A Theory of Communicating Sequential Processes

S. D. BROOKES

Carnegie-Mellon University, Pittsburgh, Pennsylvania

AND

C. A. R. HOARE AND A. W. ROSCOE

Oxford University Programming Research Group, Oxford, England

Abstract. A mathematical model for communicating sequential processes is given, and a number of its interesting and useful properties are stated and proved. The possibilities of nondeterminism are fully taken into account.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*; *syntax*; D.3.2 [**Programming Languages**]: Language Classifications; D.3.3 [**Programming Languages**]: Language Constructs—*concurrent programming structures*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*denotational semantics*

General Terms: Theory

Additional Key Words and Phrases: Communicating sequential processes, synchrony, asynchrony, nondeterminism, parallelism, deadlock, safety, liveness

1. *Introduction*

In the last decade there has been a remarkable growth in general understanding of the design and definition of computer programming languages. This understanding has been based upon a recognition that the text of each program expressed in the language should be given a mathematically defined meaning or denotation, in the same way as any other notational system of logic or mathematics. For a conventional sequential programming language, the simplest mathematical domain suitable for this purpose is the space of partial functions that maps from an abstract machine state before execution of a command to the state of the machine afterward. For a programming language with jumps, the appropriate mathematical domain is slightly more complicated, involving continuations. For a programming language in which subprograms are themselves assignable components of the abstract machine state, the appropriate reflexive domain of continuous functions has been

Part of this research was supported by grants from the Science and Engineering Research Council of Great Britain.

Authors' addresses: S. D. Brookes, Department of Computer Science, Carnegie-Mellon University, Schenley Park, Pittsburgh, PA 15213. C. A. R. Hoare and A. W. Roscoe, Oxford University Programming Research Group, 8-11 Keble Road, Oxford OX1 3QD England.

discovered by Scott [26]. His techniques have been applied to a variety of familiar and novel programming languages [18, 28]. The concept on which all these developments rest is the familiar mathematical concept of a partial function, and its familiarity has undoubtedly contributed to the widespread acceptance and success of the approach. However, there are two features of certain new experimental languages involving concurrency that are not so simply treated as mathematical functions.

(1) In the parallel execution of commands of a program, the effect of each command can no longer be modeled as a function from an initial state to a final state of an abstract machine; it is also necessary to model the continuing interactions of a command with its environment.

(2) In the execution of parallel programs, it is desirable to abstract from the relative rates of progress of the commands being executed in parallel. In general, this will give rise to nondeterminacy in the behavior and outcome of the program.

Both these problems arise in acute form in the treatment of a language like that of Communicating Sequential Processes (CSP) [12].

It is the purpose of this paper to construct a mathematical domain that should play the same role in defining the semantics of communicating processes as the domain of partial functions does for sequential and deterministic programming languages. Every effort has been made to keep the domain simple, and to ensure that the necessary operators over objects in the domain have elegant and intuitively valid properties. This paper is a much expanded and improved version of an Oxford University technical report with the same name [14].

The second section of the paper contains a definition of the required domain of processes. Following the lead of [9], [19], and [20] we first introduce the concept of a *transition*, which is a ternary relation between

(1) the *initial* state of a process,
(2) a sequence describing its interactions with its environment *during* its execution, and
(3) a possible state of the process *after* those interactions.

Next we note that the internal states of a process are not observable by its environment. We therefore define the concept of an *observation* of a process, which is a finitely describable experiment to which a process can be subjected. We then postulate that two processes are identical if they cannot be distinguished by any such finite observation. This reasoning leads directly to the construction of our proposed mathematical space of processes.

The next section shows that this space has the usual ordering properties required of a semantic domain. The relevant partial ordering is simply set inclusion in the *reverse* of the normal direction, so that one process is an approximation to another if it is less deterministic. This partial order on the space of all processes, which is shown to be complete, is similar in spirit to the usual Smyth ordering [27].

The important consequence of this is that every set of recursive equations in process-valued variables has a least solution; and this permits the use of recursion both in a programming language and in its formal definition.

The fourth section defines a wide range of operators over the domain of processes; these include sequential composition, conditional composition, two forms of parallel composition, and (perhaps most crucial of all) a concealment operator, which permits abstraction from the details of internal communications between processes connected in a network. These operators enjoy a number of elegant and useful algebraic properties. We hope that this range of defined operators will be a sufficient basis in terms of which to define all other operations required in the semantics of a parallel programming language, without any further concern for the details of the underlying mathematical model. Thus these operators should play the same role as the basic operators defined by Scott for the LAMBDA calculus, which shield the practicing user from the complexities of the underlying domain.

The fifth section gives some examples of the application of the model, by showing that it can be used to define some complex but useful programming language constructs, and to describe some simple but interesting parallel algorithms.

The sixth section contains a discussion of related work and future directions for research.

The seventh section discusses the prospects for the development of formal methods in increasing reliability of implementation and use of a programming language that includes parallelism.

The final section is an appendix that contains proofs of some of the paper's more interesting results. In addition it describes some techniques that can be used to prove the correctness of processes defined within the model.

2. *Definition of a Process*

The ultimate unit in the behavior of a process is an *event*. Events are regarded as instantaneous; if we wish to represent an activity with duration, we must introduce two events to represent its start and finish so that other events can occur between them. We shall not be interested in the length of the time interval that separates events, but only in the relative order in which they occur. We let A stand for the set of all events with which we shall be concerned. The behavior of a process up to some moment in time can be recorded as the sequence of all events in which it has participated; that is known as a *trace*. We postulate that a process can only perform a finite number of events in any finite time, and thus all traces have finite length. The set of all possible traces is denoted by A^*.

Let s be a trace and let P and Q be processes. A *transition* is a proposition

$$P \xrightarrow{s} Q,$$

which means that s is a possible trace of the behavior of P up to some moment in time, and that the subsequent behavior of P may be the same as that of Q. Thus if t is a possible trace of Q, after which it may behave like R, then clearly st (s followed by t) is also a possible trace of P, after which it can also behave like R.

This fact is formalized as a general *law*:

$$P \xrightarrow{s} Q \mathbin{\&} Q \xrightarrow{t} R \Rightarrow P \xrightarrow{st} R. \tag{L1}$$

Conversely, if $P \xrightarrow{st} R$, then there must exist some intermediate process Q that behaves exactly like P would behave after doing s but before starting on t. This is expressed in the law

$$P \xrightarrow{st} R \Rightarrow \exists Q.P \xrightarrow{s} Q \mathbin{\&} Q \xrightarrow{t} R. \tag{L2}$$

The empty trace $\langle\ \rangle$ is the sequence with no events. It describes the behavior of a process that has not yet engaged in any externally recordable event. We adopt the convention that after doing nothing a process may remain unchanged. Moreover, if before performing any visible event a process remains unchanged, we can regard all intermediate stages that it may have gone through as equivalent.

$$P \xrightarrow{\langle\rangle} Q \ \&\ Q \xrightarrow{\langle\rangle} P \Leftrightarrow P = Q. \tag{L3}$$

If $Q \neq P$, then the possibility of the transition $P \xrightarrow{\langle\rangle} Q$ means that P may make internal progress, which cannot be observed from outside, after which it can behave like Q rather than P. Since, in general, a process is nondeterministic, its internal progress will require the making of arbitrary choices, which are wholly uncontrollable and invisible from outside. Such a choice can only reduce the range of possible future behaviors of P, by excluding behaviors that would have remained possible if some alternative choice had been made. Thus the effect of a nondeterministic choice made by a process will be to constrain the ability of the process to perform events on the next and subsequent steps.

The *initials* of a process P are those events in which it can engage on the very first step; they are defined as

$$\text{initials}(P) = \{a \in A \mid \exists Q.P \xrightarrow{\langle a\rangle} Q\},$$

where $\langle a\rangle$ is the sequence containing the single event a. The choice of which of these events, if any, will actually occur will depend (at least in part) on the environment in which the process is placed. Let X be the set of events that are possible for that environment. Then the event that actually occurs must be in the intersection ($X \cap \text{initials}(P)$). If this intersection is empty, then nothing further can happen: the process and its environment remain locked forever in deadly embrace [7]. Unfortunately, if P is nondeterministic, deadly embrace is still possible even when the intersection is nonempty. This occurs when P can progress invisibly to become Q, and the intersection ($X \cap$ initials (Q)) is empty. In such a case, we say that X is a possible *refusal* of P, and that P can *refuse* X.

We want to be able to distinguish between processes by observing their behavior in *finite* environments. It will be possible to distinguish between P and Q if and only if there is a finite sequence s of events possible for P but not for Q (or vice versa), or there is a sequence s that is possible for both and a finite set X of events such that P can refuse X after doing s but Q cannot (or vice versa). We adopt this view of distinguishability because we consider a *realistic* environment to be one that is at any time capable of performing only a finite number of events. Bearing these remarks in mind, we define the set of P's refusals as

$$\text{refusals}(P) = \{X \mid X \text{ finite } \& \ \exists Q.P \xrightarrow{\langle\rangle} Q \ \&\ X \cap \text{initials}(Q) = \varnothing\}.$$

From this definition it follows that

(1) $\varnothing \in \text{refusals}(P)$;
(2) if $Y \in \text{refusals}(P)$ and $X \subseteq Y$, then $X \in \text{refusals}(P)$;
(3) if $X \in \text{refusals}(P)$ and Y is a finite subset of $(A - \text{initials}(P))$, then $(X \cup Y) \in \text{refusals}(P)$.

$(A - \text{initials}(P))$ is the set of events that P *cannot* perform. The third theorem above states that P can refuse these events, together with any other set of events that it can refuse.

A *trace* of a process is a sequence of events in which it may engage up to some moment in time. The set of all such traces is defined:

$$\text{traces}(P) = \{s \in A^* \mid \exists Q.P \xrightarrow{s} Q\}.$$

From this definition it follows that

$$\langle\,\rangle \in \text{traces}(P),$$
$$st \in \text{traces}(P) \Rightarrow s \in \text{traces}(P).$$

The second theorem states that any prefix (initial subsequence) of a trace of P is also a trace of P. We shall write $s \leq u$ when s is a prefix of u.

If s is a trace of P, and if, after engaging in the events of s, P can refuse the finite set X, we say that the pair (s, X) is a *failure* of the process P. The set of all such failures is defined:

$$\text{failures}(P) = \{(s, X) \mid \exists Q. P \xrightarrow{s} Q \;\&\; X \in \text{refusals}(Q)\}.$$

Since $\varnothing \in \text{refusals}(Q)$, it follows that s is a trace of P if and only if $(s, \varnothing)$ is a failure of P. From this definition it follows that the set $F = \text{failures}(P)$ has the properties:

(P1) $(s, X) \in F \Rightarrow s \in A^* \;\&\; X \subseteq A \;\&\; X$ finite,
(P2) $(\langle\,\rangle, \varnothing) \in F$,
(P3) $(st, \varnothing) \in F \Rightarrow (s, \varnothing) \in F$,
(P4) $X \subseteq Y \;\&\; (s, Y) \in F \Rightarrow (s, X) \in F$,
(P5) $(s, X) \in F \;\&\; (s\langle c\rangle, \varnothing) \notin F \Rightarrow (s, X \cup \{c\}) \in F$.

Note that (P5) implies that whenever (s, X) is a failure of P and Y is a finite set of events such that $s\langle c\rangle$ is not a trace of P, for all $c \in Y$, then $(s, X \cup Y)$ is also a failure of P. This can be interpreted as saying that impossible events can always be refused.

The failures of a process represent possible externally observable aspects of its behavior. The fact that $(s, X) \in \text{failures}(P)$ means that it is possible for P to do s and then refuse to do any more, in spite of the fact that its environment allows any of the events of X. Our next postulate states that there exists a process corresponding to any possible set of failures.

If F satisfies the five properties of the previous paragraph, then there exists a process P such that $\text{failures}(P) = F$. (L4)

Finally, we postulate that the failures of a process are the only externally observable aspects of its behavior. Thus two processes that fail in exactly the same circumstances are indistinguishable by external observation. Since we deliberately choose to ignore the details of the internal construction of processes, it is reasonable to adopt the principle of identity of indiscernibles:

$$\text{failures}(P) = \text{failures}(Q) \Rightarrow P = Q. \qquad \text{(L5)}$$

Postulates (L4) and (L5) together state that a process is uniquely defined by its failure set. In the future, we shall *identify* a process with its failure set and *define* the transition relation thus:

$$P \xrightarrow{s} Q \equiv (\forall t, X. (t, X) \in Q \Rightarrow (st, X) \in P).$$

This definition is consistent with the laws (L1)–(L3). From the definition we deduce (using conditions P1–P5)

$$P \xrightarrow{st} Q \equiv \exists R. (P \xrightarrow{s} R \;\&\; R \xrightarrow{t} Q),$$
$$P \xrightarrow{\langle\,\rangle} Q \equiv Q \subseteq P,$$
$$\text{traces}(P) = \{s \mid (s, \varnothing) \in P\},$$
$$\text{initials}(P) = \{a \mid (\langle a\rangle, \varnothing) \in P\},$$
$$\text{refusals}(P) = \{X \mid (\langle\,\rangle, X) \in P\},$$
$$\text{failures}(P) = P.$$

Since transitions can be defined in terms of failure sets and failure sets in terms of transitions, it is permissible to use either method in the definition of any particular process. It will be found convenient to give an intuitive explanation of the intended behavior of a process by giving laws governing its transitions, followed by a formal definition in terms of failure sets. Usually, the laws given will only specify sufficient conditions for the transitions of the process being defined. The formal definition will then specify a failure set whose transitions are precisely those deducible from the given laws using (L1)–(L3). In this precise sense, the formal definition using transitions specifies the required failure set.

It might be argued that modeling a process in terms of the *negative* aspects of its behavior is unnatural. However, we are primarily interested in two types of properties of processes, usually referred to as *safety* and *liveness* [16]. Safety properties of behavior can be treated well in a *traces* model [29]. Liveness properties, in particular absence of *deadlock*, cannot be treated in a model based on traces alone, because traces only give possible *positive* information about what *might* happen. By giving possible *negative* information; that is, failures or refusals, we are also able to support reasoning about what *must* happen. An alternative formulation of our model could have been based on the dual concept of *acceptances*. However, this approach seems to lead to rather more conceptual difficulties than the present approach.

We end this section with some examples of processes definable in our model.

Example 1. The simplest process is STOP, a process that never does anything, and therefore always refuses to do anything. It obeys the law

$$\mathrm{STOP} \xrightarrow{\langle\rangle} \mathrm{STOP}.$$

Furthermore, we can show, using this defining law and (L1)–(L3), that this is the only law that it obeys; that is,

$$\mathrm{STOP} \xrightarrow{s} Q \Rightarrow s = \langle\,\rangle \;\&\; Q = \mathrm{STOP}.$$

The process that has these properties is defined:

$$\mathrm{STOP} = \{(\langle\,\rangle, X) \mid X \subseteq A \;\&\; X \text{ finite}\}.$$

Clearly, it refuses to do whatever its environment may offer.

