

A CALCULUS BASED ON ABSENCE OF ACTIONS

PADMANABHAN KRISHNAN

Department of Computer Science

University of Canterbury, PBag 4800

Christchurch, New Zealand

E-mail: paddy@cosc.canterbury.ac.nz

ABSTRACT

In this article we present a process algebra where the behaviour can be specified when certain actions cannot be exhibited. This is useful in specifying time outs, interrupts etc. We present a few properties which form the basis for a sound and complete axiomatisation of a bisimulation equivalence relation. A comparison with other approaches is presented.

1. Introduction

Most approaches to concurrency and synchronisation are based on the presence of information. The rules that govern behaviour usually state that if a certain type of behaviour is possible, then another type of behaviour is also possible. But such a framework is not sufficient especially when one has to include concepts such as interrupts and priorities. To specify the semantics (and hence to implement) features such as interrupts and priorities, it is essential to have both the presence of and the absence of information. That is, we need to specify that if a certain behaviour is impossible, then some other behaviour is possible. The use of negative information has many uses including default reasoning in artificial intelligence¹⁴ and the select-else construct in Ada¹. In the default reasoning situation, the classical example is the assumption that all birds can fly which is discarded when penguin is a bird and penguins cannot fly is asserted. Thus the validity of the assertion that all birds can fly requires the absence of information on penguins. In Ada, the 'else' alternative in a 'select' statement is executed only if the other 'entries' cannot be accepted. To execute the 'else' alternative, knowing that there are no pending entry calls is essential.

While there have been various approaches to include priorities and interrupts in the context of concurrency, the work by Saraswat¹⁵ et. al. is the only one that we are aware of to consider a general framework for the absence of information. But their main concern is that of a non-monotonic logic and its denotational semantics.

Process algebras such as ACP³, CCS¹² and CSP⁸ are a popular approach to study concurrency. Unlike Saraswat¹⁵ et. al. who study negative information in the context of logic programming, we present a calculus with negative information using ideas from process algebra.

While studying negative information, it is easy to define a calculus whose syntax does not include negative information but whose semantics is based on absence of information. However, if such calculi are to be meaningful (i.e., have a sound semantics) the operational rules have to follow certain rules. See the work by Groote⁷ for the technical details. In certain situations the ideas expressed by Camilleri and Winskel⁶ are also applicable. We present a calculus where the behaviour in the absence of information is specified as part of the syntax and the semantics does not use any negative rules.

The syntax we consider is a variant of CCS. As usual we will consider a countable set of actions with a bijection $(\bar{\cdot})$ such that for every action μ , $\bar{\bar{\mu}} = \mu$. The bijection identifies complimentary actions which are used for synchronisation. The synchronisation of two processes is represented by a special τ action. For the sake of simplicity we do not consider relabelling but consider a syntax for specifying behaviour in the absence of actions.

$$P ::= 0 \mid (\mu.P) \mid [\neg S, P] \mid [\neg S, P] \mid (P + P) \mid (P \mid P) \mid (P \setminus H) \mid X \mid (\text{rec } X:P)$$

The intuitive semantics of processes expressed in the above syntax is as follows. The process 0 represents termination (or deadlock) and make no further progress. The process $(\mu.P)$ can exhibit a positive action (μ)

and then behave as P. The process $[\neg S, P]$ represents behaviour in the context of negative information. If the environment in which P executes cannot exhibit any action in S, the behaviour specified by P is exhibited. The process $[\neg S, P]$ is a stronger version of $[\neg S, P]$, in that the requirement of $\neg S$ persists for the entire behaviour of P. Strictly speaking, this form is not essential. One can use recursion and $[\neg S, P]$ over the entire behaviour of P. But the stronger form is useful when specifying behaviour and acts a convenient shorthand. The combinators $+$, $|$ and \backslash represent non-deterministic choice, concurrency and hiding respectively. When considering $(P | Q)$ we consider Q be in the operating environment of P and vice-versa. The term X and $(\text{rec } X:P)$ is used to define recursive processes. We assume that in $(\text{rec } X:P)$ the term P is well guarded so that the recursive process is well defined.

