CSP

CSP originated in the context of concurrent software typically implemented on a single sequential processor. Major issues were synchronisation and communication. And CSP unified these in the notion of an atomic event which involved the joint participation of, normally two, processes. The joint participation involved both partners in such events mutually blocking. The event could only proceed when both were ready: there was a ‘handshake’ between partners.

As technology advanced, CSP came to be used to describe situations involving true concurrency. In particular, the theory was used to design and program massively parallel arrays of transputers, although since individual transputers were not usually synchronous across arrays, it might be argued that the concurrency involved was still interleaving.

In consort, CSP was used to design **occam**, and that introduced new concepts. It introduced parallel assignment as in `p,q := q,p`. The semantics was equivalent to true concurrency although it was not expressed in that way. However, the assignments were not regarded as CSP events, so that did not immediately raise the question of true concurrency in CSP.

But **occam** also introduced priority as PRI PAR and PRI ALT, and these were obviously direct extensions of the CSP operations $\|$ or $|||$ and $\Box$. However there was no formal semantics for these extensions. The first attempt to provide such was made by Fidge in [21], but since fundamental issues of recursion and fixed points were not considered it was incomplete and so did not succeed.

The topic of priority is sometimes said to be a ‘real-time’ issue rather than a matter of ‘logical’ design, and therefore legitimately separated from matters of correctness. This is an appealing ‘separation of concerns’ argument at first sight: it seems to have more force when applied to $\overleftarrow{\|}$ and $\|$, the parallel operators, rather than to $\overleftarrow{\Box}$ and $\Box$, the external choice operators. But many programs, especially those involving PRI ALT, depend on priority for their correctness.

It would be absurd to require a second precise language with another rigorous semantics to establish what PRI ALT meant and to be able to prove the correctness of programs utilising it. Far better to extend CSP rigorously which is what $\mathcal{CSPP}$ achieves. And that is done in a way that maintains ‘separation of concerns’. For PRI ALT is a refinement of ALT. That is if we establish the correctness of a program that employs ALT, then it follows that ALT can be replaced by PRI ALT, and the program is still correct.

Although CSP and **occam** were used in the design of hardware including the transputer, direct use in hardware compilation was a later development. The hardware was usually a synchronous circuit and was truly concurrent. **occam** simultaneous assignment was used extensively. And when two processes engaged in a joint event, the ‘handshake’ was typically implemented with a signal in each direction. But there were some circuits where the receiver was ‘always ready’: it was a waste of hardware to provide redundant signals. So the idea of a process that was ‘always ready’ to engage in an event arose. Although this idea seems disturbing to some only familiar with software, it is still a valid interpretation of CSP and

$\mathcal{CSPP}$. We can maintain the fiction that such a ready process could in principle inhibit the event, but it ‘chooses’ not to do so.

Once one has admitted that interpretation of an event, it opens the door to the more radical idea that ‘actions’ like assignments might also be regarded as events. Quantum physicists are very familiar with the idea that a measurement can affect the subsequent observed behaviour of a system. And that measurement involves the reception of information.

While conventional assignment can be described perfectly well by CSP, that normally requires an explicit model of a variable. By extending the idea of an event to include assignments where an observer is ‘always ready’ to see the action, we can capture a large part of the semantics of an **occam**-like language within the framework of an extended CSP. Since sequential execution is included, this provides a radical unifying theory for imperative programming in general. There is a further extension of $\mathcal{CSPP}$ which covers such matters. For now, merely note that events in the theory below may have non traditional interpretations.

Hardware also throws up the idea of events that involve more than two processes. In software, events normally include precisely two processes joined by a channel (which is formally merely a set of events). The most obvious example of an event involving many processes in hardware is a clock edge in a synchronous circuit.

The underlying theory of CSP did not restrict events to two processes. In fact the number of processes involved in an event is not even well defined as evidenced by the identity

$$(a \rightarrow Skip) \parallel (a \rightarrow Skip) = (a \rightarrow Skip) \quad .$$

Almost all descriptions of CSP aimed at software deal with events as channels. Below we concentrate on the underlying events for the reasons indicated above, and because the semantics is more obvious. Channels are used in applications and examples.

Here only the modest extension of CSP to $\mathcal{CSPP}$ is addressed, but this also serves as a foundation for further extensions including $\mathcal{HCSP}$. Only interleaving concurrency is covereded here. $\mathcal{HCSP}$ includes true concurrency. The main extensions to standard CSP here are:

- A miraculous process, $\top$. This is an unimplementable process introduced for technical reasons: it ensures that the structure is a complete lattice under refinement.
- Priority and compliant processes. Thus $\Box$ has refinements $\overset{\leftrightarrow}{\Box}$, $\overset{\leftarrow}{\Box}$ and $\overset{\rightarrow}{\Box}$, for example.
- ‘Fair’ refinements of standard operations like interleaving. Thus $\widehat{|||}$ is a refinement of interleaving that must eventually favour each partner. This is related to priority and infinite behaviour and expresses ideas found in temporal logic.

3 Why denotational semantics?

Much recent work in CSP concentrates on operational semantics. That work is largely concerned with modelling and proving systems correct particularly with the aid of model checking [22]. FDR [23] is a model checker for CSP which is built around such an operational semantics.

Here there is a different emphasis: the main thrust is in building tools and designing languages for codesign, especially for hardware design. In particular transformation laws, in effect algebraic semantics, are required which can be used in compilers. Acceptance semantics is abstract and simple, yet can capture more detail than standard CSP and can be applied to a wide range of situations. It can establish the algebraic identities needed for practical tools. And the insight that it affords in aiding understanding and recognising new realms of application should not be dismissed.

4 Some intuitions underlying Acceptances

4.1 Events and Traces

Classical CSP models concurrency by interleaving: no more than one event can occur at any instant. Since such events may be separated by arbitrarily small times, at least in untimed CSP, this suffices for many purposes. If we wish to model two events that occur together, that is done by supposing that closer examination would reveal that the events actually did happen at different times, and so can be represented by a trace: a particular sequence. In fact, the more precise measurement of the order is not available, so that information is missing, and consequently the order is indeterminate. Thus if we wish to model two events a and b that appear to happen together either because that is the reality, or because we cannot determine the times of occurrence sufficiently precisely, then that is handled as the pair of orders $\langle ab\rangle$ and $\langle ba\rangle$. The corresponding elementary process is $(a \rightarrow \mathit{Skip}) \mid\mid\mid (b \rightarrow \mathit{Skip}) = (a \rightarrow b \rightarrow \mathit{Skip}) \sqcap (b \rightarrow a \rightarrow \mathit{Skip})$: one or other order will happen, we know not which. This identification is established rigorously in section 5.9 on page 28.

On occasions this model is inadequate. Modelling synchronous hardware in full detail including explicit clocks requires a more realistic model. $\mathcal{HCSP}$ [18] is an extension of $\mathcal{CSPP}$ which is further extended to include true concurrency. But $\mathcal{CSPP}$ is a more conservative extension of standard CSP and retains interleaving semantics.

