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

$\mathcal{HCSP}$: Imperative State and True Concurrency

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

Abstract. $\mathcal{HCSP}$ is an extension of $\mathcal{CSPP}$ which captures the semantics of hardware compilation. Because it is a superset of $\mathcal{CSPP}$, it can describe both hardware and software and so is useful for co-design. The extensions beyond $\mathcal{CSPP}$ include: true concurrency; new hardware constructors; and a simple and natural way to represent imperative state. Both $\mathcal{CSPP}$ and $\mathcal{HCSP}$ were invented to cope with problems that arose while the author was trying to prove that the hardware that he had designed correctly implemented *channels* between a processor and an FPGA. Standard CSP did not capture priority, yet the circuits in the FPGA and the occam processes in the transputer both depended on priority for their correctness. The attempt to extend CSP rigorously to handle such problems of co-design has led to develoments that seem to have a much wider significance including a new way of unifying theories for imperative programming. This paper reports on the current state of $\mathcal{HCSP}$ and focuses on handling imperative state and true concurrency. The acceptance denotational semantics is described briefly.

Key words: CSP, CSPP, HCSP, true concurrency, denotational semantics, formal methods, hardware compilation, VLSI design, specification, parallel systems.

$\mathcal{HCSP}$ is an extension of CSP aimed at specifying and describing hardware: it is 'Hardware CSP'. There is a very short introduction to CSP in the companion paper [1]. One of the great merits of standard CSP is that it supports various levels of abstraction. One way is through hiding. And importantly there is an explicit refinement relation. If $P \sqsupseteq Q$, then process Q is more abstract that P: the latter is more specific. So a CSP description can be relatively abstract, omitting detail and simplifying circuits to their essence. Or it can include considerable detail. Unfortunately, there are hard limits on how much detail. One such limit was encountered by the author when trying to prove that the interaction of a circuit generated by a hardware compilation system based on **occam** (Handel-AS) with a software **occam** process correctly implemented an array of CSP channels. An immediate problem was that the circuit depended for its correctness on priority: but CSP could not describe priority. So that was the first issue to be addressed. And it would be useful for software as well: priority in ordinary **occam** also had no formal semantics. A new denotational semantics was invented to define $\mathcal{HCSP}$: acceptances. And the first extension $\mathcal{CSPP}$ defined, as described in the companion paper [1]: this differs from full $\mathcal{HCSP}$ in that it does not include true concurrency.

When there is a limit to the detail which a theory can capture as with CSP and priority above, then one must change to some other theory to prove a design correct. In the case of circuits, one might consider VHDL or Verilog. However both of those hardware languages are complex, confusing and have not been designed around a proper mathematical theory. And they had no formal semantics when this investigation started. Even if proof was feasible there is still the need for a 'translation' layer to move between the high level rigorous description and the detailed circuit language.

How much better if an extension of CSP could cope with fine circuit detail? Hardware is inherently parallel. CSP is the pre-eminent parallel language. **occam** had already been used to describe hardware, although not in a formal way. So an extension of CSP to capture circuit detail was an obvious step. The ability to work at the highest level of abstraction is retained. Yet we can drop to full circuit detail and relate that to the higher level using refinement. This is the idea which has driven the development of $\mathcal{HCSP}$.

occam was designed around CSP. $\mathcal{HCSP}$ is intended to provide the same sort of foundation for hardware extensions of **occam** and for new rigorous hardware description languages. And just as CSP delivers simplicity, rigor and the ability to abstract away from all the inevitable complexities of a concrete language, its extension $\mathcal{HCSP}$ can play the same role for a hardware designer.

1 Introduction

$\mathcal{HCSP}$ is an extension of $\mathcal{CSPP}$ which is in turn an extension of CSP. It aims to extend CSP to capture idioms useful for hardware compilation using an **occam**-like language. An exploratory version was introduced in [2].

The first step was to extend CSP to capture priority: that is achieved in $\mathcal{CSPP}$, described in a companion paper [1].

The second step involved finding a way to capture the semantics of synchronous circuits where the state of many registers is updated simultaneously on a clock edge. Where such registers are independent that is modelled by standard CSP interleaving semantics, and **occam** usage rules ensure the independence.

However, that is unnecessarily restrictive when true concurrency is present: **occam**-usage rules are there to eliminate undesirable non-determinism which arises only in interleaving. Thus:

```
PAR
  c ! x
  d ? x
```

is *invalid* in **occam** because an implementation would be entitled to perform the two communications sequentially in either order. So if x is 3 before execution, and a value 4 is sent on channel d, the final value of x can be either of 3 or 4, and the value sent on channel c is also uncertain.

Yet **occam** already included true concurrency in that it includes simultaneous assignments like '`x,y := y,x`'.

These assignments are not conventionally regarded as CSP events: they do not appear to involve communication or synchronisation with another process. Yet, as noted in [1], we can extend the idea of a CSP event to include assignment. Entertain processes that are 'always ready': this idea arises naturally in hardware compilation. Consider an environment which includes all variable names that can appear in a process together with their values. An assignment may need to read such values, and in general will modify a value: that is a communication. And this environment is 'always ready' and it engages in the communication which is also a synchronisation: the environment receives information that the assignment has occurred.

Regardless of the underlying intuition, it is a valid interpretation of the $\mathcal{CSPP}$ and $\mathcal{HCSP}$ semantics to admit assignments as 'events'. The idea has been used informally by **occam** programmers reasoning about their programs, the real innovation here is to define the idea precisely using *auras* described below. The idea of an assignment as an event has a little in common with the notion of an 'action' in formalisms like the Temporal Logic of Actions [3].

Every **occam** programmer is familiar with

`x := e`	=	`CHAN OF x.type c:`
		`PAR`
		`  c ! e`
		`  c ? x`

The right hand side is a process composed of a pair engaging in a CSP event, although we may worry about how far that event is hidden outside the scope of the channel declaration. Thus the idea that the left hand side also represents (a process engaging in) an event is already implicit in the definition of **occam**. We will see that in $\mathcal{HCSP}$ the transformation above really amounts to attaching different labels to the same events: they may be called (`x:= e`), `c.e`, `c!e`, `c?x` or `c.x.e`, but these all represent a value determined by the expression e being passed into the variable x. In all cases there is a communication with an imperative *aura*[1], an environment – it becomes an explicit parallel process below – maintaining a variable x and its value, if any. After the event, we have $\{x \mapsto v\}$, where v is the result of evaluating e.

Classical CSP does not include explicit imperative state: it is normally taken to be a declarative language. One of the objectives of $\mathcal{HCSP}$ is to capture the semantics of **occam**-like languages easily and *intuitively*, so an imperative interpretation to stand along side the declarative understanding is very useful, and is essential when assignment is present. The use of one or more parallel processes, *auras*, to maintain the imperative state is of far wider significance, and provides a way of unifying theories of imperative languages in general, even purely sequential languages.