Before we present the formal details a few examples to illustrate the use of negative information are presented.

Example: Given two Ada tasks A and B defined as follows:

```
task A ... accept a do P else accept b do Q ...
task B ... A.b or A.a
```

This specifies that the entry a has higher priority than entry b . Task A can be translated into our calculus as: $([\neg\{\bar{a}\}, b \cdot Q] + a \cdot P)$ where the issuing of the entry call in task B becomes \bar{a} and \bar{b} .

Thus the overall system will be $(([\neg\{\bar{a}\}, b \cdot Q] + a \cdot P) | (\bar{b} \cdot 0 + \bar{a} \cdot 0)) \backslash \{a, b\}$.

In this particular situation the behaviour is equivalent to $(\tau \cdot P) \{a, b\}$.

If instead of task B one had tasks B and as follows:

```
task B ... A.b
task C ... A.a
```

the entry call from C will be accepted while entry call from task B will be suspended. The system in this case will be $(([\neg\{\bar{a}\}, b \cdot Q] + a \cdot P) | (\bar{b} \cdot 0 | \bar{a} \cdot 0)) \backslash \{a, b\}$

In both the cases, the presence of $\neg\{\bar{a}\}$ ensures that a has higher priority over b . The presence of the term $a \cdot P$ indicates that the action a can be selected.

Example: The behaviour of a CPU can be specified as a cyclical execution of the sequence fetch, decode and execute. This can be interrupted by an interrupt say (\bar{i}) at any given instant in the cycle. When the interrupt line is lowered (and hence the action \bar{i} disappears) the cycle is resumed. The above behaviour is specified below.

$\text{CPU} = [\neg\{\bar{i}\}, \text{NB}]$

$\text{NB} = \text{fetch} \cdot \text{decode} \cdot \text{execute} \cdot \text{NB}$

The process generating and holding the interrupt can be specified as

$\text{Intr} = \text{start} \cdot \text{Do}$

$\text{Do} = [\neg\{\text{done}\}, \bar{i} \cdot \text{Do}] + \overline{\text{done}} \cdot \text{Intr}$

The CPU can continue processing till the interrupt generator is started. Once it is started, the process Do holds the interrupt till it receives a request to complete in which case it reverts back to Intr. The negative information for Do ensures that action 'done' has a higher priority than \bar{i} and hence cannot be ignored by the process Do. Thus on completing the interrupt handling, the process Do has to disable \bar{i} , letting the CPU continue its regular processing.

The absence of information is required if the techniques used by Krishnan ¹⁰ are to be extended to verify the behaviour of a CPU in the presence of interrupts.

Example: Imprecise computation ¹¹ especially in the case of iterative improvements can be specified as follows.

$$\begin{aligned} C &= r_1 \cdot r_2 \cdot \dots \cdot r_n \cdot \text{Final} \\ \text{Final} &= r_f \cdot \text{Final} \\ \text{Muncher} &= [\neg \{hurry\}, \overline{r_1} \cdot \overline{r_2} \dots \cdot 0] \\ T &= \text{do_something} \cdot (\text{obtain_info} \cdot 0 \mid \text{HL}) \\ \text{HL} &= hurry \cdot \text{HL} \\ \text{Val} &= \overline{\text{obtain_info}} \cdot \sum_i \overline{r_i} \cdot v_i \cdot 0 \\ \text{Sys} &= (C \mid T \mid \text{HL} \mid \text{Val}) \{ \text{obtain_info}, r_1, r_2, \dots, r_n \} \end{aligned}$$

The process C is the main computation process whose body is specified as a sequence of actions which can be suspended at any given instant by enabling *hurry*. The process T is a timer which after ‘doing something’ activates both *hurry* which is persisted and a process Val which inspects the state of C (via synchronisation) and prints an appropriate value (v_i).

It is important to note that the hiding involves the r_i ’s. Hence if Muncher is absent, the process C will be unable to advance as it will be unable to exhibit the r_i ’s due to the restriction on Sys. Furthermore, even though r_i is restricted, the process Muncher cannot advance after *hurry* has been asserted. Hence after a *hurry* the only possible synchronisation is between C and Val.

