AN ASYNCHRONOUS CALCULUS BASED ON
THE ABSENCE OF ACTIONS

PADMANABHAN KRISHNAN
Department of Computer Science, University of Canterbury
Private Bag 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. Processes of the form \([\sim S, P]\) represent a behaviour which is specified by \(P\) but only when the environment cannot perform any action in \(S\). This is useful in specifying priority, time outs, interrupts etc. We present a few examples which illustrate the use of the extended calculus. A bisimulation relation induced by a labelled transition system is then considered. We present a few properties which form the basis for a sound and complete axiomatisation of a bisimulation equivalence relation. This requires an extension of the syntax. This is because the absence of information from the environment used in the operational semantics is captured syntactically. 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 intelligence\(^{16}\) and the select-else construct in Ada\(^1\). 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 inter-

\(^1\)Dedicated to Nidhi
rups in the context of concurrency, the work by Saraswat 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 ACP, CCS and CSP are a popular approach to study concurrency. Unlike Saraswat et. al. who study negative information in the context of logic programming, we present a calculus with negative information using ideas from process algebra.

Within the context of concurrency, there are a number of approaches which deal with specific concepts such as priority and preemption or interrupts. But they do not consider behaviour in the context of the absence of information. We present a calculus where the behaviour in the absence of information is specified as part of the syntax. Section presents a more in depth comparison with other approaches.

The syntax we consider is a variant of CCS. As usual we will consider a countable set of actions with a bijection such that for every action μ, ∼μ = μ. The bijection identifies complimentary actions which are used for synchronisation. The synchronisation of two processes is represented by a special action. For the sake of simplicity we do not consider relabelling. The novel aspect of this work is a syntax for specifying behaviour in the absence of actions.

\[
P ::= 0 \mid (μ \cdot P) \mid [\neg S, P] \mid \bar{[\neg S, P]} \mid (P + P) \mid (P \mid P) \mid (P \setminus H) \mid X \mid (\text{rec} \ X:P)
\]

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 (μ · 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 \bar{[\neg S,P]} is a stronger recursive 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 \ 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 (rec X:P) is used to define recursive processes. We assume that in (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:

\text{task } A \ldots \text{accept } a \text{ do } P \text{ else accept } b \text{ do } Q \ldots
\text{task } B \ldots A.b \text{ or } A.a

This specifies that the entry \( a \) has higher priority 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 \((\{\neg\{\bar{a}\}, b\cdot Q + a\cdot P\} \cup \{\bar{b}\cdot 0 + \bar{a}\cdot 0\}) \setminus \{a,b\}\).

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

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

\text{task } B \ldots A.b
\text{task } C \ldots A.a

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

In both the cases, the presence of \(\neg\{a\}\) 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 \( i \) at any given instant in the cycle. When the interrupt line is lowered (and hence the action \( i \) disappears) the cycle is resumed. The above behaviour is specified below.

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

The absence of information is required if the techniques used by Krishnan\textsuperscript{12} are to be extended to verify the behaviour of a CPU in
the presence of interrupts. For example, the general behaviour of traps in the SPARC v9\textsuperscript{13} architecture can be specified as follows.

\[
NB = \text{fetch} \cdot \text{decode} \cdot \left( \frac{\text{precise} \cdot \text{trap} \cdot \text{NB} + \text{def} \cdot \text{trap} \cdot \text{NB} + \text{dist} \cdot \text{trap} \cdot \text{NB} + \text{reset} \cdot \text{trap} \cdot \text{NB} + \text{non} \cdot \text{trap} \cdot \text{NB}}{\text{AH}} \right)
\]

\[
\text{PrTH} = \text{precise} \cdot \text{trap} \cdot (\text{wait} \cdot 0 | \text{AH})
\]

\[
\text{DefTH} = \text{def} \cdot \text{trap} \cdot \text{delay} \cdot (\text{wait} \cdot 0 | \text{AH})
\]

\[
\text{Retry} = \text{retry} \cdot \text{accept} \cdot \text{wait} \cdot \text{Retry}
\]

\[
\text{Sys} = (\parallel \cdot \{\text{wait}\} \cdot \text{NB} | \text{PrTH} | \text{DefTH} | \text{Retry}) \\{\text{wait}, \text{precise} - \text{trap}, \text{def} - \text{trap} \ldots\}
\]

The process NB is a refinement of the process NB described earlier. After the decoding phase various trap types can be indicated. If a precise trap were raised, the action wait suspends the behaviour of NB immediately while a deferred trap has a delay action before NB is suspended. On issuing a retry instruction, the wait action disappears and NB can continue its regular behaviour. The process AH is left unspecified and represents the actual trap handler.