4.2 Priority, simultaneous offers and concurrency

$\mathcal{CSPP}$ is aimed at capturing priority. Consider two events a and b again, and a process that gives priority to a in preference to b. If the process is presented with both a and b *simultaneously*, then the process will select a. At first sight, we seem to be faced with the need for true concurrency once again. Yet priority arises in real software systems implemented on entirely sequential processors, so that cannot be the case. The explanation here is that the state of the system, in particular what events are available, is *sampled* at various points.

Acceptance semantics captures these ideas, but can also be extended as in $\mathcal{HCSP}$ to handle true concurrency. It is intended to be close in spirit to Failures semantics, but easier to understand: it is simpler to identify what a process *accepts* rather than what it refuses. Hence the name.

4.3 Environments, simultaneous offers, compliant responses, and events.

A basic idea is that of an *environment* with which a process interacts. That environment is formed by the concurrent partners of the process if any. But at the outer level, it may be thought of as a passive observer. The only purpose of the environment is to engage in common events. At the outermost level, a passive observer merely 'accepts' – observes – any event that the process chooses to perform. This is an example of a 'compliant' environment.

Thus if a particular process has a repertoire of events a, b and c, then a compliant environment might make the compliant 'offer' $X = \{a, b, c\}$. That means that the environment is prepared to engage in precisely one of the events in X. It is compliant in that it leaves the choice to the process. If the process opted for event a, then we will describe that by saying that the process 'accepted' $\{a\}$ in response to the offer $\{a, b, c\}$.

But the environment is also the mechanism for capturing the interaction of cooperating (and also contending) processes. If a process is placed in parallel with another process only prepared to engage in the shared event a, that would in effect constitute an environment offering $X = \{a\}$. If our process is only prepared to perform another event b, then deadlock follows.

An offer like $\{a, b, c\}$ is a simultaneous *presentation* of a choice among 3 events. This is quite distinct from simultaneous *events* which are only present in $\mathcal{HCSP}$: the compliant offer is to perform exactly one of the events. This is just what we need to capture priority. Notice that explicit priority is sometimes only required when true concurrency is not available: if both events a and b are available and we can execute them together, there is no need to use priority to select just one. But there are uses for explicit priority even when true concurrency is available: there may still be mutually exclusive choices with a preference when both are available.

5 Simple examples

The examples here serve to introduce notation and fix ideas. It is assumed that the reader has had some previous exposure to CSP.

5.1 Stop

Consider *Stop*. Like all processes, it starts with an empty trace $\langle\rangle$: initially it has done nothing: And *Stop* continues to do nothing: if it is offered the set of events X, then it accepts nothing. We write the acceptance semantics as

$$\langle\rangle : X \rightsquigarrow \emptyset \quad \text{or} \quad \mathit{Stop} ::: \langle\rangle : X \rightsquigarrow \emptyset \quad \text{when we need the process name.}$$

Stop is an example of a process which has only one *behaviour*, namely $\langle\rangle : X \rightsquigarrow \emptyset$. That will be explicit in section 13.1 on page32.

5.2 Prefixing: $a \rightarrow \mathit{Stop}$

$a \rightarrow \mathit{Stop}$ also has a single behaviour. If it is offered the event a initially, it accepts it:

$$\langle\rangle : \{a\} \rightsquigarrow \{a\}$$

More generally

$$\langle\rangle : X \rightsquigarrow \{a\} \cap X.$$

Once the only behaviour of $a \rightarrow \mathit{Stop}$ has accepted a, the trace of its past actions is $\langle a\rangle$. It subsequent behaviour is

$$\langle a\rangle : X \rightsquigarrow \emptyset \quad .$$

$X \rightsquigarrow \{a\}$ can be pronounced *X accepts $\{a\}$* or *X may accept a*, but $X \rightsquigarrow \emptyset$ is probably better pronounced as *X may refuse* or *X may accept nothing*.

A *behaviour* has a set of traces, here just $\{\langle\rangle, \langle a\rangle\}$, and associates each trace with the responses to all possible offers X.

5.3 Nondeterminism: $\mathit{Stop} \sqcap (a \rightarrow \mathit{Stop})$

Consider $\mathit{Stop} \sqcap (a \rightarrow \mathit{Stop})$. This is a process that may behave like *Stop* or like $(a \rightarrow \mathit{Stop})$. And the choice is *internal*: the environment cannot influence which. An implementation might consist of either process alone in which case $\mathit{Stop} \sqcap (a \rightarrow \mathit{Stop})$ can be regarded as a *specification*. Or a system might be capable of behaving like either component process, but makes arbitrary choice between the options when it is run.

These two possibilities are mapped very literally here: there is a distinct *behaviour* matching each of the component processes. The process is modelled by this set of two behaviours $\{b_1, b_2\}$.

Even when a is among the events offered initially, it may be refused: $\langle\rangle : X \rightsquigarrow \emptyset$. But we still have $\langle\rangle : X \rightsquigarrow \{a\}$ when $a \in X$ as well. Each corresponds to one of the two *behaviours* of the component processes. We write

$$\begin{array}{lllcl} b_1 & :: \langle\rangle & : & X \rightsquigarrow & \emptyset \\ b_2 & :: \langle\rangle & : & X \rightsquigarrow & X \cap \{a\} \\ b_2 & :: \langle a\rangle & : & X \rightsquigarrow & \emptyset \end{array}$$

for the two behaviours b_1 and b_2. Since these behaviours are uniquely associated with the component processes here, we can abuse notation in such cases to write:

$$\begin{array}{lllcl} \mathit{Stop} & :: \langle\rangle & : & X \rightsquigarrow & \emptyset \\ (a \rightarrow \mathit{Stop}) & :: \langle\rangle & : & X \rightsquigarrow & X \cap \{a\} \\ (a \rightarrow \mathit{Stop}) & :: \langle a\rangle & : & X \rightsquigarrow & \emptyset \end{array}$$

So in general a process is identified with a set of partial functions each of which represents a possible behaviour. A behaviour b takes a possible trace s and yields another function which describes what is accepted when a set of events X is offered. Thus $b :: \langle\rangle : X \rightsquigarrow \emptyset$ means that $b(\langle\rangle)$ is the function $X \mapsto \emptyset$. More precisely

$$b(\langle\rangle) = \{X \mapsto \emptyset \mid X \subseteq \Sigma\}$$

where Σ is the set of all events.

5.4 Termination: ✓ and *Skip*

The processes illustrated above simply cease activity, but useful processes normally terminate. That is how a process passes control to a successor on successful completion. In CSP this is done with a special token written as ✓. An associated process is *Skip* which has a single behaviour which does nothing except terminate:

$$\langle\rangle : X \rightsquigarrow \{\checkmark\}$$

In $\mathcal{CSPP}$ ✓ has a special status: it cannot be offered to an event, and it cannot appear in a trace. In classical CSP, ✓ is treated as an ordinary event for most purposes so it may appear in traces. To do this consistently requires considerable ingenuity and awkwardness in Failures semantics. These problems do not arise in Acceptance semantics.

A characteristic property of ✓ is

$$(a \rightarrow \mathit{Skip}) \fatsemi (b \rightarrow \mathit{Skip}) = (a \rightarrow b \rightarrow \mathit{Skip})$$

This has a behaviour with

$$\langle ab\rangle : X \rightsquigarrow \{\checkmark\}$$

The ✓ which passes control from $a \rightarrow \mathit{Skip}$ to $b \rightarrow \mathit{Skip}$ is a hidden synchronisation between the two processes, and does not appear in the trace: rather it is associated with the instance of the sequential constructor $\fatsemi$ which “glues” the two processes together.