Example: Our final example is a modified version of the example presented by Baeten ² et. al. Consider a system with a file server, a key board and a display. The key board generates signals which are either displayed directly or are requests to the file server to display the status. Hence the keyboard generates interrupts to the file server. The formal specification is as given below. FS is the main file server while FI is the interrupt handler. The process Display and Keyboard specify the behaviour of the display and key board respectively.

$$\begin{aligned} \text{FS} &= [\neg \{f_int\} \text{BF}] \mid \text{FI} \\ \text{FI} &= \overline{f_int} \cdot \overline{f_display} \cdot \text{FI} \\ \text{Display} &= \overline{n_display} \cdot \overline{n_done} \cdot \text{Display} + f_display \cdot f_done \cdot \text{Display} \\ \text{Keyboard} &= \overline{n_display} \cdot \text{Keyboard} + f_int \cdot \text{Keyboard} \\ \text{System} &= (\text{FS} \mid \text{Display} \mid \text{Keyboard}) \setminus \{n_display, f_int, f_display\} \end{aligned}$$

The synchronisation between FI and Keyboard on f_int ensures that the interrupt is handled and FS resumes its regular service.

2. Formal Details

An operational semantics based on labelled transition systems ¹³ is given in figure 1. To define the semantics of absence of information, it is essential to know the information available; i.e., all actions that are possible. All other actions are deemed to be impossible at this stage. This is characterised from the syntax of the process as follows.

Definition: 1 Define the set of possible actions a process (say P) makes available (written as $\text{ready}(P)$) as follows.

$$\begin{aligned} \text{ready}(0) &= \emptyset \\ \text{ready}(\mu \cdot P) &= \{\mu\} \\ \text{ready}(P + Q) &= \text{ready}(P) \cup \text{ready}(Q) \\ \text{ready}(P \mid Q) &= \text{ready}(P) \cup \text{ready}(Q) \end{aligned}$$

$$\begin{array}{c}
\langle \mu.P, Q \rangle \xrightarrow{\mu} P \\
\\
\frac{\langle P, Q \rangle \xrightarrow{\mu} P'}{\langle [\neg S, P], Q \rangle \xrightarrow{\mu} P'} S \cap \text{ready}(Q) = \emptyset \\
\\
\frac{\langle P, Q \rangle \xrightarrow{\mu} P'}{\langle \llbracket \neg S, P \rrbracket, Q \rangle \xrightarrow{\mu} \llbracket \neg S, P \rrbracket} S \cap \text{ready}(Q) = \emptyset \\
\\
\frac{\langle P_1, P_2 \rangle \xrightarrow{\mu} P'_1}{(P_1 + P_2) \xrightarrow{\mu} P'_1} \\
(P_2 + P_1) \xrightarrow{\mu} P'_1 \\
\\
\frac{\langle P_1, P_2 \rangle \xrightarrow{\mu} P'_1}{(P_1 | P_2) \xrightarrow{\mu} (P'_1 | P_2)} \\
(P_2 | P_1) \xrightarrow{\mu} (P_2 | P'_1) \\
\\
\frac{\langle P_1, P_2 \rangle \xrightarrow{\mu} P'_1}{(P_1 | P_2) \xrightarrow{\tau} (P'_1 | P_2)} \\
(P_2 | P_1) \xrightarrow{\tau} (P_2 | P'_1) \\
\\
\frac{\langle P, Q \rangle \xrightarrow{\mu} \langle P', Q \rangle}{(P \setminus H) \xrightarrow{\mu} (P' \setminus H)} (\mu, \bar{\mu} \notin H) \\
\\
\frac{\langle P, Q \rangle \xrightarrow{\mu} \langle P', Q \rangle}{(\text{rec } X:P) \xrightarrow{\mu} P'(X/(\text{rec } X:P))}
\end{array}$$

Figure 1: Operational Semantics

$$\text{ready}([\neg S, P]) = \text{ready}(P)$$

$$\text{ready}(\llbracket \neg S, P \rrbracket) = \text{ready}(P)$$

$$\text{ready}(P \setminus H) = \text{ready}(P) - (H \cup \bar{H})$$

$$\text{ready}(\text{rec } X:P) = \text{ready}(P)$$

In the presentation of the rules, we have abused notation for the sake of simplifying the presentation. Technically one should have different rules for the behaviour of a process in an environment and that of a process by itself. We use a single relation \rightarrow to indicate both the behaviours. Hence at the surface level the rules appear very similar to the CCS rules.

