Communicating Process Architectures 2004
Ian East, Jeremy Martin, Peter Welch, David Duce, and Mark Green (Eds.)
IOS Press, 2004

Towards a Semantics for Prioritised Alternation

Ian EAST
Dept. for Computing, Oxford Brookes University, Oxford OX33 1HX, England.
ireast@brookes.ac.uk

Abstract. A new prioritised alternation programming construct and CSP operator have previously been suggested by the author to express behaviour that arises with machine-level prioritised vectored interruption. The semantics of each is considered, though that of prioritisation is deferred given the current lack of consensus regarding a suitable domain. Defining axioms for the operator are tentatively proposed, along with possible laws regarding behaviour. Lastly, the issue of controlled termination of component and construct is explored. This is intended as only a first step towards a complete semantics.

1 Introduction

Reactive behaviour has traditionally been regarded as the province of the operating system alone. Systems programmers were required to have additional programming skill, an understanding of concurrency, and knowledge of hardware. It was accepted that they would fall back on assembly language. The vast majority of applications could be designed and programmed with less capability, and using a purely sequential language. For the unavoidable concurrent and reactive behaviour within the system and with the the environment, they would defer to the operating system.

Such an approach is frequently no longer adequate for the engineering of software. First, the need for an operating system to satisfy the needs of every application has led it to become bloated, expensive, and unreliable. Second, the market for software application has moved away from the desktop and into the consumer product, as a result of a dramatic decrease in hardware cost. Embedded applications are themselves typically both concurrent and reactive, yet must be both cheap and reliable. A method is needed by which they can be engineered rapidly, but with high integrity and low cost, by people who can readily be hired.

Honeysuckle [1] is intended to provide the means to implement concurrent and reactive systems, secure against pathology such as deadlock and priority inversion. Static verification of formal design rules guarantees security against many errors, without the additional skill, cost, and delay, usually associated with the use of formal methods.

A *service* is a protocol between two processes that identifies a strict sequence of communications. The idea is derived from the master/servant protocol of Per Brinch-Hansen [2]. It may be drawn as a directed arc connecting two nodes within a *service digraph*. A design rule, denying any circuit in such a graph, has been proven to guarantee deadlock-freedom [3]. A companion paper shows how the protocol may be recast as statically verifiable conditions separably defining service and service network [4]. The addition of mutual exclusion and dependency between services proves necessary for compositionality, ensuring every system is a valid component and *vice versa*.

Like occam before it, Honeysuckle is a derivative of Hoare's Communicating Sequential Processes (CSP) [5]. Unlike occam, it models prioritisation as the ability of one process to interrupt another. An interrupting process is often cyclic. An interrupted process resumes

only when a cycle is complete or when its interruptor terminates. A new CSP operator has been proposed which captures such behaviour, along with a programming construct (when) that provides mechanisms for components to asynchronously communicate and precipitate the termination of (disable) each other [6].

The purpose of this paper is to take a first step towards establishing the semantics of both operator and construct by discussing desirable behaviour and resolving certain issues that arise. The aforementioned companion paper also extends service architecture to incorporate prioritised service provision and shows how *a priori* deadlock-freedom may be retained, along with immunity to priority conflict and inversion.

2 Prioritised Alternation

2.1 Indirect Expression

Because reactive behaviour is unavoidable in practice, processor design typically includes a mechanism by which normal control flow can be interrupted. Since there is usually a number of distinct events that cause interruption, the mechanism often allows for the provision of a *vector* to direct control to the appropriate subroutine. *Interrupt service routines* may or may not be re-entrant and sometimes may pre-empt one another according to some recorded prioritisation. It is thus possible to *alternate* behaviour according to prioritised events (internal or external) by programming such hardware directly. This is commonly done using assembly language, but becomes tiresome and error-prone when the system is other than trivial.