Example 2. If Q is a process and a is an event, then the process $(a \rightarrow Q)$ is a process that first does a and then behaves like Q:

$$Q \xrightarrow{s} R \Rightarrow (a \rightarrow Q) \xrightarrow{\langle a\rangle s} R.$$

We also permit Q to make internal progress while waiting for a:

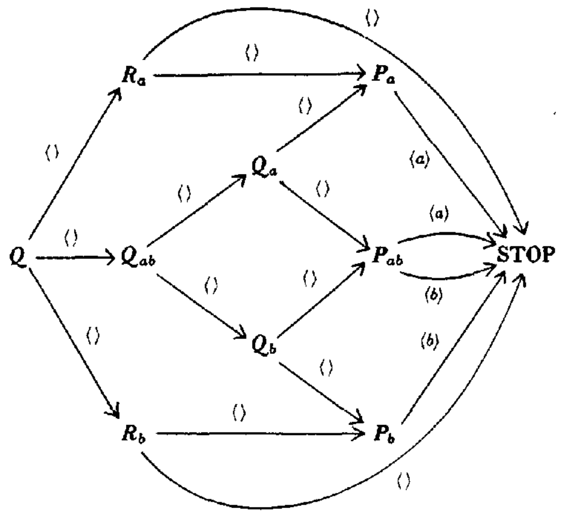
$$Q \xrightarrow{\langle\rangle} Q' \Rightarrow (a \rightarrow Q) \xrightarrow{\langle\rangle} (a \rightarrow Q').$$

The process specified by these laws is

$$\begin{aligned}(a \rightarrow Q) = {} & \{(\langle\,\rangle, X) \mid X \subseteq (A - \{a\}) \;\&\; X \text{ finite}\}\\ & \cup \{(\langle a\rangle s, X) \mid (s, X) \in Q\}.\end{aligned}$$

Clearly, it cannot initially refuse to perform a if offered; but it may (indeed must) refuse everything else. Two examples of processes built using this construction are

$$P_a = (a \rightarrow \mathrm{STOP}), \qquad P_b = (b \rightarrow \mathrm{STOP}).$$



$$\begin{tikzcd}
 & R_a \arrow[rr, "\langle\rangle"] \arrow[ddrrr, bend left=40, "\langle\rangle"] & & P_a \arrow[ddr, "\langle a\rangle"] & \\
 & & Q_a \arrow[ur, "\langle\rangle"] \arrow[dr, "\langle\rangle"] & & \\
Q \arrow[uur, "\langle\rangle"] \arrow[r, "\langle\rangle"] \arrow[ddr, "\langle\rangle"'] & Q_{ab} \arrow[ur, "\langle\rangle"] \arrow[dr, "\langle\rangle"'] & & P_{ab} \arrow[r, bend left, "\langle a\rangle"] \arrow[r, bend right, "\langle b\rangle"'] & \mathrm{STOP} \\
 & & Q_b \arrow[ur, "\langle\rangle"] \arrow[dr, "\langle\rangle"] & & \\
 & R_b \arrow[rr, "\langle\rangle"] \arrow[uurrr, bend right=40, "\langle\rangle"'] & & P_b \arrow[uur, "\langle b\rangle"'] &
\end{tikzcd}$$

FIGURE 1

Example 3. Let B be a subset of A, and let $P(x)$ be a process for each x in B. Then $(x:B \to P(x))$ is a process that first does any event b in the set B and then behaves like $P(b)$.

$$P(b) \xrightarrow{s} Q \Rightarrow (x:B \to P(x)) \xrightarrow{\langle b\rangle s} Q, \qquad \text{any} \quad b \in B.$$

Again we permit internal progress to be made while waiting for the first event:

$$(\forall x \in B.P(x) \xrightarrow{\langle\,\rangle} P'(x)) \Rightarrow (x:B \to P(x)) \xrightarrow{\langle\,\rangle} (x:B \to P'(x)).$$

The process specified by these laws is

$$\begin{aligned}(x:B \to P(x)) = {} & \{(\langle\,\rangle, X) \mid X \subseteq (A - B) \;\&\; X \text{ finite}\} \\ & \cup \{(\langle b\rangle s, X) \mid b \in B \;\&\; (s, X) \in P(b)\}.\end{aligned}$$

When B is a singleton set $\{a\}$ this reduces to the definition of $(a \to P(a))$, and when B is empty the definition coincides with that of STOP.

Note that x is a bound variable of this construction, so that

$$(x:B \to P(x)) = (y:B \to P(y)).$$

An example of a process using this construction is

$$P_{ab} = (x:\{a, b\} \to \mathrm{STOP}).$$

Example 4. With the above definitions of P_a, P_b, and P_{ab}, let

$$\begin{aligned}Q_a &= P_a \cup P_{ab}, \\ Q_b &= P_b \cup P_{ab}, \\ Q_{ab} &= P_a \cup P_b, \\ R_a &= P_a \cup \mathrm{STOP}, \\ R_b &= P_b \cup \mathrm{STOP}, \\ Q &= Q_{ab} \cup \mathrm{STOP}.\end{aligned}$$

Figure 1 shows the transitions between these processes (other than those deducible by transitivity).

Process	Initials	Refusals
Q	$\{a, b\}$	$\varnothing, \{a\}, \{b\}, \{a, b\}$
R_a	$\{a\}$	$\varnothing, \{a\}, \{b\}, \{a, b\}$
R_b	$\{b\}$	$\varnothing, \{a\}, \{b\}, \{a, b\}$
Q_{ab}	$\{a, b\}$	$\varnothing, \{a\}, \{b\}$
Q_a	$\{a, b\}$	$\varnothing, \{b\}$
Q_b	$\{a, b\}$	$\varnothing, \{a\}$
P_{ab}	$\{a, b\}$	$\varnothing$
P_a	$\{a\}$	$\varnothing, \{b\}$
P_b	$\{b\}$	$\varnothing, \{a\}$
STOP	$\varnothing$	$\varnothing, \{a\}, \{b\}, \{a, b\}$

FIGURE 2

If $A = \{a, b\}$, Figure 2 shows the initials and refusals of each of these processes, proving that they are distinct.

Example 5. RUN is a process that will always do anything offered by the environment. Thus it satisfies the law:

$$\text{RUN} \xrightarrow{s} \text{RUN}, \qquad \text{for all} \quad s \in A^*.$$

The required definition is

$$\text{RUN} = \{(s, \varnothing) \mid s \in A^*\}.$$

Clearly, RUN can never refuse anything. A similar process RUN_B that will always perform events drawn from a subset $B \subseteq A$ can be defined as

$$\text{RUN}_B = \{(s, X) \mid s \in B^* \;\&\; X \subseteq A - B \;\&\; X \text{ finite}\}.$$

Example 6. CHAOS is a process that can do anything at all; but in contrast to RUN, it can also at any time refuse to do anything at all. Indeed, it can decide at any stage to behave like any other process.

$$\text{CHAOS} \xrightarrow{s} P, \qquad \text{for all} \quad s \in A^*, \quad \text{and all} \quad P.$$

The required definition is

$$\text{CHAOS} = \{(s, X) \mid s \in A^* \;\&\; X \subseteq A \;\&\; X \text{ finite}\}.$$

Example 7. Given a nonempty, prefix-closed set T of traces, there is a process $\det(T)$ with trace set T, which at any stage refuses only *impossible* events. Its definition is

$$\det(T) = \{(t, X) \mid t \in T \;\&\; (X \text{ finite} \;\&\; \forall x \in X. t\langle x\rangle \notin T)\}.$$

Thus the failures of this process are precisely those deducible from knowledge of its trace set and laws (P1)–(P5). Such processes can be thought of as *deterministic*, because, in general, a process P satisfies the condition

$$P \xrightarrow{\langle\rangle} Q \Rightarrow P = Q,$$

so that no internal decision by P can reduce the range of its possible future actions, if and only if $P = \det(\text{traces}(P))$.

3. *Nondeterminism*

This section investigates the properties of nondeterminism. The transition relation $\xrightarrow{\langle\rangle}$ is a natural partial order on the space of processes corresponding to a measure of nondeterminism, the maximal elements with respect to this ordering being precisely the deterministic processes. Indeed, this partial ordering gives the structure of a complete semilattice to the space of the processes. This important fact is proved in the Appendix. We use the methods of lattice theory [28] to show how every recursive definition uniquely determines a process; the mathematics required is not difficult, and is fully explained.

3.1. NONDETERMINISTIC COMPOSITION. If P and Q are processes, the combination $P \sqcap Q$ is a process that behaves exactly like P or like Q; but the choice between them is wholly nondeterministic: It is made autonomously by the process (or by its implementor), and cannot be influenced or even observed by the environment. Thus $P \sqcap Q$ can do (or refuse to do) everything that P or Q can do (or refuse to do):

$$P \xrightarrow{s} R \vee Q \xrightarrow{s} R \Rightarrow (P \sqcap Q) \xrightarrow{s} R.$$

The process determined by this law is simply

$$P \sqcap Q = P \cup Q.$$

This operation is clearly associative, commutative, and idempotent. It has CHAOS as its zero.

$$\begin{array}{rll} (P \sqcap Q) \sqcap R &= P \sqcap (Q \sqcap R) & \text{(associative)}, \\ P \sqcap Q &= Q \sqcap P & \text{(commutative)}, \\ P \sqcap P &= P & \text{(idempotent)}, \\ \text{CHAOS} \sqcap P &= \text{CHAOS} & \text{(zero)}. \end{array}$$

The following relation indicates the intimate connection between nondeterministic composition and the transition relation $\xrightarrow{\langle\rangle}$:

$$P \xrightarrow{\langle\rangle} Q \Leftrightarrow (P \sqcap Q) = P.$$

This fact is closely connected with the partial-order properties of $\xrightarrow{\langle\rangle}$.

3.2. DISTRIBUTIVITY. One of the main reasons for specifying a nondeterministic process such as $P \sqcap Q$ is to allow an implementor the freedom to select and implement either P or Q, whichever of them is cheaper or gives better performance. Suppose F is some function from processes to processes. $F(\cdot)$ may be regarded as an assembly with a vacant slot, into which an arbitrary component may be plugged, producing $F(P)$ or $F(Q)$, for example. The behavior of the assembly is then a function of the behavior of this component. Suppose that an implementor has to implement $F(P) \sqcap F(Q)$. The straightforward way to do this is to implement $F(P)$ *and* $F(Q)$ and then select between them. An alternative way is first to select the component, and plug in just that one. This alternative is the same as the standard way of implementing $F(P \sqcap Q)$. We would like to ensure that both implementations give the same result, that is, that

$$F(P \sqcap Q) = F(P) \sqcap F(Q).$$

A function F that satisfies this condition for all processes P and Q is said to be *distributive*. Another reason for preferring distributive functions is that they simplify proofs of the properties of processes by allowing case analysis of the alternative behaviors.

As an example, the construction $(a \rightarrow \cdot)$ is distributive, since

$$(a \rightarrow (P \sqcap Q)) = (a \rightarrow P) \sqcap (a \rightarrow Q).$$

This means that there is no discernible difference regardless of whether the choice between P and Q is made before or after the occurrence of a.

A function of two or more arguments is distributive if it is distributive in each argument separately. Thus nondeterministic composition is itself distributive, because

$$P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap (P \sqcap R) \qquad \text{and} \qquad (Q \sqcap R) \sqcap P = (Q \sqcap P) \sqcap (R \sqcap P).$$

Furthermore, the construction $(x : B \rightarrow P(x))$ is distributive in $P(x)$ for all $x \in B$:

$$(x : B \rightarrow (P(x) \sqcap Q(x))) = (x : B \rightarrow P(x)) \sqcap (x : B \rightarrow Q(x)).$$

Thus all operations introduced so far are distributive. We shall normally make this a requirement for all operators introduced hereafter. The only exceptions will be operators that may need to call more than one version of an operand into existence. For example, if a one-place operator *op* is defined by means of the two-place distributive operator *op** by the law

$$op(P) = op^*(P, P),$$

then we see that

$$op(P \sqcap Q) = op(P) \sqcap op(Q) \sqcap op^*(P, Q) \sqcap op^*(Q, P),$$

which may very well be strictly more nondeterministic than $op(P) \sqcap op(Q)$. The extra nondeterminism is brought about by the fact that the operator may select a different implementation of its operand on each occasion when it is used.

3.3. LIMITS. The relation $P \xrightarrow{\langle\rangle} Q$ means that the process P may, as the result of internal progress, transform itself automatically into the process Q. A *chain* of processes is an infinite sequence $\langle P_i \mid i \geq 0 \rangle$, each member of which may transform itself into its successor; thus, it satisfies the law:

$$P_i \xrightarrow{\langle\rangle} P_{i+1}, \qquad \text{for all} \quad i.$$

For each such chain there exist a *limit* process, denoted $\bigsqcup_i P_i$, which can make a transition if and only if every member of the chain can:

$$(\forall i . P_i \xrightarrow{s} Q) \Leftrightarrow \left(\bigsqcup_i P_i\right) \xrightarrow{s} Q.$$

For justification, recall that the transition relation is

$$P \xrightarrow{\langle\rangle} Q \equiv P \supseteq Q,$$

the reversion inclusion relation on failure sets. It is easy to prove that the intersection of a chain of processes is again a process (see the Appendix). It follows that the desired limit process may be defined as

$$\bigsqcup_i P_i = \bigcap_i P_i, \qquad \text{provided} \quad \forall i . P_i \xrightarrow{\langle\rangle} P_{i+1}.$$

The limit process can do (or refuse) anything that every member of the chain can do (or refuse); every failure of the limit is a failure of all P_i. This operation is again distributive:

$$\bigsqcup_i (P_i \sqcap Q_i) = \left(\bigsqcup_i P_i\right) \sqcap \left(\bigsqcup_i Q_i\right),$$

provided that $\langle P_i \mid i \geq 0 \rangle$ and $\langle Q_i \mid i \geq 0 \rangle$ are chains.

The fact that this operation produces a limit with respect to the nondeterminism ordering $\xrightarrow{\langle\rangle}$ is expressed:

$$P_j \xrightarrow{\langle\rangle} \left(\bigsqcup_i P_i\right), \qquad \text{for all} \quad j,$$

and for all processes Q,

$$(\forall j. P_j \xrightarrow{\langle\rangle} Q) \Rightarrow \left(\bigsqcup_i P_i\right) \xrightarrow{\langle\rangle} Q.$$

The relation $P \xrightarrow{\langle\rangle} Q$ means simply that the set Q is contained in the set P, as we remarked above. Thus everything that Q can *do* so can P:

$$\text{traces}(Q) \subseteq \text{traces}(P),$$

and everything that Q can *refuse* so can P:

$$\text{refusals}(Q) \subseteq \text{refusals}(P).$$

In other words, P differs from Q only in that it is less deterministic, and that Q can result from P by resolution of some of P's inherent nondeterminism. In the case of a chain, where $P_i \xrightarrow{\langle\rangle} P_{i+1}$ for all i, this can mean that there is a potential infinity of nondeterministic decisions to be taken; but perhaps *none* of them will actually reach the limit $\bigsqcup_i P_i$. Thus $\bigsqcup_i P_i$ can be regarded as an "ideal" element, of which the P_i are an ever-improving sequence of approximations, getting as close as we wish to the limit but perhaps never actually reaching it. However, in implementing the limit process, we wish to allow an implementor (if so desired) to make *all* the nondeterministic choices in advance of delivering the product.

3.4. CONTINUITY. Let F be a distributive function from processes to processes. Then F is *monotonic* because

$$\begin{aligned} P \xrightarrow{\langle\rangle} Q &\Leftrightarrow (P \sqcap Q) = P \\ &\Rightarrow F(P \sqcap Q) = F(P) \\ &\Rightarrow F(P) \sqcap F(Q) = F(P) \\ &\Rightarrow F(P) \xrightarrow{\langle\rangle} F(Q), \end{aligned}$$

for all P and Q. Let $\langle P_i \mid i \geq 0 \rangle$ be a chain. Suppose that an implementor is faced with the problem of implementing $F(\bigsqcup_i P_i)$. The straightforward method would be to obtain the limit $\bigsqcup_i P_i$ and then plug it into the assembly, producing $F(\bigsqcup_i P_i)$. But suppose that the limit process is in some sense unattainable. Then we can apply F to each of the approximations P_i, obtaining the chain $\langle F(P_i) \mid i \geq 0 \rangle$, and then take the limit of that. We would like to be sure that both implementations are the same:

$$\bigsqcup_i F(P_i) = F\left(\bigsqcup_i P_i\right).$$

Then, even if the limit $\bigsqcup_i F(P_i)$ is unattainable, we can be sure of getting as close as we need by the sequence of approximations $F(P_i)$. If this condition holds for all chains, then F is said to be *continuous*. Another good reason for preferring continuous functions is that they simplify proofs of the properties of processes and allow an elegant treatment of recursively defined processes. This will be explained in more detail in the next section.

