Journal of Universal Computer Science, vol. 3, no. 12 (1997), 1415-1428
submitted: 22/7/97, accepted: 13/10/97, appeared: 28/12/97 © Springer Pub. Co.

An Asynchronous Calculus Based on the Absence of Actions [1]

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 in which the behaviour in the absence of certain actions can be specified. Processes of the form $[\neg\ S,\ P]$ represent a behaviour which is specified by P but only in an environment which cannot perform any action in S. If the environment can perform an action in S, the process is suspended. 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.

Key Words: process algebra, absence of information, bisimulation

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 [Reiter, 1980] and the select-else construct in Ada [Ada, 1983]. 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 reported in [Saraswat et al., 1995] 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 [Bergstra and Klop, 1988], CCS [Milner, 1989] and CSP [Hoare, 1985] are a popular approach to study concurrency. Unlike the work in [Saraswat et al., 1995] which studies negative information in the context

[1] A preliminary version appeared at the Eighth International Symposium on Intensional Programming: ISLIP'95

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. A few typical examples include [Baeten et al., 1987], [Cleaveland and Hennessy, 1990] and [Camilleri and Winskel, 1991]. Preemption or interrupts is considered in [Baeten et al., 1986] and [Berry, 1993]. 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 4 presents a more in depth comparison with other approaches and discusses a few potential extensions.

While studying negative information, one can 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. The work reported in [Groote, 1990] and [Verhoef, 1994] discusses a number of the technical issues. In certain situations the ideas expressed in [Camilleri and Winskel, 1991] are also applicable. We present a calculus where the behaviour in the absence of information is specified as part of the syntax and the operational 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 (say Act) with a bijection ($\bar{\cdot}$) such that for every action μ in Act, $\bar{\bar{\mu}} = \mu$. The bijection identifies complimentary actions which are used for synchronisation. The synchronisation of two processes is represented by a special τ action. We let the set $Actions$ denote ($Act \cup \{\tau\}$). 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.

$$\begin{aligned} P ::= \mathbf{0} \mid (\mu{\cdot}P) \mid [\neg\, S, P] \mid [\![\neg\, S, P]\!] \mid (P + P\,) \mid \\ (P \mid P\,) \mid (P \setminus H) \mid \mathrm{X} \mid (\mathrm{rec\ X}{:}P\,) \end{aligned}$$

where $\mu \in Actions$, $H \subseteq Act$ and $S \subseteq Actions$.

The intuitive semantics of processes expressed in the above syntax is as follows. The process $\mathbf{0}$ represents termination (or deadlock) and make no further progress. The process $(\mu{\cdot}\, P)$ can exhibit the action (μ) and then behave as P. The process $[\neg\, S, P]$ represents behaviour where the state of the environment is considered. If the environment in which $[\neg\, S, P]$ executes cannot exhibit any action in S, the behaviour as specified by P is exhibited. The process $[\![\neg\, S, P]\!]$ 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 $+$, $\mid$ and $\setminus$ represent non-deterministic choice, concurrency and hiding respectively. When considering $(P \mid Q)$ 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 1. 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:

$$([\neg\{\overline{a}\},\, b{\cdot}Q] + a{\cdot}P)$$

where the issuing of the entry calls in task B is translated as $\overline{a}$ and $\overline{b}$ respectively.

Thus the overall system will be

$$(\ ([\neg\{\overline{a}\},\, b{\cdot}Q] + a{\cdot}P) \mid (\overline{b}{\cdot}\mathbf{0} + \overline{a}{\cdot}\mathbf{0})\ ) \setminus \{a,b\}$$

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

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

$$(\ ([\neg\{\overline{a}\},\, b{\cdot}Q] + a{\cdot}P) \mid (\overline{b}{\cdot}\mathbf{0} \mid \overline{a}{\cdot}\mathbf{0})\ ) \setminus \{a,b\}$$

In both the cases, the presence of $\neg\ \{\overline{a}\}$ 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 2. 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 $(\overline{i})$ at any given instant in the cycle. When the interrupt line is lowered (and hence the action $\overline{i}$ disappears) the cycle is resumed. The above behaviour is specified below.

$$CPU = [\![\ \neg\{\overline{i}\},\, NB\ ]\!]$$
$$NB = fetch \cdot decode \cdot execute \cdot NB$$