When using a high-level programming language (one permitting a measure of abstraction), it is common to portray a set of alternating processes as concurrent. Some pre-emptive scheduler, outside application program control, is then relied upon to deliver alternation. This can be expressed directly, for example, using occam:

```
PRI PAR i=0 for n
  WHILE running
    SEQ
      input[i] ? request
      ... respond to request
```

Since, by definition, *no two responses may execute concurrently*, this is hardly ideal. Concurrency implies equivalence among all interleavings, but in an alternation each interleaving has meaning and therefore should be distinguished. There is also the issue of resumption. The solution above allows an interrupted process to resume when its interruptor is blocked. Prioritised vectored interruption (PVI) provides no mechanism for this but imposes the minimum overhead on latency. Disallowing resumption until completion is arguably simpler, both in abstraction and in implementation, though less efficient, in the sense that the alternation as a whole is blocked where otherwise it need not be. It might also be argued that a programming language should provide abstraction of *all* patterns of control flow offered by hardware, including PVI.

Pre-emptive scheduling offers a worsening solution as interaction between responses increases. Ultimately, some central process will be required to maintain state, communicating with every other process after each response. Scheduling will incur (possibly unacceptable) overheads on both performance and latency. Abstraction becomes complicated and obscure.

Although it is now common practice to simply rely on excess performance available, sometimes even discarding pre-emption, it surely remains sensible to pursue efficiency and the lowest possible latency. It is surely also sensible to provide direct abstraction, and thus transparent and efficient expression of the behaviour of any event-driven system.

2.2 Interrupts and Alternation in CSP

In his seminal book, Hoare accounts for both interruption and alternation [5, #5.4], but not priority. He denotes a process P_1, interruptible by P_2, by

$$P_1 \hat{} P_2 \tag{1}$$

The process thus formed starts and continues behaving as specified by P_1 until some event with which P_2 can start occurs. It then behaves as P_2. P_1 is never resumed.

He adds two qualifications. "To avoid problems", termination must not lie in the alphabet of P_1, αP_1. This means that P_1 is not in fact a sequential process at all, since it cannot be combined with another via the sequence operator ';'. (One obvious problem would be the interpretation following termination of P_1.) This permits the description of processes which can cease only via interruption. In fact, it's worse than that. If we compound the (associative, non-commutative) operator

$$(P_1 \hat{} P_2) \hat{} P_3 = P_1 \hat{} (P_2 \hat{} P_3) \tag{2}$$

we see that, in a string of interruptible processes, none may terminate save the last. In practice, we do not wish to be restricted in this way.

In order to preserve determinism and simplify reasoning about operators, Hoare further requires any interrupting event to lie outside the alphabet of the interrupted process:

$$P_2^0 \cap \alpha P_1 = \emptyset \tag{3}$$

(P^0 refers to the set of events with which P is willing to start. The function $initials(P)$ is used in later CSP texts.) In contrast, this is entirely acceptable. Only the environment can trigger interruption. The interruptible process is blind to the interrupting event.

Hoare goes on to provide abstraction for the class of events which prevent a process continuing (cause 'catastrophe') and for processes which subsequently restart. *Checkpoints* provide for the preservation of a state, upon an event denoted by '©', to which a process may return upon catastrophe, instead of restarting. A third interrupting event is defined, denoted by '$\otimes$', which causes two processes to alternate. (A circumflex is introduced here to distinguish event and operator.)

$$P_1 \hat{\otimes} P_2 \tag{4}$$

Hoare's second stipulation means that $\otimes$ lies in the alphabet of neither P_1 nor P_2, but in that of the process produced by the operator. $\otimes$ also secures a checkpoint. The state of the interrupted process is preserved.

2.3 Prioritised Alternation in CSP

A simple, but useful, interpretation of the term 'prioritisation' has been previously proposed by the author, together with a new *prioritised alternation* CSP operator [6]. A brief summary follows.

Consider a process composed via interruption, as described by Hoare [5, p. 180]. (The notation used by Roscoe is now preferred [7, p. 235].)

$$\Pi = ((P_1 \triangle P_2) \triangle P_3) \ldots \triangle P_n \tag{5}$$

It would only be natural to interpret the order in which processes appear, indexed by *i*, as a form of prioritization. Other interpretations of the term remain possible but surely this one is useful for describing the behaviour (desired or actual) of reactive systems.

Following interruption, no process resumes, and only the last of the given list may terminate. In practice, we usually wish no response to any interruption to disappear after completion. The solution is to allow any such process to be *cyclic* about interruption but also require it to be non–re-entrant. Thus P_2 starts with interruption of P_1. Instead of a single switching event, the new operator adds a dedicated event for each clause, marking the *completion* of a response. Upon completion, P_1 resumes, while P_2 awaits further interruption. Such behaviour might be denoted by a new operator '$\hookleftarrow$' with which to compose processes.

$$\Pi = ((P_1 \hookleftarrow P_2) \hookleftarrow P_3) \ldots P_n \tag{6}$$

A serial operator, concisely denoting a list of alternating processes, could also be defined, subject to an enumerating index.