The advantage of include absence of information in the syntax is demonstrated. The normal behaviour of a system can be described without undue worry about the operating environment and without a description of the potential interrupts. Later during system composition, the appropriate interrupts can be included as a wrapper over the normal behaviour.

\textbf{Example:} Imprecise computation\textsuperscript{13} especially in the case of iterative improvements can be specified as follows.

\[
C = r_1 \cdot r_2 \cdot \ldots \cdot r_n \cdot \text{Final}
\]

\[
\text{Final} = r_f \cdot \text{Final}
\]

\[
\text{Muncher} = (\parallel \cdot \{\text{hurry}\} \cdot \vec{r_1} \cdot \vec{r_2} \cdot \ldots \cdot 0)
\]

\[
T = \text{do} \cdot \text{something} \cdot (\text{obtain} \cdot \text{info} \cdot 0 | \text{HL})
\]

\[
\text{HL} = \text{hurry} \cdot \text{HL}
\]

\[
\text{Val} = \text{obtain} \cdot \text{info} \cdot \sum_i r_i \cdot v_i \cdot 0
\]

\[
\text{Sys} = (C | T | \text{HL} | \text{Val}) \{\text{obtain} \cdot \text{info}, r_1, r_2, \ldots, r_n\}
\]

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. Here again the benefits of having absence information in the syntax is clear. One can describe regular computation without worrying about the imprecise nature of the desired computation. By constructing Muncher etc. the process is simplified. Note that in general, the process C will be the most complex while process such as Muncher, T, HL can be reused in many situations without much change.

**Example:** Our final example is a modified version of the example presented by Baeten\(^3\) et. al. Consider a system with a file server, a keyboard and a display. The keyboard 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 keyboard respectively.

\[
FS = [ \neg \{ f \cdot int \} \text{ BF } ] | FI
\]

\[
FI = f \cdot int \cdot f \cdot display \cdot FI
\]

\[
\text{Display} = n \cdot display \cdot n \cdot done \cdot Display + f \cdot display \cdot f \cdot done \cdot Display
\]

\[
\text{Keyboard} = \overline{n \cdot display} \cdot \text{Keyboard} + f \cdot int \cdot \text{Keyboard}
\]

\[
\text{System} = (FS | \text{Display} | \text{Keyboard}) \setminus \{ n \cdot display, f \cdot int, f \cdot display \}
\]

The synchronisation between FI and Keyboard on \( f \cdot int \) ensures that the interrupt is handled and FS resumes its regular service.

2. **Formal Details**

An operational semantics based on labelled transition systems\(^15\) 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 ready(P)) as follows.

\[ \text{ready}(0) = \emptyset \]
\[ \text{ready}(\mu \cdot P) = \{ \mu \} \]
\[ \text{ready}(P + Q) = \text{ready}(P) \cup \text{ready}(Q) \]
\[ \text{ready}(P | Q) = \text{ready}(P) \cup \text{ready}(Q) \]
\[ \text{ready}(\langle \neg S, P \rangle) = \text{ready}(P) \]
\[ \text{ready}(\langle \neg S, P \rangle) = \text{ready}(P) \]
\[ \text{ready}(P \setminus H) = \text{ready}(P) \cdot (H \cup \overline{H}) \]
\[ \text{ready}(\text{rec } X : P) = \text{ready}(P) \]

In the presentation of the rules, we have abused notation for the sake of simplifying the presentation. The development of the syntactic structure of our calculus did not involve any term of the form \( \langle P, Q \rangle \). Yet, we have used them in the operational rules. The interpretation of \( \langle P, Q \rangle \) is that P operates in an environment described by Q. 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. The ready set plays the role of an explicit environment as against an implicit environment used by Camilleri and Winskel to model priority choice.