The third step is to introduce new operators needed for hardware compilation. Many of these operators involve true concurrency so that synchronous circuits can be specified. This does not preclude the generation of asynchronous systems, perhaps mixed with synchronous components.

Much of the power of the CSP family of languages comes from the high level of abstraction. But we also need to be able to specify fine detail in concrete languages including those designed for hardware compilation. It is sometimes asserted that CSP is not suitable for both tasks. The development of $\mathcal{CSPP}$ and $\mathcal{HCSP}$ shows that to be untrue. We maintain the highest level of abstraction: we can stay in the core CSP language if we so wish. But we also have refinements which can capture fine detail, and all within one rigorous framework.

Thus $\mathcal{HCSP}$ is an extension of CSP which:

- includes $\mathcal{CSPP}$ and so captures priority and is fully descriptive of infinite behaviour;
- extends CSP to handle true concurrency as well as interleaving;
- can support an imperative interpretation; and
- includes constructors useful in hardware compilation.

As with all variants of CSP, $\mathcal{HCSP}$ is intended for specification as well as concrete programming with the full rigour of a mathematical language. It first application is to define the formal semantics of codesign-oriented variants of **occam**.

2 Imperative and Declarative Views

$\mathcal{HCSP}$ has two ways of treating variables: the usual declarative way and an imperative option. Consider

$$(c?x \to Skip) \fatsemi d!x \to Stop \tag{1}$$

[1]Prototype *auras* were called *halos* in [2]: the new name avoids any confusion with non-standard analysis.

In the declarative view, x is not free: the input is just an external choice among the events $\mid c \mid$ constituting the channel c. The only possible interpretation is

$$\mathop{\Box}_{c.x \in |c|} \big(c.x \rightarrow \mathit{Skip} \mathbin{;} d.x \rightarrow \mathit{Stop}\big) = \mathop{\Box}_{c.x \in |c|} \big(c.x \rightarrow d.x \rightarrow \mathit{Stop}\big) \tag{2}$$

which shows x to be bound by the quantifier $\Box$ whose scope is the whole expression. x represents a value in a binding. In the declarative view, it is not even clear that the expression in equation (1) is well formed. $;$ would usually be taken as terminating the scope of the implied $\Box$ constructor in $c?x$. We return to this point below.

In the imperative view of equation (1) x is the constant name of a variable, and the incoming value is stored in there. The input is still just an external choice among events in $\mid c \mid$, but those events all update a particular x. Notice that $c?y \rightarrow d!y \rightarrow \mathit{Stop}$ involves a distinct set of events, those that update y rather than x. In the declarative view x and y are bound by an implied quantifier and so the two processes are the same. But there are two distinct processes when the free $x \neq y$ in the imperative view. There is still an implied quantifier, but the external choice is over a restricted range of the events of the channel: our two imperative processes are each only prepared to engage in a subset of the possible events. The first allows only events of the sort $c?x \in\mid c \mid$, and the second only the distinct events $c?y \in\mid c \mid$.

It is the latter understanding of equation (1) which is natural for an imperative programmer, especially for a hardware engineer who is mapping intuition about registers and memory arrays onto variable names. $\mathcal{HCSP}$ is not intended to be esoteric, but accessible to the ordinary imperative programmer after the notation has been introduced.

With $P(x) = c?x \rightarrow \mathit{Skip}$ and $Q(x) = d!x \rightarrow \mathit{Skip}$ consider $P(x) \mathbin{;} Q(x)$. This represents *quite different* processes in the declarative and imperative views. In a declarative context, $P(x)$ is normally interpreted as an external choice with a bound x. So the x in $P(x)$ has no connection with the x in the body. But the x in $Q(x)$ *is* the same x as that in its body. So the free variable x in the second component is the same as that in $P(x)$ but they have no necessary connection with any input in the body of P. This is far from the natural interpretation. Equation (2) here is excluded by the implication that $P(x)$ is a well defined process.

In the imperative context, the $P(x)$ is prepared to engage in any event of c that updates x. And the natural decomposition is permitted because x is free in both $P(x)$ and $Q(x)$.

Now consider $c!1 \rightarrow \mathit{Skip}$. In the imperative view this is a compliant external choice over all the events corresponding to the transmission of 1 over c. If the events of c are all of the type that update a variable from a set V, $\mid c \mid = \{c.n.v \mid n \in \mathbb{Z} \wedge v \in V\}$, perhaps, then it has the form:

$$\mathop{\overset{\leftrightarrow}{\Box}}_{c.1.v \in |c|} c.1.v \rightarrow \mathit{Skip}$$

Notice that this confirms that distinct events $c.1.x$ and $c.1.y$ occur in the imperative

$$(c!1 \rightarrow \mathit{Skip}) \parallel (c?x \rightarrow \mathit{Skip}) \quad \text{and} \quad (c!1 \rightarrow \mathit{Skip}) \parallel (c?y \rightarrow \mathit{Skip})$$

The distinction between the declarative and imperative views is only relevant when variables appear in expressions. If a is a constant event, $a \rightarrow \mathit{Stop}$ has only one interpretation. The semantics of the constructors is the same in either interpretation.

The imperative view is useful even when assignment is absent. Consider

$$(\,(c_1?x_1 \rightarrow d_1?y_1 \rightarrow \mathit{Skip}) \mathbin{|||} (c_2?x_2 \rightarrow d_2?y_2 \rightarrow \mathit{Skip})\,) \mathbin{;} \mathit{out}!(x_1 + x_2 + y_1 + y_2) \tag{3}$$

In a declarative view, this is either malformed or if *all* the variables are free then either it is does not match the usual intuition or there are implied quantifiers and sorting out their scope is tedious. The sequential constructor $\fatsemi$ is inside that scope. The process then does *not* decompose into two sequential processes.

On the other hand the imperative interpretation matches a natural intuition: the particular values of x_1, x_2, y_1 and y_2 communicated in the first component are passed across the sequential constructor to the second component. This is done by the use of the common free imperative variables and a parallel aura. Furthermore equation (3) represents a very common idiom in **occam** programs: data is collected in parallel, and then some result subsequently computed. Yet this is decidedly awkward to express in the usual declarative version of CSP.

The context of an $\mathcal{HCSP}$ expression usually indicates whether it is necessary to distinguish between the imperative and declarative interpretations: that is whether variables are free or bound. Otherwise the phrases *imperative* $\mathcal{HCSP}$ and *declarative* $\mathcal{HCSP}$ can be used.