Following Milner¹² a bisimulation relation induced by \rightarrow can be defined. A direct definition of a bisimulation relation (\sim) based only on observational behaviour would not be a congruence. This is due to the presence of the $|$ combinator. If two processes are equivalent, it is essential that their behaviour be identical in all environments. The definition of \sim is as follows.

Definition: 2 Process P and Q are bisimilar ($P \sim Q$) iff for all processes R

$$\langle P, R \rangle \xrightarrow{\mu} P' \text{ implies that } \langle Q, R \rangle \xrightarrow{\mu} Q' \text{ and } P' \sim Q'$$

$$\langle Q, R \rangle \xrightarrow{\mu} Q' \text{ implies that } \langle P, R \rangle \xrightarrow{\mu} P' \text{ and } P' \sim Q'$$

It is easy to check that \sim is the smallest relation that is a congruence.

We now present a few laws that are satisfied by \sim .

Proposition 1 *If P and Q are CCS processes (that is do not use the absence of information construct) and P and Q are bisimilar under the semantics presented for CCS, P and Q are indeed bisimilar under the semantics presented here.*

The above proposition shows that our extension is consistent with CCS. That is, the new rules do not distinguish processes in the absence of the use of negative information.

Proposition 2 *If $[\neg S_1, P] \sim [\neg S_2, Q]$ and $P \xrightarrow{\mu} P', S_1 = S_2$.*

Proof: If S_1 and S_2 are different (say in μ') $\langle [\neg S_1, P], \mu' \cdot 0 \rangle$ and $\langle [\neg S_2, Q], \mu' \cdot 0 \rangle$ will have different behaviours. \square

Proposition 3 *Let P be $[\neg S_1, \sum_{i \in I} a_i \cdot P_i]$ and Q be $[\neg S_2, \sum_{j \in J} b_j \cdot Q_j]$.*

If $S_1 \cap \{b_j, j \in J\}$ and $S_2 \cap \{a_i, i \in I\}$ are both nonempty then $(P \mid Q) \sim 0$.

If $S_1 \cap \{b_j, j \in J\} = \emptyset$ but $S_2 \cap \{a_i, i \in I\} \neq \emptyset$ then $(P \mid Q) \sim R$ where

$$R = [\neg S_1, \sum_i a_i \cdot (P_i \mid Q)]$$

If $S_2 \cap \{a_i, i \in I\} = \emptyset$ but $S_1 \cap \{b_j, j \in J\} \neq \emptyset$ then $(P \mid Q) \sim R$ where

$$R = [\neg S_2, \sum_j b_j \cdot (P \mid Q_j)]$$

Proof: It is easy to see that $\text{ready}(P) = \{a_i, i \in I\}$ while $\text{ready}(Q) = \{b_j, j \in J\}$. Hence if $(P \mid Q) \xrightarrow{a_i}$, it is clear that $(S_1 \cap \text{ready}(Q))$ has to be empty. Other cases are similar. Hence depending on the relationship between the S_k 's and the ready sets the appropriate behaviour will be exhibited. \square

In the last two results in the above proposition, the negative information guard is maintained as the bisimilarity has to be preserved over all contexts. If one removes the negative information guard in R , it is easy to devise an environment (as shown in the following example) where they are not bisimilar.

Example: Consider the process $[\neg\{a\}, b \cdot 0] \mid [\neg\{b\}, c \cdot 0]$.

This process is bisimilar to

$$[\neg\{a\}, b \cdot (0 \mid [\neg\{b\}, c \cdot 0])].$$

If the $\neg\{a\}$ is removed, the behaviours of the two processes in the context of $a \cdot 0$ are not identical.