Following Milner 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 \( \langle \rangle \) 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

\[ \langle P, R \rangle \rightarrow^\mu P' \text{ implies that } \langle Q, R \rangle \rightarrow^\mu Q' \text{ and } P' \sim Q' \]
\[ \langle Q, R \rangle \rightarrow^\mu Q' \text{ implies that } \langle P, R \rangle \rightarrow^\mu P' \text{ and } P' \sim Q' \]
\[ \langle R, P \rangle \rightarrow^\mu R' \text{ implies that } \langle R, Q \rangle \rightarrow^\mu R' \]
\[ \langle \mu.P, Q \rangle \xrightarrow{\mu} P \]

\[ \langle P, Q \rangle \xrightarrow{\mu} P' \]
\[ \langle [-S, P], Q \rangle \xrightarrow{\mu} P' \quad S \cap \text{ready}(Q) = \emptyset \]
\[ \langle [\neg S, P], Q \rangle \xrightarrow{\mu} [\neg S, P'] \quad S \cap \text{ready}(Q) = \emptyset \]
\[ \langle P_1, P_2 \rangle \xrightarrow{\mu} P_1' \]
\[ (P_1 + P_2) \xrightarrow{\mu} P_1' \]
\[ (P_2 + P_1) \xrightarrow{\mu} P_1' \]
\[ \langle P_1, P_2 \rangle \xrightarrow{\mu} P_1' \]
\[ (P_1 | P_2) \xrightarrow{\mu} (P_1 | P_2) \]
\[ (P_2 | P_1) \xrightarrow{\mu} (P_2 | P_1) \]
\[ \langle P_1, P_2 \rangle \xrightarrow{\tau} (P_1 | P_2) \]
\[ (P_2 | P_1) \xrightarrow{\tau} (P_2 | P_1) \]
\[ \langle P, Q \rangle \xrightarrow{\mu} \langle P', Q \rangle \quad (\mu, \bar{\mu} \notin H) \]
\[ (P \setminus H) \xrightarrow{\mu} (P' \setminus H) \]
\[ \langle P, Q \rangle \xrightarrow{\mu} \langle P', Q \rangle \quad (\mu, X \notin H) \]
\[ \langle P, Q \rangle \xrightarrow{\mu} \langle P', Q \rangle \]
\[ \text{(rec X:P)} \xrightarrow{\mu} P'(X/(\text{rec X:P})) \]

Fig. 1. Operational Semantics
The third condition for the bisimulation relation is required as not only can other processes affect \( P \) and \( Q \), \( P \) and \( Q \) can affect other processes as well. That is, \( R \) can either be the environment or \( P \) (and hence \( Q \)) can also be the environment. 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.

Although, the definition of bisimulation involved a universal quantifier, the following proposition is useful when it comes to detecting bisimilarity.

**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 \( \mu' \)) \( [\neg S_1, P], \mu' \cdot 0 \) and \( [\neg S_2, Q], \mu' \cdot 0 \) will have different behaviours. \( \square \)

We now address the issue of obtaining a sound a complete equational characterisation for the bisimulation equivalence. As we are still within the domain of interleaving semantics for the ‘\( \parallel \)’ combinator, we should be able to obtain a form of expansion theorem. The use of the ready set, which is based on the syntactic structure of processes, causes some difficulty in obtaining a satisfactory expansion theorem as the following example shows.

**Example:** Let \( P \) be \( [\neg\{a\}, b \cdot 0] \) and \( Q \) be \( [\neg\{b\}, a \cdot 0] \). Operationally \( (P \mid Q) \) behaves as \( 0 \). However, \( (P \mid Q) \) is not bisimilar to \( 0 \), as the context \( [\neg\{a,b\}, c \cdot 0] \) distinguishes the two. This is because the ready set of \( 0 \) is empty while the ready set of \( (P \mid Q) \) is \( \{a,b\} \) though neither action can be exhibited.