5.5 Recursion: $\mu P \bullet a \rightarrow P$

CSP includes solutions to recursive equations like

$$Q = a \rightarrow Q \quad .$$

which defines an infinite process. This can be written

$$Q = \mu P \bullet a \rightarrow P$$

$\mu P \bullet f(P)$ denotes denotes the unique solution of the equation $P = f(P)$ if one exists. If there is more than one solution, then it selects the *least*, that is the most non deterministic, of the available solutions, if any. Determining under what conditions such solutions exist is one of the primary tasks in setting up a denotational semantics for any CSP variant.

Acceptance semantics based on behaviours shows that $\mathcal{CSPP}$ constitutes a complete metric space. This shows that all functions f which are contracting with respect to that metric have unique solutions.

With the addition of a ‘miraculous’ $\top$ process, $\mathcal{CSPP}$ is also a complete lattice and therefore also a complete partial order. The order relates processes with common behaviours where those that are more deterministic are ‘better than’ those which exhibit more internal choice. This refinement order is defined below. It then follows from standard results about fixed points that all sensible recursions in $\mathcal{CSPP}$ have ‘best’ – most deterministic – solutions.

Be that as it may, note for the moment that $\mu P \bullet a \rightarrow P$ is an infinite process with traces consisting of a sequence of a’s. Its semantics is

$$\langle a^n \rangle : X \rightsquigarrow \{a\} \cap X$$

where $\langle a^n \rangle$ is a trace of n consecutive a’s. It is unbounded: it can perform more than n events for any n.

This is our first example of a *behaviour* with an infinite domain. The domain is the set of all finite traces consisting of sequences of as. This ability to represent infinite behaviour using only finite traces is simple, yet significant. It is the extra layer of structure at the individual *behaviour* level that yields the expressive power to distinguish truly infinite behaviour.

5.6 Hiding, divergence and ✗: $(a \rightarrow b \rightarrow Stop) \setminus \{a\}$

An important feature of CSP is that it includes hiding. An example:

$$(a \rightarrow b \rightarrow Stop) \setminus \{a\} = b \rightarrow Stop\,.$$

So a becomes an ‘internal event’ invisible to the environment. This is the primary abstraction mechanism. It introduces nondeterminism and divergence. So

$$\big(a \rightarrow (\mu P \bullet b \rightarrow P)\big) \setminus \{b\}\,, \tag{1}$$

is a process which performs a, but then goes into an infinite loop which no longer interacts with its environment, and there is no way to exercise control. This livelocked process is said to be *divergent* in analogy with uncontrolled infinite behaviour. Such situations are captured with the aid of another pseudo-event ✗. The notation is intended to indicate undesirable non-terminating behaviour contrasting with ✓. For equation (1) the behaviour is:

$$\begin{array}{rcl} \langle\rangle : & X & \rightsquigarrow & \{a\} \cap X \\ \langle a \rangle : & X & \rightsquigarrow & \{\text{✗}\}\,. \end{array}$$

When priority is present, processes like

$$\big((a \to \mathit{Stop})\overleftarrow{\Box}(b \to \mathit{Stop})\big) \setminus \{a\} = \mathit{Stop}\,,$$

arise. $(a \to \mathit{Stop})\overleftarrow{\Box}(b \to \mathit{Stop})$ is a process which always performs a when it is available, but otherwise performs b. But

$$\big((a \to \mathit{Stop}) \Box (b \to \mathit{Stop})\big) \setminus \{a\} = \mathit{Stop} \sqcap (b \to \mathit{Stop})\,.$$

Here $(a \to \mathit{Stop}) \Box (b \to \mathit{Stop})$ treats the events a and b on an equal basis. Both of the above equations are true in the semantics presented below.

5.7 External choice: $(a \to \mathit{Stop}) \Box (b \to \mathit{Stop})$

A simple example of external choice is, $(a \to \mathit{Stop}) \Box (b \to \mathit{Stop})$. It is a process which is partly controlled by the environment as we can see in

$$\begin{array}{rrll}
b_1 :: & \langle\rangle : & X \rightsquigarrow & \{a,b\} \cap X \\
b_2 :: & \langle\rangle : & X \rightsquigarrow & \{a\} \blacktriangleleft a \in X \blacktriangleright \{b\} \cap X \\
b_3 :: & \langle\rangle : & X \rightsquigarrow & \{b\} \blacktriangleleft b \in X \blacktriangleright \{a\} \cap X \\
 & \langle a\rangle : & X \rightsquigarrow & \emptyset \\
 & \langle b\rangle : & X \rightsquigarrow & \emptyset\,.
\end{array} \tag{2}$$

$E_1 \blacktriangleleft \mathit{boolean} \blacktriangleright E_2$ is a notation borrowed from CSP itself: if the boolean is true the result is the expression E_1, otherwise it is E_2. Our process is nondeterministic because there are in general three possible responses to an initial offer of X. The last two lines in equation (2) represent the responses common to $b1, b2$ and $b3$.

$b2 :: \langle\rangle : \{a,b\} \rightsquigarrow \{a\}$ shows that the process may choose to perform the event a when given a choice between a and b. But suppose that we have a process which is compliant in the sense that it wishes to conform to the selection made by a parallel partner in its shared environment. It expresses that by responding with both a and b: $b1 :: \langle\rangle : \{a,b\} \rightsquigarrow \{a,b\}$.

Thus $\Box$ allows any of these possibilities: it abstracts from those details. Priority is a means of choosing just one class of these behaviours.

5.8 The pseudo events $\{\checkmark, \text{✗}\}$

$\checkmark$ and ✗ model termination and livelock respectively. We work with a global alphabet $\Sigma \cup \{\checkmark, \text{✗}\}$ where Σ is a set which includes all the 'ordinary' events that may arise.

Later we will encounter the odd process $\mathit{Skip}\overleftrightarrow{\Box}\mathsf{div}$. It always accepts $\{\checkmark, \text{✗}\}$. That is $X \rightsquigarrow \{\checkmark, \text{✗}\}$ for every offer X.

$\{\checkmark, \text{✗}\}$ has the form of a compliant response, but since an environment can only offer real events, it cannot 'select' between $\checkmark$ and ✗: there is no way to make $\mathit{Skip}\overleftrightarrow{\Box}\mathsf{div}$ comply with a request to terminate because the environment has no way of making such a request.

How then does $\mathit{Skip}\overleftrightarrow{\Box}\mathsf{div}$ differ from $\mathit{Skip} \sqcap \mathsf{div}$? The acceptance semantics certainly differ. The first has a single compliant behaviour; the second two deterministic behaviours. The nondeterminism in $\mathit{Skip} \sqcap \mathsf{div}$ is between the two possible behaviours. We interpret any unresolved compliance also as a nondeterministic outcome. So $\mathit{Skip}\overleftrightarrow{\Box}\mathsf{div}$ is deterministic in that there is only one behaviour, but nondeterministic in that no environment can choose between $\checkmark$ and ✗ in the response of that behaviour. So no single experiment can distinguish $\mathit{Skip}\overleftrightarrow{\Box}\mathsf{div}$ and $\mathit{Skip} \sqcap \mathsf{div}$.