3 Events

Pure CSP does *not* require that an event involve exactly two processes.

$$\begin{aligned}(a \to Skip \parallel a \to Skip \parallel a \to Skip) &= (a \to Skip \parallel a \to Skip)\\ &= a \to Skip\end{aligned}$$

shows that it may not be possible to decide how many processes are associated with any particular event.

In clasical CSP semantics where $\checkmark$ is an event, albeit rather special, the process

$$(a \to Skip \parallel b \to Skip) \fatsemi (c \to Stop)$$

involves 3 partners synchronising on $\checkmark$.

Yet events are almost always regarded as involving precisely two processes in concrete situations at least conceptually and explicitly when channels are defined. This is a useful simplification apart from $\mathcal{CSPP}$ and $\mathcal{HCSP}$, not least in **occam**. Even in the case of `SHARED` channels, ordinary **occam** still deals with communication events involving precisely two processes. But when we model hardware in CSP we need to include 'events' which involve many processes. The most obvious case is a clock in a synchronous circuit: the clock edges are naturally regarded as events involving perhaps hundreds of processes. And if we regard assignment as an 'event', then it will involve only a single process in a concrete language. In all these concrete cases the aura represents the imperative state and is disregarded except when considering hiding and its relation to scope.

$\mathcal{HCSP}$ puts that on a proper basis: assignments are events which update state. In that light, `x,y := y,x`, which interchanges the values of x and y represents a pair of *simultaneous events*. It can be nothing else. It cannot be an interleaving, for execution in either order gives quite different results. Moreover, we need this idea of simultaneous events, or rather events 'tied' together, for other reasons when compiling hardware. That is particularly clear for synchronous hardware, especially when there may be several clock domains.

So $\mathcal{HCSP}$ extends the conventional presentation of CSP in these respects:

- events involving one or more processes are standard;
- simultaneous events are introduced: *true concurrency* is present; and
- atomic assignment is treated like other events.

Assignment must be *atomic* as it is in occam and in many synchronous state circuits. Thus the change of state associated with an assignment event is uniquely determined by that event. This does not preclude hiding sets of events: that has the standard semantics. But when modelling scope in a concrete language, a matching local aura is imported and also hidden. If the implementation of an event is extended in time and involves fine grained underlying actions, those actions must not be visible in any concurrently executing event: there must be no interference between overlapping event implementations. This is of course ensured by **occam** usage rules in the interleaving case. An assignment, or indeed any, event must behave as if it had an instantaneous implementation.

4 $\mathcal{HCSP}$ is compositional

$\mathcal{HCSP}$ and $\mathcal{CSPP}$ are *compositional*: the properties of a process or circuit follow from those of its components. There can be no interference from the surroundings or other processes. That means that any interaction must be explicit as shared events. This allows understanding, proof of correctness and simplicity. $\mathcal{HCSP}$ and $\mathcal{CSPP}$ also support separation of concerns partly through compositionality. One should not have to worry about too many aspects of a design at once. $\mathcal{HCSP}$ supports and encourages the style in which one builds simple understandable units, transparently correct, preferably proved correct, and then assembles them into larger units which again can be seen, and hopefully proven, to be correct. Once the properties of the smaller units are established, they must not be undermined by assembling them into a larger system.

One style of hardware compilation makes extensive use of simultaneous assignments as in:

```
SYNC
  SEQ
    x := y
    u := y
  SEQ
    y := x
    v := x
```

=

```
SEQ
  SYNC
    x := y
    y := x
  SYNC
    u := y
    v := x
```

=

```
SEQ
  x,y := y,x
  u,v := y,x
```

Here SYNC is a constructor for true concurrency: it is a synchronous, truly concurrent, version of **occam** PAR. The left hand side may not *appear* to be compositional at first sight. However, remember that assignments here are events shared with an aura recording the value of variables. So the behaviour is determined by the interactions with parallel partners, and the system *is* compositional. Which is more obvious in the second and third ways of writing the process above.

Abstraction is also a fundamental tool in compositional design: the minimum level of detail or specification should be used. This is very economical: the same process might be implemented in an asynchronous self-timed circuit, or in synchronous hardware with one or more clocks. But probably more important is that a human has less to worry about at any given point. The prospect of getting a system correct is much greater if one can dismiss detail irrelevant at a particular stage. Identifying what is relevant and handling the detail properly and rigorously when it becomes necessary, but not before, is central to compositional design with abstraction. These are the characteristics of co-design using $\mathcal{HCSP}$, which are largely missing from other methods not based on CSP.

As a small aside, it is often required that real-time safety-critical systems be deterministic. While that may be a necessary requirement for systems designed with inadequate tools, it is misconceived. What is required is that a system must be proved to meet the timing constraints. Any further specification is an artificial and possibly dangerous additional constraint. It is normally far better and safer to have a nondeterministic system, if only so that it has the flexibility to work around partial system failure, work efficiently, and relieve the system designers of the burden of meeting a more complicated system specification. Of course, languages that deal explicitly with time, like those based on $\mathcal{HCSP}$, make these tasks much easier.

4.1 Examples of compositional design introducing new constructors.

To illustrate compositionality, suppose that we have a process which is to be compiled into a circuit: it could be an adder, for example. The process might be as simple as

```
SEQ
  PAR
    c1 ? x
    c2 ? y
  out ! x + y
```

Think of `c1` and `c2` as two input buses to a circuit that places the result on `out`, the output bus. That can be written in imperative $\mathcal{HCSP}$ as

$$(c1?x \to \mathit{Skip}) \mathbin{\#\!\!\#} (c2?y \to \mathit{Skip}) \ ⨾ \ \mathit{out}!(x+y) \to \mathit{Skip}$$

$\mathcal{HCSP}$ includes an extended form of 'interleaving', $\#\!\!\#$, which allows two disjoint events to occur simultaneously. So $\#\!\!\#$ includes *true concurrency* as well as interleaving. Using interleaving as in standard CSP or $\mathcal{CSPP}$ in $(c1?x \to \mathit{Skip}) \mid\mid\mid (c2?y \to \mathit{Skip}) ⨾ \mathit{out}!(x{+}y) \to \mathit{Skip}$ would only permit one input to occur at one time. But in synchronous circuits, such inputs will often be ready on the same clock edge, and be executed together. When clocks are explicit, we find that we require true concurrency to capture the behaviour of the circuit.

In $\mathcal{HCSP}$ $\mid\mid\mid$ is extended to permit the possibility of two events happening together coincidentally.

Example 4.1

$$(a \to \mathit{Stop}) \mid\mid\mid (b \to \mathit{Stop}) = (a \diamond b \to \mathit{Stop}) \ \Box \ (a \to b \to \mathit{Stop}) \ \Box \ (b \to a \to \mathit{Stop})$$

where $a \diamond b$ represents the events a and b happening together.