Thus we could leave the parallel combinator as an essential primitive in the syntax. But this is unsatisfactory especially in the context of interleaving. What is necessary is the ability to remember the ready set even when the underlying process is changed. Hence we extend the syntax to include what we term as kill sets. The purpose of the kill set is to indicate which actions can result in interrupting other processes (even if they never really occur). This is necessary as we defined the ready set purely syntactically and this information needs to be preserved by semantic transformations.

Thus the process \( C(K, P) \) represents the behaviour \( P \) with a kill set \( K \) where \( K \) is the set of actions. For the purposes of extending the operational semantics
the ready set of $C(K, P)$ is defined to be the union of $K$ and the ready set of $P$. The operational behaviour of $C(K, P)$ is derived to be identical to that of $P$. We have not considered $C(K, P)$ to be part of the original syntax. We have not found any particular use of the kill set and we have preferred to leave it as a part of the extended syntax purely for the purposes of a satisfactory expansion theorem.

Now the propositions 3 and 4 which are variations of the original expansion theorem are valid.

**Proposition 3** Let $P$ be $C(K_1, [\neg S_1, \sum_{i \in I} a_i \cdot P_i])$ and $Q$ be $C(K_2, [\neg S_2, \sum_{j \in J} b_j \cdot Q_j])$.

Let $K = \text{ready}(P) \cup \text{ready}(Q)$ with $R_p = \text{ready}(P)$ and $R_q = \text{ready}(Q)$.

- If $(S_1 \cap R_p)$ and $(S_2 \cap R_p)$ are both nonempty then $(P \mid Q) \sim C(K, 0)$

- If $(S_1 \cap R_p) = \emptyset$ but $(S_2 \cap R_p) \neq \emptyset$ then $(P \mid Q) \sim C(K, R)$ where

  $$ R = [\neg S_1, \sum_{i} a_i \cdot (P_i \mid Q)] $$

- If $(S_2 \cap R_p) = \emptyset$ but $(S_1 \cap R_p) \neq \emptyset$ then $(P \mid Q) \sim C(K, R)$ where

  $$ R = [\neg S_2, \sum_{j} b_j \cdot (P \mid Q_j)] $$

**Proof:** Notice that $\text{ready}(P) = R_p$ while $\text{ready}(Q) = R_q$. Hence if $(P \mid Q) \xrightarrow{a_i} \cdot$, 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.

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. Similarly the union of the kill sets are also maintained.

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

This process is bisimilar to $C([b, c], [\neg \{a\}, b \cdot (0 \mid [\neg \{b\}, c \cdot 0]])$.

This is because the presence of $b$ prevents the exhibition of $c$. If the $\neg \{a\}$ at the top level is removed, the behaviours of the two processes in the context of $a \cdot 0$ are not identical. For a similar reason the guard for the action $c$ is also retained.

The proposition 3 represents some form of merging when a process is disabled due to the presence of certain action. The following proposition represents an unconstrained progress of two processes which do not disable each other.
Proposition 4 Let \( P \) be \( C(K_1, \neg S_1, \sum_{i \in I} a_i \cdot P_i) \) and \( Q \) be \( C(K_2, \neg S_2, \sum_{j \in J} b_j \cdot Q_j) \).

Let \( K \) denote \( \text{ready}(P) \cup \text{ready}(Q) \).

If \( S_1 \cap (K_2 \cup \{\{b_j, j \in J\}\}) \) and \( S_2 \cap (K_1 \cup \{a_i, i \in I\}) \) are both empty then \( (P | Q) \sim C(K, R) \) where

\[
R = \neg S_1, \sum_{i \in I} a_i \cdot (P_i | Q) + \\
\neg S_2, \sum_{j \in J} b_j \cdot (P | Q_j) + \\
\neg (S_1 \cup S_2), \sum_{i,j, a_i=b_j} \tau \cdot (P_i | Q_j)
\]