The process generating and holding the interrupt can be specified as

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

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 $\overline{i}$ and hence cannot be ignored by the process Do Thus on completing the interrupt handling, the process Do has to disable $\overline{i}$, letting the process CPU continue its regular processing. The absence of information is required if the techniques used in [Krishnan, 1994] 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 [Weaver and Germond, 1994] architecture can be specified as follows.

$$NB = fetch \cdot decode{\cdot}(\overline{preciseTrap}{\cdot}NB + \overline{defTrap}{\cdot}NB + \overline{nonTrap}{\cdot}NB\ )$$
$$PrTH = preciseTrap{\cdot}\ (wait{\cdot}\mathbf{0} \mid AH)$$
$$DefTH = defTrap \cdot delay{\cdot}\ (wait{\cdot}\mathbf{0} \mid AH)$$
$$Retry = retry \cdot \overline{wait}{\cdot}\ Retry$$
$$Sys = ([\![\ \neg\ \{wait\},\, NB]\!] \mid PrTH \mid DefTH \mid Retry)\backslash \{wait,preciseTrap,defTrap,retry\}$$

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 $\overline{preciseTrap}$), 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 action *retry*), the *wait* action disappears and NB can continue its regular behaviour. The process AH is left unspecified and represents the actual trap handler. But we assume that the process AH will issue the action *retry*. The action $\overline{nonTrap}$ indicates a normal instruction and is not handled by any other process.

The advantage of including 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.

Example 3. Our final example concerns imprecise computation [Lin et al., 1987]. An imprecise computation involves some form of iterative improvement of some value. Either the computation is allowed to reach its natural conclusion or in the case of some significant event (like the arrival of some deadline) the computation is terminated. The result of the computation is the current value. Hence if the computation reached its natural conclusion, the result will be best possible, otherwise it is as close to the best as possible given other constraints.

An example in our calculus is presented below.

$$
\begin{aligned}
&C = r_1 \cdot r_2 \cdot \ldots r_{n-1} \cdot Final\\
&Final = r_n \cdot Final\\
&Muncher = [\![\neg \{hurry\}, \overline{r_1} \cdot \overline{r_2} \ldots \cdot \mathbf{0}]\!]\\
&T = deadline \cdot (obtain \cdot \mathbf{0} \mid HL)\\
&HL = hurry \cdot HL\\
&Val = \overline{obtain} \cdot \sum_{i \in 1 \ldots n} \overline{r_i} \cdot v_i \cdot \mathbf{0}\\
&Sys = (C \mid T \mid Muncher \mid Val) \setminus \{ obtain, r_1, r_2, \ldots, r_n \}
\end{aligned}
$$

The process C is the main computation process whose body is specified as a sequence of actions whose normally termination is indicated by the process $Final$. That is, the process $Final$ represents normal termination and hence no improvement is possible (it always exhibits r_n). The process T is a timer which after the action *deadline* activates both *hurry* (via process HL) which is persistent and a process Val which inspects the state of C. The process Val inspects C (via synchronisation) and prints an appropriate value (v_i). The process $Muncher$, in the absence of *hurry*, synchronises with C thus letting C progress from r_i to r_{i+1}.

It is important to note that the hiding involves the r_i's. Hence if the process $Muncher$ 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. The key to our example is the fact that the process $Muncher$ cannot advance after *hurry* has been enabled. After the action *deadline* has been exhibited, the only possible synchronisation is between C and Val. Also the process Val is activated only after exhibiting *deadline* as it is awaiting synchronisation on the action *obtain*. If the action *deadline* occurs after process C has finished, the action *hurry* has no effect as the process $Muncher$ has terminated.

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 desired semantics of imprecise computation can be obtained. 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.

2 Formal Details

An operational semantics based on labelled transition systems [Plotkin, 1981] is given in figure 1. To define the semantics of absence of information, it is essential to know the state of the environment. The environment is also characterised as a process and hence we introduce a notion of a system which is a pair of processes. The pair $\langle P,Q \rangle$ represents the process P in the environment Q. Hence to know the state of the environment, a ready set which can be computed from the syntax of the process is defined as follows.

Definition 1. Define the set of possible actions a process (say P) can make available (written as $ready(P)$) as follows.