5.9 *Interleaving:* $(a \rightarrow \mathit{Stop}) \mathbin{|||} (b \rightarrow \mathit{Stop})$

The interleaving $(a \rightarrow \mathit{Stop}) \mathbin{|||} (b \rightarrow \mathit{Stop})$ has 3 *behaviours* characterised by

$$
\begin{array}{rlcl}
b1 :: \langle\rangle : & X & \rightsquigarrow & \{a,b\} \cap X \\
b2 :: \langle\rangle : & X & \rightsquigarrow & \{a\} \blacktriangleleft a \in X \blacktriangleright \{b\} \cap X \\
b3 :: \langle\rangle : & X & \rightsquigarrow & \{b\} \blacktriangleleft b \in X \blacktriangleright \{a\} \cap X \\
\langle a\rangle : & X & \rightsquigarrow & \{b\} \cap X \\
\langle b\rangle : & X & \rightsquigarrow & \{a\} \cap X \\
\langle ab\rangle : & X & \rightsquigarrow & \emptyset \\
\langle ba\rangle : & X & \rightsquigarrow & \emptyset\ ,
\end{array}
$$

where the responses after the first event for a particular trace are common. This is just

$$(a \rightarrow b \rightarrow \mathit{Stop}) \mathbin{\Box} (b \rightarrow a \rightarrow \mathit{Stop})\ ,$$

so the process is prepared to perform a and b in either order. And when offered $\{a,b\}$ it may choose a, b or be noncommittal.

5.10 *Parallel:* $(a \rightarrow \mathit{Stop}) \mathbin{\underset{\{a\}}{\|}} (a \rightarrow b \rightarrow \mathit{Stop})$

$(a \rightarrow \mathit{Stop}) \mathbin{\underset{\{a\}}{\|}} (a \rightarrow b \rightarrow \mathit{Stop})$ is a process that synchronises on the event a. The two component processes can only engage in a simultaneously so there is only one *behaviour* given by:

$$
\begin{array}{rlcl}
\langle\rangle : & X & \rightsquigarrow & \{a\} \cap X \\
\langle a\rangle : & X & \rightsquigarrow & \{b\} \cap X \\
\langle ab\rangle : & X & \rightsquigarrow & \emptyset\ .
\end{array}
$$

b is an independent event, so one partner can engage in it without the participation of its compatriot. So

$$(a \rightarrow \mathit{Stop}) \mathbin{\underset{\{a\}}{\|}} (a \rightarrow b \rightarrow \mathit{Stop}) = (a \rightarrow b \rightarrow \mathit{Stop})\ .$$

6 Alphabets and traces

As above, there is an alphabet Σ of ordinary events: this will be large enough to include all the visible events of any process that we need to describe. To this we add the pseudo events $\checkmark$ and ✗, writing $\Sigma^{\checkmark\text{✗}} = \Sigma \cup \{\checkmark, \text{✗}\}$.

Traces are sequences, empty or finite, of events drawn from Σ. $\langle\rangle$ is the empty sequence. The set of all finite traces drawn from Σ is written as Σ^*. The acceptance semantics here based on *behaviours* needs only finite traces.

7 Behaviours

We specify the meaning of a process by describing its responses after it has performed some trace of events. Traces are members of Σ^*. Given such a trace, we then specify an acceptance function as $\{X \rightsquigarrow U\}$. The offer X is some subset of the alphabet Σ: that is $X \subseteq \Sigma$. And a response U is a subset of $\Sigma^{\checkmark\text{✗}}$. So there is a partial function $b : \Sigma^{*} \nrightarrow \left(\mathbb{P}\,\Sigma \rightarrow \mathbb{P}\,\Sigma^{\checkmark\text{✗}}\right)$ representing each possible behaviour.

Thus we have a description which is a set $\mathcal{B}\,P$ of such behaviours:

$$\mathcal{B}\,P : \mathbb{P}\left(\Sigma^{*} \nrightarrow \left(\mathbb{P}\,\Sigma \rightarrow \mathbb{P}\,\Sigma^{\checkmark\text{✗}}\right)\right) \tag{3}$$

$[\![P]\!]$ is the usual notation for the semantic function describing the behaviour of the process P, but $\mathcal{B}\,P$ is more intuitive and used here. The set of traces of the process is just the union of the traces of the behaviours:

$$traces(P) = \bigcup\{traces(b) \mid b \in \mathcal{B}\,P\}\,,$$

where $traces(b) = \operatorname{dom} b$.

We sometimes identify a process directly with its behaviours where the context warrants. And that leads to a simple matching *normal form*.

8 p-priority

$\mathcal{CSPP}$ extends CSP with extra operators including $\overset{\leftarrow}{\Box}$ and $\overset{\leftarrow}{\|}$.

8.1 Process Priority and external choice.

$P = P_1 \overset{\leftarrow}{\Box} P_2$ extends external choice and provides a semantics for PRI ALT in **occam**. The first event is selected in favour of P_1. So $P_{ab} = (a \rightarrow Stop) \overset{\leftarrow}{\Box} (b \rightarrow Stop)$, with a minimal alphabet $\Sigma = \{a, b\}$ for simplicity, has the behaviour:

$$\begin{array}{rrcl}
\langle\rangle : & \{a,b\} & \leadsto & \{a\} \\
\langle\rangle : & \{a\} & \leadsto & \{a\} \\
\langle\rangle : & \{b\} & \leadsto & \{b\} \\
\langle\rangle : & \emptyset & \leadsto & \emptyset \\
\langle a\rangle : & X & \leadsto & \emptyset \\
\langle b\rangle : & X & \leadsto & \emptyset \quad .
\end{array} \tag{4}$$

$P_1 \overset{\leftrightarrow}{\Box} P_2$ is the symmetrical version. So the behaviour of $P = (a \rightarrow Stop) \overset{\leftrightarrow}{\Box} (b \rightarrow Stop)$ is:

$$\begin{array}{rrcl}
\langle\rangle : & \{a,b\} & \leadsto & \{a,b\} \\
\langle\rangle : & \{a\} & \leadsto & \{a\} \\
\langle\rangle : & \{b\} & \leadsto & \{b\} \\
\langle\rangle : & \emptyset & \leadsto & \emptyset \\
\langle a\rangle : & X & \leadsto & \emptyset \\
\langle b\rangle : & X & \leadsto & \emptyset
\end{array} \tag{5}$$

$P_1 \overset{\leftrightarrow}{\Box} P_2$ embodies the antithesis of p-priority in that it refuses to choose between a and b but rather treats them symmetrically. But it will always respect the p-priority of a parallel partner.

9 Fairness

The presence of priority in $\mathcal{CSPP}$ provides a way to express degrees of fairness. Since infinite behaviour is also captured properly, this includes *eventual fairness*.

If $A = \mu p \bullet a \rightarrow p$ and $B = \mu p \bullet b \rightarrow p$, consider $P_1 = A \mid\mid\mid B$ where there are no other processes involved so that both component processes are always ready. Then an implementation of P_1 can always favour A over B, that is $A \overleftarrow{\mid\mid\mid} B \sqsupseteq P_1$, and no b is ever performed.