Although we still call the symbol $\mid\mid\mid$ "interleaving", as example (4.1) shows, there is now the possibility of coincidental execution. In a context in which true concurrency is possible, it would be unnatural to exclude it. Of course one can still generate a process like

$$(a \to b \to \mathit{Stop}) \ \Box \ (b \to a \to \mathit{Stop})$$

which is $(a \to \mathit{Stop}) \mid\mid\mid (b \to \mathit{Stop})$ in CSP and $\mathcal{CSPP}$. This could also be generated from the $\mathcal{HCSP}$ $(a \to \mathit{Stop}) \mid\mid\mid (b \to \mathit{Stop})$ by removing $a \diamond b$ from the alphabet, or by using a parallel partner imposing such a restriction.

$\#\!\!\#$ is like $\mid\mid\mid$, but captures the frequent case where we desire efficiency:

Example 4.2

$$(a \to \mathit{Skip}) \mathbin{\#\!\!\#} (b \to \mathit{Skip}) = (a \diamond b \to \mathit{Skip}) \overleftarrow{\Box} \big((a \to b \to \mathit{Skip}) \ \Box \ (b \to a \to \mathit{Skip})\big)$$

Example 4.2 matches the earlier process:

```
SEQ
  PAR
    c1 ? x
    c2 ? y
  out ! x + y
```

The programmer will expect a circuit that will perform the inputs on `c1` and `c2` simultaneously whenever possible. Notice how priority arises naturally in capturing the semantics of circuits like this.

We can *require* that the two inputs be simultaneous by writing:

```
SEQ
  SYNC        -- ◇
    c1 ? x
    c2 ? y
  out ! x + y
```

which is:

$$(c1?x \rightarrow Skip) \diamond (c2?y \rightarrow Skip) \fatsemi out!(x+y) \rightarrow Skip$$

in imperative $\mathcal{HCSP}$. And if further it is to be clocked by `Clock`:

```
TIE Clock     -- TIE is ◁|
  SEQ
    SYNC      -- The ◇ constructor
      c1 ? x
      c2 ? y
    out ! x + y
```

which represents the compositional $\mathcal{HCSP}$ process:

$$\left((c1?x \rightarrow Skip) \diamond (c2?y \rightarrow Skip) \fatsemi out!(x+y) \rightarrow Skip\right) \lhd\!| \, Clock$$

Any events of the adder now happen only when the `Clock` process generates events. Presumably these will be standard clock edges determined by `Clock`. But the process above can only handle simultaneous inputs. Contrast:

```
TIE Clock
  SEQ
    PAR
      c1 ? x
      c2 ? y
    out ! x + y
```

or:

$$\Big((c1?x \rightarrow Skip) \interleave (c2?y \rightarrow Skip) \fatsemi out!(x+y) \rightarrow Skip\Big) \lhd\!| \, Clock$$

which will also complete a communication on a single input if it is ready on a clock edge.

A clocked circuit cannot usually inhibit its controlling clock. The circuit may be inactive on some clock edges, while it awaits data from a microprocessor, perhaps. So while the circuit can only perform actions synchronously with the clock, the clock can continue ticking independently.

We have seen that $\Diamond$ binds the events of processes even more closely. It should not be confused with the `sync` event used in the Kroc compiler.

Notice that now we can write '$x, y := y, x$' as:

```
SYNC
  x := y
  y := x
```

or:

$$(x := y \to Skip) \Diamond (y := x \to Skip) = (x := y) \diamond (y := x) \to Skip$$

In passing we have introduced $\diamond$, $\Diamond$ and $\triangleleft\!|$ as well as $\mathbin{+\!\!\!+\!\!\!+}$. $\diamond$ and $\Diamond$ are closely related. $\diamond$ is used to 'glue' events together to form a compound event, and is the primary way that we capture true concurrency. This is placed on a proper basis later. $\Diamond$ is a constructor that 'glues' processes together concurrently. $\triangleleft\!|$ is particularly useful in synchronous hardware and makes sure that the events of one processes can only occur in synchrony with those of another, typically a clock. The symbol for this 'tie' operation has been changed from that used in [2]: the new version suggests the way clock signals are usually drawn on schematics. It is the task of $\mathcal{HCSP}$ to define precisely such entities.

5 Infinite behaviour

When is $|||$ useful? We have seen that $\mathbin{+\!\!\!+\!\!\!+}$ is usually preferred to specify an efficient circuit which eagerly does as much as possible in parallel.

One potential application for $|||$ is in *specifying* clocks. Suppose that a system consists of two synchronous components driven by their own clocks $C_1 = \mu\, C \bullet t_1 \to C$ and $C_2 = \mu\, C \bullet t_2 \to C$. These might be two chips each with their own local crystal oscillators: the clocks are unlikely to be correlated. How do we describe the overall trace, say of just the clock components?

An extraordinarily cumbersome way would be to have an explicit composite clock like:

$$t_1 \to t_1 \to t_1 \to t_2 \to t_1 \to t_1 \to t_1 \to t_2 \to t_1 \to t_1 \diamond t_2 \to \ldots$$

which includes the very unlikely case of clock edges exactly coinciding. Then C_1 and C_2 could be extracted by hiding. This is so ugly and inelegant that it can be dismissed immediately for any normal use.

$C_1 \;|||\; C_2$ is close to what we need, but it includes too much: a refinement is $C_1 \overleftarrow{|||} C_2 = C_1$ which does not allow C_2 to run at all when C_1 is ready.

$C_1 \overleftrightarrow{|||} C_2$ is almost right, but again one or other of the clocks could always be excluded: the non-determinism in selecting a component process at each step could always be resolved in favour of C_1, for example.

Fortunately $\mathcal{CSPP}$ and $\mathcal{HCSP}$ include a form of interleaving which *eventually* admits both partners. This is a virtue of acceptance semantics based on behaviours, and permits the description and prescription of varying degrees of fairness. Here we want $C_1 \;\widehat{|||}\; C_2$. The behaviours of $C_1 \;\widehat{|||}\; C_2$ are those of $C_1 \;|||\; C_2$, except that only behaviours which give priority to both C_1 and to C_2 at some stage in their evolution are admitted. Perhaps C_1 is favoured on $\langle\rangle$, the acceptance is compliant for all traces of length between 1 and 10, and C_2 has priority for any trace of length 11.

More precise control can be given using the construction $P_1 \overset{n}{\widehat{|||}} P_2$ which guarantees that any segment of any trace of length n is "fair": that is P_1 and P_2 are given priority at least once in such a segment. Clearly this only makes sense when $n > 1$: $P_1 \overset{\frown}{\underset{}{\overset{1}{|||}}} P_2 = \top$. So $C_1 \overset{\frown}{\overset{2}{|||}} C_2$ would alternate ticks from each clock if they were both always ready.

