Submitted to J. ACM. Tony
I've sent this to Misra.
Note §2.3 +
Laws numbered.
Mark

A Theory of Asynchronous Processes

Mark B. Josephs, C. A. R. Hoare and He Jifeng

Oxford University Programming Research Group

February 11, 1989

Abstract. A theory of asynchronous processes (nondeterministic data flow networks) is presented. It consists of a mathematical model and a process algebra. The intention is to provide a better theoretical underpinning to the Jackson System Development method. The model is so constructed as to be compatible with the failures model of Hoare's Communicating Sequential Processes. The process algebra describes the laws that govern a collection of CSP-like operators, convenient for constructing asynchronous process networks from their components. The operators are defined in terms of the model and so their algebraic properties can be verified. As in CSP, the laws are sufficiently complete to transform every network to a sequential form. This is important for a design method like JSD, where the resulting programs must be implemented on a traditional sequential computer.

Categories and Subject Descriptors: F.1.0. [Computation by Abstract Devices]: General; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages-*algebraic approaches to semantics, denotational semantics*
General Terms: Theory
Additional Key Words and Phrases: Asynchronous processes, data flow networks, JSD, CSP, failures model, process algebra

---

This research was undertaken on the Software Engineering Project at the Oxford University Computing Laboratory. The project is funded by the Science and Engineering Research Council of Great Britain.

Authors' current address: Oxford University Computing Laboratory, Programming Research Group, 11 Keble Road, Oxford OX1 3QD, U.K.

1 Introduction

An asynchronous process is one which communicates with its environment on buffered channels of unbounded capacity. This paper models such processes as a subset of Communicating Sequential Processes [2, 3, 7], defined by certain additional closure properties. Each asynchronous process is characterized by its failures (traces and refusal sets) and its divergences, as in CSP; but the refusal set is simplified almost out of existence. The closure properties are expressed by insisting that the traces of a process can always be reordered in certain ways.

The advantages of modelling asynchronous processes within CSP are

1. Many theoretical results concerning CSP, including some algebraic laws, remain valid for asynchronous processes.

2. Systems with a mixture of asynchronous and synchronised communication become amenable to formal reasoning.

3. The relation between the model and the reality of processor networks is well understood.

The disadvantage is that we cannot deal with traditional problems of fairness.

The objective of this study is to provide a better theoretical underpinning to the Jackson System Development method [9, 5], which is currently used to advantage in commercial and real-time programming projects. Possible applications of the theory include

1. More expressive methods for specifying requirements.

2. More powerful transformation strategies for taking the requirements down to efficient implementation.

3. The solution of certain practical problems that have come to light in the application and implementation of JSD.

This paper models the communication aspects of a JSD process network. It formalizes the description in [5] of JSD processes communicating asynchronously.

Summary

In section 2 we introduce and explain the definition of an asynchronous process, in terms of its traces, its failures and its divergences; these must satisfy certain closure properties. The relationship between this and the failures model of CSP is established. In section 3, we define for asynchronous processes a collection of CSP-like operators. In each case it is necessary to prove that these operators preserve the closure properties of their operands, so that a network constructed from asynchronous processes is itself an asynchronous process. The algebraic laws satisfied by asynchronous processes are also investigated. The relationship with previous work is summarised in the conclusion.

2 A Mathematical Model

2.1 Channels

Let In and Out name disjoint sets of input channels and output channels respectively, connecting a particular process to its environment. Typical channels will be denoted by $a, b \in In$ and $c, d \in Out$. Data consists of values denoted by variables $v, w \in V$. An input event $a.v \in In\_Com$ records the environment sending along channel a the value v to the process. An output event $c.v \in Out\_Com$ records the environment receiving on channel c the value v from the process. For an asynchronous process, each channel acts as a buffer of unbounded capacity. Thus these events performed by the environment precede or follow the actual receipt or transmission of the value by the process. These latter events are hidden from the environment. The set $Com = In\_Com \cup Out\_Com$ consists only of the communications with the environment. This is the *alphabet* of the process, in the sense of CSP.

2.2 Finite Sequences of Events

A *trace* $t \in Com^*$ is a finite sequence of events in which a particular process might participate with its environment. After engaging in t, the environment may be unable to obtain further output from the process. Usually this refusal arises because the output channels of the process are empty

and remain so, at least until the environment has supplied further input. In this case, t is called a *quiescent trace* [13, 10]. However, a refusal may also arise because the environment is always pre-empted by hidden events (infinite chatter) from receiving data from the output channels. In this case, t is called a *divergence* [3, 7]. (We say that the process is capable of diverging.) In either case, the trace t is a *failure* of the process. We will model an asynchronous process as a pair of sets, the set F of all its failures and the set D of all its divergences. Clearly, the sets must satisfy the condition $D \subseteq F$.

Consider a process which is capable of outputting forever. This means that the process can engage indefinitely in hidden events, in which the process transmits new data along its output channels. These hidden events can always pre-empt the receipt of data from those channels by the environment. We therefore conclude that a process which is capable of outputting forever is also capable of diverging.