$ready(\mathbf{0}) = \emptyset$
$ready(\mu \cdot P) = \{\mu\}$
$ready(P + Q) = ready(P) \cup ready(Q)$
$ready(P \mid Q) = ready(P) \cup ready(Q)$
$ready([\neg\ S, P]) = ready(P)$
$ready([\![\neg\ S, P]\!]) = ready(P)$
$ready(P \setminus H) = ready(P)$ - $(H \cup \overline{H})$
$ready(recX : P) = ready(P)$

The labelled transition relation $\longrightarrow$ is defined to be a subset of $Sys\times$ $Actions$ $\times Process$ where $Process$ is the set of all possible processes and Sys is the set of all process pairs.

The intuition is that given a process P and an environment E, the relation specifies the action that can be exhibited by P and a new process P' that is derived from P. The behaviour of P' is then described in the context of its environment. Thus to know if a particular process can exhibit a particular action, its operating environment also needs to be specified. As the behaviour of a process may be dependent only on the ready set of its environment, we need not consider all possible structures of the environment. Thus, in the semantics for $(P \mid Q)$ in the context of E, the behaviour of P is determined by the ready set of both Q and E. It does not really matter whether one considers $(Q \mid E)$ or $(E \mid Q)$. Hence only of the rules is presented. As a notational convenience, we abuse notation slightly when we write $(P \xrightarrow{\mu} Q)$ where P and Q are processes when we should have written $(\langle P,\mathbf{0} \rangle \xrightarrow{\mu} Q)$.

Following [Milner, 1989] a bisimulation relation induced by $\longrightarrow$ 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

$$\langle \mu{\cdot}P,\, Q\rangle \xrightarrow{\mu} P$$

$$\frac{\langle P,Q\rangle \xrightarrow{\mu} P'}{\langle [\neg\, S,\, P],Q\rangle \xrightarrow{\mu} P'}\ S \cap \mathrm{ready}(Q) = \emptyset$$

$$\frac{\langle P,Q\rangle \xrightarrow{\mu} P'}{\langle [\![\ \neg\, S,\, P\ ]\!],Q\rangle \xrightarrow{\mu} [\![\ \neg\, S,\, P'\ ]\!]}\ S \cap ready(Q) = \emptyset$$

$$\frac{\langle P_1,P_2\rangle \xrightarrow{\mu} P_1'}{\begin{array}{c}\langle (P_1 + P_3),P_2\rangle \xrightarrow{\mu} P_1' \\ \langle (P_3 + P_1),P_2\rangle \xrightarrow{\mu} P_1'\end{array}}$$

$$\frac{\langle P_1,(P_2 \mid P_3)\rangle \xrightarrow{\mu} P_1'}{\begin{array}{c}\langle (P_1 \mid P_2),P_3\rangle \xrightarrow{\mu} (P_1' \mid P_2) \\ \langle (P_2 \mid P_1),P_3\rangle \xrightarrow{\mu} (P_2 \mid P_1')\end{array}}$$

$$\frac{\begin{array}{c}\langle P_1,(P_2 \mid P_3)\rangle \xrightarrow{\mu} P_1' \\ \langle P_2,(P_1 \mid P_3)\rangle \xrightarrow{\overline{\mu}} P_2'\end{array}}{\begin{array}{c}\langle (P_1 \mid P_2),\, P_3\rangle \xrightarrow{\tau} (P_1' \mid P_2) \\ \langle (P_2 \mid P_1),\, P_3\rangle \xrightarrow{\tau} (P_2 \mid P_1')\end{array}}$$

$$\frac{\langle P,Q\rangle \xrightarrow{\mu} P'}{\langle (P \setminus H),Q\rangle \xrightarrow{\mu} (P' \setminus H)}\ (\mu,\, \overline{\mu} \notin H)$$

$$\frac{\langle P,Q\rangle \xrightarrow{\mu} P'}{\langle (\mathrm{rec}\ \mathrm{X}{:}P),Q\rangle \xrightarrow{\mu} P'(\mathrm{X}/(\mathrm{rec}\ \mathrm{X}{:}P))}$$

Figure 1: Operational Semantics

$|$ 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. A relation $\mathcal{R}$ is a bisimulation, if for every $(P,\, Q)$ belonging to $\mathcal{R}$, for every process E the following conditions hold.

$\langle P,E\rangle \xrightarrow{\mu} P'$ implies that $\langle Q,E\rangle \xrightarrow{\mu} Q'$ and $(P',\, Q')$ belong to $\mathcal{R}$.
$\langle Q,E\rangle \xrightarrow{\mu} Q'$ implies that $\langle P,E\rangle \xrightarrow{\mu} P'$ and $(P',\, Q')$ belong to $\mathcal{R}$.
$\langle E,P\rangle \xrightarrow{\mu} E'$ implies that $\langle E,Q\rangle \xrightarrow{\mu} E'$

Two processes P and Q are bisimilar $(P \sim Q)$ if there exists a bisimulation containing (P,Q).