As we are still within the domain of interleaving semantics for the ' \mid ' combinator the following proposition is valid.

Proposition 4 *Let P be $[\neg S_1, \sum_{i \in I} a_i \cdot P_i]$ and Q be $[\neg S_2, \sum_{j \in J} b_j \cdot Q_j]$.*

If $S_1 \cap \{b_j, j \in J\}$ and $S_2 \cap \{a_i, i \in I\}$ are both empty then $(P \mid Q) \sim R$ where

$$\begin{aligned} R = & [\neg S_1, \sum_{i \in I} a_i \cdot (P_i \mid Q)] + \\ & [\neg S_2, \sum_{j \in J} b_j \cdot (P \mid Q_j)] + \\ & [\neg (S_1 \cup S_2), \sum_{i, j, a_i = \bar{b}_j} \tau \cdot (P_i \mid Q_j)] \end{aligned}$$

Proof: As $S_1 \cap \{b_j, j \in J\}$ is empty, $\langle P, Q \rangle \xrightarrow{a_i}$ for every a_i . Similarly $\langle Q, P \rangle \xrightarrow{b_j}$ for every b_j . Hence both asynchronous behaviour and synchronisation moves are possible. \square

The above result is straightforward generalisation of the expansion theorem for CCS. Propositions 3 and 4 together cover all possible interleaved behaviour.

$$\begin{array}{ll}
P + P = P & 0 \mid P = P \\
P + 0 = P & 0 \setminus H = 0 \\
P + Q = Q + P & P \mid Q = Q \mid P \\
(P + Q) + R = P + (Q + R) & (P \mid Q) \mid R = P \mid (Q \mid R)
\end{array}$$

Figure 2: Equations

Proposition 5 *Other properties include*

$$\begin{aligned}
& \sum_{i \in I} a_i \cdot P_i \sim [\neg \emptyset, \sum_{i \in I} a_i \cdot P_i] \\
& [\neg S_1, [\neg S_2, P]] \sim [\neg (S_1 \cup S_2), P] \\
& [\neg S, (\mu_1 \cdot P_1 + \mu_2 \cdot P_2)] \sim [\neg S, \mu_1 \cdot P_1] + [\neg S, \mu_2 \cdot P_2] \\
& [\neg S, P] \setminus H \sim [\neg S, (P \setminus H)] \\
& (P + Q) \setminus H \sim (P \setminus H) + (Q \setminus H) \\
& (\mu \cdot P) \setminus H \sim 0 \text{ if } \mu \text{ or } \bar{\mu} \in H \\
& (\mu \cdot P) \setminus H \sim \mu \cdot (P \setminus H) \text{ if } \mu \text{ and } \bar{\mu} \notin H \\
& [\neg S_1, 0] \sim [\neg S_2, 0]
\end{aligned}$$

It is easy to derive a sound and complete axiomatisation of the bisimulation relation for finite processes. That is we do not consider recursion and $[\![\]\!]$. One can translate the above rules into equations (and add a few axioms such as associativity, commutativity, idempotence etc.) to obtain the axiomatisation. The proof follows the usual lines of defining a standard form and proving that every bisimilar process can be reduced to the same standard form. The standard that needs to be considered is $[\neg S, P]$ where P is in CCS standard form (i.e., of the form $\sum_{i \in I} a_i \cdot P_i$ where each P_i is in standard form). The following propositions formalise the above description.

Definition: 3 *A process is in CCS standard form if it is of the form $\sum_{i \in I} a_i \cdot P_i$ where each P_i is in standard form. Note that 0 is in CCS standard form as 0 can be expressed as an empty choice.*

A process in our calculus is in standard form if it is of the form $[\neg S, P]$ where S is a set of action (perhaps empty) and P is in CCS standard form.

Proposition 6 *Every process can be converted to a process in standard form using the equation form of the results related to bisimulation and the axioms in figure 2.*

The use of standard forms is to get a handle on the structure of process, given a specific behaviour. That is, given that a process P in standard form, and if P can exhibit an action (say μ), the syntactic structure of P can be assumed to be of the form $[\neg S, (\mu \cdot P_1 + P_2)]$. This observation is then used to prove the following lemma.