2.3 Reordering of Traces

It is sometimes possible to reorder a trace without affecting the behaviour of a process. Consider the following three cases:

1. Suppose the environment sends to a process a value v along some input channel a and then sends it another value w along a different input channel b. This is recorded by the trace $\langle a.v, b.w \rangle$. Both values will eventually become available to the process, but as a consequence of buffering by the two channels the process will not be in a position to determine which value was sent first. Whatever the process does next would still have been possible had the environment sent the value w first. We say that $\langle b.w, a.v \rangle$ reorders $\langle a.v, b.w \rangle$ and write

   $$\langle b.w, a.v \rangle \sqsubseteq \langle a.v, b.w \rangle.$$

2. Suppose the environment receives from a process a value v along some output channel c and then receives another value w along a different output channel d. This is recorded by the trace $\langle c.v, d.w \rangle$. Buffering on the channels means that the process might have transmitted these values in either order. In particular, the environment could have received the value w first. Furthermore, the actual receipt of a message

by the environment cannot affect the behaviour of a process. We have that

$$\langle d.w, c.v \rangle \sqsubseteq \langle c.v, d.w \rangle.$$

3. Suppose the environment receives from a process a value v along some output channel c and then sends to the process a value w along an input channel a. This is recorded by the trace $\langle c.v, a.w \rangle$. The process did not require input to be available on channel a in order for it to transmit v on channel c. The behaviour of the process might therefore have been unaffected had the environment sent the value w first. We have that

$$\langle a.w, c.v \rangle \sqsubseteq \langle c.v, a.w \rangle.$$

A formal definition of $s \sqsubseteq t$ is as the smallest binary relation on finite sequences of events with the above three properties, together with

4. It is a pre-order, that is, a reflexive and transitive relation.

5. It is respected by catenation:

$$s \sqsubseteq s' \wedge t \sqsubseteq t' \Rightarrow s \frown t \sqsubseteq s' \frown t'.$$

Thus $s \sqsubseteq t$ (s reorders t) means that s is obtainable from t by moving inputs before outputs, and by changing the interleaving of communications on distinct channels. (The possibility of reordering traces was recognized by Misra [13]. Chandy [6] has also defined a similar relation between traces.) **Exercise** How might the reordering relation be extended to model *lossy* input and output channels? □

2.4 Failures

Not all sets of failures define an asynchronous process; the set must satisfy certain closure properties, defined with the aid of the reordering relation $\sqsubseteq$. For a set F to define an asynchronous process we require that a reordering of a failure is itself a failure:

$$F \neq \emptyset \wedge (s \sqsubseteq t \wedge t \in F \Rightarrow s \in F). \tag{1}$$

Further closure conditions will be given later.

Note that reordering a failure involves shifting input left and output right. The only proviso is that a communication on a given channel may not be shifted past another communication on the same channel. Thus each failure may be reordered to one in which all input occurs before all output, all communications on a given channel are contiguous, and the relative ordering among input and among output channels is arbitrary. Failures in this form resemble the history functions of Kahn [11]. They are the minimal elements of the $\sqsubseteq$ relation, and serve as a kind of normal form. Unfortunately, the Brock-Ackermann anomaly [1] shows that they do not constitute an adequate semantics for asynchronous processes. In effect, our model goes to the opposite extreme: each process is effectively defined by its maximal failures, because the other ones can be restored by the closure condition.

The following properties will be used later.

Proposition 1 *Let $S \subseteq Com^*$ and $C \subseteq Out$. If S is closed under $\sqsubseteq$, then so is $\{s \restriction \overline{C} \mid s \in S\}$, where $s \restriction \overline{C}$ removes from s all output events on all channels in C.* $\square$

From the failures of a process it is possible to derive the set $\hat{F}$ of all its traces. These are just the prefixes of its failures.

$$s \in \hat{F} \Leftrightarrow \exists t \in F.\ s \leq t.$$

To check that this definition is reasonable, we need to consider the case of a process that can output forever. Suppose such a process has engaged in any sequence s of output events. If the process remains capable of outputting indefinitely, then as we have already observed it is also capable of diverging. The trace s is a divergence, which is a failure as required. (Misra [13] chose instead to introduce *infinite* quiescent traces in modelling processes that can always output.)

From the definition we have that $\hat{F}$ is closed under both $\leq$ and $\sqsubseteq$:

Proposition 2 $\langle\rangle \in \hat{F} \wedge (s \leq t \wedge t \in \hat{F} \Rightarrow s \in \hat{F})$. $\square$

Proposition 3 $s \sqsubseteq t \wedge t \in \hat{F} \Rightarrow s \in \hat{F}$. $\square$

Two additional closure conditions are conveniently expressed in terms of the traces $\hat{F}$ of an asynchronous process. First, because input channels have unbounded capacity, a trace can always be extended by an input event:

$$s \in \hat{F} \Rightarrow s \frown \langle a.v \rangle \in \hat{F}. \tag{2}$$

As a result $In\_Com^* \subseteq \hat{F}$. Second, if a process is not capable of diverging, then it must become quiescent after a finite sequence of outputs and *no* inputs have been recorded:

$$s \in \hat{F} \Rightarrow \exists\, t \in Out\_Com^*.\ s \frown t \in F. \tag{3}$$

2.5 Divergences

The set D of divergences of a process must also satisfy certain closure properties. Divergence cannot be avoided by reordering a trace:

$$s \sqsubseteq t \wedge t \in D \Rightarrow s \in D. \tag{4}$$

A process that diverges after the environment has received data from it may diverge before the environment receives that data:

$$s \frown \langle c.v \rangle \in D \Rightarrow s \in D. \tag{5}$$

Thus, if s is a divergence, so is s restricted to its input events:

Proposition 4 $s \in D \Rightarrow s \restriction In \in D$. □

The remaining two closure conditions are necessary for consistency with the failures model of CSP [3, 7]. They also facilitate the algebraic treatment of asynchronous processes presented in section 3. A process that can engage in an unbounded amount of output before requiring input from the environment is considered capable of diverging:

$$s \in \hat{F} \wedge (\forall\, n \in \mathbb{N}.\ \exists\, t \in Out\_Com^*.\ \#t > n \wedge s \frown t \in \hat{F}) \Rightarrow s \in D. \tag{6}$$

A process that can diverge is so undesirable that we shall effectively treat its behaviour as undefined. It is modelled as the totally chaotic process, that is, it may do anything whatsoever or fail to do anything whatsoever:

$$s \in D \wedge s \leq t \Rightarrow t \in D. \tag{7}$$

A surprising (and perhaps undesirable) consequence of this condition is that any process which performs only output, and never waits for input, is modelled as the totally chaotic process.