The third condition for the bisimulation relation is required as not only can the environment process affect P and Q, P and Q can act as part of the environment for other processes as well. In other words, not only can the process E

can either be viewed as an environment for P (and hence Q), the processes P (and Q) can be viewed as the environment for E.

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

Proposition 3. If P and Q are CCS processes (that is do not use the absence of information construct) and P are 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/identify extra 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.

Lemma 4. If $[\neg\ S_1,\ P] \sim [\neg\ S_2,\ Q]$ and $P \xrightarrow{\mu} P'$, $S_1 = S_2$.

Proof: Let $[\neg\ S_1,\ P] \sim [\neg\ S_2,\ Q]$ such that S_1 and S_2 are different. Without loss of generality assume μ' belongs to S_1 but not to S_2. The system $\langle[\neg\ S_1,\ P],\ \mu' \cdot \mathbf{0}\rangle$ cannot exhibit any action while $\langle[\neg\ S_2,\ Q],\ \mu' \cdot \mathbf{0}\rangle$ can exhibit the action μ'. Hence they cannot be bisimilar as assumed. $\square$

The requirement ($P \xrightarrow{\mu} P'$) is essential. The process $[\neg\ S_1,\mathbf{0}]$ is bisimilar to the process $[\neg\ S_2,\ \mathbf{0}]$ as in both cases the process $\mathbf{0}$ cannot exhibit any action.

Proposition 5. Other properties of bisimulation are presented below.

1. $\sum_{i \in I} a_i \cdot P_i \sim [\neg\emptyset, \sum_{i \in I} a_i \cdot P_i]$
2. $[\neg\ S_1\ ,\ [\neg\ S_2, P]] \sim [\neg\ (S_1 \cup\ S_2),\ P]$
3. $[\neg\ S,\ (\mu_1 \cdot P_1\ +\ \mu_2 \cdot P_2)] \sim [\neg\ S,\ \mu_1 \cdot P_1] + [\neg\ S,\ \mu_2 \cdot P_2]$
4. $[\neg\ S,\ P] \setminus H \sim [\neg\ S,\ (P \setminus H)]$
5. $(P\ +\ Q) \setminus H \sim (P \setminus H)\ +\ (Q \setminus H)$
6. $[\neg\ S_1,\ \mathbf{0}] \sim [\neg\ S_2,\ \mathbf{0}]$
7. $(\mu{\cdot}P) \setminus H \sim \mathbf{0}$ if (μ or $\overline{\mu} \in H$) and $((ready(P) \cap (H \cup \overline{H})) = \emptyset)$
8. $(\mu{\cdot}P) \setminus H \sim \mu({\cdot}P \setminus H)$ if μ and $\overline{\mu} \notin H$

While all the bisimulations can be proven, we only provide an intuitive explanation for some of them. Statement 1 indicates that a process which is not disabled by any action in its environment is related to a basic CCS-like process. As the ready set is computed from the syntax of the process and the fact that the ready set can be used to disable other processes, it is essential that the ready sets of bisimilar processes be identical. This requirement is partially captured in statement 7. This issue is taken up in more detail in the rest of the paper.

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 '|' 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 4. Let P be $[\neg\{a\}, b{\cdot}\mathbf{0}]$ and Q be $[\neg\{b\}, a{\cdot}\mathbf{0}]$. Operationally the process $(P \mid Q)$ behaves as $\mathbf{0}$. However, $(P \mid Q)$ is not bisimilar to $\mathbf{0}$, as the process $[\neg\{a,b\},c{\cdot}\mathbf{0}]$ (say R) can be used to distinguish the effect of P and Q. This is because the ready set of $\mathbf{0}$ is empty while the ready set of $(P \mid Q)$ is $\{a,b\}$ though neither action can be exhibited. The process R in the context of $\mathbf{0}$ can exhibit the action c. However, the process R in the context of $(P \mid Q)$ cannot exhibit c as the the presence of actions a and b disable the process R.