Even $P_2 = A \overleftrightarrow{\mid\mid\mid} B$ could also behave in just the same way. Although no behaviour of P_2 favours a over b, but is neutral, the nondeterministic resolution of that neutrality, conceptually resolved by the environment in this case, might happen to be consistently unfair.

Acceptance semantics based on behaviours allows us to define $P_0 = A \widehat{\mid\mid\mid} B$ which is a process which cannot be consistently unfair. It guarantees that *eventually* there is an instance when a has priority, and likewise an instance when b has priority. To be precise, $A \widehat{\mid\mid\mid} B$ consists of those behaviours of $A \mid\mid\mid B$ which have at least one trace matching an acceptance giving priority to A and another giving priority to B.

More precise control is given by $A \overset{\widehat{n}}{\mid\mid\mid} B$ which ensures fairness in the sense above over every sequence of n events.

10 Abstracting from priority

When two processes are combined, one or the other may have priority, or the result may be compliant. So if the first process accepts X_1 and the other X_2, the overall acceptance is determined by X_1 if the first process has priority, by X_2 when the second process has priority and by $X_1 \cup X_2$ in the compliant case. Each case corresponds to a particular variant of the joint process. But we also need a general version which abstracts from those details. We need it to model situations in which we lack full information, and for specification where we wish to leave the implementation choices open. This last is especially important in a programming language where we wish the compiler to generate the most efficient circuit or code.

A more general and fairly extreme example is $e : E \rightarrow \mathit{Stop}$ when E is a large set. This is a process that may perform any event from E that may be offered. But if several events from E are offered simultaneously, we do not wish to constrain how the final choice of a single event is made. In a hardware implementation, the most efficient circuit is quite likely to be one with a definite priority hardwired. A biased implementation is acceptable. But a compliant implementation is just as acceptable, as is any intermediate sort that might be compliant with respect to $\{e_1, e_2\}$ say, but biased in favour of $\{e_3\}$.

11 Some definitions and notation

The discussion so far has been introductory. There is no room in a conference paper for full technical detail. Here we lay out the foundations, in particular the axioms or 'health conditions' and simply give a taste of the full semantics by way of a small number of examples.

Definition 11.1 *When $X \subseteq \Sigma$, $X^{\checkmark}$ is used to denote $X \cup \{\checkmark\}$ and $X^{\checkmark\text{✗}}$ denotes $X \cup \{\checkmark, \text{✗}\}$.*

$\#s$ is the length of s: $\#\langle\rangle = 0$ and $\#(t ^\frown \langle e \rangle) = \#t + 1$. We extend this notation to behaviours: $\#b = \max\{\#s \mid s \in \operatorname{dom} b\}$ which is well defined when the lengths are bounded. Otherwise $\#b = \infty$.

12 Health Conditions (Axioms)

A set of partial functions $\mathcal{B}P : \mathbb{P}\left(\Sigma^* \nrightarrow \left(\mathbb{P}\Sigma \rightarrow \mathbb{P}\Sigma^{\checkmark\times}\right)\right)$ describing a process P, must satisfy the axioms or health conditions below. We usually write $b(s)(X)$ as bsX following the usual conventions for curried functions. This is equivalent to writing $b :: s : X \leadsto bsX$.

Each behaviour starts with a clean slate:

$$\textbf{H1:} \qquad \forall\, b \in \mathcal{B}P \bullet \quad \langle\rangle \in traces(b)$$

$traces(b) = \operatorname{dom} b$ above. The miraculous process has no behaviours: $\mathcal{B}\top = \emptyset$.

The traces of a behaviour are prefix closed, and extend while any event can be accepted:

$$\begin{array}{ll} \textbf{H2:} & \forall\, b \in \mathcal{B}P \bullet \forall\, s \in \Sigma^* \bullet \forall\, x \in \Sigma \bullet \\ & \qquad s ^\frown \langle x\rangle \in traces(b) \;\Leftrightarrow\; s \in traces(b) \wedge (\exists\, X \subseteq \Sigma \bullet x \in bsX) \end{array}$$

Notice that these closure conditions determine all behaviours when the acceptances are specified for a general trace.

Every acceptance of an event is one of those offered. And there is a response to every offer because each $b(s)$ is a total function:

$$\textbf{H3:} \qquad \forall\, b \in \mathcal{B}P \bullet \forall\, s \in traces(b) \bullet \forall\, X \subseteq \Sigma \bullet \qquad bsX \subseteq X^{\checkmark\times}$$

If an offer can be refused, then so can any smaller offer. And if an event can be accepted, then no offer including that event can be refused. However the accepted event may differ from the original, perhaps because the second offer includes an event of higher priority:

$$\begin{array}{ll} & \forall\, b \in \mathcal{B}P \bullet \forall\, s \in traces(b) \bullet \forall\, X, Y \subseteq \Sigma \bullet \\ & \qquad bsX = \emptyset \wedge Y \subseteq X \Rightarrow bsY = \emptyset \\ \textbf{H4:} & \qquad\qquad \wedge \\ & \qquad bsX \cap Y \neq \emptyset \Rightarrow bsY \neq \emptyset \end{array}$$

If an offer can be accepted, then smaller offers including accepted events can also be accepted:

$$\begin{array}{ll} \textbf{H5:} & \forall\, b \in \mathcal{B}P \bullet \forall\, s \in traces(b) \bullet \forall\, X, Y \subseteq \Sigma \bullet \\ & \qquad bsX \cap Y^{\checkmark\times} \neq \emptyset \wedge Y \subseteq X \Rightarrow bsY = bsX \cap Y^{\checkmark\times}\,. \end{array}$$

Combining conditions from **H4** and **H5** shows that all behaviours $b \in \mathcal{B}P$ have acceptances which obey

$$\begin{array}{lllll} \textbf{(C1)} & bsX = \emptyset & \wedge\;\; Y \subseteq X & \Rightarrow & bsY = \emptyset \\ \textbf{(C2)} & bsX \cap Y \neq \emptyset & & \Rightarrow & bsY \neq \emptyset \\ \textbf{(C3)} & bsX \cap Y^{\checkmark\times} \neq \emptyset & \wedge\;\; Y \subseteq X & \Rightarrow & bsY = bsX \cap Y^{\checkmark\times}\,, \end{array}$$

when s is one of their traces and X and $Y \subseteq \Sigma$.

These requirements for any individual behaviour b are collected in the following abbreviation.

Definition 12.1 $\mathsf{behave}(b)$ *is an abbreviation for*

$$\begin{gathered}
b : \Sigma^{*} \nrightarrow \left(\mathbb{P}\,\Sigma \to \mathbb{P}\,\Sigma^{\text{✓✗}}\right) \\
\wedge \\
\langle\rangle \in \operatorname{dom} b \\
\wedge \\
\forall\, s \in \operatorname{dom}(b) \bullet \forall\, X \subseteq \Sigma \bullet\ bsX \subseteq X^{\text{✓✗}} \\
\wedge \\
\forall\, s \in \Sigma^{*} \bullet \forall\, x \in \Sigma \bullet \\
s ^\frown \langle x\rangle \in \operatorname{dom}(b) \ \Leftrightarrow\ s \in \operatorname{dom}(b) \wedge (\exists\, X \subseteq \Sigma \bullet x \in bsX) \\
\wedge \\
\forall\, s \in \operatorname{dom} b \bullet \forall\, X, Y \subseteq \Sigma \bullet \\
\begin{array}{lcccl}
bsX = \emptyset & \wedge\ Y \subseteq X & \Rightarrow & bsY = \emptyset & \wedge \\
bsX \cap Y \neq \emptyset & & \Rightarrow & bsY \neq \emptyset & \wedge \\
bs(X) \cap Y^{\text{✓✗}} \neq \emptyset & \wedge\ Y \subseteq X & \Rightarrow & bsY = bsX \cap Y^{\text{✓✗}} & .
\end{array}
\end{gathered}$$

13 Semantics

There is only room here to give the precise semantics for some of the simpler cases and to highlight refinement and recursion. In particular, the definition of the parallel operators requires a little technical infrastructure which we omit.