Another way of regarding the divergence set is as follows. Misra [13] introduces the concept of an *operating region*. An operating region represents a "precondition" which the environment guarantees to meet. If the environment fails to meet the guarantee, the behaviour of the process is undefined – a particular implementation of the process might behave in an entirely arbitrary fashion. Our divergence set is therefore nothing but the complement of Misra's operating region.

In summary, we have characterized an asynchronous process by its failures F and its divergences D, which must satisfy conditions (1)-(7).

2.6 Refinement

Refinement involves replacing a process representing the specification of a system by a more deterministic process representing the implementation. Let $P_i = (F_i, D_i), i = 1, 2$. P_1 is refined by P_2 if $P_1 \supseteq P_2$, that is, $F_1 \supseteq F_2$ and $D_1 \supseteq D_2$. Refinement is a partial order with least element $Chaos = (Com^*, Com^*)$.

In section 3, the asynchronous processes denoted by the terms of our process algebra will satisfy a further condition, namely, a process can exhibit unboundedly nondeterministic behaviour only if it is also capable of diverging. That is, a trace that is not a divergence can only be extended by one of a finite set of output events:

$$s \in \widehat{F} - D \Rightarrow \{c.v \in Out\_Com \mid s \frown \langle c.v \rangle \in \widehat{F}\} \text{ is finite.}$$

When restricted to such processes, refinement is a complete partial order.

2.7 Relationship with CSP

In this section we show how every asynchronous process can be modelled as a CSP process, and every CSP process can be converted to an asynchronous process. The whole section should be omitted by a reader not familiar with CSP.

In CSP, a failure is a pair (s, X) where s is a trace and X a set of events (a refusal set). In our model of asynchronous processes the refusal set has

been simplified almost out of existence. However, for a given asynchronous process $P = (F, D)$, it is still possible to obtain the corresponding CSP process $P^{\sim}$ as follows.

The CSP process $P^{\sim}$ has the same set D of divergences. Suppose s is a divergence of P. Then (s, X) is a failure of $P^{\sim}$ for any set of events X because a process capable of diverging can refuse anything whatsoever. Suppose instead s is simply a trace of P. Certainly after s no input event can be refused, so X can only consist of output events. Moreover, if P were to become quiescent after a further sequence t of output events, then after s the environment would be able to obtain the first output event on any channel c recorded in t. More formally, (s, X) is a failure of $P^{\sim}$ if and only if the following condition is met:

$$
\begin{array}{l}
s \in D \vee \\
(s \in \widehat{F} \wedge X \subseteq \mathit{Out\_Com} \wedge \\
\quad (\exists\, t \in \mathit{Out\_Com}^{*}. \\
\qquad s \frown t \in F \wedge X \cap \{c.v \in \mathit{Out\_Com} \mid \langle v \rangle \leq (t \downarrow c)\} = \emptyset)),
\end{array}
$$

where $t \downarrow c$ is the sequence of values on channel c recorded in t.

The validity of this transformation is established by the following theorem.

Theorem 1 *If P is an asynchronous process, then $P^{\sim}$ is a CSP process. Furthermore, $Chaos^{\sim}$ is the chaotic process of CSP and $P \supseteq Q$ implies that $P^{\sim} \supseteq Q^{\sim}$ in CSP.* $\square$

In fact, $\sim$ is an isomorphism from the set of asynchronous processes to a subset of CSP. Any CSP process Q can be converted to another CSP process Q^{*} by attaching an unbounded buffer to every input and output channel. Because a chain of unbounded buffers is also an unbounded buffer, it is the case that $Q^{**} = Q^{*}$. An asynchronous process is essentially a CSP process having the property $Q^{*} = Q$:

Theorem 2 *If P is an asynchronous process, then $(P^{\sim})^{*} = P^{\sim}$. Conversely, if Q is a CSP process satisfying $Q^{*} = Q$, then there exists an unique asynchronous process P such that $P^{\sim} = Q$.* $\square$

A fuller treatment of this comparison will be given in another paper.

3 Process Algebra

In this section, we take an algebraic approach to asynchronous processes. We investigate the properties of a collection of operators from which processes can be constructed. The operators are defined in terms of the model of section 2 and preserve the closure properties of their operands. All the operators are continuous and so the meaning of a recursion $\mu X.F(X)$ is the least fixed point $\bigcap_{n\geq 0} F^n(Chaos)$ of F.

3.1 Nondeterministic Choice

The process $P_1 \sqcap P_2$ behaves nondeterministically like P_1 or P_2.

$$P_1 \sqcap P_2 = (F_1 \cup F_2, D_1 \cup D_2).$$

The nondeterministic choice operator is commutative, associative, idempotent and has *Chaos* as a zero. Also note that refinement can be defined in terms of nondeterministic choice:

$$(P \sqcap Q = P) \;\Leftrightarrow\; (P \sqsupseteq Q).$$

3.2 Stop

We have already met the process *Chaos* which diverges immediately. *Stop* is a process with a more deterministic behaviour. It represents the behaviour of a deadlocked process: it will never output or diverge. Of course, its input channels can always accept more data. Formally,

$$Stop = (In\_Com^*, \emptyset).$$

3.3 Prefix

There are two forms of prefixing, which are described below.