Thus we could leave the parallel combinator as an essential primitive in the syntax. But this is unsatisfactory especially in the context of an interleaving semantics. What is necessary is the ability to remember the original 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 $\mathsf{C}(K,P)$ 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 $\mathsf{C}(K,P)$ is defined to be the union of K and the ready set of P. That is formally specified as: $\mathit{ready}(\mathsf{C}(K,P)) = K \cup \mathit{ready}(P)$

The operational behaviour of $\mathsf{C}(K,P)$ is derived to be identical to that of P. This precise rule is as follows.

$$\frac{\langle P,Q\rangle \xrightarrow{\mu} P'}{\langle \mathsf{C}(K,P),Q\rangle \xrightarrow{\mu} P'}$$

We did not consider $\mathsf{C}(K,P)$ to be part of the original syntax as there was no particular use of the kill set. We have preferred to leave it as a part of the extended syntax purely for the purposes of a satisfactory expansion theorem.

Now the lemmas 6 and 7 which are variations of the original expansion theorem are valid.

Lemma 6. Let P be $\mathsf{C}(K_1,[\neg\, S_1, \sum_{i\in I} a_i \cdot P_i])$ and Q be $\mathsf{C}(K_2,[\neg\, S_2, \sum_{j\in J} b_j \cdot Q_j])$.
Let $R_p = \mathit{ready}(P)$, $R_q = \mathit{ready}(Q)$ and $K = R_p \cup R_q$.

1. If $(S_1 \cap\ R_q) \neq \emptyset$ and $(S_2 \cap\ R_p) \neq \emptyset$, then $(P \mid Q) \sim \mathsf{C}(K,\mathbf{0})$
2. If $(S_1 \cap\ R_q) = \emptyset$ and $(S_2 \cap\ R_p) \neq \emptyset$ then $(P \mid Q) \sim \mathsf{C}(K,R)$ where
$$R = [\neg\, S_1, \sum_{i\in I} a_i \cdot (P_i \mid\ Q)]$$
3. If $(S_2 \cap\ R_p) = \emptyset$ and $(S_1 \cap\ R_q) \neq \emptyset$, then $(P \mid Q) \sim \mathsf{C}(K,R)$ where
$$R = [\neg\, S_2, \sum_{j\in J} b_j \cdot (P \mid\ Q_j)]$$

Proof: We prove part 2 of the above lemma. The other cases are similar.

Consider the relation $\mathcal{A}$ defined as
$\{((P \mid Q), \mathsf{C}(K,R)) \mid P, Q, R \text{ and } K \text{ as above }\} \cup \{(X,X) \mid X \text{ any process}\}$.

We show that $\mathcal{A}$ is a bisimulation.

Consider any process E such that $ready(E) \cap S_1 = \emptyset$.

Now, $\langle (P \mid Q), E\rangle \xrightarrow{a_i} (P_i \mid Q)$. This is because $S_1 \cap R_q = \emptyset$. For the same reason, $\langle R,E\rangle \xrightarrow{a_i} (P_i \mid Q)$ due to which $\langle \mathsf{C}(K,R),E\rangle \xrightarrow{a_i} (P_i \mid Q)$. As identical processes belong to $\mathcal{A}$, the relation is a bisimulation.

As $S_2 \cap R_p \neq \emptyset$, $\langle (P \mid Q), E\rangle \overset{b_j}{\not\longrightarrow}$, only the behaviour of P needs to be considered. The behaviours of $\langle E, (P \mid Q)\rangle$ and $\langle E, \mathsf{C}(K,R)\rangle$ are identical. That is, $\langle E, (P \mid Q)\rangle \xrightarrow{\mu} E'$ iff $\langle E, \mathsf{C}(K,R)\rangle \xrightarrow{\mu} E'$.

This is because the ready set of $(P \mid Q)$ is identical to the ready set of $\mathsf{C}(K,R)$, namely, K.

Again as identical processes belong to $\mathcal{A}$, it is a bisimulation.

$\square$.

In the above lemma, 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 5. Consider the process $[\neg\{a\}, b{\cdot}\mathbf{0}] \mid [\neg\{b\}, c{\cdot}\mathbf{0}]$. This process is bisimilar to $\mathsf{C}(\{b,c\},\ [\neg\{a\},\ b{\cdot}(\ \mathbf{0} \mid [\ \neg\{b\},\ c{\cdot}\mathbf{0}])])$.

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 $a{\cdot}\mathbf{0}$ are not identical. For a similar reason the guard for the action c is also retained.

Lemma 6 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.