A number of semantic issues surround prioritised alternation. For example, when process P_2 interrupts P_1, one must consider the possibility that P_1 is blocked awaiting synchronization with the environment. P_1 may also comprise multiple concurrent processes, each of which may be blocked. Upon interruption, all offers of communication must be withdrawn, pending completion of the response, whereupon they are re-established.

2.4 The Honeysuckle `when` *Construct*

The Honeysuckle programming language [1] introduces a dedicated prioritized alternation construct – `when`. For example:

```
when
  transfer draft to publisher
    ... celebrate
  acquire draft from editor
    ... check
  idle
    sleep
```

Each guard initiates a service. At least two clauses must be given and are listed in order of priority. The lowest priority clause may employ the symbol `idle` to indicate that it is unguarded. No clause is re-entrant. Part of each response may be to disable further interruption by either the same, or any other, event. Note that an alternation does not terminate until *all* interruption is disabled.

The order in which clauses may appear is constrained by an explicit process interface in Honeysuckle. A guard may be compounded by selection, allowing interruption by any member of a designated set of services, known as a *bunch* (see next section). This is supported by admitting a selection construct in place of a single guard.

Clauses may share memory, which should immediately raise concern regarding the possibility of interference. Each visible object (variable) may be assigned value within at most one clause (component), which is said to *own* it. One form of interference, commonly cited in textbooks on concurrency, is where interruption of (read or write) access leads to an outcome (final state) different to that given no interruption. Honeysuckle might protect against this by disabling interruption during every access. Assignment would be atomic, as is any procedure encapsulated within an object. (Shared objects may then be modelled using CSP. See Section 4.) The price paid for such security is in the form of extended latency when large objects are shared. Achieving security and adequate performance is then the responsibility of the designer, who must understand the issue, as indeed existing practitioners must, when using PVI explicitly.

Interference can only be defined with regard to the intended function of each component, specified, for example, by pre- and post-assertions. Interference-freedom between concurrent processes requires that every interleaving thereof yields precisely the same outcome. It remains a serious general issue in programming concurrency. (Because processes can model variables, distributing memory does *not* automatically confer a solution [8].)

Components of an alternation are *not* concurrent. Indeed, the interleaving that actually occurs *has meaning* and may thus legitimately affect outcome. It remains to prove that each interleaving has only the desired outcome. This is not a simple issue, nor one which currently knows a solution. Increasing atomicity according to granularity (disabling interruption longer when accessing larger objects) does not confer a complete solution for an alternation [8, #1.6]. It is not difficult to contrive an example where an interruption between two related accesses interferes with the outcome.

All that is claimed here is that the situation is no worse than it would be using pre-emptive scheduling (*e.g.* `PRI PAR` in occam) to address the same problem.

3 Regarding the Semantics of Prioritised Alternation

Viewed from the outside, an alternation consists of an enumerated list of server *bunches*. (A 'server' is the providing end of a service, *i.e.* a connection, not a process. A "server bunch" refers to a set of servers providing mutually exclusive services.) For the sake of simplicity, we shall consider only cases where there is just a single server per bunch.

Termination will be addressed in the next section and only briefly discussed here.

3.1 Alphabet

Rather like a parallel composition, the alphabet of an alternation is the union of that of its components:

$$\alpha(P_1 \hookleftarrow P_2) = \alpha P_1 \cup \alpha P_2 \tag{7}$$

On the other hand, component alphabets are disjoint. In an alternation, components do *not* communicate directly. While interleaving common events might be useful, we choose here the simpler option.

$$\alpha P_1 \ \cap \ \alpha P_2 = \emptyset \tag{8}$$

Each component P_i is characterised by its guards g_i^j, and a *completion* event h_i which marks the resumption of any interrupted process. The initial event set of the alternation is just the union of the initials of both components:

$$initials(P_i \hookleftarrow P_j) \ = \ initials(P_i) \cup initials(P_j) \tag{9}$$

Completion cannot force actual resumption, compelling the interrupted process to engage in its next event. There is no sense of *urgency* in our definition. Completion merely denotes the granting of *permission* to proceed. If further interruption occurs before it resumes then, well, it had the chance. As Hoare noted [5, p. 80], any requirement to take advantage of opportunity, and not be "infinitely overtaken", must be met in implementation, which in practice should not be difficult.