These clock combinations are primarily of use in proving the correctness of a circuit: they permit an abstract modelling of the clocks. They would not normally be used in the *compilation* of crystal oscillator clocks.

6 Bags

This is a short digression to discuss *bags* or *multisets*. A bag is a set which may include repeated elements. At face value that statement is nonsense: it is inherent in the idea of a set that it can contain at most one instance of any particular element. A bag is really an integer valued function on a set: the bag containing two copies of an element a is identified with the partial function $\{a \mapsto 2\}$.

We have to face the fraught topic of notation. One standard notation for the bag above is $[\![a, a]\!]$, but this overloads the symbols $[\![$ and $]\!]$. And it carries connotations of lists rather than sets. There is an alternative notation sometimes used in Z, but it is awkward to write neatly freehand: typeset symbols which are easily and quickly approximated by hand are preferred. Here we choose to write $\{\!|a, a|\!\}$ with just the right suggestion of a set.

$\{\!|a, a|\!\}$ is a shorthand for $\{\!|a \mapsto 2|\!\}$ where strictly speaking the bag brackets should be replaced by set brackets in the second case if we wished to be unhelpful. If a maps to 3, and b to 2, that can be written in many ways including $\{\!|a, a, a, b, b|\!\}$, $\{\!|a, b, a, b, a|\!\}$ and $\{\!|a \mapsto 3, b \mapsto 2|\!\}$.

We write $x \Subset \beta$ to indicate that x is a member of the bag β, although we sometimes overload $\in$ and just write $x \in \beta$ where the context warrants. Both mean $x \in \operatorname{dom} \beta$: the codomain of a bag is $\mathbb{N}_1$. Similarly bag intersection and union are written as the similarly decorated symbols $\Cap$ and $\uplus$. We write $\beta \Supset \beta'$, $\beta \supseteq \beta'$, $\beta \Subset \beta'$ and $\beta \subseteq \beta'$ when β properly contains β', contains β' or is a proper sub bag of β' or is any sub bag of β'. We can write the empty bag as $\emptyset$ without ambiguity but we may also write $\{\!|\,|\!\}$.

7 Merged events

If e_1 and e_2 are events, we write $e_1 \diamond e_2$ to represent the pair of events e_1 and e_2 happening together. An example is:

$$(a \to Stop) \Diamond (b \to Stop) = a \diamond b \to Stop$$

In:

$$(a \to Stop) \mathbin{|\!|\!|\!|} (b \to Stop) \tag{4}$$

a and b can happen together as $a \diamond b$. As usual, the environment must offer $a \diamond b$ if the simultaneous pair is to occur. A compliant offer of $\{a, b, a \diamond b\}$, for example, is an offer to perform one of three mutually exclusive alternatives: either of the lone events a and b, or both together as $a \diamond b$. The response to the offer above will be $\{a \diamond b\}$.

The *same* event may happen concurrently in a synchronous circuit where there are multiple copies of a particular server. If there happens to be a demand for this service from several independent processes on a particular clock edge, then the servers may all engage in multiple copies of the same event simultaneously, for example $a \diamond a \diamond a$. Yet these events are not inherently tied together for they can happen independently. Nor do they represent synchronisation between the various servers: only with the clients using the servers. This is the natural extension of interleaving in conventional CSP and is modelled with $\mathbin{|\!|\!|\!|}$. In this version of the semantics, no distinction is drawn between events coinciding 'accidentally' in such ways, and those intentionally tied together: the aim is simplicity with a minimal set of primitives.

In this light, we identify *coincident events* like $a \diamond b$ or $a \diamond a$ with *bags of events* which happen, accidentally or inherently, at the same instant.

So for the process in equation (4) when a and b are distinct, acceptances include

$$
\begin{array}{llrcl}
b1 \mid b2 \mid b3 & :: \langle\rangle : & \{a, b, a \diamond b\} & \rightsquigarrow & \{a \diamond b\} \\
b1 & :: \langle\rangle : & \{a, b\} & \rightsquigarrow & \{a\} \\
b2 & :: \langle\rangle : & \{a, b\} & \rightsquigarrow & \{b\} \\
b3 & :: \langle\rangle : & \{a, b\} & \rightsquigarrow & \{a, b\} \\
b1 \mid b2 \mid b3 & :: \langle\rangle : & \{a\} & \rightsquigarrow & \{a\} \\
b1 \mid b2 \mid b3 & :: \langle\rangle : & \{b\} & \rightsquigarrow & \{b\} \\
b1 \mid b2 \mid b3 & :: \langle\rangle : & \{a \diamond b\} & \rightsquigarrow & \{a \diamond b\}
\end{array} \tag{5}
$$

and so on. The interpretation of the first line:

$$b1 \mid b2 \mid b3 \,::\, \langle\rangle \,:\, \{a, b, a \diamond b\} \rightsquigarrow \{a \diamond b\}$$

is that a compliant environment offers to perform either the event a alone, or the event b alone, or both a and b simultaneously. All of the behaviours $b1, b2$ and $b3$ respond by accepting the offer of $a \diamond b$. Notice that $\Diamond$ can generate environments which offer only joint events, so an offer of $\{a \diamond b\}$ in the last line makes sense: the environment may be prepared to do events jointly which it refuses to allow alone.

This is an easy and minor modification of Acceptance semantics as described in [1]. The acceptances for a behaviour have the type $\mathbb{P}\,\Sigma \rightarrow \mathbb{P}\,\Sigma^{\checkmark,\text{✗}}$, but now Σ consists of bags representing coincident events.

It is not even necessary to modify the concept of traces in $\mathcal{HCSP}$, although they have an underlying structure of sequences of bags of bare events. As usual, all processes have a trace $\langle\rangle$ before they have performed any events. But now a process like

$$a \rightarrow\ (b \rightarrow c \rightarrow \mathit{Stop}) \mathbin{|\!|\!|} \big(b \rightarrow (c \rightarrow \mathit{Stop} \mathbin{|\!|\!|} c \rightarrow \mathit{Stop})\big)$$

has $\langle \{\!|a|\!\}, \{\!|b, b|\!\}, \{\!|c, c, c|\!\} \rangle$ which we normally write as $\langle a, b{\diamond}b, c{\diamond}c{\diamond}c\rangle$ among its traces. We write singleton bags as the corresponding event in the usual notation. So $\langle \{\!|a|\!\} \rangle$ is generally abbreviated as $\langle a \rangle$.