Input prefix

The process $a?x;P(x)$ will behave like $P(v)$ once v has been input at a. Let $P(v) = (F(v), D(v))$, for all $v \in V$. Then, $a?x;P(x) = (F', D')$, where

1. If t is a divergence of $P(v)$, for some $v \in V$, then $\langle a.v \rangle \frown t$ is a divergence of $a?x;P(x)$, and so is every reordering of it:

$$t' \in D' \Leftrightarrow \exists\, v \in V, t \in D(v).\ t' \sqsubseteq \langle a.v \rangle \frown t.$$

2. The process $a?x;P(x)$ refuses to output until the environment satisfies its demand for input on channel a. So $\langle\rangle$ is one of its failures and so is every sequence of inputs on all other channnels. Also, if t is a failure of $P(v)$, for some $v \in V$, then $\langle a.v \rangle \frown t$ is a failure of $a?x;P(x)$, and so is every reordering of it:

$$\begin{aligned} t' \in F' \Leftrightarrow\ & (t' \in In\_Com^* \wedge t' \restriction \{a\} = \langle\rangle) \vee \\ & (\exists\, v \in V, t \in F(v).\ t' \sqsubseteq \langle a.v \rangle \frown t). \end{aligned}$$

Laws

As in CSP, prefixing with an input event is distributive, that is,

$$a?x;(P(x) \sqcap Q(x)) = (a?x;P(x)) \sqcap (a?x;Q(x)). \qquad \text{[Law 1.]}$$

A process that waits for input on channel a and then deadlocks cannot be distinguished from the process *Stop*, since data is always accepted on any input channel. Thus, input prefixing has *Stop* as its zero:

$$a?x;Stop = Stop. \qquad \text{[Law 2.]}$$

On the other hand, $a?x;Chaos$ cannot diverge before input is received on channel a and so is distinct from the process *Chaos*.

If a process must wait for input on two different channels before it can continue, it does not matter which of the two channels it waits on first:

$$a?x;b?y;P(x,y) = b?y;a?x;P(x,y). \qquad \text{[Law 3.]}$$

Output prefix

The process $c!v;P$ is capable of outputting v on c before behaving like P. Because of buffering on the channel c, the visible effect of this output might be delayed until after some communications on other channels. Let $P = (F, D)$. Then, $c!v;P = (F', D')$, where

1. If t is a divergence of P, then $\langle c.v \rangle \frown t$ is a divergence of $c!v;P$, and so is every reordering of it. Further, if P can diverge without the environment ever receiving output from c, so can $c!v;P$ (i.e., the output may be buffered by the channel, and then P may diverge before the message is delivered). After divergence anything is possible, including the environment receiving *any* value from c:

$$t' \in D' \Leftrightarrow \exists t \in D.\ (t \restriction \{c\} = \langle\rangle \wedge t \leq t') \vee (t' \sqsubseteq \langle c.v \rangle \frown t).$$

2. If P may refuse to output after t, then $c!v;P$ may refuse to output after $\langle c.v \rangle \frown t$, where the prefix $\langle c.v \rangle$ shows that the environment has received the initial output v of the process from the channel c:

$$t' \in F' \Leftrightarrow t' \in D' \vee (\exists t \in F.\ t' \sqsubseteq \langle c.v \rangle \frown t).$$

Laws

As in CSP, output prefixing is distributive, that is,

$$c!v;(P \sqcap Q) = (c!v;P) \sqcap (c!v;Q). \qquad \text{[Law 4.]}$$

We have defined output prefixing in a way that allows a process that is capable of diverging to "corrupt" any values buffered by its channels, before the environment has received them. As a result, output prefixing is strict, that is, it has *Chaos* as its zero.

$$c!v;Chaos = Chaos. \qquad \text{[Law 5.]}$$

This means that every recursion should be prefixed by at least one *input* (outputs will not do).

The order in which a process transmits messages on distinct channels does not determine the order in which the messages are received by the environment:

$$c!v;d!w;P = d!w;c!v;P. \qquad \text{[Law 6.]}$$

Here are some examples of output prefixing:

X1. The process $c!0;Stop$ outputs 0 once on channel c, and then deadlocks.

X2. The process $c!0; c!0; \mathit{Stop}$ outputs 0 twice on channel c.

X3. The process $\mu X.\,(a?x; c!x; X)$ copies data from channel a to channel c.

X4. The process $\mu X.\,(c!0; X)$ is equivalent to *Chaos*.

In the last example the successive approximations to the fixed point are *Chaos*, $c!0$; *Chaos*, $c!0; c!0$; *Chaos*, etcetera. All of these are equal to *Chaos*, and so is their fixed point.

3.4 Guarded Choice

The guarded choice operator is similar to the ALT construct of Occam [8]. A *guarded process* G is of the form

$$\mathit{skip} \rightarrow P \qquad \text{or} \qquad a?x \rightarrow P(x).$$

If S is a finite set of guarded processes, a *guarded choice* (written $[S]$) is a process that selects one of the set in accordance with the following rules. If input is available on a channel a, then any process guarded on that channel may be selected. Whether or not input is available, any *skip*-guarded process may be selected. If no input is ever available on any of the input channels acting as guards, then selection of a *skip*-guarded process (if there is one) cannot be indefinitely delayed. Thus the *skip* guard acts like a hidden event in CSP, or a τ event in CCS [12]. (Note that we could if we so wished define an output-guarded process $c!v \rightarrow P$ so that it was equivalent to the *skip*-guarded process $\mathit{skip} \rightarrow c!v; P$.)

With each guarded process $G \in S$, it is convenient to associate a set $\mathcal{F}(G)$ of failures and a set $\mathcal{D}(G)$ of divergences. If $P = (F, D)$, then

$$\mathcal{F}(\mathit{skip} \rightarrow P) = F$$
$$\mathcal{D}(\mathit{skip} \rightarrow P) = D.$$

If $P(v) = (F(v), D(v))$, for all $v \in V$, then

$$t' \in \mathcal{F}(a?x \rightarrow P(x)) \;\Leftrightarrow\; \exists\, v \in V, t \in F(v).\; t' \sqsubseteq \langle a.v \rangle \frown t$$
$$t' \in \mathcal{D}(a?x \rightarrow P(x)) \;\Leftrightarrow\; \exists\, v \in V, t \in D(v).\; t' \sqsubseteq \langle a.v \rangle \frown t.$$

We shall also write $\mathit{skipfree}(S)$ to mean that S contains no *skip*-guarded processes, and $\mathit{chan}(S)$ for the set of input channels on which processes in S are guarded. Then, $[S] = (F', D')$ where