On the other hand, in practice, we *do* require that the operator guarantees acceptance of any offer of a guard by the environment, and according to the defined prioritization. Without introducing timing, we cannot however stipulate the delay. That must also be a matter for implementation.

It may be that the lowest priority process is required to run continuously without awaiting any particular event. In other words, it may be unguarded. This is identified with a *null* guard. occam employed *skip* as a null guard in an `alt` (alternative) construction. However, *skip* is a *process*, not an event – a distinction which proves necessary in a consistent algebra. Rather than introduce a null event, we shall simply allow a component to be unguarded. This is the interpretation place upon the notation *idle*, used in Honeysuckle. There seems no reason to restrict this possibility to the component with lower priority. One law is suggested as a direct consequence, at least in the traces domain. If the interrupting process is unguarded and non-terminating, an alternation is indistinguishable from that process alone:

$$initials(P_2) = \{\tau\} \ \wedge \ \checkmark \notin \alpha P_2 \ \Rightarrow \ (P_1 \hookleftarrow P_2) = P_2 \tag{10}$$

P_2 may only engage in internal events and not communicate with the environment. Note that much depends upon whether or not a component terminates. It is often easier to understand alternation with cyclic, non-terminating, components.

3.2 Traces

Once again, an alternation may be compared with parallel composition. Every trace of each component is also a trace of the alternation. To these must be added those formed as a result of interruption:

$$\begin{aligned} traces(P_1 \hookleftarrow P_2) = {} & traces(P_1) \ \cup \ traces(P_2) \cup \\ & \{s = p \frown u \frown q \ \mid \ u \in traces(P_2)),\ h \in |\, u \,|,\ p \frown q \in traces(P_1)\} \ \cup \\ & \{s = p \frown u \frown q \frown v \frown r \mid \ u, v \in traces(P_2),\ h \ \in |\, u \,|,\ h \ \in |\, v \,|, \\ & \qquad\qquad p \frown q \frown r \in traces(P_1)\} \ \cup \\ & \ldots \end{aligned} \tag{11}$$

Note that we take care to ensure that any interruption is followed by a completed response. After g_2, no further progress by P_1 is allowed until completion h_2. There can be no interleaving.

3.3 An Axiom

It is useful here to define a Boolean operator to infer event precedence. We shall denote "x precedes y in trace s" by $x \overset{s}{\rightsquigarrow} y$[1], so that:

$$x \overset{s}{\rightsquigarrow} y \iff 1 \geqslant \ |\, s \downarrow x - s \downarrow y \,| \ \geqslant 0 \tag{12}$$

assuming both x and y occur just once within any cycle – a condition fulfilled by both guard and completion of any clause. Note that the condition remains appropriate even if the clause concerned is not cyclic, and simply terminates.

A prioritised alternation may be understood as something which guarantees completion of a higher priority service before one of lower priority. To be more precise:

Condition 1. *If a higher priority event precedes the completion of a response to one of lower priority then the completion of its own response does also:*

$$\forall\, s \in traces(P_1 \hookleftarrow P_2).\ g_2 \overset{s}{\rightsquigarrow} h_1 \implies h_2 \overset{s}{\rightsquigarrow} h_1 \tag{13}$$

[1]Not to be confused with use of the same symbol by Schneider to denote an *evolution* — a state transition over time [9, p. 270].

3.4 Prioritisation

Equation 13 is still not quite enough to guarantee the desired behaviour. Suppose the environment offers interruption simultaneously with the next communication of the current process. Having deprived it of the liberty to interleave high and low priority responses (Eq. 11), we must somehow compel interruption:

Condition 2. *If a higher priority guard is ever offered then it will be immediately accepted.*

Unfortunately, neither traces nor failures provide an adequate domain in which to express this precisely. Proof of some of the laws suggested in the next section would also require an adequate domain. Both are left for future publication. However, a brief review follows of the issue and existing literature on the issue.