As an example, the construction $(a \rightarrow \cdot)$ is continuous, because

$$\left(a \rightarrow \bigsqcup_i P_i\right) = \bigsqcup_i (a \rightarrow P_i),$$

whenever $\langle P_i \mid i \geq 0 \rangle$ is a chain.

A function of two or more arguments is continuous if it is continuous in each argument separately. Thus nondeterministic composition is continuous, because for every process Q and every chain $\langle P_i \mid i \geq 0 \rangle$ we have (by elementary properties of union and intersection)

$$\left(\bigsqcup_i P_i\right) \sqcap Q = \bigsqcup_i (P_i \sqcap Q) \quad \text{and} \quad Q \sqcap \left(\bigsqcup_i P_i\right) = \bigsqcup_i (Q \sqcap P_i).$$

Furthermore, the construction $(x : B \rightarrow P(x))$ is continuous in $P(x)$ for all $x \in B$:

$$\left(x : B \rightarrow \left(\bigsqcup_i P_i(x)\right)\right) = \bigsqcup_i (x : B \rightarrow P_i(x)),$$

provided $\langle P_i(x) \mid i \geq 0 \rangle$ is a chain for each $x \in B$.

Finally, the limit construction is itself continuous:

$$\bigsqcup_i \left(\bigsqcup_j P_{ij}\right) = \bigsqcup_j \left(\bigsqcup_i P_{ij}\right),$$

provided that for all i, $\langle P_{ij} \mid j \geq 0 \rangle$ is a chain, and for each j, $\langle P_{ij} \mid i \geq 0 \rangle$ is a chain.

Thus all of the operators introduced so far are continuous, and we shall make this a requirement for all operators introduced hereafter. This will ensure that any expression composed from named components by applying continuous operators will also be continuous in each of its named components.

3.5. Recursion. Let F be a continuous function from processes to processes. We define the n-fold composition of F by induction on n:

$$F^0(P) = P, \qquad F^{n+1}(P) = F(F^n(P)).$$

Since F is continuous, it is also monotonic. Since CHAOS is the most nondeterministic process of all, it follows that the sequence

$$\langle F^n(\text{CHAOS}) \mid n \geq 0 \rangle$$

constitutes a chain; and its limit is defined by

$$\mu p \cdot F(p) = \bigsqcup_n F^n(\text{CHAOS}).$$

Note that in this notation, p plays the role of a bound variable, so that

$$\mu p \cdot F(p) = \mu q \cdot F(q).$$

Provided that F is continuous, it is clear that $\mu p \cdot F(p)$ is a *fixed point* of F, in the sense that it satisfies the equation $P = F(P)$.

$$\begin{aligned} F(\mu p \cdot F(p)) &= F\left(\bigsqcup_n F^n(\text{CHAOS})\right) \\ &= \bigsqcup_n F(F^n(\text{CHAOS})) \quad \text{by continuity} \\ &= \bigsqcup_n F^{n+1}(\text{CHAOS}) \\ &= \mu p \cdot F(p). \end{aligned}$$

Furthermore, this is the most general solution, in the sense that it can progress autonomously to any other solution:

$$Q = F(Q) \Rightarrow \mu p.F(p) \xrightarrow{\langle\rangle} Q.$$

Equivalently, $\mu p.F(p)$ is the *least* fixed point of F. Thus the equation $p = F(p)$ can be regarded as a *recursive definition* of the process $\mu p.F(p)$; for example, we could have defined

$$\begin{aligned} \text{RUN} &= \mu p.(x{:}A \to p), \\ \text{RUN}_B &= \mu p.(x{:}B \to p), \qquad \text{for any} \quad B \subseteq A. \end{aligned}$$

For another example, the least fixed point $\mu p.p$ of the identity function is simply CHAOS.

A similar construction can be used to find the solution of mutually recursive equations such as

$$p = F(p, q), \qquad q = G(p, q),$$

even (in some cases) when the number of equations is infinite. We will give more details in the examples of later sections and in the Appendix.

The desire to define processes freely by recursion and to be able to manipulate recursive definitions in order to prove properties of such processes is one of the major motives for requiring operators to be continuous.

4. *Operators on Processes*

In this section we define the most important primitive operators on processes, and state their chief properties. The section is sadly devoid of examples; these will be found in the next section. Proofs of some of the more interesting results appear in the Appendix.

4.1. PARALLEL COMPOSITION BY INTERSECTION. The combination $(P \| Q)$ is intended to behave like both P and Q, progressing in parallel. Thus an event can occur only when both P and Q are able to participate in it simultaneously. The same is therefore true of sequences of events:

$$P \xrightarrow{s} P' \;\&\; Q \xrightarrow{s} Q' \Rightarrow (P \| Q) \xrightarrow{s} (P' \| Q'). \tag{1}$$

The process determined by this law is defined as

$$(P \| Q) = \{(s, X \cup Y) \mid (s, X) \in P \;\&\; (s, Y) \in Q\}.$$

Thus, $(P \| Q)$ can refuse a set of events if P can refuse part of it and Q can refuse the rest.

The operator $\|$ is distributive, continuous, associative, and commutative. It has STOP as its zero and RUN as its unit; that is,

$$P \| \text{STOP} = \text{STOP}, \qquad P \| \text{RUN} = P.$$

Furthermore,

$$(x{:}B \to P(x)) \| (y{:}C \to Q(y)) = (z{:}(B \cap C) \to (P(z) \| Q(z))).$$

A partial converse to the defining relation (1) above can be proved:

$$(P \| Q) \xrightarrow{s} R \Rightarrow \exists P', Q'.P \xrightarrow{s} P' \;\&\; Q \xrightarrow{s} Q' \;\&\; R \xrightarrow{\langle\rangle} (P' \| Q').$$

4.2. CONDITIONAL COMPOSITION. The process $(P \,\Box\, Q)$ behaves either like P or like Q, but it differs from $(P \sqcap Q)$ in that the choice between them can be influenced by the environment on the very first step. If the environment offers an event a that is possible for P but not for Q, then P is selected; and conversely for Q. But if the environment offers an event that is possible for both processes, the selection between them is nondeterminate, and the environment does not get a second chance to influence it. Thus

$$P \xrightarrow{\langle a\rangle s} R \vee Q \xrightarrow{\langle a\rangle s} R \Rightarrow (P \,\Box\, Q) \xrightarrow{\langle a\rangle s} R.$$

Before occurrence of the first event, P and Q may progress independently:

$$P \xrightarrow{\langle\,\rangle} P' \;\&\; Q \xrightarrow{\langle\,\rangle} Q' \Rightarrow (P \,\Box\, Q) \xrightarrow{\langle\,\rangle} (P' \,\Box\, Q').$$

The process determined by these laws is defined

$$\begin{aligned}(P \,\Box\, Q) = {} & \{(\langle\,\rangle, X) \mid (\langle\,\rangle, X) \in P \;\&\; (\langle\,\rangle, X) \in Q\} \\ & \cup \{(s, X) \mid s \neq \langle\,\rangle \;\&\; ((s, X) \in P \vee (s, X) \in Q)\}.\end{aligned}$$

$P \,\Box\, Q$ initially refuses a set if and only if it is refused by both P and Q.

The operator $\Box$ is distributive, continuous, associative, commutative, and idempotent. It has unit STOP. Furthermore, it admits distribution thus:

$$\begin{aligned}P \sqcap (Q \,\Box\, R) &= (P \sqcap Q) \,\Box\, (P \sqcap R),\\ (x{:}B \to P(x)) \,\Box\, (y{:}C \to Q(y)) &= (z{:}(B \cup C) \to R(z)),\end{aligned}$$

where

$$\begin{aligned}R(z) &= P(z) && \text{if } z \in B - C,\\ &= Q(z) && \text{if } z \in C - B,\\ &= P(z) \sqcap Q(z) && \text{otherwise.}\end{aligned}$$

When $P = Q$, the last theorem is much more simply expressed:

$$(x{:}B \cup C \to P(x)) = (x{:}B \to P(x)) \,\Box\, (x{:}C \to P(x)).$$

4.3. PARALLEL COMPOSITION BY INTERLEAVING. The process $(P \,|||\, Q)$ behaves like P and Q operating in parallel, but it differs radically from $(P \| Q)$ in that each event requires participation of only one of the processes rather than both. Thus each trace of $(P \,|||\, Q)$ is an interleaving of a trace of P and a trace of Q, as stated in the law

$$P \xrightarrow{s} P' \;\&\; Q \xrightarrow{t} Q' \Rightarrow (P \,|||\, Q) \xrightarrow{u} (P' \,|||\, Q'),$$

where u is an interleaving of s and t.

The process determined by this law is

$$\begin{aligned}P \,|||\, Q = \{(u, X) \mid \exists s, t.(s, X) \in P \;\&\; (t, X) \in Q \\ \&\; u \text{ is an interleaving of } s \text{ and } t\}.\end{aligned}$$

$P \,|||\, Q$ can initially refuse a set only if both P and Q refuse it.

The operator $|||$ is distributive, continuous, associative, and commutative. It has unit STOP and zero RUN. Furthermore, if $P = (x{:}B \to P(x))$ and $Q = (y{:}C \to Q(y))$, then

$$P \,|||\, Q = (x{:}B \to (P(x) \,|||\, Q)) \,\Box\, (y{:}C \to (P \,|||\, Q(y))).$$

Thus if an event can be performed by both processes, which of them actually performs it is nondeterministic.

4.4. SEQUENTIAL COMPOSITION. Let ✓ (pronounced *tick*) denote an event that we interpret as successful termination of a process. Then SKIP is defined as a process that does nothing but terminate successfully:

$$\begin{aligned}\text{SKIP} &= (\checkmark \to \text{STOP})\\ &= \{(\langle\,\rangle, X) \mid \checkmark \notin X\}\\ &\quad \cup \{(\langle \checkmark \rangle, X) \mid X \subseteq A \ \&\ X \text{ is finite}\}.\end{aligned}$$

The process $(P;Q)$ behaves like P until P terminates, after which it behaves like Q. However, the occurrence of $\checkmark$ at the end of P does not appear in any trace of $(P;Q)$. It occurs automatically, without the knowledge or participation of the environment. Thus, if s does not contain $\checkmark$ we formulate the laws:

$$\begin{aligned}&P \xrightarrow{s} P' \ \&\ Q \xrightarrow{\langle\rangle} Q' \Rightarrow (P;Q) \xrightarrow{s} (P';Q'),\\ &P \xrightarrow{s\langle\checkmark\rangle} P' \ \&\ Q \xrightarrow{t} Q' \Rightarrow (P;Q) \xrightarrow{st} Q'.\end{aligned}$$

Note that we allow Q to make internal progress while waiting for P to finish.

The definition that satisfies these laws is

$$\begin{aligned}P;Q = {} &\{(s, X) \mid s \text{ does not contain } \checkmark \ \&\ (s, X \cup \{\checkmark\}) \in P\}\\ &\cup \{(st, X) \mid s \text{ does not contain } \checkmark \ \&\ (s\langle\checkmark\rangle, \varnothing) \in P \ \&\ (t, X) \in Q\}.\end{aligned}$$

This definition shows that while P is still running, $(P;Q)$ cannot refuse a set X unless P can *also* refuse to terminate successfully.

In general, it is a useful convention that $\checkmark$ should be used *only* in the process SKIP. In particular, in the construction $(x{:}B \to P(x))$, the set B should never contain $\checkmark$; in all of our examples we will assume that this convention is observed.

Sequential composition is distributive, continuous, and associative. Furthermore,

$$\begin{aligned}\text{SKIP};P &= P,\\ \text{STOP};P &= \text{STOP},\\ (x{:}B \to P(x));Q &= (x{:}B \to P(x);Q), \qquad \text{if } \checkmark \notin B,\\ (\text{SKIP} \mathbin{\Box} P);Q &= Q \sqcap (Q \mathbin{\Box} (P;Q)).\end{aligned}$$

The process $(\text{SKIP} \mathbin{\Box} P)$ can either terminate immediately or behave like P. The sequential composition $(\text{SKIP} \mathbin{\Box} P);Q$ may choose arbitrarily the first alternative; that is, $\text{SKIP};Q (= Q)$, or it may leave the choice to the environment; that is, $(\text{SKIP};Q) \mathbin{\Box} (P;Q)$.

4.5. ITERATION. The process $*P$ behaves like an infinite sequential composition of the process P:

$$P;P;P;\ldots$$

It can be simply defined by recursion:

$$*P = \mu q.(P;q).$$

Iteration is continuous, but not distributive. It fails to be distributive for the reason described earlier: $*P$ may well need to call into existence many copies of P, and different implementations can be used. In addition to continuity, iteration has the following properties:

$$\begin{aligned}(P;*P) &= *P,\\ (*(x{:}B \to P(x)));Q &= *(x{:}B \to P(x)), \qquad \text{if } \checkmark \notin B,\\ *\text{STOP} &= \text{STOP},\\ *\text{SKIP} &= \text{CHAOS}.\end{aligned}$$

The last result might seem surprising: it may seem more intuitive that $*$SKIP should equal STOP. Indeed, it is permitted to implement $*$SKIP as STOP. But

$\underset{*}{}$SKIP behaves like a process engaging in a nonterminating internal computation, never interacting with its environment. When a process is defined as a set of failures, it is important to be able to determine whether a particular failure is *outside* the set. Since $\underset{-}{*}$SKIP never interacts with its environment, the environment cannot rule out at any stage the possibility that the process might eventually perform some observable action. In such a situation it is only reasonable to identify the process with the wholly arbitrary process CHAOS. Note that the identity $\underset{-}{*}$SKIP = CHAOS is a direct consequence of our definition: CHAOS is the least fixed point of the equation $p = \mathrm{SKIP};p$.

A terminating form of iteration can be defined

$$P \textbf{ until } Q = \mu p.(Q \,\Box\, (P;p)).$$

This repeats P any number of times, possibly ending with a single execution of Q. It has the following properties, where we have assumed that $\checkmark \notin B$:

$$\underset{-}{*}P = P \textbf{ until } (\underset{-}{*}P) = P \textbf{ until } \mathrm{STOP},$$
$$((x:B \to P(x)) \textbf{ until } (y:C \to Q(y));R = (x:B \to P(x)) \textbf{ until } (y:C \to (Q(y);R),$$
$$\mathrm{SKIP} \textbf{ until } Q = (\mathrm{CHAOS} \,\Box\, Q).$$

The third result is again surprising: it could be argued that in the implementation of (SKIP **until** Q) the opportunity to behave like Q occurs infinitely often; and it is "unfair" to neglect such an opportunity forever. But it seems impossible to define a notion of fairness such that a "fair" process can be distinguished from an "unfair" one by any finite observation. That is why our theory makes no stipulation of fairness.

Some of these problems can be avoided if we insist that $\underset{-}{*}$ and **until** are used only on processes whose first event cannot be $\checkmark$. In such cases, we have the identities

$$\underset{-}{*}(x:B \to P(x)) = \mu p.(x:B \to P(x);p),$$
$$(x:B \to P(x)) \textbf{ until } (y:C \to Q(y)) = \mu p.((y:C \to Q(y)) \,\Box\, (x:B \to (P(x);p))).$$

The same technique can be used to define a *parallel* iteration, in which each activation of the body of the loop progresses in parallel with all previous activations:

$$\underset{-}{**}(x:B \to P(x)) = \mu p.(x:B \to (P(x) \,|||\, p)).$$

Unfortunately, this technique cannot be applied when a similar problem arises in the next section.