13.1 *Stop*

$$\langle\rangle : X \rightsquigarrow \emptyset \tag{6}$$

The only behaviour has a single empty trace and no events are accepted. This means

$$\mathcal{B}\,\mathit{Stop} = \left\{\{\langle\rangle \mapsto \{X \rightsquigarrow \emptyset) \mid X \subseteq \Sigma\}\}\right\}, \tag{7}$$

so the only behaviour b has domain $\{\langle\rangle\} = \mathit{traces}(b)$ and $b\langle\rangle X = \emptyset$ for every $X \subseteq \Sigma$.

13.2 *Skip*

$$\langle\rangle : X \rightsquigarrow \{\checkmark\} \tag{8}$$

Again there is only one behaviour, the only trace is empty, but the process always offers to terminate.

$$\mathcal{B}\,\mathit{Skip} = \left\{\{\langle\rangle \mapsto \lambda\, X \bullet \{\checkmark\}\}\right\} \tag{9}$$

13.3 div

We define a simple div here. It is a process which immediately livelocks:

$$\langle\rangle : X \rightsquigarrow \{\text{✗}\}$$

There is a single behaviour $\{\langle\rangle \mapsto \lambda\, X \bullet \{\text{✗}\}\}$.

13.4 $\perp$

$\perp$ is the most unpredictable of processes: it includes every behaviour.

$$\mathcal{B}\perp = \left\{b \mid \mathsf{behave}(b)\right\} \tag{10}$$

13.5 $\top$

$\top$ is a miracle. And a fraud. It has no behaviours, and so is not implementable. But it refines every process. And ensures that every monotone recursion is well defined.

$$\mathcal{B}\top = \emptyset \tag{11}$$

13.6 Prefix choice

Consider $e : E \to P(e)$ with $E \subseteq \Sigma$. In general there are many possible initial behaviours $b\langle\rangle$:

$$b :: \langle\rangle : X \rightsquigarrow \emptyset \blacktriangleleft X \cap E = \emptyset \blacktriangleright U$$

where $U \subseteq X \cap E$ is not empty. So $b\langle\rangle$ must satisfy $b\langle\rangle X = \emptyset$ when $X \cap E = \emptyset$ and $\emptyset \neq b\langle\rangle X \subseteq X \cap E$ otherwise;

$$\mathcal{B}\,(e : E \to P(e)) =$$
$$\left\{ b \;\middle|\; \begin{array}{c} \mathsf{behave}(b) \\ \wedge \\ \forall X \subseteq \Sigma \bullet b\langle\rangle X \subseteq X \cap E \wedge (b\langle\rangle X = \emptyset \Rightarrow X \cap E = \emptyset) \\ \wedge \\ \forall e \in E \bullet \exists p \in \mathcal{B}P(e) \bullet \forall s \in \Sigma^* \bullet \langle e\rangle ^\frown s \in traces(b) \Rightarrow b(\langle e\rangle ^\frown s) = ps \end{array} \right\} \tag{12}$$

The behaviours represent all possible ways of assigning or refraining from assigning priority among the events of E. They match one of those in $\mathcal{B}\,P(e)$ after the initial event.

13.7 Compliant Prefix choice

The fully compliant refinement of prefix choice, $e : \overleftrightarrow{E} \to P(e)$, is sometimes useful. An initial behaviour accepts anything from E: $\langle\rangle : X \rightsquigarrow X \cap E$. So

$$\mathcal{B}\,(e : \overleftrightarrow{E} \to P(e)) =$$
$$\left\{ b \;\middle|\; \begin{array}{c} \mathsf{behave}(b) \;\wedge\; \forall X \subseteq \Sigma \bullet b\langle\rangle X = X \cap E \\ \wedge \\ \forall e \in E \bullet \exists p \in \mathcal{B}P(e) \bullet \forall s \in \Sigma^* \bullet \langle e\rangle ^\frown s \in traces(b) \Rightarrow b(\langle e\rangle ^\frown s) = ps \end{array} \right\} \tag{13}$$

13.8 Non deterministic choice

$$\mathcal{B}\,(P_1 \sqcap P_2) = \mathcal{B}\,P_1 \cup \mathcal{B}\,P_2 \tag{14}$$

We extend the definition to sets of processes, writing

$$\mathcal{B}\left(\bigsqcap_{i \in I}\{P_i\}\right) = \bigcup_{i \in I}\{\mathcal{B}\,P_i\} \tag{15}$$

13.9 Compliant external choice

The process $P = P_1 \Box P_2$ abstracts away from implementation details. If given the choice, it may select between available initial events of P_1 and P_2. If it always favours events from P_1, that amounts to p-priority, but in general there are a large number of other possibilities. Among them is the case where the process abstains entirely from making a choice, and is completely symmetrical in its treatment of P_1 and P_2. It will always comply with the wishes of whatever mechanism makes the final decision:

$$\mathcal{B}\,(P_1 \overset{\leftrightarrow}{\Box} P_2) =$$

$$\left\{ b \;\middle|\; \begin{array}{c} \mathsf{behave}(b) \\ \wedge \\ \exists (p_1, p_2) \in \mathcal{B}\,P_1 \times \mathcal{B}\,P_2 \bullet \\ \left(\begin{array}{c} \forall X \subseteq \Sigma \bullet b\langle\rangle X = p_1\langle\rangle X \cup p_2\langle\rangle X \\ \wedge \\ \forall e \in \Sigma \bullet \forall s \in \Sigma^* \bullet \langle e\rangle ^\frown s \in traces(b) \Rightarrow \\ \left(\begin{array}{c} \exists X \subseteq \Sigma \bullet e \in p_1\langle\rangle X \wedge b(\langle e\rangle ^\frown s) = p_1(\langle e\rangle ^\frown s) \\ \vee \\ \exists X \subseteq \Sigma \bullet e \in p_2\langle\rangle X \wedge b(\langle e\rangle ^\frown s) = p_2(\langle e\rangle ^\frown s) \end{array} \right) \end{array} \right) \end{array} \right\} \tag{16}$$