Lemma 1 Absorption Lemma *If P and Q are in standard form such that $P \sim Q$, $P + Q = P = Q$*

Proof Outline: If P and Q cannot exhibit an action, they are of the form $[\neg S, 0]$ and the result is obvious. Otherwise as $P \sim Q$, for every R , $\langle P, R \rangle \xrightarrow{\mu} P'$ implies that $\langle Q, R \rangle$ has a matching move. For this to occur P must be of the form $[\neg S, (\mu \cdot P_1 + P_2)]$ and Q of the form $[\neg S, (\mu \cdot Q_1 + Q_2)]$. Note that in both P and Q the negative part must be identical. Hence $[\neg S, (\mu \cdot P_1 + P_2)] + [\neg S, \mu \cdot Q_1]$ is equal to P . By repeating the process the entire of Q_2 can be absorbed into P . \square

Proposition 7 *If $P \sim Q$, it can be proved that $P = Q$.*

Proof: The proof follows the usual steps. The first is to use the above proposition and convert P and Q to standard form. Hence it suffices to consider P and Q already in standard form. If P and Q in standard form are bisimilar, the absorption lemma shows that $(P + Q) = P$ and $(Q + P) = Q$ and as $(P + Q) = (Q + P)$, $(P = Q)$. \square

The modal μ -calculus¹⁶ has been used to obtain a logical characterisation of bisimulation. However in our case it is not clear how the semantics of satisfaction of a formula by a process of the form $[\neg S, P]$ should be defined. One can adopt the view that $[\neg S, P] \models \varphi$ iff $P \models \varphi$. This view is satisfactory as far as observation behaviour of processes is concerned. However, this is not sufficient to characterise bisimulation as both $[\neg\emptyset, \mu\cdot 0]$ and $[\neg\{b\}, \mu\cdot 0]$ satisfy $\langle\mu\rangle\text{True}$ but clearly the two processes are not bisimilar. While it is possible to define satisfaction of a formula φ for the term $[\neg S, P]$ as

$$[\neg S, P] \models \varphi \text{ iff } \forall R, ([\neg S, P] \mid R) \models \varphi$$

the definition is unsatisfactory due to the universal quantification over the set of processes (R). This invalidates the use of traditional model checking techniques. Hence a more discerning form of satisfaction is essential and this is a topic of future work.

But we can present a few results related to the simple definition of satisfaction for processes whose behaviour depends on the absence of other actions.

Proposition 8 *If $P \models \langle a \rangle \text{True}$, and $Q \models [a] \text{False}$, then if $([\neg S, P] \mid Q) \models \langle a \rangle \text{True}$, then for every $\mu \in S$, $Q \models [\mu] \text{False}$*

As Q cannot perform an a action, and P can, the only way for $([\neg S, P] \mid Q)$ to exhibit an a action was for Q not to disable P . Hence the ready set of Q cannot contain any action in S .

Proposition 9 *For every P such that $P \models [c] \text{False}$, $([\neg\{a\}, b \cdot 0] \mid P) \models [c] \text{False}$*

The above proposition states that as long as P cannot perform a c action, placing it in an environment which cannot perform a c action will not magically enable c .

3. Related Work

Bolognesi and Lucidi⁵ present two calculi in the context of real-time systems. The first deals with urgent actions and is restricted to a single process. That is, if a process can perform an urgent action, it cannot idle. Hence this is useful in controlling choice in the presence of time outs. The second calculus deals with a binary operator which is used to disable other processes. Once a process is disabled, it does not make any contribution to further behaviours. They achieve their main aim of providing with a single very powerful operator. Even with the powerful binary combinator, it is hard to specify concepts such as temporary suspension. Furthermore, being a binary operator, the environment has to be encoded in. In our case we can first specify the system and then worry about the environment. Of course, we have not added any concept related to time. But that is easily achieved using the well known techniques^{9,17}.