4.6. CONCEALMENT. Let b denote an event (other than $\checkmark$) that is to be regarded as an internal operation of the process P. For example, it may be an interaction between some component processes from which P has been constructed. We wish such events to occur automatically whenever they can, without the participation or even the knowledge of the environment of P. We therefore define $P\backslash b$ as the process that behaves like P except that every occurrence of b is removed from its traces; it therefore satisfies the law

$$P \xrightarrow{s} Q \Rightarrow (P\backslash b) \xrightarrow{s\backslash b} (Q\backslash b),$$

where $s\backslash b$ is formed from s by removing all occurrences of b.

For reasons explained in the previous section, if P can engage in an infinite sequence of occurrences of b, so that $P\backslash b$ can perform an unbounded sequence of hidden actions, without ever interacting with its environment, then $P\backslash b$ equals CHAOS:

$$(\forall n.P_n \xrightarrow{\langle b\rangle} P_{n+1}) \Rightarrow (P_0\backslash b) \xrightarrow{\langle\,\rangle} \mathrm{CHAOS}.$$

The required definition is

$$\begin{aligned} P\backslash b = &\{(s\backslash b, X) \mid (s, X \cup \{b\}) \in P\} \\ &\cup \{((s\backslash b)t, X) \mid \forall n.(sb^n, \varnothing) \in P \;\&\; (t, X) \in \text{CHAOS}\}, \end{aligned}$$

where sb^n is s followed by n occurrences of b.

This operation is distributive and continuous, and

$$(P\backslash b)\backslash c = (P\backslash c)\backslash b, \qquad (P\backslash b)\backslash b = P\backslash b.$$

Therefore, if B is any finite set of events, $\{b_1, \ldots, b_n\}$, we can define

$$P\backslash B = (\cdots((P\backslash b_1)\backslash b_2)\backslash\cdots\backslash b_n).$$

Other theorems are

$$\begin{aligned} \text{STOP}\backslash b &= \text{STOP}, \\ \text{RUN}\backslash b &= \text{CHAOS}, \\ \text{CHAOS}\backslash b &= \text{CHAOS}, \\ (b \rightarrow P)\backslash b &= P\backslash b, \\ (x:B \rightarrow P(x))\backslash b &= (x:B \rightarrow P(x)\backslash b), && \text{if} \quad b \notin B, \\ ((b \rightarrow P) \,\Box\, (x:B \rightarrow P(x)))\backslash b &= (P\backslash b) \sqcap ((P\backslash b) \,\Box\, (x:B \rightarrow P(x)\backslash b)), && \text{if} \quad b \notin B. \end{aligned}$$

The Appendix contains proofs of some of the interesting properties of the hiding operator.

4.7. Inverse Image. Let f be any (total) function from events to events. Then we define $f^{-1}(P)$ to be a process that can do a whenever P could have done $f(a)$:

$$P \xrightarrow{f(s)} Q \Rightarrow f^{-1}(P) \xrightarrow{s} f^{-1}(Q),$$

where the sequence $f(s)$ is obtained by applying f to each element of s.

The required definition is

$$f^{-1}(P) = \{(s, X) \mid (f(s), f(X)) \in P \;\&\; X \text{ is finite}\},$$

where we have used $f(X)$ for the set $\{f(x) \mid x \in X\}$. We shall also use $f^{-1}(B)$ for the inverse image of B under f; that is, $\{a \mid f(a) \in B\}$.

f^{-1} is distributive and continuous; furthermore

$$\begin{aligned} f^{-1}(g^{-1}(P)) &= (g \circ f)^{-1}(P), \\ f^{-1}(\text{STOP}) &= \text{STOP}, \\ f^{-1}(\text{RUN}) &= \text{RUN}, \\ f^{-1}(x:B \rightarrow P(x)) &= (y:f^{-1}(B) \rightarrow f^{-1}(P(f(y)))). \end{aligned}$$

f^{-1} distributes through $\Box$, $\|$, $|||$, and ; (provided $f^{-1}(\checkmark) = \{\checkmark\}$) and

$$f^{-1}(P\backslash B) = f^{-1}(P)\backslash f^{-1}(B),$$

provided $f^{-1}(B)$ is finite and each event in B lies in the range of f.

The Appendix contains proofs of some of these properties.

4.8. Direct Image. Let f be any total function from events to events with the property that the set $f^{-1}(a)$ is finite for all $a \in A$. (This is called the finite pre-image property.) We define $f(P)$ to be a process that can perform $f(a)$ whenever P could have done a:

$$P \xrightarrow{s} Q \Rightarrow f(P) \xrightarrow{f(s)} f(Q).$$

The required definition is

$$f(P) = \{(f(s), X) \mid (s, f^{-1}(X)) \in P\}.$$

This operation is distributive and continuous; furthermore

$$\begin{aligned} f(g(P)) &= (f \circ g)(P), \\ f(\text{STOP}) &= \text{STOP}, \\ f(\text{RUN}) &= \text{RUN}_{f(A)}, \\ f(a \to P) &= (f(a) \to f(P)). \end{aligned}$$

f distributes through $\Box$, $\|$ (provided f is injective), $|||$, and ; (provided $f^{-1}(\checkmark) = \{\checkmark\}$). Some connections between the inverse and direct image operations are

$$\begin{aligned} f^{-1}(f(P)) &= P, && \text{when } f \text{ is injective;} \\ f(f^{-1}(P)) &= P, && \text{when } f \text{ is surjective.} \end{aligned}$$

Moreover, when f is a bijection, the direct image of P under f^{-1} agrees with the inverse image of P under f, as we would expect.

5. *Applications*

In this section we give a number of examples of the use of the operators defined above in the definition of simple processes. In each case, we use laws about transitions to specify the required behavior of a process before constructing it.

5.1. A COUNT Register. A COUNT is a process that behaves like an unbounded nonnegative integer register, with initial value zero. It engages in three kinds of event:

up	increments the register, and can occur at any time.
down	decrements the register, and cannot occur when its value is zero.
iszero	can occur only when the value is zero.

Thus the behavior of COUNT is specified by the law:

$$\begin{aligned} \text{COUNT} \xrightarrow{s} Q \Rightarrow\ & (\text{EQ}(s)\ \&\ \text{initials}(Q) = \{\text{up, iszero}\}) \\ & \vee\ (\text{LESS}(s)\ \&\ \text{initials}(Q) = \{\text{up, down}\}), \end{aligned}$$

where EQ(s) means that the number of occurrences of "up" and of "down" in s are equal, and LESS(s) means that there are fewer occurrences of "down" than "up" in s.

A simple definition of a process COUNT_0, which satisfies these laws, can be given by an infinite mutual recursion (indexed by the natural numbers). The process COUNT_n defines the behavior of a count register holding the value n.

$$\begin{aligned} \text{COUNT}_0 &= (\text{iszero} \to \text{COUNT}_0)\ \Box\ (\text{up} \to \text{COUNT}_1), \\ \text{COUNT}_{n+1} &= (\text{down} \to \text{COUNT}_n)\ \Box\ (\text{up} \to \text{COUNT}_{n+2}). \end{aligned}$$

Another process that satisfies these laws is ZERO, where

$$\text{ZERO} = (\text{iszero} \to \text{ZERO})\ \Box\ (\text{up} \to (\text{POS;ZERO}))$$

and

$$\text{POS} = (\text{down} \to \text{SKIP})\ \Box\ (\text{up} \to (\text{POS;POS})).$$

Note that POS terminates successfully when it first performs one more "down" than "up". In order to compensate for an initial "up", it needs to perform *two* more "down"s than "up"s. This is achieved by first performing one more, and then one more again. A third definition of the same process is C_0, where

$$\begin{aligned} C_0 &= (\text{iszero} \rightarrow C_0) \,\Box\, (\text{up} \rightarrow C_1), \\ C_{n+1} &= \text{POS};C_n. \end{aligned}$$

5.2. CHANNEL NAMING. In this and later sections we shall assume that the only events are communications between processes, which are linked by named channels. Thus each event consists of two parts "m.t", where m is the name of a *channel* along which the communication takes place, and t is the *content* of the message that passes. We define

$$\begin{aligned} \text{chan}(\text{m}.t) &= \text{m}, \\ \text{cont}_{\text{m}}(\text{m}.t) &= t. \end{aligned}$$

If P is a process, then (m.P) is the process that engages in m.t whenever P would have engaged in t; this is the direct image of P under the renaming function

$$\begin{aligned} \mathbf{m}(a) &= \text{m}.a, \qquad \text{for all} \quad a \in A, \\ (\text{m}.P) &= \mathbf{m}(P). \end{aligned}$$

For example,

$$\text{m.COUNT}_3 = (\text{m.down} \rightarrow \text{m.COUNT}_2) \,\Box\, (\text{m.up} \rightarrow \text{m.COUNT}_4).$$

We can now construct two separate COUNTs, communicating along differently named channels and operating in parallel:

$$(\text{n.COUNT}_0) \,|||\, (\text{m.COUNT}_3).$$

Suppose now that a process MASTER requires to use a count register, communicating with it along some channel named m. To use the register, it engages in the events m.up, m.down, and m.iszero. By using the operator ‖, we can ensure that the process m.COUNT engages in these events at the same time as the MASTER. But first we need to ensure that m.COUNT will *ignore* all communications of the MASTER, except those that are directed along the channel m. This is done by using the interleaving operator. Let $M = \{\text{m.up, m.down, m.iszero}\}$. Then define

$$P \textbf{ ignoring } B = (P \,|||\, \text{RUN}_B),$$

for any set B of events. We wish to run the "slave" COUNT process in parallel with its MASTER, but ignoring events outside of the set M, and we also want to hide the internal communications between the master and slave process. To this end, we define the master–slave construction $[\text{m}:P \,\|\, Q]$ and use it as follows:

$$[\text{m:COUNT}_3 \,\|\, \text{MASTER}] = (((\text{m.COUNT}_3) \textbf{ ignoring } (A - M)) \,\|\, \text{MASTER})\backslash M.$$

If the MASTER requires to use two differently named COUNTs, we can similarly define

$$[\text{n:COUNT}_0 \,\|\, [\text{m:COUNT}_3 \,\|\, \text{MASTER}]].$$

For example, the MASTER may contain the following process code, which terminates successfully when it has added the current value of m to the current value of n, leaving the former unchanged:

$$\text{ADD} = \mu p.((\text{m.iszero} \rightarrow \text{SKIP}) \,\Box\, (\text{m.down} \rightarrow (\text{n.up} \rightarrow (p;(\text{m.up} \rightarrow \text{SKIP}))))).$$

ADD has the property that

$$\begin{aligned}&[\mathrm{n{:}COUNT}_i \parallel [\mathrm{m{:}COUNT}_j \parallel (\mathrm{ADD;RESTOFMASTER})]]\\ &\quad = [\mathrm{n{:}COUNT}_{i+j} \parallel [\mathrm{m{:}COUNT}_j \parallel \mathrm{RESTOFMASTER}]].\end{aligned}$$

This example shows how simultaneous participation in events by parallel processes can achieve the effect of communication between them.

It is possible (with care) to use the master–slave relation recursively, as shown by yet another example of the COUNT register:

$$\mathrm{COUNT} = \mu p.[\mathrm{m}{:}p \parallel \mathrm{LOOP}],$$

where

$$\begin{aligned}\mathrm{LOOP} = \mu q.[&(\mathrm{iszero} \rightarrow q)\\ &\square\ (\mathrm{up} \rightarrow \mu r.(\mathrm{up} \rightarrow \mathrm{m.up} \rightarrow r\\ &\quad \square\ \mathrm{down} \rightarrow (\mathrm{m.down} \rightarrow r)\ \square\ (\mathrm{m.iszero} \rightarrow q)))].\end{aligned}$$

The process LOOP is initially able to reveal that its value is zero or to accept an increment "up". Subsequent "up"s are relayed to the slave, as are "down"s when the slave is prepared to accept them. If the slave has value zero it will not cooperate in "m.down" but will instead communicate "m.iszero", a signal for the LOOP to return to its initial state.

It can be shown that all of our recursive definitions of COUNT registers define the same process:

$$\mathrm{COUNT} = \mathrm{COUNT}_0 = \mathrm{C}_0 = \mathrm{ZERO}.$$

The Appendix contains some of the details of the proof, and illustrates the techniques available in proving such results.

5.3. BUFFERS AND CHAINS. We define a BUFFER (of type T) as a process that inputs any sequence of values from the set T along a channel named "in" and outputs the same sequence of values along a channel named "out". Let m be a channel name, and let

$$\begin{aligned}\mathrm{m}.T &= \{\mathrm{m}.t \mid t \in T\},\\ (s\restriction \mathrm{m}) &= \mathrm{cont_m}(s\backslash(A - \mathrm{m}.T)),\\ X\restriction \mathrm{m} &= \{t \mid \mathrm{m}.t \in X\}.\end{aligned}$$

Less formally, $(s\restriction \mathrm{m})$ is the sequence of values whose communication along channel "m" is recorded in the trace s. Now a BUFFER is a process that satisfies the law

$$\begin{aligned}&\mathrm{BUFFER} \xrightarrow{s} Q \Rightarrow\\ &\quad s \in (\mathrm{in}.T \cup \mathrm{out}.T)^* \ \&\\ &\quad (s\restriction\mathrm{out} \le s\restriction\mathrm{in})\ \&\\ &\quad (s\restriction\mathrm{out} = s\restriction\mathrm{in} \Rightarrow \mathrm{initials}(Q) = \mathrm{in}.T)\ \&\\ &\quad (s\restriction\mathrm{out} \ne s\restriction\mathrm{in} \Rightarrow \mathrm{initials}(Q) \cap \mathrm{out}.T \ne \varnothing).\end{aligned}$$

The third line states that an empty buffer must input any value of type T, and the fourth line states that a nonempty buffer must always be prepared to output some value of type T. The condition that the output sequence be a *prefix* of the input sequence (line 2) guarantees that the values are transmitted in the correct order. It is left undetermined whether a nonempty buffer may refuse to input.

A simple example that meets this specification is the single-portion buffer B1:

$$\mathrm{B1} = \underset{-}{*}(x{:}(\mathrm{in}.T) \rightarrow (\mathrm{out}.(\mathrm{cont_{in}}(x)) \rightarrow \mathrm{SKIP})).$$

In future we shall use abbreviations:

$$\begin{aligned}(?x{:}T \to P(x)) &\quad \text{for} \quad (y{:}(\text{in}.T) \to P(\text{cont}_{\text{in}}(y))),\\ !x &\quad \text{for} \quad (\text{out}.x \to \text{SKIP}).\end{aligned}$$

Thus the example B1 could have been written

$$\text{B1} = \overset{*}{\cdot}(?x{:}T \to !x),$$

or, using the definition of the iteration operator,

$$\text{B1} = \mu p.(?x{:}T \to !x;p).$$

Let us define the *bound* of a buffer (if it exists) to be the minimum number of items it can contain and refuse to input any further item. Thus, B1 is a buffer with bound 1. An unbounded buffer can be defined by an infinite set of mutually recursive equations, indexed on the current content of the buffer, which starts empty:

$$\begin{aligned}\text{BUFF}_{\langle\rangle} &= (?x{:}T \to \text{BUFF}_{\langle x\rangle}),\\ \text{BUFF}_{s\langle x\rangle} &= (?y{:}T \to \text{BUFF}_{\langle y\rangle s\langle x\rangle}) \,\Box\, (!x;\text{BUFF}_s).\end{aligned}$$

The process $(P \gg Q)$ is one in which everything output by P on channel "out" is simultaneously input by Q on channel "in"; and all such communications are concealed from their common environment. Thus all external communication on channel "in" is received by P and ignored by Q, and all external communication on channel "out" is sent by Q and ignored by P. Communication between P and Q is achieved by transforming each event "out.x" of P and each event "in.x" of Q to the *same* event x. This is achieved by applying the function $\mathbf{strip}_{\text{m}}$ that removes channel name m from messages:

$$\begin{aligned}\mathbf{strip}_{\text{m}}(x) &= t, \quad &&\text{if} \quad x = \text{m}.t,\\ &= x, &&\text{otherwise.}\end{aligned}$$

Assuming that T is finite, we may define

$$(P \gg Q) = [((\mathbf{strip}_{\text{out}}(P)) \textbf{ ignoring } \text{out}.T) \,\|\, ((\mathbf{strip}_{\text{in}}(Q)) \textbf{ ignoring } \text{in}.T)]\backslash T.$$

We normally only use the operator $\gg$ for processes whose traces are built from events in.t and out.t for $t \in T$. This operation can be thought of as *chaining* the two processes together.