13.10 p-prioritised external choice

$$\mathcal{B}\,(P_1 \overset{\leftarrow}{\Box} P_2) =$$

$$\left\{ b \;\middle|\; \begin{array}{c} \mathsf{behave}(b) \\ \wedge \\ \exists (p_1, p_2) \in \mathcal{B}\,P_1 \times \mathcal{B}\,P_2 \bullet \\ \left(\begin{array}{c} \forall X \subseteq \Sigma \bullet b\langle\rangle X = p_2\langle\rangle X \blacktriangleleft p_1\langle\rangle X = \emptyset \blacktriangleright p_1\langle\rangle X \\ \wedge \\ \forall \langle e\rangle \in traces(b) \bullet \forall s \in \Sigma^* \bullet \langle e\rangle ^\frown s \in traces(b) \Rightarrow \\ \left(\begin{array}{c} \exists X \subseteq \Sigma \bullet e \in p_1\langle\rangle X \wedge b(\langle e\rangle ^\frown s) = p_1(\langle e\rangle ^\frown s) \\ \vee \\ \exists X \subseteq \Sigma \bullet p_1\langle\rangle X = \emptyset \;\wedge\; e \in p_2\langle\rangle X \wedge b(\langle e\rangle ^\frown s) = p_2(\langle e\rangle ^\frown s) \end{array} \right) \end{array} \right) \end{array} \right\} \tag{17}$$

This simply says that the process always behaves like P_1 unless P_1 refuses in which case it behaves like P_2. It allows P_1 to perform any event, terminate or diverge if it is capable of doing so. So if P_1 is active in any sense, it is let loose:

$$\langle\rangle : X \rightsquigarrow U \quad \text{if} \quad P_1 ::: \langle\rangle : X \rightsquigarrow U \quad \text{apart from } U = \emptyset$$

P_2 is only allowed to be active when P_1 is not. If P_1 shows any sign of life, even pathological life, it executes.

13.11 External choice

We define $P_1 \Box P_2$ abstractly as $(P_1 \overset{\leftarrow}{\Box} P_2) \sqcap (P_1 \overset{\leftrightarrow}{\Box} P_2) \sqcap (P_1 \overset{\rightarrow}{\Box} P_2)$:

$$\mathcal{B}\,(P_1 \Box P_2) = \mathcal{B}\,(P_1 \overset{\leftarrow}{\Box} P_2) \cup \mathcal{B}\,(P_1 \overset{\leftrightarrow}{\Box} P_2) \cup \mathcal{B}\,(P_1 \overset{\rightarrow}{\Box} P_2) \tag{18}$$

13.12 Sequential Composition

$$\mathcal{B}\,(P_1 \fatsemi P_2) =$$

$$\left\{ b \;\middle|\; \begin{array}{c} \mathsf{behave}(b) \\ \wedge \\ \exists (p_1, p_2) \in \mathcal{B}\,P_1 \times \mathcal{B}\,P_2 \bullet \forall s \in traces(b) \bullet \forall X \subseteq \Sigma \bullet \\ \left(\begin{array}{c} s \in traces(p_1) \wedge \checkmark \notin p_1 s X \wedge bsX = p_1 s X \\ \vee \\ s \in traces(p_1) \wedge \checkmark \in p_1 s X \wedge bsX = ((p_1 s X) \setminus \{\checkmark\}) \cup p_2 \langle\rangle X \\ \vee \\ \exists (s_1, s_2) \in traces(p_1) \times traces(p_2) \bullet \exists Y \subseteq \Sigma \bullet \\ \checkmark \in p_1 s_1 Y \wedge s_2 \neq \langle\rangle \wedge s_2(0) \in p_1 s_1 Y \wedge s = s_1 {}^\frown s_2 \wedge bsX = p_2 s_2 X \end{array} \right) \end{array} \right\} \qquad (19)$$

Notice that we cover processes like $\mu p \bullet (Skip \sqcap a \to p) \fatsemi Q$. And also that it is trivial to check that $Skip \fatsemi P = P \fatsemi Skip = P$.

13.13 Refinement

Refinement is as usual

$$P_1 \sqsupseteq P_2 \;\Leftrightarrow\; P_2 = P_2 \sqcap P_1 \qquad (20)$$

which simply maps onto set inclusion on the behaviours:

$$P_1 \sqsupseteq P_2 \;\Leftrightarrow\; \mathcal{B}\,P_1 \subseteq \mathcal{B}\,P_2 \qquad (21)$$

13.13.1 $\top$ and $\bot$

The most nondeterministic process which has all possible behaviours is evidently below any other process in this order: it is the least element $\bot$ of $\sqsupseteq$.

$\top$ has no behaviours, and is not implementable. Its contribution is to form the final brick in building a complete lattice. The structure then has the weaker property of being a complete partial order to which we can apply standard fixed point theorems to define recursion.

13.13.2 Meets and joins

Obviously meets correspond to unions and joins to intersections of behaviours which always represent processes. Thus meet is just $\sqcap$ which is the motivation for the choice of symbol. For example, the meet of $(a \to Stop)$ and $(b \to Stop)$ is $(a \to Stop) \sqcap (b \to Stop)$.

Joins are written analogously as in $(a \to Stop) \sqcup (b \to Stop) = \top$. All meets and joins exist so the set of processes is a complete lattice under refinement. And so also a Complete Partial Order(CPO).

13.13.3 Unbounded nondeterminism

$\top$ handles cases of unbounded nondeterminism in a clean way. A standard example with an infinite Σ is:

Example 13.1 *Consider the set of processes*

$$P_n = \bigsqcap \{a_i \to Stop \mid i \geq n\}$$

for $n \in \mathbb{N}$ where all the a_i are distinct.

Here any finite set of the P_i has an upper bound, namely P_m where m is the largest index. So we have a directed set. Yet for every $n \in \mathbb{N}$, a_n cannot be performed by members of $\{P_i \mid i > n\}$, so there can be no process behaviour which refines every element. But since $\top$ has no behaviours, it is the join: $\sqcup\{P_n \mid n \in \mathbb{N}\} = \top$. ■[13.1]

Another of the standard examples of an awkward directed set involves infinite behaviour even when Σ is finite.

Example 13.2 *Let $\Sigma = \{a, b\}$ and*

$$
\begin{aligned}
P_0 &= \mu p \bullet (b \to p) && (22)\\
P_1 &= a \to P_0 && (23)\\
P_2 &= b \to a \to P_0 && (24)\\
P_3 &= b \to b \to a \to P_0 && (25)\\
&\ldots && (26)
\end{aligned}
$$

so that P_n performs an a as the n^{th} event in an otherwise unbroken stream of b's. Write

$$D_n = \bigsqcap_{i>n} P_i \quad .$$

Then $D = \{D_n \mid n \in \mathbb{N}\}$ is a directed set.

A finite set of D_n is refined by a process that performs an a any time after the n^{th} b. It might be thought that a possible candidate for a process that refines every member of D is P_0. Yet is is clear that this is not true here for it consists of a single behaviour which is not present in any other P_n. Clearly the only candidate for an upper bound of D is given by intersection of the behaviours of members of D. That is obviously empty, so yields $\top$.

13.14 Recursion

$\mu p \bullet f(p)$ denotes a fixed point of the function f. This is often the least fixed point with respect to the refinement order in standard untimed CSP. Semantics based on Failures often have problems in this area: when unbounded non determinism or infinite traces are present, refinement is no longer a CPO.

The Acceptance semantics based on *behaviours* presented here is the first denotational semantics that completely overcomes all those problems. It naturally includes infinite behaviour yet, as we have seen above, it is not only a CPO, but also a particularly simple sort of complete lattice under refinement.