1. If any guarded process of S can diverge, $[S]$ can diverge in the same circumstances:

$$D' = \bigcup_{G \in S} \mathcal{D}(G).$$

2. Since any *skip*-guarded process of S may be selected nondeterministically, all its failures are included in the failures of $[S]$. If there are no such processes, $[S]$ will refuse to output until at least one input is available on at least one of the channels in $chan(S)$. After input on one such channel has occurred, the failures of $[S]$ will include those of any of the processes guarded on that channel:

$$t \in F' \Leftrightarrow (t \in In\_Com^* \wedge skipfree(S) \wedge t \upharpoonright chan(S) = \langle\rangle) \vee (\exists\, G \in S.\ t \in \mathcal{F}(G)).$$

In future, the symbol $\Box$ will be used to separate the guarded processes in S whenever they are listed. $\rightarrow$ binds tighter than $\Box$. So, for example,

$$[G \Box S] = [\{G\} \cup S] \qquad \text{and} \qquad [G_1 \Box G_2 \Box G_3] = [\{G_1, G_2, G_3\}].$$

X5. The process that merges input from channels a and b onto channel c is

$$\mu X.[a?x \rightarrow c!x; X \Box b?y \rightarrow c!y; X].$$

As in CSP, this is a fair merge; if the process supplying data to one of the input channels indefinitely delays doing so, the merge process will successfully copy from the other input channel to c for as long as required.

Laws

Guarded choice distributes through nondeterministic choice:

$$[skip \rightarrow (P \sqcap Q) \Box S] = [skip \rightarrow P \Box S] \sqcap [skip \rightarrow Q \Box S] \qquad \text{[Law 7.]}$$

$$[a?x \rightarrow (P(x) \sqcap Q(x)) \Box S] = [a?x \rightarrow P(x) \Box S] \sqcap [a?x \rightarrow Q(x) \Box S]. \qquad \text{[Law 8.]}$$

If there are no guarded processes to choose between, the process deadlocks:

$$[\,] = Stop. \qquad \text{[Law 9.]}$$

Choice with only one alternative is no real choice at all:

$$[skip \rightarrow P] = P \qquad \text{[Law 10.]}$$

$$[a?x \rightarrow P(x)] = a?x; P(x). \qquad \text{[Law 11.]}$$

A *skip*-guarded process may be selected nondeterministically at any time:

$$[skip \rightarrow P \,\Box\, S] \supseteq P. \qquad \text{[Law 12.]}$$

In particular, $[skip \rightarrow Chaos \,\Box\, S] = Chaos$. The offer of a guarded process that deadlocks can be withdrawn in an implementation, because the resulting process will only be an improvement:

$$[skip \rightarrow Stop \,\Box\, S] \supseteq [S] \qquad \text{[Law 13.]}$$

$$[a?x \rightarrow Stop \,\Box\, S] \supseteq [S]. \qquad \text{[Law 14.]}$$

The choice between *skip*-guarded processes is nondeterministic:

$$[skip \rightarrow P \,\Box\, skip \rightarrow Q \,\Box\, S] = [skip \rightarrow (P \sqcap Q) \,\Box\, S]. \qquad \text{[Law 15.]}$$

In particular,

$$[skip \rightarrow P \,\Box\, skip \rightarrow Q] = P \sqcap Q. \qquad \text{[Law 16.]}$$

These laws permit every guarded choice to be reduced to one with at most one *skip*-guarded process.

Nondeterminism also arises when the same input channel is used to guard two processes:

$$[a?x \rightarrow P(x) \,\Box\, a?y \rightarrow Q(y) \,\Box\, S] = [a?z \rightarrow (P(z) \sqcap Q(z)) \,\Box\, S]. \qquad \text{[Law 17.]}$$

The following three rather complicated laws are also valid in Occam, as described in [15]. The offer of a process guarded on an input channel can be postponed until after a *skip*-guarded process has been selected:

$$[skip \rightarrow [a?x \rightarrow P(x) \,\Box\, S_1] \,\Box\, a?y \rightarrow Q(y) \,\Box\, S_2] \qquad \text{[Law 18.]}$$
$$= [skip \rightarrow [a?z \rightarrow (P(z) \sqcap Q(z)) \,\Box\, S_1] \,\Box\, S_2].$$

A deeply nested nondeterministic choice can be unnested by the law

$$[skip \to [skip \to P \,\Box\, S_1] \,\Box\, S_2] = [skip \to P \,\Box\, (S_1 \cup S_2)]. \qquad \text{[Law 19.]}$$

The last of the three laws expresses a convexity property:

$$[skip \to [S_1] \,\Box\, skip \to [S_1 \cup S_2] \,\Box\, S_3] = [skip \to [S_1] \,\Box\, (S_2 \cup S_3)]. \qquad \text{[Law 20.]}$$

In general we cannot substitute the guarded process $a?x \to b?y; P(x,y)$ for $b?y \to a?x; P(x,y)$ in a guarded choice. For example, the process

$$[skip \to c!0; Stop \,\Box\, a?x \to b?y; Stop]$$

must eventually output 0 on channel c if no input is available on channel a. However, it only requires input to be available on channel b for the process

$$[skip \to c!0; Stop \,\Box\, b?y \to a?x; Stop]$$

to be capable of deadlocking. Nevertheless, we do have the following two laws. Suppose S consists only of guarded processes of the form

$$skip \to a?x; P(x) \qquad \text{and} \qquad b?y \to a?x; Q(x,y),$$

so that $a \notin chan(S)$. Define $S(x)$ to consist of the same guarded processes, stripped of their input prefixes on channel a. Thus, each guarded process in $S(x)$ is of the form

$$skip \to P(x) \qquad \text{and} \qquad b?y \to Q(x,y).$$

Under these circumstances we may distribute input prefixing through guarded choice:

$$a?x; [S(x)] = [S]. \qquad \text{[Law 21.]}$$

Our final law for guarded choice allows a process $P(x,y)$, guarded first on channel a and then on channel b, to be re-offered (guarded on channel a) after a process guarded on channel b has been selected:

$$\begin{aligned} & [a?x \to [b?y \to P(x,y) \,\Box\, S_1(x)] \qquad \text{[Law 22.]} \\ & \Box\, b?z \to [skip \to Q(z) \,\Box\, S_2(z)] \\ & \Box\, S_3] \\ = \; & [a?x \to [b?y \to P(x,y) \,\Box\, S_1(x)] \\ & \Box\, b?z \to [a?x \to P(x,z) \,\Box\, skip \to Q(z) \,\Box\, S_2(z)] \\ & \Box\, S_3]. \end{aligned}$$