The operator $\gg$ is partially associative: provided the traces of P, Q, and R are all contained in the set $(\text{in}.T \cup \text{out}.T)^*$ and there is no possibility of an unbroken infinite sequence of hidden internal communications in either $P \gg Q$ or $Q \gg R$, then we have the identity $P \gg (Q \gg R) = (P \gg Q) \gg R$. Note that the associative law always holds when P, Q, and R are buffers, since any infinite sequence of internal communications in $P \gg Q$ could only arise from P outputting an infinite number of items after it had only input a finite number, which no buffer can do.

Understandably, there is a close relationship between buffers and the chain operator; there are several interesting results that demonstrate this. For example, if P and Q are two processes such that $\text{traces}(P \sqcap Q) \subseteq (\text{in}.T \cup \text{out}.T)^*$, then whenever two of the processes P, Q and $P \gg Q$ are buffers, so is the third. This result can be used both to justify the construction of large buffers from smaller ones and to prove buffers correct by various means.

For example, a buffer with bound 2, which stores two portions, may be defined:

$$\text{B2} = \text{B1} \gg \text{B1}.$$

In general, a buffer B_n that stores n portions is defined inductively:

$$\begin{aligned} B_1 &= B1, \\ B_{n+1} &= B_n \gg B1. \end{aligned}$$

Two unbounded buffers may be defined:

$$\begin{aligned} B_\infty &= \mu p.(?x:T \rightarrow (p \gg !x;B1)), \\ B_\infty^* &= \mu p.(?x:T \rightarrow (p \gg !x;p)). \end{aligned}$$

Two buffers whose capacities grow steadily in proportion to the number of items they have output may be defined:

$$\begin{aligned} B^* &= \mu p.(?x:T \rightarrow (B1 \gg !x;p)) \\ B' &= \mu p.(B1 \gg (?x:T \rightarrow !x;p)). \end{aligned}$$

A buffer that may have any bound or none can be defined:

$$B_? = \mu p.(B1 \sqcap (?x:T \rightarrow (p \gg !x;B1))).$$

Note that it is not possible to define in our model a buffer with a nondeterministically chosen finite bound, without also allowing an unbounded buffer as an implementation. This is because there is no finite test that could demonstrate that a buffer is unbounded.

The following identities may be proved by various methods:

(a) $B_\infty^* = B_\infty = \text{BUFF}_{\langle\rangle}$,
(b) $(B_\infty \gg B_\infty) = (B_n \gg B_\infty) = (B_\infty \gg B_n) = B_\infty$,
(c) $(B_n \gg B_m) = B_{n+m}$,
(d) $(\text{BUFF}_v \gg \text{BUFF}_w) = \text{BUFF}_{vw}$,
(e) $B^* = B'$.

There are more interesting connections between buffers and the chain operator. One of these states that whenever $P \gg Q$ is a buffer, so is $?x:T \rightarrow (P \gg !x;Q)$. This result can be extended recursively in several ways: for example, it can be shown that if two processes P and Q satisfy the equation

$$P \gg Q = ?x:T \rightarrow (P \gg !x;Q)$$

then $P \gg Q$ is a buffer. A generalization of this last result is discussed in the Appendix, which also contains proof methods for establishing the above results.

Let $f: T^* \rightarrow T^*$ be a prefix-preserving function on strings; that is, $f(s)$ is always a prefix of $f(st)$. A process P is said to be a *pipe* for f if it satisfies the law

$$\begin{aligned} &P \xrightarrow{s} Q \Rightarrow \\ &\quad s \in (\text{in}.T \cup \text{out}.T)^* \ \& \\ &\quad (s\restriction\text{out} \leq f(s\restriction\text{in}))\ \& \\ &\quad (s\restriction\text{out} = f(s\restriction\text{in}) \Rightarrow \text{initials}(Q) = \text{in}.T)\ \& \\ &\quad (s\restriction\text{out} \neq f(s\restriction\text{in}) \Rightarrow \text{initials}(Q) \cap \text{out}.T \neq \varnothing). \end{aligned}$$

Thus a buffer is just a pipe for the identity function. If P is a pipe for f and Q is a pipe for g, then $(P \gg Q)$ is a pipe for $(g \circ f)$. A simple example is a pipe for the sine function:

$$\text{SIN} = \underset{\sim}{*}(?x:\text{REAL} \rightarrow !\mathit{sine}(x)),$$

and so are $(\text{SIN} \gg B_3)$, $(B_8 \gg \text{SIN})$, etc.

Suppose now a MASTER process requires to use the SIN process to compute sines, using a channel named sin. It sends the argument x by sin!x (an abbreviation for (sin.in.$x \rightarrow$ SKIP)), and it inputs the result by (sin?y: REAL $\rightarrow P(y)$), which is also an abbreviation for something similar. (Note the coding trick that assimilates output by the master with input by the slave.) The required effect can be achieved by the combination

$$[\text{sin:SIN} \parallel \text{MASTER}].$$

A pipe for the tangent function can be defined:

$$[\text{sin:SIN} \parallel [\text{cos:COS} \parallel \text{TAN}]],$$

where COS is defined similarly to SIN, and where

$$\begin{aligned}\text{TAN} = {}&\overset{*}{}((?x\!:\text{REAL} \rightarrow \text{sin}!x);\\ &(\text{sin}?y\!:\text{REAL} \rightarrow (\text{cos}?z\!:\text{REAL} \rightarrow !(y/z)))).\end{aligned}$$

A process is said to be a *variable* (of type T) if it is always prepared to input a new value, and, once it has been initialized, it is always prepared to output the value it has most recently input; that is, for all Q,

$$\begin{aligned}P \xrightarrow{s} Q \Rightarrow {}&(s\restriction\text{in} = \langle\,\rangle \Rightarrow \text{initials}(Q) = \text{in}.T)\ \&\\ &(s\restriction\text{in} \neq \langle\,\rangle \Rightarrow \text{initials}(Q) = (\text{in}.T \cup \{\text{out}.t\})),\end{aligned}$$

where

$$t = \text{last}(s\restriction\text{in}).$$

A process definition satisfying these laws is

$$\text{VAR}_T = (?x\!:T \rightarrow \text{V}_x),$$

where

$$\text{V}_x = (?y\!:T \rightarrow \text{V}_y) \,\Box\, (!x;\text{V}_x), \qquad \text{for all} \quad x \in T.$$

V_x is the behavior of a variable with initial value x. A fresh local instance of such a variable can be declared thus:

$$[\text{m:VAR}_T \parallel \text{MASTER}].$$

A *stack* (for type T) is a process P that outputs everything that it has input, on a last-in, first-out principle; and outputs the signal "isempty" when empty. It obeys the law

$$\begin{aligned}&P \xrightarrow{s} Q \Rightarrow\\ &\quad(\text{length}(s\restriction\text{in}) = \text{length}(s\restriction\text{out}) \Rightarrow \text{initials}(Q) = (\text{in}.T \cup \{\text{out.isempty}\}))\ \&\\ &\quad(\text{length}(s\restriction\text{in}) > \text{length}(s\restriction\text{out}) \Rightarrow \text{in}.T \subseteq \text{initials}(Q)\\ &\qquad\&\ \text{initials}(Q) \cap \text{out}.T \neq \varnothing).\end{aligned}$$

A trace s of a stack satisfies the condition

$$\forall t \leq s.\text{length}(t\restriction\text{in}) \geq \text{length}(t\restriction\text{out})$$

and the number of items on the stack after this trace is simply

$$\text{size}(s) = \text{length}(s\restriction\text{in}) - \text{length}(s\restriction\text{out});$$

if this number is zero, the stack is empty. If $u\langle\text{out}.x\rangle$ is a trace of a stack, then u can be written in the form $s\langle\text{in}.x\rangle t$, where t is a stack trace with size $(t) = 0$. This corresponds to the last-in, first-out property.

Three different implementations of a stack can be modeled on three different implementations of the COUNT register. We hope the reader will enjoy constructing them.

6. *Conclusions*

We have introduced a mathematical model for a powerful language of communicating processes, and demonstrated that the model enjoys many elegant mathematical properties. In our examples we showed that some interesting problems can be tackled in our model, and we outlined some proof techniques that may be used to prove properties of processes defined in the model. Our work can be seen as a step toward providing a tractable semantic model for parallel processes.

Several alternative approaches to the problems of parallelism have been proposed, and our work is most closely related to Milner's calculus of communicating systems (CCS) [20] and the work of the Edinburgh group (e.g., [9, 11]). Several authors have reported recently on connections between the underlying semantic models of CCS and CSP, notably Brookes [3] and Hennessy and de Nicola [10]. The relationship with Kennaway's work [15] is discussed in [2]. As we remarked earlier, our failures model is a direct extension of earlier models based on traces [12, 13] that were unable to cope properly with nondeterminism. The chief advantage the failures model appears to have over most other attempted approaches derived from traces (e.g., *possible futures* [25]) is its mathematical tractability.

The proof techniques of our paper, and those of [23], have been successfully applied to many interesting parallel programming problems. The mathematical elegance of our model helps considerably in such endeavors. Although, as yet, our proof methods are relatively informal, there are grounds for hope that powerful formal proof systems can be developed based on our semantic domain [2, 4, 21]. Formal proof systems for tracelike models already exist [6, 29]; Hoare-style proof systems for CSP [1, 17] are also well known.

Our model is very well suited to reasoning about the problems associated with deadlock. A related problem is *divergence*, which arises when a process performs an unbounded sequence of internal actions without responding to the requests of its environment. This problem has been touched upon briefly in this paper; for example, in the treatment of iteration and hiding. However, it can be argued that the present model does not cope entirely adequately with divergence. Extensive discussions of these points can be found in [2] and [23]. Future work will show that the model and associated proof rules can be simply adapted to give a satisfactory treatment of divergent processes [5]. In addition, the model can be adapted to cope with imperative parallel languages [24]. A similar attempt based on traces was made in [8].

Although we presented our semantics in denotational style, the failure sets of compound processes being built up from the failures of their components, our alternative formulation based on transitions can be developed into an operational semantics in the style of [11] and [22]. This issue will be elaborated in [5].

7. *Prospects*

The original objective of denotational semantics was to provide a clear, consistent, and unambiguous definition of a programming language that is likely to have more than one implementation. Such a definition could serve usefully as a national or international standard; it would give a precise specification that must be met by each implementor; and it would tell each programmer exactly what he can rely on in all implementations. Thus it would achieve the primary objective of standardization, namely, the reliable conjunction of programs and implementations from widely differing sources. The deficiencies of existing language standards can be directly attributed to their failure to take advantage of this known technology—a failure that to future generations will probably seem amazing. In the area of parallel programming languages, we hope that the development of a suitable semantic model at an early stage will forestall a repetition of the problems that have beset the development and standardization of sequential programming languages.

Apart from the improved quality of programming language standards, the techniques of mathematical semantics have much to offer in improving the quality of computer programs. In the first place, they offer the possibility that an implementor can prove with mathematical rigor that his implementation meets the standard specification of the language. Clearly, no program can be more reliable than the implementation of the language in which it is expressed for input to a computer.

A second advantage of a mathematical description of a programming language is that it offers the individual programmer the opportunity to prove the correctness of his program with respect to some description of its intended behavior. For this, he would need to identify the mathematical object denoted by his program, and then prove that this object exhibits the required mathematical properties. Unfortunately, this method of program proving is impractically laborious; it is like trying to solve differential equations using only the original definitions of derivatives in the epsilon–delta terminology of analysis. What is required for practical program development and proof is a formal calculus, similar to the assertional calculus for sequential programs, that will permit a reasonably direct expression of the purpose of each command, and a method of proving that it meets its purpose. Such a calculus must be firmly based on a proof of its conformity with an appropriate mathematical model, just as the differential calculus can be ultimately based on the Dedekind model of real numbers, and as Hoare-style logics for sequential languages can typically be based on a conventional state-transformation semantics. But these are topics for future research.

Appendix

This Appendix contains proofs of some of the results stated in the paper. The selection of example proofs provided here is intended to be illustrative of our general methods and techniques. More extensive accounts and proofs of all of the results in the paper will be found in either Brookes [2] or Roscoe [23].

After recalling the definition of a process as a set of failures satisfying a number of conditions, we show that the nondeterminism ordering $\xrightarrow{(\,)}$ gives the space of processes the structure of a *complete semilattice.* This fact was used, together with the *continuity* of all operations used in the paper, to justify our use of recursively defined processes. Next we prove some properties of parallel composition: hiding and inverse image. Finally, we introduce a proof method, based on *fixed-point induction,* for reasoning about recursively defined processes; we show how to establish in this way the properties of some of the example processes of Section 5.

A process is a subset $P \subseteq A^* \times \mathscr{P}(A)$ satisfying the conditions:

(P1) $(s, X) \in P \Rightarrow X$ is finite,
(P2) $(\langle\,\rangle, \varnothing) \in P$,
(P3) $(st, \varnothing) \in P \Rightarrow (s, \varnothing) \in P$,

(P4) $Y \subseteq X \,\&\, (s, X) \in P \Rightarrow (s, Y) \in P$,
(P5) $(s, X) \in P \,\&\, (s\langle c\rangle, \varnothing) \notin P \Rightarrow (s, X \cup \{c\}) \in P$.

Let M be the set of all such processes.

For a trace $s \in A^*$, the transition relation on processes is defined:

$$P \xrightarrow{s} Q \Leftrightarrow Q \subseteq \{(t, X) \mid (st, X) \in P\}.$$

In particular,

$$P \xrightarrow{\langle\,\rangle} Q \Leftrightarrow Q \subseteq P,$$

so that $\xrightarrow{\langle\,\rangle}$ is just the superset relation. If $P \neq Q$ and $P \xrightarrow{\langle\,\rangle} Q$ we say that P is more nondeterministic than Q.

Since any collection of sets is partially ordered by the superset relation, $\xrightarrow{\langle\,\rangle}$ is a partial order on M:

$$P \xrightarrow{\langle\,\rangle} Q \xrightarrow{\langle\,\rangle} P \Leftrightarrow P = Q,$$
$$P \xrightarrow{\langle\,\rangle} Q \xrightarrow{\langle\,\rangle} R \Rightarrow P \xrightarrow{\langle\,\rangle} R.$$

Now we show that the union of any non-empty set of processes is again a process, and that the intersection of any directed set of processes is a process. This will establish the fact that the space $(M, \xrightarrow{\langle\,\rangle})$ is a complete semilattice.