In $\mathcal{HCSP}$ all events are really bags: singleton bags can be identified with the atomic events Σ_S. Every event is a member of $\mathrm{bag}\,\Sigma_S$, the set of all nonempty bags formed from members of Σ_S. If $a, b \in \Sigma_S$ are atomic events, then a is identified with $\{\!|a|\!\}$, b with $\{\!|b|\!\}$ and $a \diamond b$ with $\{\!|a, b|\!\}$. So $\diamond$ itself is really just bag union, $\uplus$.

It may sometimes be useful if Σ, the alphabet of events, is finite. If we required that Σ be closed under bag formation, that would not be possible. So in general, the alphabet Σ is a collection of bags including all those that arise in modelling the systems of interest. Then Σ can often be finite. More formally, $\Sigma \subseteq \mathrm{bag}\,\Sigma_S$, where $\mathrm{bag}\,\Sigma_S = \Sigma_S \nrightarrow \mathbb{N}_1$ is the set of all bags containing atomic events from Σ_S. Since acceptance semantics based on behaviours has no problems with infinite behaviour, $\Sigma = \mathrm{bag}\,\Sigma_S$ is also acceptable.

And as we have seen, $\diamond$ is an alias for the bag union operation $\uplus$. That is $(b_1 \diamond b_2)(e) = b_1'(e) + b_2'(e)$ for $e \in \mathrm{dom}\, b_1 \cup \mathrm{dom}\, b_2$ and where $b_i' = b_i \cup \{e \mapsto 0 \mid e \notin \mathrm{dom}\, b_i\}$.

What about the pseudo-events $\checkmark$ and ✗? Since $(a \rightarrow \mathit{Skip}) \,;\, (b \rightarrow \mathit{Stop}) = a \rightarrow b \rightarrow \mathit{Stop}$ we see that $\checkmark$ cannot be tied to a clock. That also follows because $\checkmark$ can never appear in a trace. If an implementation does take one or more clock cycles to terminate a process in hardware, and that process is tied to a clock, then an explicit 'termination' event can be added to the model if necessary to reflect the extra clock cycles. But most synchronous implementations will not require the extra cycles: in that sense, termination is free – it requires no additional cycles.

Skip plays a special rôle in the termination of parallel processes: it acts as a sort of rendezvous. All partners must jointly offer ✓ in order for the whole process to finish. The natural extension of this is to write:

$$(a \to Skip) \diamondsuit (b \to Skip) = a \diamond b \to Skip$$

So might we expect that $\checkmark \diamondsuit \checkmark = \checkmark$ if $\diamond$ can be applied to such terms. And:

$$(a \to c \to Skip) \diamondsuit (b \to Skip) = a \diamond b \to Stop$$

because the processes cannot agree after the first action. Thus ✓ cannot be merged with any other action. We must exclude $a \diamond \checkmark$ either by removing such pairs from $\mathrm{dom}\,\diamond$ or by setting $a \diamond \checkmark = \bot_\diamond$. The simplest approach is to exclude both ✓ and ✗ from the domain of $\diamond$. A related example is:

$$(a \to Skip) \mathbin{|\!|\!|} (b \to c \to Skip)$$

If environment offers $a \diamond b$ initially, and then c, it seems clear that the natural meaning is that the event c can happen despite the fact that the first process offers no event or has terminated. Our semantics must permit the trace $\langle a \diamond b, c \rangle$ in such cases.

For the elementary processes:

$$Skip \diamondsuit Skip = Skip \quad , \quad Skip \diamondsuit Stop = Stop \quad \text{and} \quad Skip \diamondsuit \mathsf{div} = \mathsf{div}$$

Notice that:

$$(a \to Stop) \diamondsuit (b \to c \to Stop) = a \diamond b \to Stop$$

because it is enough for just one of the processes to stop to prevent any further progress for $\diamondsuit$.

✗ models a process out of control, primarily one in an infinite internal loop. Consider $P \lhd\!| \mathsf{div}$. Since the clock never in fact emits any pulses, this is a process which can make no progress. And indeed, the overall process is itself locked into an internal loop, so clearly $P \lhd\!| \mathsf{div} = \mathsf{div}$. Hence ✗ $\diamond\, t$ makes no sense in this context. $\mathsf{div} \lhd\!| C = \mathsf{div}$ follows for much the same reason, although now it is the tied process which does not respond to external clock pulses. Since the clock might continue, one might argue for a less extreme result: $\mathsf{div} \lhd\!| C = \mathsf{div}$ in effect 'stops' the clock which may well be driving other good circuits. But since one part of the overall circuit is broken, it is reasonable to regard the whole circuit as defective. Another option might be to write $\mathsf{div} \lhd\!| C = \bot$ where $\bot$ is the process which can do anything at all including diverge.

For simplicity, we treat ✗ like ✓ with $\diamond$, and do not include it in $\mathrm{dom}\,\diamond$. So $\diamond$ only produces bags of real events from Σ.

8 Acceptances

An acceptance is a set of alternative actions of a process in a particular environment. Each 'action' consists of a bag of events. For example the compliant process:

$$(a \to Stop) \overset{\leftrightarrow}{\Box} (b \to Stop)$$

accepts $\{a, b\} = \{\{\!|a|\!\}, \{\!|b|\!\}\}$ when both a and b are possible single events: at least when the environment is compliantly offering to perform either a or b but not both simultaneously. More exactly:

$$(a \to Stop) \overset{\leftrightarrow}{\Box} (b \to Stop) ::: \langle\rangle : \{a, b\} \rightsquigarrow \{a, b\}$$

This indicates that the process is also willing to perform either a or b but not both simultaneously. Thereafter:

$$\langle a\rangle : X \rightsquigarrow \emptyset$$
$$\langle b\rangle : X \rightsquigarrow \emptyset$$

This matches standard interleaving $\mathcal{CSPP}$ semantics. They ensure that there is no possibility of simultaneous execution. And this is maintained in $\mathcal{HCSP}$ with true concurrency: a and b can only be performed together when $a \diamond b$ is offered and accepted. So $(a \to Stop) \overset{\leftrightarrow}{\Box} (b \to Stop)$ will not perform $a \diamond b$: it is only able to perform a or b, even if it delegates that choice to a partner:

$$(a \to Stop) \overset{\leftrightarrow}{\Box} (b \to Stop) ::: \langle\rangle : \{a, b, a \diamond b\} \rightsquigarrow \{a, b\}$$

In contrast:

$$(a \to Stop) \mathbin{|||} (b \to Stop)$$

will respond with $\{a \diamond b\}$ when it is available:

$$(a \to Stop) \mathbin{|||} (b \to Stop) ::: \langle\rangle : \{a, b, a \diamond b\} \rightsquigarrow \{a \diamond b\}$$

so it will perform a and b simultaneously if the environment is willing to allow this. So one possible trace is $\langle a \diamond b\rangle$.