The processes *Chaos*, $c!v;P$ and $[S]$ are said to be in *pre-normal form* [15]. (Observe that *Stop*, input prefixing and nondeterministic choice can be regarded as special cases of guarded choice, by Laws 9, 11 and 16.) The remaining operators that we shall introduce satisfy enough algebraic laws that any process constructed from them can be transformed into pre-normal form.

3.5 After

The process $P/a.v$ behaves like P after v has been communicated by its environment on input channel a. The value v remains buffered by the channel until P is ready to use it. Since a has unbounded capacity it can never refuse input and so $P/a.v$ is always defined (which in CSP is not always so).

Let $P = (F, D)$. Then, $P/a.v = (F', D')$ where

$$t \in D' \Leftrightarrow \langle a.v \rangle \frown t \in D.$$
$$t \in F' \Leftrightarrow \langle a.v \rangle \frown t \in F.$$

Laws

After is distributive and has both *Chaos* and *Stop* as fixed points. Because distinct channels buffer their data independent of each other,

$$(P/a.v)/b.w = (P/b.w)/a.v. \qquad \text{[Law 23.]}$$

The value made available to a process which is waiting to input on channel a is just the one buffered by the after operation:

$$(a?x;P(x))/a.v = P(v). \qquad \text{[Law 24.]}$$

After distributes through prefixing on channels other than a:

$$(b?x;P(x))/a.v = b?x;(P(x)/a.v) \qquad \text{[Law 25.]}$$

$$(c!w;P)/a.v = c!w;(P/a.v). \qquad \text{[Law 26.]}$$

After distributes through guarded choice, so that

$$[S]/a.v = [S']. \qquad \text{[Law 27.]}$$

Here S' is formed by substituting for each guarded process $G \in S$ the guarded process $G/a.v$ defined by

$$(skip \rightarrow P)/a.v = skip \rightarrow (P/a.v)$$
$$(a?x \rightarrow P(x))/a.v = skip \rightarrow P(v)$$
$$(b?x \rightarrow P(x))/a.v = b?x \rightarrow (P(x)/a.v).$$

Finally, the following law allows us to expand the set of guarded processes in a guarded choice:

$$[S] = [a?x \rightarrow ([S]/a?x) \,\Box\, S]. \qquad \text{[Law 28.]}$$

Future laws are greatly simplified by defining $P/a.v$ to be equal to P when a is not an input channel of P. This represents the fact that events that are not in the alphabet of a process are ignored by that process.

Exercise Consider the process $a?x * P(x)$ defined by

$$a?x * P(x) = a?x; Q(x)$$

where $Q(v) = [skip \rightarrow (P(v)/a.v) \,\Box\, a?x \rightarrow Q(x)]$. Prove the following laws from those stated so far:

1. $a?x * a?y * P(y) = a?y * P(y)$.
2. $a?x; b?y * P(x,y) = b?y * a?x; P(x,y)$.
3. $a?x * b?y * P(x,y) = b?y * a?x * P(x,y)$.

The third law is surprisingly difficult to prove. The interest of the definition of $*$ is that it models the important state-vector inspection of JSD. $\Box$

3.6 Parallel Composition

The operators defined so far assume that all operands are processes with the same alphabet. As in CSP, parallel composition is an operation between processes possibly with different alphabets, whose union is the alphabet of its result.

We define a rather complex form of parallel composition $P_1 \parallel P_2$, under the constraint that the output channels of P_1 and P_2 are disjoint. The

components P_1 and P_2 may communicate with each other along any channel c which is an output channel of one and an input channel of the other. In $P_1 \parallel P_2$, the output on c is retained so that c may be connected to yet other input channels. (Hiding of output, as in CSP, is defined as a separate operation described in the next section.) If P_1 and P_2 share an input channel a, then both P_1 and P_2 will input *all* messages sent to $P_1 \parallel P_2$ on a. Input by P_1 and P_2 on a does not have to be synchronised: an implementation must copy all messages sent on a into two buffers, so that they can be consumed at different speeds by P_1 and P_2.

Let $P_i = (In_i, Out_i, F_i, D_i)$, $In_i \cap Out_i = \emptyset$, $i = 1, 2$, with $Out_1 \cap Out_2 = \emptyset$. Let T be the set of those traces consistent with both P_1 and P_2, that is,

$$s \in T \Leftrightarrow s \restriction (In_1 \cup Out_1) \in \widehat{F}_1 \wedge s \restriction (In_2 \cup Out_2) \in \widehat{F}_2.$$

Then, $P_1 \parallel P_2 = ((In_1 - Out_2) \cup (In_2 - Out_1), Out_1 \cup Out_2, F', D')$, where

1. If s is consistent with both P_1 and P_2, and one of them may diverge or they can continue to output indefinitely, then s is a divergence of $P_1 \parallel P_2$, and so is every reordering of it. After divergence anything is possible.

$$\begin{aligned} t' \in D' \Leftrightarrow \exists\, s' \leq t', s \in T.\ & s' \sqsubseteq s \wedge \\ & (s \restriction (In_1 \cup Out_1) \in D_1 \vee s \restriction (In_2 \cup Out_2) \in D_2 \vee \\ & (\forall\, n \in \mathbb{N}.\ \exists\, u \in (Out_1\_Com \cup Out_2\_Com)^*. \\ & \qquad \#u > n \wedge s \frown u \in T)). \end{aligned}$$