THEOREM 1. *The union of any nonempty set of processes is a process.*

PROOF. Let $\mathscr{D}$ be a nonempty set of processes, and let $P = \bigcup \mathscr{D}$, so that the failures of P are

$$(s, X) \in P \Leftrightarrow \exists Q \in \mathscr{D}.(s, X) \in Q.$$

We need to verify that P has the properties (P1)–(P5). This is straightforward. By way of illustration, consider (P5). Suppose

$$(s, X) \in P \,\&\, (s\langle c\rangle, \varnothing) \notin P.$$

By definition of P, there is a process $Q \in \mathscr{D}$ with

$$(s, X) \in Q \,\&\, (s\langle c\rangle, \varnothing) \notin Q.$$

But Q, being a process, has property (P5), which gives

$$(s, X \cup \{c\}) \in Q,$$

and hence, since $Q \in \mathscr{D}$, it follows that

$$(s, X \cup \{c\}) \in P,$$

as required. The remaining properties are established similarly. □

Definition 1. A set $\mathscr{D}$ of processes is *directed* if it is nonempty and

$$\forall Q_1, Q_2 \in \mathscr{D}. \exists R \in \mathscr{D}.(Q_1 \xrightarrow{\langle\,\rangle} R \,\&\, Q_2 \xrightarrow{\langle\,\rangle} R).$$

THEOREM 2. *The intersection of any directed set of processes is a process.*

PROOF. Let $\mathscr{D}$ be directed and let $P = \bigcap \mathscr{D}$, so that

$$(s, X) \in P \Leftrightarrow \forall Q \in \mathscr{D}.(s, X) \in Q.$$

Again we must prove that P satisfies conditions (P1)–(P5). Again we give details only for (P5). Suppose

$$(s, X) \in P \,\&\, (s\langle c\rangle, \varnothing) \notin P.$$

This means that

$$\forall Q \in \mathscr{D}.(s, X) \in Q,$$

but there is a process $Q_1 \in \mathscr{D}$ such that

$$(s\langle c\rangle, \varnothing) \notin Q_1.$$

We want to prove that $(s, X \cup \{c\}) \in P$, and this will be true unless there is a process $Q_2 \in \mathscr{D}$ with

$$(s, X \cup \{c\}) \notin Q_2.$$

If such a process existed, we would be able to use directedness to find a process $R \in \mathscr{D}$ such that

$$Q_1 \xrightarrow{\langle\,\rangle} R \;\&\; Q_2 \xrightarrow{\langle\,\rangle} R,$$

that is, $R \subseteq Q_1 \cap Q_2$. But then we would have, by our previous assumptions,

$$(s, X) \in R,$$
$$(s\langle c\rangle, \varnothing) \notin R,$$
$$(s, X \cup \{c\}) \notin R.$$

If R is a process, this would contradict (P5). There cannot, therefore, be any such process; and it must be the case that

$$(s, X \cup \{c\}) \in P,$$

as required. That completes the proof. □

The parallel composition of two processes P and Q was defined

$$P \| Q = \{(s, X \cup Y) \mid (s, X) \in P \;\&\; (s, Y) \in Q\}.$$

Next we show that this is a well-defined operation on processes, and then we establish its continuity.

THEOREM 3. *If P and Q are processes, so is $P \| Q$.*

PROOF. (P1)–(P3) are trivial. For (P4), let $(s, Z) \in P \| Q \;\&\; Z' \subseteq Z$. We want to show that $(s, Z') \in P \| Q$. By hypothesis, there are sets X and Y such that

$$Z = X \cup Y, \qquad (s, X) \in P, \qquad (s, Y) \in Q.$$

Let $X' = X \cap Z'$, $Y' = Y \cap Z'$. Then we have

$$Z' = X' \cup Y', \qquad (s, X') \in P, \qquad (s, Y') \in Q,$$

using (P4) for the processes P and Q. It follows by definition that $(s, Z') \in P \| Q$, as required.

For (P5), suppose that $(s, Z) \in P \| Q$ and that $(s\langle c\rangle, \varnothing) \notin P \| Q$. Again let X and Y be such that

$$Z = X \cup Y, \qquad (s, X) \in P, \qquad (s, Y) \in Q.$$

Since $s\langle c\rangle$ is not a trace of $P \| Q$, it cannot be a trace of both P and Q. Without loss of generality, assume it is not a trace of P. Then we have

$$(s, X) \in P \;\&\; (s\langle c\rangle, \varnothing) \notin P.$$

By (P5) this gives $(s, X \cup \{c\}) \in P$, from which we deduce that

$$(s, X \cup \{c\} \cup Y) \in P \| Q.$$

That completes the proof. □

Parallel composition is a symmetric operation. In order to prove continuity of this operation, it is only necessary to establish continuity in one argument.

THEOREM 4. *Parallel composition is continuous.*

PROOF. Let $\langle P_n \mid n \geq 0\rangle$ be a chain of processes with limit $P = \bigcap_n P_n$. Let Q be any process. We must show that

$$P \parallel Q = \bigcap_n (P_n \parallel Q).$$

It is easy to prove from the definitions that

$$P \parallel Q \subseteq \bigcap_n (P_n \parallel Q).$$

The converse is more difficult. Suppose (s, Z) is a failure of $\bigcap_n(P_n \parallel Q)$; choose sets X_n, Y_n accordingly, so that

$$Z = X_n \cup Y_n, \qquad (s, X_n) \in P_n, \qquad (s, Y_n) \in Q,$$

for all $n \geq 0$. Since Z is a finite set and the X_n and Y_n are subsets of Z, the list of pairs (X_n, Y_n) contains only finitely many distinct pairs. Some pair, say (X, Y), must occur infinitely often. For this pair we have

$$Z = X \cup Y \;\&\; (s, Y) \in Q,$$

and also, for infinitely many n, $(s, X) \in P_n$. Since the P_n form a chain, this means that $(s, X) \in P$. Putting these results together, we have

$$Z = X \cup Y, \qquad (s, X) \in P, \qquad (s, Y) \in Q,$$

and hence $(s, Z) \in P \parallel Q$. This shows that every failure of $\bigcap_n(P_n \parallel Q)$ is also a failure of $P \parallel Q$, as required to complete the proof. □

The hiding operation $\backslash b$ on traces simply deletes all occurrences of the event b. It may be defined inductively on the length of traces:

$$\begin{aligned} \langle\,\rangle\backslash b &= \langle\,\rangle, \\ (s\langle c\rangle)\backslash b &= s\backslash b, && \text{if} \quad c = b, \\ &= (s\backslash b)\langle c\rangle, && \text{if} \quad c \neq b. \end{aligned}$$

It is easy to see that $(st)\backslash b = (s\backslash b)(t\backslash b)$ for all s and t.
The hiding operation on processes was defined

$$\begin{aligned} P\backslash b = {} & \{(s\backslash b, X) \mid (s, X \cup \{b\}) \in P\} \\ & \cup \{((s\backslash b)t, X) \mid \forall n.(sb^n, \varnothing) \in P \;\&\; (t, X) \in \text{CHAOS}\}. \end{aligned}$$

The following lemma will be useful in proving the well definedness of hiding.

LEMMA 1. *If P is a process, then*

$$(s, \varnothing) \in P \Rightarrow (s\backslash b, \varnothing) \in P\backslash b.$$

PROOF. There are two cases to consider. If $(sb^n, \varnothing) \in P$ for all n, then by definition we see that $((s\backslash b)t, X) \in P\backslash b$ for all t, X. In particular, therefore, $(s\backslash b, \varnothing) \in P\backslash b$. The other case is when there is an integer n such that $(sb^n, \varnothing) \in P$ but $(sb^{n+1}, \varnothing) \notin P$. By (P5) this gives $(sb^n, \{b\}) \in P$, from which it follows that $((sb^n)\backslash b, \varnothing) \in P\backslash b$. Since $(sb^n)\backslash b = s\backslash b$, the result follows. □

THEOREM 5. *If P is a process so is* $P\backslash b$.

PROOF. (P1)–(P4) are straightforward, using Lemma 1 and elementary properties of $\backslash b$ on traces. For (P5), let $(u, X) \in P\backslash b$ and $(u\langle c\rangle, \varnothing) \notin P\backslash b$. The case when

$c = b$ is easily dealt with, so suppose $c \neq b$. There are two possibilities for (u, X):

(1) $\exists s.s\backslash b = u \;\&\; (u, X \cup \{b\}) \in P$,
(2) $\exists s.s\backslash b \leq u \;\&\; \forall n.(sb^n, \varnothing) \in P$.

But (2) can be ruled out, because in this case we would get $(u\langle c\rangle, \varnothing) \in P\backslash b$ too, contradicting our assumption. We may therefore assume that there is a trace s such that

$$s\backslash b = u \;\&\; (s, X \cup \{b\}) \in P.$$

If $(s\langle c\rangle, \varnothing) \in P$, then by Lemma 1 we would have

$$((s\backslash b)\langle c\rangle, \varnothing) = (u\langle c\rangle, \varnothing) \in P\backslash b,$$

contradicting our hypothesis. Thus, $(s\langle c\rangle, \varnothing) \notin P$. But we know that $(s, X \cup \{b\}) \in P$, so (P5) gives $(s, X \cup \{b, c\}) \in P\backslash b$, as required. □

The operation of removing all occurrences of a particular event from a trace is clearly commutative and idempotent:

$$(s\backslash b)\backslash c = (s\backslash c)\backslash b, \qquad (s\backslash b)\backslash b = s\backslash b.$$

When hiding a set $B = \{b_1, \ldots, b_n\}$ of events, therefore, the order of deletion is irrelevant; we write

$$s\backslash B = (\cdots(s\backslash b_1)\cdots)\backslash b_n.$$

For consistency with this notation, we adopt the convention that

$$s\backslash\varnothing = s.$$

It is easily seen that for any pair of finite sets B and C,

$$(s\backslash B)\backslash C = s\backslash(B \cup C).$$

Before we derive similar results for the hiding operation on processes, we need a technical lemma.

LEMMA 2. *Let B be a finite set of events and w be a trace. If $\langle s_n \mid n \geq 0\rangle$ is a sequence of traces such that for all n,*

$$s_n\backslash B \leq w,$$

then either infinitely many of the s_n are equal, or there is a trace s and an increasing sequence of traces $t_k \in B^k$, such that, for all k, st_k is a prefix of some s_n.

PROOF. Since w has finite length, it has only a finite number of prefixes. At least one of its prefixes must, therefore, be generated by applying $\backslash B$ to infinitely many of the s_n. So there is a trace $w' \leq w$ and an infinite subsequence $\langle s_{n_k} \mid k \geq 0\rangle$ such that for all k,

$$s_{n_k}\backslash B = w' \leq w.$$

Without loss of generality (replacing w by w' and s_k by s_{n_k}), we can assume that for all n,

$$s_n\backslash B = w.$$

This does not affect the conclusion of the lemma, but simplifies the argument. Let the trace w be

$$w = a_1 a_2 \cdots a_r,$$

where each $a_i \in A$. Each s_n must have the form

$$s_n = u_n^{(0)} a_1 u_n^{(1)} \cdots a_r u_n^{(r)},$$

where each $u_n^{(i)} \in B^*$. But B is a finite set and the events $a_1 \cdots a_r$ are fixed. If the s_n are bounded in length only finitely many of them can be distinct, since there is only a finite number of distinct traces of any fixed length having this form. In this case, then, some trace s occurs infinitely often in the sequence $\langle s_n \mid n \geq 0 \rangle$ and we have the first alternative. In the case when the s_n are unbounded in length, there must be a position i at which the $u_n^{(i)}$ terms $(n \geq 0)$ are unbounded in length. Taking i to be the smallest such index, we can apply the above argument to the traces obtained by truncating the s_n at this position. This time we deduce that there is a trace s and infinitely many n such that

$$su_n^{(i)} \leq s_n.$$

Since the $u_n^{(i)} \in B^*$ appearing here are unbounded in length, and since B is finite, we can use Konig's Lemma to deduce the existence of an increasing sequence

$$u_{n_k}^{(i)} < u_{n_{k+1}}^{(i)}, \qquad k \geq 0.$$

Putting t_k equal to the length k prefix of $u_{n_k}^{(i)}$, we get an increasing sequence $\langle t_k \mid k \geq 0 \rangle$ with

$$st_k \leq s_{n_k} \;\&\; t_k \in B^k,$$

as required for the second alternative. That completes the proof. □

THEOREM 6

$$\begin{aligned}(P\backslash b)\backslash c = {} & \{((s\backslash b)\backslash c, X) \mid (s, X \cup \{b, c\}) \in P\} \\ & \cup \{(((s\backslash b)\backslash c)t, X) \mid \forall n\ \exists u \in \{b, c\}^n.(su, \varnothing) \in P \;\&\; (t, X) \in CHAOS\}\end{aligned}$$

PROOF. It is easy to check that every failure of the right-hand side is also a failure of $(P\backslash b)\backslash c$. For the converse, let $(w, X) \in (P\backslash b)\backslash c$. We must show that either

$$(1) \quad \exists s.(s\backslash b)\backslash c = w \;\&\; (s, X \cup \{b, c\}) \in P$$

or

$$(1') \quad \exists s.(s\backslash b)\backslash c \leq w \;\&\; \forall n \exists u \in \{b, c\}^n (su, \varnothing) \in P.$$

Since $(w, X) \in (P\backslash b)\backslash c$, we have either

$$(2) \quad \exists t.t\backslash c = w \;\&\; (t, X \cup \{c\}) \in P\backslash b$$

or

$$(2') \quad \exists t.t\backslash c \leq w \;\&\; \forall n.(tc^n, \varnothing) \in P\backslash b.$$

First consider case (2). Let t be a trace such that $t\backslash c = w$ and $(t, X \cup \{c\}) \in P\backslash b$. By definition of $P\backslash b$ this means that either

$$(3) \quad \exists s.s\backslash b = t \;\&\; (s, X \cup \{b, c\}) \in P$$

or

$$(3') \quad \exists s.s\backslash b \leq t \;\&\; \forall n.(sb^n, \varnothing) \in P.$$

But since $t\backslash c = w$ it is clear that (3) $\Rightarrow$ (1) and (3′) $\Rightarrow$ (1′). It remains to consider case (2′). Now let t be a trace such that $t\backslash c \leq w$ and $(tc^n, \varnothing) \in P\backslash b$ for all n. This means that for each n, either

$$(4) \quad \exists s_n . s_n \backslash b = tc^n \;\&\; (s_n, \{b\}) \in P$$

or

$$(4') \quad \exists s_n . s_n \backslash b \leq tc^n \;\&\; \forall m . (s_n b^m, \varnothing) \in P.$$

If (4′) holds for any n we can repeat the argument of case (3′) to show that (1′) holds. The only remaining possibility is, therefore, when there is a sequence $\langle s_n \mid n \geq 0 \rangle$ such that for all n,

$$(5) \quad s_n \backslash b = tc^n \;\&\; (s_n, \{b\}) \in P.$$

Since $t \backslash c \leq w$, we also have

$$\forall n . (s_n \backslash b) \backslash c \leq w,$$

and we may apply Lemma 2 with $B = \{b, c\}$. From (5) we see that it is impossible for infinitely many of the s_n to be identical because

$$\text{length}(s_n) \geq \text{length}(t) + n.$$

There must be, therefore, a trace s and a subsequence $\langle s_{n_k} \mid k \geq 0 \rangle$, and an increasing sequence of traces $u_k \in B^k$ such that

$$\forall k . s u_k \leq s_{n_k}.$$

But for each k we also have from (5)

$$s_{n_k} \backslash b = tc^n \;\&\; (s_{n_k}, \{b\}) \in P.$$

Hence we have $(s \backslash b) \backslash c \leq w$ and, using (P3) and (P4),

$$(s_{n_k}, \varnothing) \in P \Rightarrow (s u_k, \varnothing) \in P.$$

Thus we have established (1′). That completes the proof. □

Notice that the expression derived here for $(P \backslash b) \backslash c$ is symmetric in b and c, and that putting $b = c$ produces again the failure set for $P \backslash b$. It follows that hiding is commutative and idempotent:

$$(P \backslash b) \backslash c = (P \backslash c) \backslash b, \qquad (P \backslash b) \backslash b = P \backslash b.$$

We may therefore write

$$P \backslash B = (\cdots (P \backslash b_1) \cdots) \backslash b_n$$

for any finite set $B = \{b_1, \ldots, b_n\}$. Again we adopt the convention that

$$P \backslash \varnothing = P.$$

It is clear that for any pair of finite sets B and C,

$$(P \backslash B) \backslash C = P \backslash (B \cup C).$$

Lemma 2 is also applicable in the following proof.