Lemma 7. Let P be $\mathsf{C}(K_1, [\neg\ S_1, \sum_{i\in I} a_i \cdot P_i])$ and Q be $\mathsf{C}(K_2, [\neg\ S_2, \sum_{j\in J} b_j \cdot Q_j])$.

Let $K = ready(P) \cup ready(Q)$.

If $S_1 \cap ready(Q) = S_2 \cap ready(P) = \emptyset$, then $(P \mid Q) \sim \mathsf{C}(K,R)$ where

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

Proof: The result follows directly from the following observations. Using this, a bisimulation relation can be exhibited.

The first observation is that if $(ready(E) \cap S_1 = \emptyset)$ then $\langle P, (Q \mid E)\rangle \xrightarrow{a_i} P_i$.

The second is that if $(ready(E) \cap S_2 = \emptyset)$, $\langle Q, (P \mid E)\rangle \xrightarrow{b_j} Q_j$

Hence if $(ready(E) \cap S_1 = \emptyset)$ and $(ready(E) \cap S_2 = \emptyset)$ both asynchronous and synchronisation moves are possible. The later is possible only if $(a_i = \overline{b_j})$ in which case $\langle(P \mid Q), \mathrm{E}\rangle \xrightarrow{\tau} (P_i \mid Q_j)$.

If E is blocked/enabled by $(P \mid Q)$ then E will be blocked/enabled by $\mathsf{C}(K, R)$ as the ready sets are identical.

Formally, the relation
$\{((P \mid Q), \mathsf{C}(K,R)) \mid P, Q, R$ and K as above $\} \cup \{(X,X) \mid X$ a process$\}$ can be shown to be a bisimulation. □

The above result is straightforward generalisation of the expansion theorem for CCS. Lemmas 6 and 7 together cover all possible interleaved behaviour.

Proposition 8. Other properties involving kill sets include

1. $\mathsf{C}(\emptyset, P) \sim$ P
2. $\mathsf{C}(K_1, P_1) \mid \mathsf{C}(K_2, P_2) \sim \mathsf{C}((K_1 \cup K_2), (P_1 \mid P_2))$
3. $\mathsf{C}(K_1, P_1) + \mathsf{C}(K_2, P_2) \sim \mathsf{C}((K_1 \cup K_2), (P_1 + P_2))$
4. $[\neg S, \mathsf{C}(K, P)] \sim \mathsf{C}(K, [\neg S, P])$
5. $\mathsf{C}(K_1, \mathsf{C}(K_2, P)) \sim \mathsf{C}((K_1 \cup K_2), P)$
6. $\mathsf{C}(K, P) \sim \mathsf{C}(K', P)$ where $K' = K \cup ready(P)$
7. $\mathsf{C}(K, (\mu \cdot P) \setminus H) \sim \mathsf{C}(K', \mathbf{0})$ if μ or $\overline{\mu} \in H$ and $K' = K \cup ready(P \setminus H)$

An intuitive explanation for some of the properties are presented below. Item 1 states that an empty kill set has no effect. As the kill sets are nothing more than a form of superset of the ready set, they can be combined with the ready sets. This is indicated in item 2. Item 4 recognises the fact that the kill set only disables process belonging to the enviroment. Hence it does not interefere with the disabling set. Item 7 is useful in simplifying process that cannot exhibit any action due to restriction. As the ready set is calculated syntactically, it is essential for the kill set to reflect the ready set. The definition of the ready set makes it clear that μ cannot belong to the ready set of $((\mu \cdot P) \setminus H)$.

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 $\mathsf{C}(K, [\neg S, P])$ 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.

$P + P = P$	$\mathbf{0} \mid P = P$
$P + \mathbf{0} = P$	$\mathbf{0} \setminus H = \mathbf{0}$
$P + Q = Q + P$	$P \mid Q = Q \mid P$
$(P + Q) + R = P + (Q + R)$	$(P \mid Q) \mid R = P \mid (Q \mid R)$

Figure 2: Equations

Definition 9. 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 $\mathbf{0}$ is in CCS standard form as $\mathbf{0}$ 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 actions (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 $\mathsf{C}(K, P)$ where K is a set of actions and P is in pre-standard form.

Proposition 10. 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 is 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 [¬S, $(\mu{\cdot}P_1 + P_2)$]. This observation is then used to prove the following lemma.

Lemma 11. Absorption Lemma If P and Q are in standard form such that $(P \sim Q)$, then $P + Q = P = Q$.