For $(a \to Stop) \Diamond (b \to Stop)$ we have:

$$\langle\rangle : X \rightsquigarrow X \cap \{a \diamond b\}$$

with traces $\{\langle\rangle, \langle a \diamond b\rangle\}$. A more interesting example is:

$$(a \to Stop) \Diamond (b \to Skip)$$

This engages in $a \diamond b$, but then the second component attempts to synchronously terminate by executing *Skip*. But the first component stops, so the overall effect is *Stop*. Indeed $(a \to a \to Skip) \Diamond (b \to Skip)$ also deadlocks after the first event for the same reason: one component tries to engage in a but the second wishes to terminate in a common action. Successful termination requires that both partners jointly engage in $\checkmark$.

9 Behaviours and Traces

Traces are sequences of bags drawn from $\operatorname{bag} \Sigma_S$ which is the same as sequences of elements drawn from $\Sigma \subseteq \operatorname{bag} \Sigma_S$. So this is similar to $\mathcal{CSPP}$ traces drawn from an alphabet Σ.

The acceptances of a behaviour is a function recording the response to an offer of a set $X \in \mathbb{P}\,\Sigma$ which is itself a set $Y \in \mathbb{P}\left(\Sigma^{\checkmark,\text{✗}}\right)$, written $X \rightsquigarrow Y$.

An individual *behaviour* b is a partial function from traces to the acceptances:

$$b : \Sigma^* \nrightarrow \mathbb{P}\,\Sigma \to \mathbb{P}\left(\Sigma^{\checkmark,\text{✗}}\right) \quad \text{with } traces(b) = \operatorname{dom} b\,.$$

The behaviour of a process P is a set of behaviours:

$$\mathcal{B}\,P : \mathbb{P}\left(\Sigma^* \nrightarrow \mathbb{P}\,\Sigma \to \mathbb{P}\left(\Sigma^{\checkmark,\text{✗}}\right)\right) \tag{6}$$

with:

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

10 Auras and imperative state

As we have noted, $\mathcal{HCSP}$ can also treat assignments as events. The target of an assignment is an imperative variable, so we introduce a set of $\mathcal{N}ames$ and a set of $\mathcal{V}alues$. An *aura* is a *process* which keeps a record of the value of variables, that is the state of an aura is given by a partial function $M : \mathcal{N}ames \nrightarrow \mathcal{V}alues$.

Consider $(x := 6) \to Skip$. This is *not* an aura, but when run in parallel with one as in:

$$\big((x := 6) \to Skip\big) \parallel A(M)$$

then the state of the aura A is $\{x \mapsto 6\} \oplus M$ after $(x := 6) \to Skip$ has terminated.

Thus an *aura* is a process that is prepared to engage in assignment events, and it keeps track of imperative state. It runs in parallel with the rest of a system and synchronises on all assignment events. Such events are communications with an aura.

$(x := x + y) \to Skip$ is a little more interesting. A natural implementation would involve inputting from an aura to discover the values of x and y, and then outputting the new value of x. Here we abstract all of that into a single atomic event which involves communication in both directions. Although that abstraction is a very good approximation to what happens in many synchronous circuits, that is by no means necessary. The abstraction is still valid even when implementations may require several communications over an extended period.

If we run $(x := x + y) \to Skip$ in parallel with an aura $A(\{x \mapsto 4, y \mapsto 5\})$, then after the assignment, we have $A(\{x \mapsto 9, y \mapsto 5\})$.

If we examine $(x := x + y) \to Skip$ more closely, we realise that $(x := x + y)$ really represents a compliant external choice, that is compliant prefixing, of events. So one offer is $[x := x + y]_6$, that is the event involving collection of the values of x and y, and then sending the new value of 6. That would be *refused* by the aura above which will only accept $[x := x + y]_9$.

Definition 10.1 *$[N := e]_v$ represents an assignment event which involves setting all the names in the set $N \subseteq \mathcal{N}ames$ to the value $v \in \mathcal{V}alues$.*

An aura will only engage in such an event if

1. any variables that appear in e are in the domain of M, the current state of the aura; and
2. the expression e evaluates to v.

Otherwise it refuses, so errors are signalled by a process stopping in the usual way.

The definition (10.1) uses a set N to permit assignments like $\{b_0, b_1, b_2, b_3\} := 3$ which may be useful in hardware when one driver is connected to several loads possibly on a bus. Usually N is a singleton, and then we usually write $x := e$ rather than $\{x\} := e$.

Now we can write

$$\begin{aligned}(x := x + y) \to Skip &= \overset{\leftrightarrow}{\underset{v \in \mathcal{V}alues}{\Box}} [x := x + y]_v \to Skip \\ &= e : \overleftrightarrow{\{[x := x + y]_v \mid v \in \mathcal{V}alues\}} \to Skip\end{aligned}$$

If $M : \mathcal{N}ames \nrightarrow \mathcal{V}alues$ represents an imperative state, then we extend the notation $M(exp)$ to all valid expressions exp. So among valid expressions are the names of the variables in $\operatorname{dom} M$. There is no need to be prescriptive about which expressions are to be admitted: $\mathcal{HCSP}$ is intended to have a wide range of applicability. When used to model **occam**, the expressions would be those of that language. Nor do we specify here whether expressions like $x + x$ and $2x$ are to be identified. A strict reading of the definitions 10.3 and 10.4 below however would exclude $[x := y - y]_0$ as a valid event when $y \notin \operatorname{dom} M$.

If an expression *exp* in $[N := exp]_v$ is not well formed, then the aura will refuse the event, so the overall effect will be *Stop*. This is a neat way to handle error in the spirit of **occam**, and it requires no new machinery.

A similar issue arises with mutually contradictory assignments like $(x := 1) \diamond (x := 2)$. Again, an aura refuses such pathologies which might represent bus contention or worse in a circuit.

Clearly we require that $\mathcal{N}ames$ and $\mathcal{V}alues$ be large enough to include all names and variables that appear in the alphabet of atomic events, Σ_S. The set of all events which reference one of the variable names is $\Sigma_{\mathcal{N}}$:

Definition 10.2 *$\Sigma_{\mathcal{N}} \subseteq \Sigma$ is the set of events which which contain at least one atomic assignment:*

$$\Sigma_{\mathcal{N}} = \{e \in \Sigma \mid \exists N \subseteq \mathcal{N}ames \bullet \exists v \subseteq \mathcal{V}alues \bullet \exists exp \bullet [N := exp]_v \Subset e\}$$

Definition 10.3 *The atomic assignment $[N := exp]_v$ is **valid** with respect to state $M : \mathcal{N}ames \nrightarrow \mathcal{V}alues$ when all the free variables which appear in exp are in the domain of M, and the expression exp evaluates to v.*

Definition 10.4 *An event $e \in \operatorname{bag} \Sigma_S$ is **valid** with respect to the state M precisely when every atomic assignment in e is valid and all the assigned variables are distinct.*

Thus $[p := 3]_3 \diamond [p := 4]_4$ is invalid.