THEOREM 7. *Hiding is continuous.*

PROOF. Let $\langle P_n \mid n \geq 0 \rangle$ be a chain of processes with limit P. We must show that

$$\sqcap_n (P_n \backslash b) = P \backslash b.$$

As usual, one inclusion is easy:

$$P\backslash b \subseteq \bigcap_n (P_n\backslash b).$$

For the converse, let $(u, X) \in \bigcap_n(P_n\backslash b)$; we need to prove that $(u, X) \in P\backslash b$. For this, we require either

$$(1) \quad \exists s.s\backslash b = u \;\&\; (s, X \cup \{b\}) \in P,$$

or

$$(1') \quad \exists s.s\backslash b \leq u \;\&\; \forall n.(sb^n, \varnothing) \in P.$$

By hypothesis, $(u, X) \in P_n\backslash b$ for each n, so there is a sequence $\langle s_n \mid n \geq 0\rangle$ of traces such that for each n, either

$$(2) \quad s_n\backslash b = u \;\&\; (s_n, X \cup \{b\}) \in P_n,$$

or

$$(2') \quad s_n\backslash b \leq u \;\&\; \forall m.(s_n b^m, \varnothing) \in P_n.$$

One of these conditions must hold for infinitely many n, and hence (by the chain condition) for all n. In either case Lemma 2 is applicable, with $B = \{b\}$, and we treat separately the two possible conclusions. The first case is when infinitely many of the s_n are identical to some trace, say s. Rewriting (2) and (2′), we have either

$$(3)\; s\backslash b = u \;\&\; (s, X \cup \{b\}) \in P_n \qquad \text{for infinitely many} \quad n,$$

or

$$(3')\; s\backslash b \leq u \;\&\; \forall m.(sb^m, \varnothing) \in P_n, \qquad \text{for infinitely many} \quad n.$$

Now the chain condition shows that (3) $\Rightarrow$ (1) and (3′) $\Rightarrow$ (1′). The second and final case is when the s_n are of unbounded length; we know in this case that there is a trace s and an infinite subsequence $\langle s_{n_k} \mid k \geq 0\rangle$ such that

$$sb^k \leq s_{n_k}, \qquad \text{for all} \quad k.$$

From either (3) or (3′) we can deduce (using (P3) and (P2)) that

$$s\backslash b \leq u \;\&\; (sb^k, \varnothing) \in P_{n_k}, \qquad \text{for all} \quad k.$$

But then for each k it is clear that $(sb^k, \varnothing) \in P_n$ for infinitely many, and hence for all, n. Thus,

$$s\backslash b \leq u \;\&\; \forall k.(sb^k, \varnothing) \in P,$$

as required for (1′). That completes the proof. □

In Sections 4.7 and 4.8 we introduced renaming operations, which are total functions from events to events. For any such operation f, we use the same name for the extension of f to a function on traces and for the pointwise extension of f to sets:

$$f(\langle c_1, \ldots, c_n\rangle) = \langle f(c_1), \ldots, f(c_n)\rangle,$$
$$f(X) = \{f(x) \mid x \in X\}.$$

Similarly we write

$$f^{-1}(b) = \{a \in A \mid f(a) = b\},$$
$$f^{-1}(t) = \{s \in A^* \mid f(s) = t\},$$
$$f^{-1}(X) = \{a \in A \mid f(a) \in X\}.$$

The inverse image of a process P under f is defined:

$$f^{-1}(P) = \{(s, X) \mid (f(s), f(X)) \in P \;\&\; X \text{ finite}\}.$$

Using the elementary facts that

$$\begin{aligned} f(\langle\rangle) &= \langle\rangle, & f(st) &= f(s)f(t), \\ f(\varnothing) &= \varnothing, & Y \subseteq X &\Rightarrow f(Y) \subseteq f(X), \end{aligned}$$

it is easy to verify that $f^{-1}(P)$ is a process whenever P is. We now prove a connection between the hiding operation and inverse image. In this theorem we assume that f has the finite pre-image property.

THEOREM 8. *Let f be a renaming operation and let B be a finite subset of the range of f. Then, for all processes P,*

$$f^{-1}(P\backslash B) = f^{-1}(P)\backslash f^{-1}(B).$$

PROOF. Let $C = f^{-1}(B)$. Since B is a subset of the range of f, we know that $f(C) = B$. If B is empty there is nothing to prove, since $f^{-1}(\varnothing) = \varnothing$ and $P\backslash\varnothing = P$. Assume therefore that B is nonempty. The proof relies on a simple lemma: for all t and s,

$$t\backslash B = f(s) \Leftrightarrow \exists u.t = f(u) \;\&\; u\backslash C = s.$$

Note the corollary that $f(u\backslash C) = f(u)\backslash B$. Now suppose (s, X) is a failure of $f^{-1}(P)\backslash C$. We show that (s, X) is also a failure of $f^{-1}(P\backslash B)$. By definition of $f^{-1}(P)\backslash C$ we have either

$$(1) \quad \exists u.u\backslash C = s \;\&\; (u, X \cup C) \in f^{-1}(P),$$

or

$$(2) \quad \exists u.u\backslash C \leq s \;\&\; \forall n \exists w \in C^n.(uw, \varnothing) \in f^{-1}(P).$$

In the first case we have

$$\begin{aligned} \exists u.u\backslash C = \;& s \;\&\; (f(u), f(X) \cup f(C)) \in P \\ \Rightarrow\;& \exists t.t\backslash B = f(s) \;\&\; (t, f(X) \cup B) \in P. \end{aligned}$$

But this implies $(f(s), f(X)) \in P\backslash B$, and hence $(s, X) \in f^{-1}(P\backslash B)$, as required. In the other case we have

$$\exists u.u\backslash C \leq s \;\&\; \forall n \exists w \in C^n.(f(u)f(w), \varnothing) \in P.$$

But $w \in C^n \Rightarrow f(w) \in B^n$, so (putting $t = f(u)$, $v = f(w)$) we get

$$\exists t.t\backslash B \leq f(s) \;\&\; \forall n \exists v \in B^n.(tv, \varnothing) \in P.$$

This again implies that $(f(s), f(X)) \in P\backslash B$, and again $(s, X) \in f^{-1}(P\backslash B)$. So far we have shown that

$$f^{-1}(P)\backslash C \subseteq f^{-1}(P\backslash B).$$

The converse is established in the same way, to complete the proof. □

The fifth section of this paper introduced a few of the types of results that can be proved of processes defined in our model. Many of these results have long and technical proofs, so it is not possible here to prove all of them in detail. What we can do is to indicate some of the methods that can be employed in proving such results, and illustrate our methods by applying them to some of the simpler examples. A much more extensive exposition of these topics can be found in Roscoe [23].

The results quoted earlier fall into two basic categories. First, we have the general technical results, the main examples of which were the following:

(i) partial associativity of $\gg$;
(ii) if two of P, Q and $P \gg Q$ are buffers, then so is the third;
(iii) if $P \gg Q$ is a buffer, then so is $?x{:}\, T \rightarrow (P \gg !x;Q)$;
(iv) if P is a pipe for f, and Q is a pipe for g, then $P \gg Q$ is a pipe for $g \circ f$.

Results of this type must be proved by analysis of the definitions involved, much as in the proofs of the results from Sections 3 and 4. This analysis is much assisted by such lemmas as the following:

LEMMA 3. *If $traces(P \sqcap Q) \subseteq (in.T \cup out.T)^*$, then*

$$\begin{aligned} P \gg Q = \{(s, (X \cap in.T) \cup (Y \cap out.T) \cup Z) \mid\ & \exists u, v.\ s \in filter(u, v)\ \& \\ & (u, X) \in P\ \&\ (v, Y) \in Q\ \& \\ & (X{\restriction}out \cup Y{\restriction}in) = T\ \& \\ & Z \text{ is finite}\ \&\ Z \subseteq \overline{(in.T \cup out.T)}\} \\ \cup\ \{(st, X) \mid\ & \exists^{\infty}(u, v).\ s \in filter(u, v)\ \& \\ & (u, \varnothing) \in P\ \&\ (v, \varnothing) \in Q\ \&\ (t, X) \in CHAOS\}, \end{aligned}$$

where

$$\begin{aligned} filter(u, v) = \{s \in (in.T \cup out.T)^* \mid\ & s{\restriction}in = u{\restriction}in\ \& \\ & s{\restriction}out = v{\restriction}out\ \&\ u{\restriction}out = v{\restriction}in\}. \end{aligned}$$

Here the notation $\overline{X}$ is used for the complement of a set. Note that Lemma 3 states that in the combination $P \gg Q$ the left-hand process P has control over the input events and the right-hand process Q controls the output. The traces s of $P \gg Q$ are obtained from traces u of P and v of Q that agree on the internal communications ($u{\restriction}\text{out} = v{\restriction}\text{in}$), by filtering out the output events of u and the input events of v. We omit the proof of this result, as it is a direct consequence of the definition of the $\gg$ operator. Once this and similar lemmas have been established, the proofs of the four theorems (i)–(iv) above are, although long, not too difficult.

The second group of results are those which we wish to prove of individual processes, such as

(v) C_0 = COUNT;
(vi) B_∞ is a buffer.

Proofs of such results are obtained by application of technical results of the above type together with analysis of the recursive constructions used in the definitions of the processes involved. Recall that a recursively defined process is either the least fixed point of some function from processes to processes or, when mutual recursion is used, a component of the least fixed point of some function defined on a product space of processes. Our analysis of recursive constructions will be helped by first defining a standard means of approximating the behavior of a process.

If P is a process, define $P{\downarrow}n$, the restriction of P to n steps, to be

$$\{(s, X) \mid \text{length}(s) < n\ \&\ (s, X) \in P\} \cup \{(st, X) \mid s \in \text{traces}(P)\ \&\ \text{length}(s) = n\}.$$

$P{\downarrow}n$ is the process that behaves exactly like P for n steps, and then dissolves into CHAOS. The following easily proved identities hold for processes P and Q, and all natural numbers m, n.

$$P{\downarrow}0 = \text{CHAOS},$$
$$(P{\downarrow}n){\downarrow}m = P{\downarrow}\min(m, n),$$
$$P{\downarrow}n \xrightarrow{\langle\,\rangle} P,$$
$$(\forall n. P{\downarrow}n = Q{\downarrow}n) \Rightarrow P = Q.$$

Thus, for example, any process is determined uniquely by its restrictions to finite depth. Indeed, the sequence $\langle P{\downarrow}n \mid n \geq 0\rangle$ is always a chain with limit P.

Suppose F is a function from processes to processes. If F is continuous, we know that its effect on any process P is uniquely determined by its effect on the finite restrictions of P:

$$F(P) = \bigsqcup_n F(P{\downarrow}n).$$

We say that F is *nondestructive* if for all P

$$F(P){\downarrow}n = F(P{\downarrow}n){\downarrow}n,$$

and *constructive* if

$$F(P){\downarrow}(n+1) = F(P{\downarrow}n){\downarrow}(n+1).$$

Informally, a nondestructive function can be regarded as producing results whose n-step behavior depends only on the n-step behavior of its operand; similarly, a constructive function produces results whose $n + 1$-step behavior depends only on the n-step behavior of its operand. Note that every constructive function is also nondestructive.

These results and definitions generalize in the obvious way to functions of more than one argument. Let us write M^Λ for the product space whose elements are vectors of processes indexed by a set Λ. (M^Λ is isomorphic to the function space $\Lambda \rightarrow M$, where M is the space of processes). A typical element of M^Λ can be written $\langle P_\lambda \mid \lambda \in \Lambda\rangle$, and if $\bar{P} \in M^\Lambda$ we will denote the λ-component of $\bar{P}$ by P_λ. The definition of restriction ${\downarrow}n$ on product spaces is simply:

$$\langle P_\lambda \mid \lambda \in \Lambda\rangle{\downarrow}n = \langle P_\lambda{\downarrow}n \mid \lambda \in \Lambda\rangle.$$

A function of more than one argument can be constructive or nondestructive in any or all of its arguments. For example, a two-place function $F: M \times M \rightarrow M$ is constructive in its second argument if for all P and Q and all n we have $F(P, Q){\downarrow}(n+1) = F(P, Q{\downarrow}n){\downarrow}(n+1)$.

Of our existing operators, $(a \rightarrow \cdot)$, $\Box$, $\sqcap$, $\|$, $|||$, $;$, f^{-1}, f, and $a.(\ )$ are all nondestructive in each of their arguments, and $(a \rightarrow \cdot)$, $(x{:}T \rightarrow \cdot)$ and $(?x{:}T \rightarrow \cdot)$ are all constructive. The last two of these operators can properly be regarded as functions from M^T to M.

The following results are not hard to prove.

(i) If $F: M^\Lambda \rightarrow M^\Delta$ and $G: M^\Delta \rightarrow M^\Theta$ are nondestructive, then so is $G \circ F$; if in addition one of F and G is constructive, then so is $G \circ F$.

(ii) If $F: M^\Lambda \rightarrow M^\Theta$ and $G: M^\Lambda \rightarrow M^\Delta$ are nondestructive (constructive, respectively) then so is the function H: $M^\Lambda \rightarrow (M^\Theta \times M^\Delta)$ defined $H(\bar{P}) = (F(\bar{P}), G(\bar{P}))$.

(iii) If $F: (M^\Delta \times M^\Lambda) \rightarrow M^\Lambda$ is nondestructive in its second argument and nondestructive (constructive, respectively) in its first argument, then the function

$G: M^{\Delta} \to M^{\Lambda}$ defined $G(\bar{P}) = \mu\bar{Q}.F(\bar{P}, \bar{Q})$ is nondestructive (constructive, respectively).

A corollary to these results is the fact that if F is any function defined using any combination of our operators other than hiding, then F is nondestructive. Furthermore, if all occurrences of an operand of F are directly or indirectly *guarded*, then F is a constructive function of that operand. For example, the function $F: (M \times M) \to M$ defined

$$F(P, Q) = P \square (a \to \mu p.(\text{SKIP} \square P \square (p;Q)))$$

is nondestructive in its first argument and constructive in its second.

Now suppose that R: $M^{\Lambda} \to$ {**true**, **false**} is a predicate. We will say that R is *satisfiable* if there exists some $\bar{P} \in M^{\Lambda}$ such that $R(\bar{P})$ = **true**. We will say that R is *continuous* if it satisfies the condition

$$\forall \bar{P}.((\forall n. \exists \bar{Q}.(\bar{P}{\downarrow}n = \bar{Q}{\downarrow}n) \;\&\; R(\bar{Q})) \Rightarrow R(\bar{P})).$$

A predicate is continuous if and only if its truth can be determined by examining finite restrictions of its argument.

THEOREM 9. *Suppose that $F: M^{\Lambda} \to M^{\Lambda}$ is a (monotone) constructive function with least fixed point $\bar{P}$. Suppose also that R is a continuous satisfiable predicate of M^{Λ} and that R is F-inductive in the sense that $\forall \bar{Q}.(R(\bar{Q}) \Rightarrow R(F(\bar{Q}))$. Then $R(\bar{P})$ holds.*

PROOF. By satisfiability of R, we can choose $\bar{Q}$ such that $R(\bar{Q})$ holds. It is easy to prove by induction on n that $R(F^n(\bar{Q}))$ holds for all n. We claim that in addition

$$\bar{P}{\downarrow}n = F^n(\bar{Q}){\downarrow}n$$

holds for all n. We use induction on n.