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

Proposition 5 Other properties include

\[
\sum_{i \in I} a_i \cdot P_i \sim \neg \emptyset, \sum_{i \in I} a_i \cdot P_i
\]

\[
\neg S_1, \neg S_2, P \sim \neg (S_1 \cup S_2), P
\]

\[
\neg S, (\mu_1 \cdot P_1 + \mu_2 \cdot P_2) \sim \neg S, \mu_1 \cdot P_1 + \neg S, \mu_2 \cdot P_2
\]

\[
\neg S, P \setminus H \sim \neg S, (P \setminus H)
\]

\[
(P + Q) \setminus H \sim (P \setminus H) + (Q \setminus H)
\]

\[
(\mu \cdot P) \setminus H \sim 0 \text{ if } \mu \text{ or } \overline{\mu} \in H
\]

\[
(\mu \cdot P) \setminus H \sim \mu \cdot (P \setminus H) \text{ if } \mu \text{ and } \overline{\mu} \notin H
\]

\[
\neg S_1, 0 \sim \neg S_2, 0
\]

\[
C(\emptyset, P) \sim P
\]

\[
C(K_1, P_1) | C(K_2, P_2) \sim C((K_1 \cup K_2), (P_1 | P_2))
\]

\[
C(K_1, P_1) + C(K_2, P_2) \sim C((K_1 \cup K_2), (P_1 + P_2))
\]

\[
\neg S, C(K, P) \sim C(K, \neg S, P)
\]

\[
C(K, C(K_2, P)) \sim C((K_1 \cup K_2), P)
\]

\[
C(K, P) \sim C(K', P) \text{ where } K' = K \cup \text{ready}(P)
\]
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 form that needs to be considered is $C(K, [\neg S, P])$ 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 pre-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.

A process in our extended calculus is in standard form if it is of the form $C(K, P)$ where $K$ is a set of actions and $P$ is in pre-standard form.

**Proposition 6** Every process can be converted to a process in standard form using the equational 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 $\mu$), 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 $P \sim Q$, $P + Q = P = Q$.

**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 \((P + Q) = (Q + P), (P = Q)\).

3. Modal Logic

The modal \(\mu\)-calculus\(^9\) has been used to obtain a logical characterisation of bisimulation in CCS. 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 \([\emptyset, \mu \cdot 0] and [\neg \{b\}, \mu \cdot 0] satisfy \langle \mu \rangle True but clearly the two processes are not bisimilar. While it is possible to define satisfaction of a formula \(\varphi\) for the term \([\neg S, P]\) as

\[\neg S, P] \models \varphi iff \forall R, ([\neg S, P] | R) \models \varphi\]

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. Thus we do not have a satisfactory logical characterisation of the bisimulation equivalence in our original calculus nor do we have a satisfactory answer for the extended calculus.

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. These preliminary results are sufficient to check various properties of the systems we have considered.

**Proposition 8** If \(P \models [a]True, and Q \models [a]False, then if ([\neg S, P] | Q) \models [a]True, then for every \(\mu \in S, Q \models [\mu]False.\)

As Q cannot perform an a action, and P can, the only way for ([\neg S, P] | 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.

This result is the dual of the above one. If the combination of P and Q cannot exhibit an a action, Q clearly has to disable P.

**Proposition 9** If \(P \models [a]False and Q \models [a]False, (P | Q) \models [c]False.\)

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

4. Related Work

Berry\(^6\) 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. Our calculus can be perceived as a partially synchronous one (i.e., synchronous only when absence comes into play and asynchronous otherwise).

Bolognesi and Lucidi\(^7\) 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. In the terminology of Berry\(^6\), it is a form of must preemption applied to choice. 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\(^11,20\).

Camilleri and Winskel\(^8\) describe the addition of priority choice \((\rightarrow)\) 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 \(\rightarrow\) can be expressed in our calculus. For example, \((a\cdot 0 \rightarrow b\cdot 0 \rightarrow c\cdot 0)\) can be represented as \([\neg\{a,b\}, c\cdot 0] + [\neg\{a\}, b\cdot 0] + a\cdot 0\).

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 \((a\cdot 0 \rightarrow b\cdot 0)\) \(|\) \((b\cdot 0 \rightarrow a\cdot 0)\). 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.

While we are able to express the various concepts in our calculus, we have not been able to provide a compositional translation from the other calculi to the one presented here. We are currently investigating techniques/constructs which permit a compositional translation.
5. Acknowledgements

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

6. References