Definition 10.5 *$\mathcal{A}(M) \subseteq \Sigma$ is the set of all valid events which contain at least one atomic assignment.*

With these definitions, it is easy to define an aura:

Definition 10.6 *An **aura** A is a function $A : (\mathcal{N}ames \nrightarrow \mathcal{V}alues) \rightarrow \mathcal{HCSP}$ with*

$$A(M) = \overset{\leftrightarrow}{\underset{e \in \mathcal{A}(M)}{\square}} e \rightarrow A(M')$$

where:

$$M' = \left(\bigcup_{[X := e]_v \Subset e} \{x \mapsto v \mid x \in X\} \right) \oplus M$$

In this section, so far only assignments have been considered. What of the imperative:

$$(c!42 \rightarrow Skip) \parallel (c?x \rightarrow Skip) \qquad ?$$

This too is run in parallel with an aura. Assume for now that the aura accepts the events $c?x$ and $c!42$. These are then events that involve *three* participants: the two processes above and an aura $A(M)$. But, of course, $c?x$ and $c!42$ are just descriptions of the *same* event. That event updates the variable x with 42. In fact it is precisely $[x := 42]_{42}$! Thus we see that we have already captured imperative channel communication. And we can write

$$((c!42 \rightarrow Skip) \parallel (c?x \rightarrow Skip)) \parallel A(M) = ((x := 42) \rightarrow Skip) \parallel A(M) \tag{7}$$

which shows how the occam law identifying channel communication and assignment appears rigorously in $\mathcal{HCSP}$. We can just identify the channel events with assignment events.

Thus auras capture all aspects of imperative state in a very simple way without requiring any new concepts. They are also very flexible allowing multiple auras which might be useful in modelling distributed systems. And often an aura will have an initial empty state thereby STOPping an any attempt to read an uninitialised variable.

Auras are run in parallel with imperative processes synchronising on the set $\Sigma_{\mathcal{N}}$: that is natural and intuitive. But it can be tedious to remember to specify the synchronisation set. An a-aura is an extension of an aura which synchronises vacuously also on the non-imperative events $\Sigma - \Sigma_{\mathcal{N}}$:

Definition 10.7 *An **a-aura** A is a process of the form*

$$A(M) = \left(e : \overleftrightarrow{\mathcal{A}(M)} \to A(M')\right) \overleftrightarrow{\Box} \left(e : \overleftrightarrow{(\Sigma - \Sigma_{\mathcal{N}})} \to A(M)\right)$$

where:

$$M' = \left(\bigcup_{[X:=e]_v \Subset e} \{x \mapsto v \mid x \in X\} \right) \oplus M \quad .$$

and $M : \mathcal{N}ames \twoheadrightarrow \mathcal{V}alues$.

Example 10.1

$$((x := y) \to Skip) \parallel A(\emptyset) = Stop$$

where A is an a-aura.

The initial empty state has no value for y, so the assignment is invalid. ■[10.1]

Example 10.2

$$\left(((y := 1) \to Skip) \,;\, (x := y) \to Skip\right) \parallel A(\emptyset) =$$

$$([y := 1]_1 \to [x := y]_1 \to \left(Skip \parallel A(\{x \mapsto 1, y \mapsto 1\})\right)$$

where A is an a-aura.

■[10.2]

Although $\mathcal{HCSP}$ avoids the full complexity of a complete concrete language, it is intended both as an aid in the design of such languages as well as capturing a substantial part of the semantics of existing **occam**-like languages. An approach to capturing imperative state based on auras seems to hold promise for simplifying the semantics presented in [4] and [5].

11 Health Conditions

In order for a function $\mathcal{B}\,P : \mathbb{P}\left(\Sigma^* \twoheadrightarrow \left(\mathbb{P}\,\Sigma \to \mathbb{P}\,\Sigma^{\checkmark\text{✗}}\right)\right)$ to be admissible as a description of a process, it must conform to some simple constraints. These "health conditions" are the same as for $\mathcal{CSPP}$ applied to an alphabet $\Sigma \subseteq \text{bag}\,\Sigma_S$ of bags of atomic events.

Thus $\mathcal{HCSP}$ inherits the semantics of $\mathcal{CSPP}$ described in [1] in a remarkably simple way. This is a result of maintaining traces of events, but identifying those events with bags. It captures the semantics of true concurrency especially as encountered in synchronous circuits in a very satisfying way, maintaining abstraction from the full details of time.

12 Summary and conclusions

This paper has given a brief survey of the way in which $\mathcal{HCSP}$ has evolved since its introduction in [2]. In particular, it shows how acceptance semantics based on *behaviours* is simply adapted to capture true concurrency by little more than extending events to include bags.

The acceptance base allows both priority and infinite behaviour to be properly described. It has been shown how these arise naturally and inevitably in hardware compilation. A language lacking these features will be inadequate for specification and compilation.

A novel and simple way to extend CSP to model imperative state at just the right level of abstraction has been described. This is needed since both **occam** and most hardware compilation languages are imperative and match a natural intuition about hardware. The treatment has been very substantially simplified since the original version of $\mathcal{HCSP}$, and at the same time made far more flexible and powerful. For the first time it has captured a rigorous model for the **occam** equivalence between communication and distributed assignment.

Several of the constructors of $\mathcal{HCSP}$ have been introduced: full definitions have been omitted for lack of space.

References

[1] A. E. Lawrence. Acceptances, Behaviours and infinite activity in CSPP. In *Communicating Process Architectures – 2002*, Concurrent Systems Engineering, pages 17–38, Amsterdam, Sept 2002. IOS Press.

[2] A.E. Lawrence. HCSP: Extending CSP for codesign and shared memory. In *Proceedings of WoTUG 21: Architectures, Languages and Patterns*, pages 133–156. WoTUG, 1998.

[3] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

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

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

[6] Peter T. Breuer, Carlos Delgado Kloss, Andrés Marín López, and Natividad Martínez Madrid. A refinement calculus for the synthesis of verified hardware descriptions in VHDL. *ACM Transactions on Programming Languages and Systems*, 19(4):586–616, July 1997.

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

[8] A.E. Lawrence. Extending CSP - even further. Communicating Process Architectures–2000, 2000. WoTUG.

[9] Andrew Butterfield and Jim Woodcock. Semantics of prialt in Handel-C. In *Communicating Process Architectures – 2002*, Concurrent Systems Engineering, pages 1–16, Amsterdam, Sept 2002. IOS Press.