2. If t is consistent with both P_1 and P_2 refusing to output, then it is a failure of $P_1 \parallel P_2$, and so is every reordering of it.

$$\begin{aligned} t' \in F' \Leftrightarrow\ & t' \in D' \vee \\ & (\exists\, t \in T.\ t' \sqsubseteq t \wedge \\ & \quad t \restriction (In_1 \cup Out_1) \in F_1 \wedge t \restriction (In_2 \cup Out_2) \in F_2). \end{aligned}$$

Laws

We shall not make explicit the alphabets of processes in our laws; for example the law

$$P \parallel Chaos = Chaos \qquad \text{[Law 29.]}$$

states that *Chaos* is a zero, even though the two occurrences of *Chaos* may be associated with different alphabets.

Parallel composition is commutative, associative and distributive. The following law states that it is immaterial as to whether two processes in parallel wait independently or together for input to become available on a shared channel.

$$(a?x;P(x)) \parallel (a?y;Q(y)) = a?z;(P(z) \parallel Q(z)). \qquad \text{[Law 30.]}$$

If the two processes are waiting for input on different channels a and b (neither channel being an output channel of the other process), either input may take place. Subsequently, the corresponding process uses the input value, while the other saves it up (by the after operation) for later use. Our convention that $P/a.v = P$ when a is not in the alphabet of P means that we do not have to distinguish this case.

$$\begin{aligned}&(a?x;P(x)) \parallel (b?y;Q(y)) \qquad \text{[Law 31.]}\\ &\quad = [a?x \rightarrow (P(x) \parallel (b?y;(Q(y)/a.x)))\\ &\qquad \Box\, b?y \rightarrow ((a?x;(P(x)/b.y)) \parallel Q(y))]\end{aligned}$$

a, b not output channels.

If one process is waiting for input from the environment on channel a and the other process is waiting for input from the first process on channel c, then only input on channel a may take place.

$$\begin{aligned}&(a?x;P(x)) \parallel (c?y;Q(y)) \qquad \text{[Law 32.]}\\ &\quad = a?x;(P(x) \parallel (c?y;(Q(y)/a.x)))\end{aligned}$$

c an output channel.

If each process is waiting for input from the other, we have deadlock.

$$(c?x;P(x)) \parallel (d?y;Q(y)) = Stop \qquad \text{[Law 33.]}$$

c, d output channels.

If one of the processes is prepared to output, this may happen straight away. The message is buffered by the other process for future consumption if it is in the alphabet of the process and is ignored otherwise.

$$(c!v;P) \parallel Q = c!v;(P \parallel (Q/c.v)). \qquad \text{[Law 34.]}$$

Finally, we consider the general case in which each process is a guarded choice:

$$[S_1] \parallel [S_2] = [S], \qquad \text{[Law 35.]}$$

defined as follows. Consider those guarded processes in $S_1 \cup S_2$ that are not guarded on a channel which receives its data from an output channel of the other process. Corresponding to each of them is a guarded process in S, namely

1. Corresponding to $skip \rightarrow P \in S_1$ is

   $$skip \rightarrow (P \parallel [S_2]).$$

2. Corresponding to $skip \rightarrow P \in S_2$ is

   $$skip \rightarrow ([S_1] \parallel P).$$

3. Corresponding to $a?x \rightarrow P(x) \in S_1$ is

   $$a?x \rightarrow (P(x) \parallel ([S_2]/a.x)).$$

4. Corresponding to $a?x \rightarrow P(x) \in S_2$ is

   $$a?x \rightarrow (([S_1]/a.x) \parallel P(x)).$$

(The expansion theorems of CSP and CCS are equally complicated.)

A parallel composition in which each process sends messages to the other can give rise to divergence. In the following example, the value 0 is sent back and forth between the two processes. Although each process is individually free from divergence, when composed in parallel they diverge, because they are always willing to output.

X6. The process $(c!0; \mu X.(a?x; c!x; X)) \parallel \mu Y.(c?y; a!y; Y)$ is equivalent to *Chaos*.

Parallel composition can be used to describe feedback loops, as follows.

X7. The process $P \parallel \mu X.(c?x; a!x; X)$ feeds output on channel c of P back as input to channel a of P.

3.7 Concealment of Output

A trace of the behaviour of a process with concealed output is obtained by simply removing the record of all concealed events. This leads to the following simple definition. Let $P = (In, Out, F, D)$ and $C \subseteq Out$. The process formed by concealing output from channels in C is defined by

$$P \backslash C = (In, Out - C, F', D')$$

where

$$t' \in D' \Leftrightarrow \exists t \in D.\ t' = t \upharpoonright \overline{C}$$
$$t' \in F' \Leftrightarrow \exists t \in F.\ t' = t \upharpoonright \overline{C}.$$

Laws

Concealment is distributive and has both *Stop* and *Chaos* as fixed points. Concealing nothing has no effect:

$$P \backslash \emptyset = P. \qquad \text{[Law 36.]}$$

The order in which channels are concealed is irrelevant: if $C_1 \cap C_2 = \emptyset$, then

$$(P \backslash C_1) \backslash C_2 = P \backslash (C_1 \cup C_2). \qquad \text{[Law 37.]}$$

Input is never concealed:

$$(a?x; P(x)) \backslash C = a?x; (P(x) \backslash C) \qquad \text{[Law 38.]}$$

and, more generally,

$$[S] \backslash C = [S'] \qquad \text{[Law 39.]}$$

where each guarded process in S

$$skip \rightarrow P \qquad \text{or} \qquad a?x \rightarrow P(x)$$

has a corresponding guarded process in S'

$$skip \rightarrow (P \backslash C) \qquad \text{or} \qquad a?x \rightarrow (P(x) \backslash C).$$

Output on a concealed channel is removed:

$$(c!v;P)\backslash C = P\backslash C \qquad \text{if } c \in C. \qquad \text{[Law 40.]}$$

Output on a non-concealed channel is unaffected:

$$(d!v;P)\backslash C = d!v;(P\backslash C) \qquad \text{if } d \notin C. \qquad \text{[Law 41.]}$$

There is no need to define concealment of input channels. You would never want to do so, unless that input channel has been connected to the output channel of some other process; and by the definition of parallel composition, the channel remains only as an output channel of the composite process.