Any "denotational semantics" rests on establishing a meaning for operators such as equivalence ('='). Deciding equivalence, for example, reduces to establishing whether or not two processes share a common value for some function (or set of functions) of their description. One possible attribute of a process P is the set of traces, $traces(P)$. If another process Q shares a common set of traces with P we can say that $P = Q$ "in the trace domain".

Two processes might exhibit identical trace sets but may behave quite differently under the same circumstances. It would be valuable to know what a process will do when presented with a set of offers of communication by its environment. An *acceptance set* circumscribes those events in which process would agree to engage, following a particular trace. Its complement, the *refusal set*, may be combined with the trace to form a *failure*. A set of failures may then be attributed to the process concerned (which subsumes a description of its traces). Equivalence in the failure domain tells a great deal more than one in the domain of traces, and allows many more deductions.

At this point, it is worth considering how a process is defined. It is usual to establish a list of conditions. (For example, see [5, #3.9].) When defining a language by which one can express process composition, it is essential to include a demonstration that all constituent operators are *well–defined*. By this we mean that each process produced by each operator itself obeys the same conditions. Furthermore, in order to employ recursion, one must show that each operator is *continuous* over some complete partial order (CPO). One process may then be said to *refine* another.

Failures still tell us nothing of process *preference*. If the environment offers two communications, it is possible to assert only that a process is 'willing' to engage in just one or either. In the latter case, the semantics of general choice reduces to that of a non-deterministic *internal* choice. It is not possible to describe a process that would repeatably *prefer* one action over another. Neither is it then possible to decide, say, equivalence in this sense. Some process attribute is needed that would always expose such preference. Only then could the precise meaning of any *biased*, or *asymmetric*, operator on processes be defined.

Over a decade ago, Colin Fidge took an approach similar to the use of refusal sets to discriminate between responses to the environment a process might make following each trace. He established an attribute $preferences(P)$ that described a relation between each pair of events in the process alphabet [10]. Technically, each attribute is a function of the process description. In this case, it returns a set of ordered pairs. For example, $a \rightarrow b \in preferences(P)$ implies that, should both a and b be offered by the environment, the process will accept b.

Fidge goes as far as defining a set of operators in terms of traces and preferences. However, as far as the author is aware, he did not show they were either well-defined or continuous. Hence, their semantics remain undefined.

At around the same time as Fidge, Gavin Lowe also provided a semantics for prioritisation. However, its relation to *timed* CSP makes it more complicated and less relevant to

the subject discussed here. In Lowe's model, a process offers a set of bags of events at each time-step. Like Fidge, a relation (a set of ordered pairs) is used to describe preference (bias). He shows that the language thus formed is entirely deterministic after the removal of non-deterministic choice and then proceeds to develop a probabalistic model on top of the prioritised one.

More recently, Adrian Lawrence has developed an alternative approach whereby bias may be expressed via the 'response' of a process Y to an offer X made by the environment following a trace s [11]. A domain of *triples* $\{(s, X, Y)\}$ is thus established [12]. Each response defines what a process is willing to do in a particular circumstance. It may be simply to terminate ($Y = \checkmark$) or even to do nothing at all ($Y = \{\ \}$).

Lawrence is thus able to distinguish 'soft' and 'hard' priority. In the former case, some means of arbitration is found when priorities conflict. Hard priority may result in deadlock. This contrasts with the approach taken here, where the means is sought to eliminate the possibility of priority conflict [4]. It may be argued to be at least risky to contrive a language where any such conflict may be directly expressed – a liberty surely better denied. However, a form of "prioritised interleaving" can be formulated that would seem to yield the same behaviour as prioritised alternation, though it is arguably much less transparent.

Overall, much progress has been made to introduce a semantic domain in which prioritisation can be defined, but it remains to achieve academic consensus. An appropriate operator, or set of operators, is a secondary issue, of greater concern here, where a single operator is preferred that abstracts prioritisation uniquely as interruptibility.

3.5 Laws

The following laws are suggested but not proven over any domain.

3.5.1 Unit and Zero

There appears to be no zero of of an alternation. *Stop*, however, suggests a unit:

$$P \hookleftarrow \mathit{Stop} \;=\; \mathit{Stop} \hookleftarrow P \;=\; P \tag{14}$$

The first equality is self-evident. The second is valid only if we disregard termination, and thus whether an alternation may be regarded as a sequential process (*i. e.* can be composed via the ';' operator). To the outside observer, behaviour will otherwise be identical.