Camilleri and Winskel⁶ describe the addition of priority choice (\dashv) to CCS. The operational rules appear to be more complex due to the assumption of an implicit environment. We have a simplified presentation as the environment is represented as another process. Every process using \dashv can be expressed in our calculus. For example, $(a \cdot 0 \dashv b \cdot 0 \dashv c \cdot 0)$ can be represented as $[\neg\{\bar{a}, \bar{b}\}, c \cdot 0] + [\neg\{\bar{a}\}, b \cdot 0] + a \cdot 0$.

Apart from simplifying the presentation of the operational semantics, by incorporating absence of action information in the syntax of processes, we have done away with need for a bi-level syntax. They required a bi-level syntax to avoid giving semantics to processes such as $(a \cdot 0 \dashv \bar{b} \cdot 0) \mid (b \cdot 0 \dashv \bar{a} \cdot 0)$. Hence they outlaw this by imposing constraints on the syntax. In our case this process will be equated with 0 . This is because in the definition of ready for non-deterministic choice, the union of all possibilities is taken.

Berry⁴ provides a calculus for preemption based on the synchronous language Esterel. But the main drawback of the work is the need for a large number of constructs to express various types of preemptions. They also do not present any algebraic laws. We are able encode these operators in terms of our much simpler

operators (although in fairness it must be said that not all encodings are perspicuous) which satisfy certain algebraic properties. Furthermore, our definitions are based on asynchronous behaviour. It is possible to modify the semantics to specify instantaneous behaviour by extending the environment (using the ready set) to include the current process whose behaviour is to be determined.

4. Acknowledgements

This work has been partially supported by UoC Grant No. 1787123.

5. References

1. *Ada programming language (ANSI/MIL-STD-1815A)*. Washington, D.C. 20301, January 1983.
2. J. C. M. Baeten, J. A. Bergstra, and J. W. Klop. Syntax and defining equations for an interrupt mechanism in process algebra. Technical Report CS-R8503, CWI, 1985.
3. J. A. Bergstra and J. W. Klop. Process Theory Based on Bisimulation Semantics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, LNCS 354, pages 50–122. Springer Verlag, 1988.
4. G. Berry. Preemption in Concurrent Systems. In *Foundations of Software Technology and Theoretical Computer Science*, LNCS 761, pages 72–93. Springer Verlag, 1993.
5. T. Bolognesi and F. Lucidi. Time Process Algebras with Urgent Interactions and a Unique Powerful Binary Operator. In J. deBakker, editor, *Proceedings of the REX Workshop on Real-Time: Theory in Practice*, LNCS 600, pages 124–148. Springer Verlag, 1991.
6. J. Camilleri and G. Winskel. CCS with priority choice. In *IEEE Symposium on Logic in Computer Science*, pages 246–255, Amsterdam, The Netherlands, 1991.
7. J. F. Groote. Transition System Specifications with Negative Premises. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR 90*, LNCS-458, pages 332–341. Springer Verlag, 1990.
8. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.
9. P. Krishnan. A Calculus of Timed Communicating Systems. *International Journal of Foundations of Computer Science*, 3(3):303–322, September 1992.
10. P. Krishnan. A Case Study in Specifying and Testing Architectural Features. *Microprocessors and Microsystems*, 18(3):123–130, April 1994.
11. K. Lin, S. Natarajan, and J. W. Liu. Imprecise results: Utilizing partial computations in real-time systems. In *IEEE Real-Time Systems Symposium*, pages 210–217, 1987.
12. R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.
13. G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.
14. R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.
15. V. Saraswat, R. Jagadeesan, and V. Gupta. Default Timed Concurrent Constraint Programming. In *22nd ACM Symposium on Principles of Programming Languages*, January 1995.
16. C. Stirling. An Introduction to Modal and Temporal Logics for CCS. In *Joint UK/Japan Workshop on Concurrency*, LNCS 491, pages 2–20, 1989.
17. Wang Yi. CCS+Time = An Interleaving Model for Real-Time Systems. In *ICALP -91*, LNCS 510. Springer Verlag, 1991.