Proof: The proof is essentially by structural induction on the processes.

If P and Q are both of the form $\mathsf{C}(K, [\neg S, \mathbf{0}])$ and the result is obvious.

Otherwise as $(P \sim Q)$, for every R, $\langle P,R\rangle \xrightarrow{\mu} P'$ implies that $\langle Q,R\rangle \xrightarrow{\mu} Q'$ such that $P' \sim Q'$.

As P and Q are in standard form, one can assume that P must be of the form $\mathsf{C}(K, [\neg S, (\mu \cdot P_1 + P_2)])$ and Q of the form $\mathsf{C}(K, [\neg S, (\mu \cdot Q_1 + Q_2)])$ where P_1 is identical to P$'$ and Q_1 is identical to Q$'$.

By a simple extension of lemma 4 both P and Q must have identical K and S sets. By induction hypothesis, $P_1 = Q_1$.

Thus $\mathsf{C}(K, [\neg S, (\mu \cdot P_1 + P_2)]) + \mathsf{C}(K, [\neg S, \mu \cdot Q_1])$ is equal to $\mathsf{C}(K, [\neg S, (\mu \cdot P_1 + P_2 + \mu \cdot Q_1)])$ which is equal to P.

By repeating the above step, the process Q_2 can also be absorbed into P. □

Lemma 12. If $P \sim Q$, it can be proved that $P = Q$.

Proof: Follows directly from proposition 10 and lemma 11. □

3 Modal Logic

The modal μ-calculus [Stirling, 1989] 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 observational behaviour of processes is concerned. However, this is not sufficient to characterise bisimulation as both $[\neg\emptyset, \mu{\cdot}\mathbf{0}]$ and $[\neg\{\mathrm{b}\}, \mu{\cdot}\mathbf{0}]$ satisfy $\langle\mu\rangle$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

$[\neg\ S,\ P] \models \varphi$ iff $\forall\ R,\ \langle[\neg\ S,\ P] \mid R\rangle \models \varphi)$

the definition is unsatisfactory. The reason is that a process of the form $[\neg\ S,\ a \cdot \mathbf{0}]$ where S is not empty can never satisfy the formula $\langle \mathrm{a}\rangle$True. The universal quantification over the set of processes (R) leads to the undesirable property. 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 using the kill sets.

But we can present a few results related to the simple standard 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 13. If $P \models \langle a\rangle$True, and $Q \models [a]$False, then if $([\neg\ S,\ P] \mid Q) \models \langle a\rangle$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 $([\neg\ S,\ P] \mid Q)$ to exhibit an a action was for Q not to disable P. Hence the ready set of Q cannot contain any action in S. Thus the stronger requirement of Q being unable to exhibit any action in S holds.

Proposition 14. Let Q be a standard CCS process. If $P \models \langle a\rangle$True, and $Q \models [a]$False, then if $([\neg\ S,\ P] \mid Q) \models [a]$False, then $\exists \mu \in S,\ Q \models \langle\mu\rangle$True.

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. The requirement on Q to be a standard CCS process is best illustrated by the following example.

Example 6. Let P be $[\neg,\ \{a\}\ c \cdot \mathbf{0}]$. Let Q be $([\neg\ \{b\},\ d \cdot \mathbf{0}] \mid [\neg\ \{d\},\ a \cdot \mathbf{0}])$. The ready set of Q will contain a and d and hence Q will disable P. However, Q cannot exhibit the action a. If Q cannot use negative information, the above proposition will indeed be satisfied.

Proposition 15. If $P \models [a]$False and $Q \models [a]$False $(P\ |Q) \models [a]$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, 1993] 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, 1991] 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, 1993], 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 defining only one but 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 developed in [Krishnan, 1992] and [Yi, 1991].

[Camilleri and Winskel, 1991] describes the addition of priority choice ($\not\to$) 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 $\not\to$ can be expressed in our calculus. For example, (a$\cdot\mathbf{0}$ $\not\to$ b$\cdot\mathbf{0}$ $\not\to$ c$\cdot\mathbf{0}$) can be represented as $[\neg\{\overline{a},\overline{b}\}$, c$\cdot\mathbf{0}]$ + $[\neg\{\overline{a}\}$, b$\cdot\mathbf{0}]$ + a$\cdot\mathbf{0}$.