Skip also suggests a unit, but again we must be mindful of termination:

$$P \hookleftarrow \mathit{Skip} \;=\; P$$

The reverse, $\mathit{Skip} \hookleftarrow P$, requires clarification. A definition must be sought so that the alternation itself either terminates or continues as P. The latter would be simpler and arguably would more commonly correspond with intuition and requirements. It should hold even when P itself terminates and thus affords termination of the alternation also.

$$P \hookleftarrow \mathit{Skip} \;=\; \mathit{Skip} \hookleftarrow P \;=\; P \tag{15}$$

Any law should hold over the entire process domain. Equation 15 seems to do this. Equation 14 clearly does not, and therefore lacks the status of law.

3.5.2 Association and Commutation

Alternation would seem associative:

$$(P_1 \hookleftarrow P_2) \hookleftarrow P_3 \;=\; P_1 \hookleftarrow (P_2 \hookleftarrow P_3) \tag{16}$$

but not generally commutative:

$$(P_1 = \mathit{Skip}) \;\vee\; (P_2 = \mathit{Skip}) \;\Longleftrightarrow\; P_1 \hookleftarrow P_2 \;=\; P_2 \hookleftarrow P_1 \tag{17}$$

3.5.3 Distribution

Neither sequential nor parallel composition distribute through an alternation. Though it might be appealing to suggest:

$$P \hookleftarrow (Q;\; R) \;=\; (P \hookleftarrow Q);\; (P \hookleftarrow R)$$

Equation 15 quickly denies it. (Consider $P = Q = \mathit{Skip}$.) Clearly, an alternation does not distribute through parallel or sequence either.

The *after* operator ($/s$) is expected to distribute through an alternation, exactly as it does a parallel composition:

$$(P_1 \hookleftarrow P_2)\,/s \;=\; (P_1/(s \upharpoonright \alpha P_1)) \hookleftarrow (P_2/(s \upharpoonright \alpha P_2)) \tag{18}$$

3.6 Specification

The axiomatic specification of a particular alternation looks very much like that of a parallel composition [5, p.90]:

$$\begin{aligned} &\forall\, r \in \mathit{traces}(P_1),\; \forall\, s \in \mathit{traces}(P_2).\;\; P_1 \;\mathbf{sat}\; C_1(r) \wedge P_2 \;\mathbf{sat}\; C_2(s) \;\Rightarrow \\ &\quad \forall\, t \in \mathit{traces}(P_1 \hookleftarrow P_2).\; (P_1 \hookleftarrow P_2) \;\mathbf{sat}\; (C_1(t \upharpoonright \alpha P_1) \wedge C_2(t \upharpoonright \alpha P_2)) \end{aligned} \tag{19}$$

Many reactive systems may be specified by requiring pre-emptive responses to occur to certain events (guards) according to a given prioritisation. Response latency can arguably be computed by the compiler, given adequate information regarding the platform, affording the satisfaction of certain timed requirements.

4 Termination

It becomes apparent early, as in Equation 10, that an alternation can be understood differently according to whether or not components are capable of terminating. If P is non-terminating (not a sequential process), Equation 14 is correct as it stands, at least with regard to traces observed, but not otherwise. Such equivalence might well be useful in some applications.

It may be worthwhile to consider an alternation at distinct epochs, according to whether each component is cyclic or terminating. According to Equation 15, once a component terminates, its clause may be considered deleted, and a new epoch begins.

This is precisely how a prioritised vectored interrupt system is commonly regarded. A response to some event may be to *disable* further response to the same event, some other event, or even *all* events. As noted in the earlier summary, the Honeysuckle `when` construct allows for this, and will terminate only when all interruption is disabled.

On considering how this may be modelled in CSP, one qualification regarding completion is first necessary. When an interrupting process terminates it must first have completed its response, allowing the interrupted process to resume. Completion cannot follow termination; clearly, nothing can. Termination must follow completion.

Component termination obviously requires an assurance that the environment will never again offer the corresponding guard. An interface must therefore convey information regarding when any given service becomes available and ceases to be available. The design of Honeysuckle will address this.