Exercise (Brock-Ackermann anomaly) Consider the two processes $P_i, i = 1, 2$, defined by

$$P_i \;=\; ((Merge/b.5/b.5) \parallel Copy_i)\backslash\{c\},$$

where

$$Merge = [a?x \rightarrow c!x; Merge \,\Box\, b?y \rightarrow c!y; Merge]$$
$$Copy_1 = c?x; d!x; c?y; d!y; Stop$$
$$Copy_2 = c?x; c?y; d!x; d!y; Stop.$$

Show that

$$P_1 = [d!5 \rightarrow [d!5 \rightarrow Stop \,\Box\, a?x \rightarrow d!x; Stop]$$
$$\Box\, a?x \rightarrow d!x; [d!5 \rightarrow Stop \,\Box\, a?y \rightarrow d!y; Stop]$$
$$P_2 = [d!5 \rightarrow d!5; Stop$$
$$\Box\, a?x \rightarrow [d!5 \rightarrow d!x; Stop \,\Box\, d!x \rightarrow d!5; Stop \,\Box\, a?y \rightarrow d!x; d!y; Stop].$$

It should be evident that P_2 is strictly more deterministic than P_1, even though the processes have identical input-output histories. In a naive theory which identifies the two processes, an anomaly arises when each value output from d is incremented by one and supplied as input to a. P_1 permits a 5 followed by a 6 to be output from d, whereas P_2 does not. □

4 Conclusion

A mathematical model has been provided for asynchronous processes. This is based upon the concept of a failure. Misra [13] has also considered the same kind of processes and has stated less formally many of our closure properties. Thus, section 2 of this paper can be regarded as a formalization of Misra's work. Misra's ideas have also been formalized in a different way by Jonsson [10]. Misra and Jonsson introduce *infinite* quiescent traces. This is a complication that we have been able to avoid; we observed that a process that can produce output forever must also be capable of diverging. We have treated divergence in a way that is compatible with the failures model of CSP [3]. While a process remains within its operating region [13] it should not in any case be capable of diverging.

Previous theories of asynchronous processes, such as those of Park [14], Broy [4] and Staples and Nguyen [16], have been based on Kahn's model [11] of deterministic processes. The Brock-Ackermann anomaly [1] has made these theories rather complicated. However, Misra [13] and Jonsson [10] observed that a simple solution to the anomaly is to take into consideration the traces of a process.

A second contribution of this paper has been the exploration of an algebra for asynchronous processes. We have considered a number of useful CSP-like operators by which processes can be constructed from smaller components. Associated with these operators is a rich set of algebraic laws. The algebra provides a convenient way of specifying and transforming networks of processes. Indeed, sufficient laws have been provided to permit the transformation of any network of processes into a single sequential process. Because all our operators have been defined in terms of the model, it is possible to verify that they do in fact possess their stated algebraic properties.

Acknowledgements

We are most grateful to Michael Jackson and John Cameron for helping us to understand JSD. Ralph Back and Bengt Jonsson were kind enough to introduce us to relevant literature. Thanks are due to Bengt and also to Geoff Barrett for their comments on this paper.

References

[1] Brock, J.D., and Ackermann, W.B. Scenarios: a model of non-determinate computation. *Lect. Notes in Comp. Sci. 107* (1981), 252-259.

[2] Brookes, S.D., Hoare, C.A.R., and Roscoe, A.W. A theory of communicating sequential processes. *J. ACM 31,* 7 (1984), 560-599.

[3] Brookes, S.D., and Roscoe, A.W. An improved failures model for communicating sequential processes. *Lect. Notes in Comp. Sci. 197* (1984), 281-305.

[4] Broy, M. Semantics of finite and infinite networks of concurrent communicating agents. *Distributed Computing 2* (1987), 13-31.

[5] Cameron, J.R. An overview of JSD. *IEEE Trans. Soft. Eng. 12,* 2 (1986), 222-240.

[6] Chandy, K.M. Theorems on computations of distributed systems. Caltech-CS-TR-88-6 (1988).

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

[8] INMOS Ltd. *Occam 2 Reference Manual.* Prentice-Hall International, London (1988).

[9] Jackson, M.A. *System Development.* Prentice-Hall International, London (1983).

[10] Jonsson, B. A model and proof system for asynchronous processes. *Proc. 4th ACM Symp. on Principles of Distributed Computing* (1985), 49-58.

[11] Kahn, G. The semantics of a simple language for parallel programming. *Information Processing 74: Proc. IFIP Congress,* North-Holland, New York (1974), 471-475.

[12] Milner, R. A calculus of communicating systems. *Lect. Notes in Comp. Sci. 92* (1980).

[13] Misra, J. Reasoning about networks of communicating processes. *Unpublished.* Presented at INRIA Advanced Nato Study Institute on Logics and Models for Verification and Specification of Concurrent Systems, La Colle-sur-Loupe, France (1984).

[14] Park, D. The "fairness" problem and nondeterministic computing networks. *Foundations of Computer Science IV Part 2, Amsterdam, Math. Centre Tracts 159* (1982), 133-161.

[15] Roscoe, A.W., and Hoare, C.A.R. The laws of occam programming. *Theor. Comp. Sci. 60,* 2 (1988), 177-229.

[16] Staples, J., and Nguyen, V.N. A fixpoint semantics for nondeterministic data flow, *J. ACM 32,* 2 (1985), 411-444.