Standard theorems now ensure that every monotone function f has a least fixed point, and so $\mu p \bullet f(p)$ is well defined. All the ordinary operators of $\mathcal{CSPP}$ are monotone with respect to the refinement order $\sqsupseteq$: this follows from the fact that they are defined in terms of individual behaviours, so they all distribute over $\sqcap$.

As noted earlier, a restriction metric can also be defined, and that is also complete. Contraction mappings then have unique fixed points: the corresponding recursions in CSP are known as constructive. The simplicity of establishing the uniqueness of the fixed points is often useful in proofs.

14 Conclusions

Acceptances employing *behaviours* provides a simple satisfying powerful and intuitive denotational semantics for CSP and $\mathcal{CSPP}$. It incorporates infinite behaviour naturally in contrast to other denotational theories for CSP which require awkward extensions to do the same.

The standard operators distribute over $\sqcap$ which is related to refinement in the usual way. With the additional of a 'top' or 'miraculous' process, refinement yields a complete lattice. Thus all standard recursions are well defined and have fixed points. This intuitive result has been difficult to establish in other denotational semantics, but the present theory shows that the intuition was well founded.

It extends CSP so that there is the *option* of refining to the level of detail required to describe priority, yet in an abstract way which does not require a fully timed theory.

It forms a foundation for wider extensions of CSP needed for codesign, especially hardware compilation, but also for capturing a larger part of the semantics of **occam**-like languages than is traditional. These further developments will be reported elsewhere.

15 Acknowledgements

Bill Roscoe noticed a deficiency in an early version of Acceptance semantics which did not include what are now called compliant processes. His prompting led to their formulation, but he is not to blame for the details.

Support from Jeremy Martin when $\mathcal{CSPP}$ was just an experiment was crucial. The WoTUG community and discussions at CPA conferences have inspired many of the ideas underlying $\mathcal{CSPP}$.

References

[1] C.A.R.Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1978.

[2] Per Brinch Hansen, editor. *The Origin of Concurrent Programming*. Springer-Verlag, 2002.

[3] C.A.R Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

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

[5] Inmos Limited. *occam 2 Reference Manual*, 1988. Document 72 occ 45 01.

[6] M.H. Goldsmith, A.W. Roscoe, and B.G.O. Scott. Denotational semantics for occam 2, part 1. *Transputer Communications*, 1:65–91, 1993.

[7] M.H. Goldsmith, A.W. Roscoe, and B.G.O. Scott. Denotational semantics for occam 2, part 2. *Transputer Communications*, 2:25–67, 1994.

[8] B.M.Cook and R.M.A.Peel. Occam on Field Programmable Gate Arrays - Steps towards the para-PC. In Barry Cook, editor, *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering*, pages 211–228, Amsterdam, April 1999. IOS Press.

[9] R.M.A.Peel and B.M.Cook. Occam on Field Programmable Gate Arrays – Optimizing for Performance. In P.H.Welch and A.W.P.Bakkers, editors, *Communicating Process Architectures, Proceedings of WoTUG 23*, volume 58 of *Concurrent Systems Engineering*, pages 227–238. World occam and Transputer User Group (WoTUG), IOS Press, Netherlands, September 2000.

[10] I Page and W Luk. Compiling occam into fpgas. In Will R Moore and Wayne Luk, editors, *FPGAs*. Abingdon EE&CS Books, 1991.

[11] A.E. Lawrence. *HARP (TRAMple) manual. Volume 1. User Manual for HARP1 and HARP2.* Oxford University, 1992-95.

[12] Adrian Lawrence & Andrew Kay & Wayne Luk & Toshio Nomura & Ian Page. Using reconfigurable hardware to speed up product development and performance. In *JFIT Conference*. Oxford University, 1994.

[13] A.E. Lawrence. Extending CSP. In P. H. Welch & A. P. Bakkers, editor, *Proceedings of WoTUG 21: Architectures, Languages and Patterns*, volume 52 of *Concurrent Systems Engineering*, pages 111–131, Amsterdam, April 1998. WoTUG, IOS Press.

[14] A. E. Lawrence. Hard and soft priority in CSP. In Barry M Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems.*, volume 57 of *Concurrent Systems Engineering*, pages 169–195, Amsterdam, Apr 1999. WoTUG, IOS Press.

[15] A. E. Lawrence. CSPP and event priority. In *Communicating Process Architectures – 2001*, Concurrent Systems Engineering, pages 67–92, Amsterdam, Sept 2001. IOS Press.

[16] A. E. Lawrence. Successes and Failures: Extending CSP. In *Communicating Process Architectures – 2001*, Concurrent Systems Engineering, pages 49–65, Amsterdam, Sept 2001. IOS Press.

[17] Oxford University Computing Laboratory. *Two Papers on CSP*, number PRG-67 in PRG Technical Monographs, July 1988.

[18] A. E. Lawrence. HCSP, imperative state and true concurrency. In *Communicating Process Architectures – 2002*, Concurrent Systems Engineering, pages 39–55, Amsterdam, Sept 2002. IOS Press.

[19] A.W. Roscoe. An alternative order for the failures model. In *Two Papers on CSP* [17].

[20] A.W. Roscoe. Unbounded nondeterminism in CSP. In *Two Papers on CSP* [17].

[21] C.J.Fidge. A formal definition of priority in CSP. *ACM Transactions on Programming Languages and Systems*, 15(4):681–705, September 1993.

[22] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[23] Formal Systems (Europe) Ltd, 3, Alfred Street, Oxford OX1 4EH UK. *FDR2 User Manual*, May 2000.

[24] Jeff Magee & Jeff Kramer. *Concurrency: State Models & Java Programs*. John Wiley, 1999.

[25] Jeremy Malcolm Randolph Martin. *The Design and Construction of Deadlock–Free Concurrent Systems*. PhD thesis, University of Buckingham, 1996.

[26] A.W. Roscoe, editor. *A Classical Mind*. Prentice Hall Series in Computer Science. Prentice Hall, 1994. Essays in Honour of C.A.R. Hoare.

[27] Gavin Lowe. Prioritized and probabilistic models of Timed CSP. Technical Report PRG-TR-24-91, OUCL, 1991.

[28] Gavin Lowe. Prioritized and probabilistic models of timed CSP. *Theoretical Computer Science*, 1994. Special Issue on Mathematical Foundations of Programming Semantics conference.

[29] Carl A. Gunter. *Semantics of Programming Languages*. The MIT Press, 1992.

[30] A.E.Lawrence. HCSP: Extending CSP for Codesign and Shared Memory. In P.H.Welch and A.P.Bakkers, editors, *Proceedings of WoTUG-21: Architectures, Languages and Patterns for Parallel and Distributed Applications*, volume 52 of *Concurrent Systems Engineering*, pages 133–156, Amsterdam, April 1998. IOS Press.

[31] A. E. Lawrence. Infinite traces, Acceptances and CSPP. In *Communicating Process Architectures – 2001*, Concurrent Systems Engineering, pages 93–102, Amsterdam, Sept 2001. IOS Press.

[32] Andrew Butterfield and Jim Woodcock. Semantics of prialt in Handel-C. In *Communicating Process Architectures – 2002*, Concurrent