A *disable* command might be implemented via a shared variable, which in CSP can be regarded as a process D_i composed in parallel with an alternation:

$$D_i = send.d_i \rightarrow receive.d_i \tag{20}$$

One such process is required for each component P_i of an alternation.

A `when` clause might then be expressed generically (in normal form):

$$P_i = ((g_i \rightarrow R_i);\ P_i)\ \Box\ (receive.d_i \rightarrow Skip) \tag{21}$$

One possibility is that a response includes disabling of further interruption, in which case R_i communicates $send.d_i$. After completion (the final event of R_i), P_i should terminate. Here, again we appear to face the need for bias. It is desirable that, should both initial events, g_i and $receive.d_i$, be offered, that the latter is always chosen. However, this circumstance should never arise. It would expose a design flaw that could (should?) lead to deadlock. A component of an alternation should never be disabled when another occurrence of its guard event might occur.

Some applications might call for more than one component to have the capacity to disable a peer. Later attempts to disable an already disabled component could be accommodated via a slightly more complicated "shared variable" process. However, its recursion must then be terminated somehow. Alternatively, the ability to disable each component might be limited to the same or just one other and required to occur just once. This would be statically verifiable but may narrow the application domain. Because it represents the simpler path, this will be the rule for `when` in Honeysuckle.

It has always been the intention to explicitly share variables between components of `when`, as discussed earlier in Section 2.4. These might be modelled in a similar manner. Further work is needed here with regard to interference.

5 Conclusion

While the semantics of both prioritised alternation ($\leftrightarrow$) and the `when` programming construct remain incomplete, some progress has been made. An axiom has been proposed, along with various laws that characterize behaviour. Some key issues have been explored, such as the behaviour desired when components terminate and how termination might be brought about. Further work is needed with regard to interference and communication within a `when` construct.

Prioritisation in alternation has been shown to reduce to the same issue as with other 'asymmetric' operators, such as biased choice. An appropriate semantic domain still needs to gain consensus, though strong candidates exist. Further work is needed in order to secure a complete semantics of both operator and programming construct. Once that has been achieved, proof can then be sought of freedom from both deadlock and priority conflict in systems with *prioritised service architecture*.

It is hoped soon to complete the definition of the Honeysuckle programming language, and implement a compiler. A demonstration of the complete methodology will then become possible.

Acknowledgements

I am very grateful for a number of conversations with Mark Green, Jeremy Martin, and Adrian Lawrence with regard to this work.

References

[1] Ian R. East. The Honeysuckle programming language: An overview. *IEE Software*, 150(2):95–107, 2003.

[2] Per Brinch Hansen. *Operating System Principles*. Automatic Computation. Prentice Hall, 1973.

[3] Jeremy M. R. Martin. *The Design and Construction of Deadlock-Free Concurrent Systems*. PhD thesis, University of Buckingham, Hunter Street, Buckingham, MK18 1EG, UK, 1996.

[4] Ian R. East. Prioritised service architecture. In East and Martin et al., editors, *Communicating Process Architectures 2004*, Series in Concurrent Systems Engineering, pages 55–69. IOS Press, 2004.

[5] C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice Hall International, 1985.

[6] Ian R. East. Programming prioritized alternation. In H. R. Arabnia, editor, *Parallel and Distributed Processing: Techniques and Applications 2002*, pages 531–537, Las Vegas, Nevada, USA, 2002. CSREA Press.

[7] A. W. Roscoe. *The Theory and Practice of Concurrency*. Series in Computer Science. Prentice-Hall, 1998.

[8] C. B. Jones. Wanted: A compositional model for concurrency. In Annabelle McIver and Carroll Morgan, editors, *Programming Methodology*, Monographs in Computer Science, pages 1–15. Springer-Verlag, 2003.

[9] Steve Schneider. *Concurrent and Real-Time Systems: The CSP Approach*. Wiley, 2000.

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

[11] Adrian E. Lawrence. Hard and soft priority in CSP. In Barry Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, Series in Concurrent Systems Engineering, pages 169–195. IOS Press, 1999.

[12] Adrian E. Lawrence. Triples. In East and Martin et al., editors, *Proceedings of Communicating Process Architectures 2004*, Series in Concurrent Systems Engineering, pages 157–184. IOS Press, 2004.