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$\cdot\mathbf{0} \not\to \overline{b}\cdot\mathbf{0}$) | (b$\cdot\mathbf{0} \not\to \overline{a}\cdot\mathbf{0}$). Hence they outlaw this by imposing constraints on the syntax. In our case this process will be equated with $\mathbf{0}$. This is because in the definition of ready for non-deterministic choice, the union of all possibilities is taken.

Acknowledgements

This work has been partially supported by UoC Grant No. 1787123. Many thanks to the anonymous referees for their helpful comments.

References

[Ada, 1983] (1983). *Ada programming language (ANSI/MIL-STD-1815A)*. Washington, D.C. 20301.

[Baeten et al., 1986] Baeten, J., Bergstra, J., and Klop, J. (1986). Syntax and Defining Equations for an Interrupt Mechanism in Process Algebra. *Fundamenta Informaticae*, IX(2):127–168.

[Baeten et al., 1987] Baeten, J., Bergstra, J., and Klop, J. (1987). Ready Trace Semantics for Concrete Process Algebras with the Priority Operator. *The Computer Journal*, 30(6):498–506.

[Bergstra and Klop, 1988] Bergstra, J. A. and Klop, J. W. (1988). Process Theory Based on Bisimulation Semantics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume LNCS 354, pages 50–122. Springer Verlag.

[Berry, 1993] Berry, G. (1993). Preemption in Concurrent Systems. In *Foundations of Software Technology and Theoretical Computer Science*, volume LNCS 761, pages 72–93. Springer Verlag.

[Bolognesi and Lucidi, 1991] Bolognesi, T. and Lucidi, F. (1991). Time Process Algebras with Urgent Interactions and a Unique Powerful Binary Operator. In deBakker, J., editor, *Proceedings of the REX Workshop on Real-Time: Theory in Practice*, volume LNCS 600, pages 124–148. Springer Verlag.

[Camilleri and Winskel, 1991] Camilleri, J. and Winskel, G. (1991). CCS with priority choice. In *IEEE Symposium on Logic in Computer Science*, pages 246–255, Amsterdam, The Netherlands.

[Cleaveland and Hennessy, 1990] Cleaveland, R. and Hennessy, M. (1990). Priorities in Process Algebra. *Information and Computation*, 87:58–77.

[Groote, 1990] Groote, J. F. (1990). Transition System Specifications with Negative Premises. In Baeten, J. C. M. and Klop, J. W., editors, *CONCUR 90*, volume LNCS-458, pages 332–341. Springer Verlag.

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

[Krishnan, 1992] Krishnan, P. (1992). A Calculus of Timed Communicating Systems. *International Journal of Foundations of Computer Science*, 3(3):303–322.

[Krishnan, 1994] Krishnan, P. (1994). A Case Study in Specifying and Testing Architectural Features. *Microprocessors and Microsystems*, 18(3):123–130.

[Lin et al., 1987] Lin, K., Natarajan, S., and Liu, J. W. (1987). Imprecise results: Utilizing partial computations in real-time systems. In *IEEE Real-Time Systems Symposium*, pages 210–217.

[Milner, 1989] Milner, R. (1989). *Communication and Concurrency*. Prentice Hall International.

[Plotkin, 1981] Plotkin, G. D. (1981). A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University.

[Reiter, 1980] Reiter, R. (1980). A logic for default reasoning. *Artificial Intelligence*, 13:81–132.

[Saraswat et al., 1995] Saraswat, V., Jagadeesan, R., and Gupta, V. (1995). Default Timed Concurrent Constraint Programming. In 22^{nd} *ACM Symposium on Principles of Programming Languages*.

[Stirling, 1989] Stirling, C. (1989). An Introduction to Modal and Temporal Logics for CCS. In *Joint UK/Japan Workshop on Concurrency*, volume LNCS 491, pages 2–20.

[Verhoef, 1994] Verhoef, C. (1994). A congruence theorem for structured operational semantics with predicates and negative premises. In Jonsson, B. and Parrow, J., editors, *CONCUR 94*, volume LNCS-836, pages 433–448. Springer Verlag.

[Weaver and Germond, 1994] Weaver, D. L. and Germond, T. (1994). *The SPARC Architecture Manual: Version 9*. Sparc International.

[Yi, 1991] Yi, W. (1991). CCS+Time = An Interleaving Model for Real-Time Systems. In *ICALP -91*, volume LNCS 510, pages 217–228, Madrid, Spain. Springer Verlag.