The base case $n = 0$ is easy, because $\bar{P}{\downarrow}0 = \bar{Q}{\downarrow}0 = \text{CHAOS}^{\Lambda}$.

Now suppose that $\bar{P}{\downarrow}n = F^n(\bar{Q}){\downarrow}n$. Then we have

$$\begin{aligned} \bar{P}{\downarrow}(n+1) &= F(\bar{P}){\downarrow}(n+1) && \text{since } \bar{P} = F(\bar{P}), \\ &= F(\bar{P}{\downarrow}n){\downarrow}(n+1) && \text{by constructivity,} \\ &= F(F^n(\bar{Q}){\downarrow}n){\downarrow}(n+1) && \text{by hypothesis,} \\ &= F(F^n(\bar{Q})){\downarrow}(n+1) && \text{by constructivity,} \\ &= F^{n+1}(\bar{Q}){\downarrow}(n+1). \end{aligned}$$

This establishes the claim that $\bar{P}{\downarrow}n = F^n(\bar{Q}){\downarrow}n$ for all n. Since we now have

$$\forall n.(R(F^n(\bar{Q})) \;\&\; (\bar{P}{\downarrow}n = F^n(\bar{Q}){\downarrow}n))$$

we can infer $R(\bar{P})$, by continuity of R. □

This theorem gives us a general method for proving properties of recursively defined processes. Informally it tells us that if the truth of a reasonable (i.e., satisfiable, continuous) predicate is preserved by the function of a sufficiently well-defined (monotone, constructive) recursion then we may infer the truth of that predicate on the least fixed point.

In fact, it is easy to show that any constructive function has only one fixed point. This is a corollary to the above result when we put $R(\bar{P}) = (\bar{P} = \bar{Q})$, where $\bar{Q}$ is chosen to be any fixed point of F. This R is continuous and satisfiable, and satisfies $R(\bar{P}) \Rightarrow R(F(\bar{P}))$. It follows that the least fixed point is identical to $\bar{Q}$, and hence that there is a unique fixed point for F.

Theorem 9 can be generalized to certain nonconstructive recursions that can be proved independently to have unique fixed points. The class of allowable predicates in such generalizations may need to be different from the one used above.

The class of continuous predicates is large. A few examples are listed below for functions of a single variable.

LEMMA 4. *The following predicates are continuous.*

(*i*) $R(P) \equiv (P = Q)$, *any* Q,
(*ii*) $R(P) \equiv (P \xrightarrow{\langle\,\rangle} Q)$,
(*iii*) $R(P) \equiv (Q \xrightarrow{\langle\,\rangle} P)$,
(*iv*) *"P is a buffer,"*
(*v*) *P is free from deadlock* $(\equiv \forall s. \neg(P \xrightarrow{s} STOP))$,
(*vi*) $\forall s \in traces(P).\psi(s)$ (ψ *any predicate on* Σ^*),
(*vii*) $\bigwedge_{i \in I} R_i(P)$, *all* R_i *continuous*,
(*viii*) $R_1(P) \vee R_2(P)$, R_1, R_2 *continuous*.

We are now sufficiently well equipped to be able to tackle some of our examples. The first example will be to prove that ZERO = COUNT$_0$.

Recall that the COUNT processes are defined by means of the following function F from $M^N \to M^N$, where N is the set of natural numbers:

$$F(\bar{P}) = \bar{Q},$$

where

$$Q_0 = (\text{iszero} \to P_0) \mathbin{\Box} (\text{up} \to P_1),$$
$$Q_{n+1} = (\text{down} \to P_n) \mathbin{\Box} (\text{up} \to P_{n+2}).$$

Let $\underline{\text{COUNT}} = \langle \text{COUNT}_n \mid n \in N \rangle$ denote the least fixed point of F. We wish to show that COUNT$_0$ = ZERO, where ZERO satisfies

$$\text{ZERO} = (\text{iszero} \to \text{ZERO}) \mathbin{\Box} (\text{up} \to \text{POS; ZERO}),$$
$$\text{POS} = (\text{down} \to \text{SKIP}) \mathbin{\Box} (\text{up} \to \text{POS; POS}).$$

We will prove the following predicate of $\underline{\text{COUNT}}$

$$R(\bar{P}) = \forall n. P_n = \text{POS}^n;\text{ZERO},$$

where POS^0 = SKIP and POS^{n+1} = POS;POS^n. This predicate on M^N is easily seen to be continuous and satisfiable. The function F of the COUNT recursion is constructive, since all recursive calls are guarded. To prove $R(\underline{\text{COUNT}})$ it is sufficient to prove that, for all $\bar{P}$, we have $R(\bar{P}) \Rightarrow R(F(\bar{P}))$; the result then follows by fixed point induction, using Theorem 9. To this end, suppose $R(\bar{P})$ holds. Let $\bar{Q} = F(\bar{P})$. Then we have

$Q_0 = (\text{iszero} \to P_0) \mathbin{\Box} (\text{up} \to P_1)$	by definition of F,
$\quad = (\text{iszero} \to \text{ZERO}) \mathbin{\Box} (\text{up} \to \text{POS;ZERO})$	by hypothesis that $R(\bar{P})$ holds,
$\quad = \text{ZERO}$	by definition of ZERO.

Also, by a similar argument, we have

$Q_{n+1} = (\text{down} \to P_n) \mathbin{\Box} (\text{up} \to P_{n+2})$	by definition of F,
$\quad = (\text{down} \to \text{POS}^n;\text{ZERO}) \mathbin{\Box} (\text{up} \to \text{POS}^{n+2};\text{ZERO})$	by hypothesis that R holds,
$\quad = ((\text{down} \to \text{SKIP}) \mathbin{\Box} (\text{up} \to \text{POS;POS}));\text{POS}^n;\text{ZERO}$	
$\quad = \text{POS;POS}^n;\text{ZERO}$	by definition of POS,
$\quad = \text{POS}^{n+1};\text{ZERO}$.	

This shows that $R(\bar{Q})$ holds, and completes the proof. Notice that this particular application of our rule can be interpreted as an instance of the unique fixed point property of constructive functions.

The constructiveness of functions is not quite so easy to establish when hiding is used, as is the case in recursions that use the master–slave operator $[\,\|\, m{:}]$ and the chaining operator $\gg$. The function

$$F(Q) = [P \,\|\, m{:}Q]$$

is not in general constructive; however, some conditions can be imposed on P that make the function constructive. For example, if the alphabet used to communicate with the slave Q is B, then F will be constructive if P satisfies the condition

$$s \in \text{traces}(P) \Rightarrow \text{length}(s) \geq 2 \times \text{length}(s \restriction B).$$

Intuitively, this condition requires that P does not communicate with its slave too often between other actions.

Similarly, the function $G(P) = (P \gg Q)$ is nondestructive, if Q is constrained to satisfy

$$s \in \text{traces}(Q) \Rightarrow \text{length}(s \restriction \text{out}) \geq \text{length}(s \restriction \text{in}).$$

Likewise, if P satisfies the condition

$$s \in \text{traces}(P) \Rightarrow \text{length}(s \restriction \text{in}) \geq \text{length}(s \restriction \text{out}),$$

then the function $H(Q) = (P \gg Q)$ is nondestructive. This result can be used to show that the function

$$F(Q) = ?x{:}\, T \rightarrow (Q \gg !x;\text{B}1)$$

is constructive. Then we may show that B_∞ is a buffer by a simple argument. Assume P is a buffer. Then $(P \gg \text{B}1)$ is also a buffer. This implies that $?x{:}\, T \rightarrow (P \gg !x;\text{B}1)$ is a buffer. Thus the predicate "is a buffer" is preserved by the function F above. Since B_∞ is the least fixed point of this function, it follows by Theorem 9 that B_∞ is a buffer.

This method of proof can be used for most of the results in Section 5 that refer to individual processes. In some cases, however, it is not sufficiently sophisticated. Several modifications to the method are possible in order to extend considerably the class of problems that can be tackled. In particular, it is straightforward to generalize to functions defined on sets of processes.

For a set $\mathcal{M}$ of processes, define $\mathcal{M}\downarrow n = \{P\downarrow n \mid P \in \mathcal{M}\}$. For a set-valued function $F{:}\ \mathcal{P}(M) \rightarrow \mathcal{P}(M)$, say F is constructive if, for all sets of processes $\mathcal{M} \subseteq M$ and all integers n, we have

$$F(\mathcal{M})\downarrow(n+1) = F(\mathcal{M}\downarrow n)\downarrow(n+1).$$

This generalizes the notion of constructiveness to functions defined on sets of processes. For any predicate R of M, we can apply R to a set of processes $\mathcal{M}$ in the obvious way; we will write $R(\mathcal{M})$ for the conjunction $\bigwedge_{P \in \mathcal{M}} R(P)$. The proof method based on Theorem 9 for constructive functions of M generalizes also, as follows.

THEOREM 10. *Suppose $F{:}\ \mathcal{P}(M) \rightarrow \mathcal{P}(M)$ is constructive and R is a continuous satisfiable predicate of M. Suppose that for all $\mathcal{M} \subseteq M$ we have*

$$(R(\mathcal{M}) \Rightarrow R(F(\mathcal{M}))).$$

Then whenever $\mathcal{M}$ satisfies the condition $\mathcal{M} \subseteq F(\mathcal{M})$ we can infer $R(\mathcal{M})$.

We sketch here an example using this method. Suppose that $\{(P_\lambda, Q_\lambda) \mid \lambda \in \Lambda\}$ is an indexed set of pairs of processes. Suppose also that for every $\lambda \in \Lambda$ there is a function g_λ: $T \rightarrow \Lambda$ such that the process $P_\lambda \gg Q_\lambda$ satisfies

$$P_\lambda \gg Q_\lambda = ?x : T \rightarrow (P_{g_\lambda(x)} \gg !x; Q_{g_\lambda(x)}).$$

If we now define an appropriate F (on sets of pairs of processes), we can show that the predicate "is a buffer" is preserved by application of the function. The inductive proof rule of Theorem 10 can then be used to show that each of the processes $P_\lambda \gg Q_\lambda$ is a buffer. For a more detailed proof of this result and an explanation of how it can be used to prove correctness of a wide variety of buffers, see Roscoe [23]. In particular, consider the general result that states that whenever P and Q are processes such that

$$P \gg Q = ?x : T \rightarrow (P \gg !x; Q),$$

then $P \gg Q$ is a buffer. This was stated without proof in Section 5. It is now an immediate corollary of the above result, obtained by choosing the obvious g_λ.

ACKNOWLEDGMENTS. This work has been inspired and guided by the pioneering research reported in [19] and [20], and by fruitful discussions with Bill Rounds. The exposition has benefited from suggestions by Geraint Jones and by the referees.

REFERENCES

1. APT, K. R., FRANCEZ, N., AND DE ROEVER, W. P. A proof system for communicating sequential processes. *ACM Trans. Program Lang. Syst. 2*, 3 (July 1980), 359–385.
2. BROOKES, S. D. A model for communicating sequential processes. D.Phil. dissertation, Oxford Univ., Oxford, England, 1983.
3. BROOKES, S. D. On the relationship of CCS and CSP. In *Proceedings of the 1983 International Conference on Automata, Languages, and Programming (ICALP 83)* Lecture Notes in Computer Science, vol. 154. Springer-Verlag, New York, 1983.
4. BROOKES, S. D. A semantics and proof system for communicating processes. In *Proceedings of the NSF/ONR Conference on Logics of Programs.* Lecture Notes in Computer Science, vol. 164. Springer-Verlag, New York, 1983.
5. BROOKES, S. D., AND ROSCOE, A. W. An improved failures model for communicating processes, to be published as CMU Tech. Rep., 1984.
6. CHANDY, R. M., AND MISRA, J. An axiomatic proof technique for networks of communicating processes. Tech. Report TR-98, Univ. of Texas, Austin, Tex.
7. DIJKSTRA, E. W. Cooperating sequential processes. In *Programming Languages,* F. Genuys, Ed. Academic Press, New York, 1968.
8. FRANCEZ, N., LEHMANN, D., AND PNEULI, A. A linear history semantics of languages for distributed programming. In *Proceedings of the 21st IEEE Foundations of Computer Science Symposium* IEEE, New York, 1980.
9. HENNESSY, M., AND MILNER, R. On observing nondeterminism and concurrency. In *Proceedings of the 1980 International Conference on Automata, Languages, and Programming (ICALP 80).* Lecture Notes in Computer Science, vol. 85. Springer-Verlag, New York, 1980.
10. HENNESSY, M., AND DE NICOLA, R. Testing equivalences for processes. In *Proceedings of the 1983 International Conference on Automata, Languages, and Programming (ICALP 83).* Lecture Notes in Computer Science, vol. 154. Springer-Verlag, New York, 1983.
11. HENNESSY, M., AND PLOTKIN, G. A term model for CCS. In *Proceedings of the 9th Conference on Mathematical Foundations of Computer Science* Lecture Notes in Computer Science, vol. 88 Springer-Verlag, New York, 1980.
12. HOARE, C. A. R. Communicating sequential processes. *Commun ACM 21,* 8 (Aug. 1978), 666–676.
13. HOARE, C. A. R. A model for communicating sequential processes. Tech. Report PRG-22, Oxford Univ. Programming Research Group, Oxford, England, 1981.
14. HOARE, C. A. R., BROOKES, S. D., AND ROSCOE, A. W. A theory of communicating sequential processes. Tech. Report PRG-16, Oxford Univ. Programming Research Group, Oxford, England, 1981.

15. KENNAWAY, J.R. Formal semantics of nondeterminism and parallelism, D.Phil. dissertation, Oxford Univ., Oxford, England, 1981.
16. LAMPORT, L. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng. SE-3*, 2 (Mar. 1977).
17. LEVIN, G.L. A proof technique for communicating sequential processes (with an example). Ph.D. dissertation, Cornell Univ., Ithaca, N.Y., 1979.
18. MILNE, R., AND STRACHEY, C. *A Theory of Programming Language Semantics.* Chapman Hall, London, and Wiley, New York, 1976.
19. MILNER, R. Algebras for communicating systems. Tech. Report CSR-25-78, Computer Science Dept., Edinburgh Univ., Edinburgh, England, 1978.
20. MILNER, R. *A Calculus of Communicating Systems* Lecture Notes in Computer Science, vol. 92. Springer-Verlag, New York, 1980.
21. DE NICOLA, R. A complete set of axioms for a theory of communicating sequential processes. In *Proceedings of the 1983 Conference on the Fundamentals of Computation Theory.* Lecture Notes in Computer Science, vol. 158. Springer-Verlag, New York, 1983.
22. PLOTKIN, G. An operational semantics for CSP. In *Proceedings of the W.G.2.2 Conference,* 1982.
23. ROSCOE, A.W. A mathematical theory of communicating processes. D.Phil. dissertation, Oxford Univ., Oxford, England, 1982.
24. ROSCOE, A.W. Denotational Semantics for *occam* In preparation.
25. ROUNDS, W.C., AND BROOKES, S.D. Possible futures, acceptances, refusals and communicating processes. In *Proceedings of the 22nd IEEE Foundations of Computer Science Symposium.* IEEE, New York, 1981.
26 SCOTT, D.S. Data types as lattices. *SIAM J Comput. 5* (1976), 522–587.
27. SMYTH, M.B. Powerdomains. *J Comput. Syst. Sci 16* (1978).
28. STOY, J.E. *Denotational Semantics* MIT Press, Cambridge, Mass., 1977.
29. ZHOU, C.C., AND HOARE, C.A.R. Partial correctness of communicating processes and protocols. Tech. Report PRG-20, Oxford Univ. Programming Research Group, Oxford, England, 1981.

RECEIVED AUGUST 1981; REVISED SEPTEMBER 1983 AND OCTOBER 1983; ACCEPTED OCTOBER 1983