DENOTATIONAL SEMANTICS FOR OCCAM

A.W. Roscoe
Oxford University Computing Laboratory
8-11 Keble Road, Oxford OX1 3QD,
United Kingdom.

ABSTRACT. A denotational semantics is given for a large subset of occam, a programming language for concurrent systems. The semantic domain used is a "failure-sets" model modified to allow machine states to be properly dealt with. The applications of the semantics are discussed briefly, and we see how the natural congruence induced by the semantics allows us to prove simple algebraic laws relating occam programs.

0. Introduction The occam programming language [10] was designed with the philosophy of eliminating unnecessary complexities, thus keeping the language simple and elegant. Fundamental to its design was the idea that the language should have close ties with formal methods of designing and verifying programs. The main aim of this paper is to develop a theory which will allow us to apply existing techniques of program verification to occam, as well as giving us a rigorous mathematical framework for developing new ones.

The main difficulty in developing a useful theory of occam is the fact that it is a concurrent language. Indeed the idea of concurrency is central in occam, making an adequate treatment of concurrency essential in such a theory. There has been a considerable amount of effort expended in recent years in developing theories of concurrency. This subject is rather harder than the study of purely sequential computation, because of the emergence of such phenomena as nondeterminism, deadlock and livelock. Fortunately occam is close in spirit to CSP [6], a language which has been one of the main vehicles for research into concurrency.

Since the problems of dealing with concurrency in isolation are quite considerable, many authors have chosen to omit several "conventional" programming constructs from the example languages they have used when reasoning about concurrent systems. In particular they have often omitted those constructs, such as assignment and declaration, which deal with a machine's internal state (or store). Several of the most successful theories of concurrency have been based on these "purely parallel" languages. To handle occam we need a theory which, while retaining its full capacity for dealing with concurrency, is extended to handle machine states.

This paper presents one possible approach to this problem by constructing a mathematical model for communicating processes with internal states. As a basis for our treatment of concurrency we take the "failure-sets" (or "refusal-sets") model for communicating processes, originally developed as a model for a purely parallel version of CSP. It was introduced in [4], and developed and improved in [3,5,12]. It provides a reasonably simple mathematical structure within which most of the important features of concurrency are easy to study. The fact that it was developed as a model for CSP makes it well

suited to occam. It is necessary to add to the model a mechanism for dealing with the state transformations induced by occam's "conventional" constructs. The framework chosen for this is the well-known idea of regarding a program as a relation between initial and final states. This is sufficient because it turns out that knowledge of intermediate states is not required. One of the aims when putting these models together was that on purely parallel programs the results obtained would correspond closely to the old failure-sets model, and that on sequential programs the results would be relations on states.

The first part of the paper is concerned with the construction of the revised model. The second part shows how it can be used to give a denotational semantics in the style of [11,13,14] to occam. The third section discusses a few applications of these semantics, and derives some algebraic laws relating occam terms.

Throughout this paper $\mathcal{P}(X)$ will denote the full powerset of X (the set of all subsets of X), while $\mathcal{P}(X)$ will denote the finite powerset of X (the set of all finite subsets of X). X* will denote the set of all finite sequences of elements of X. <> denotes the empty sequence, and <a,b,...,z> denotes the sequence containing a,b,...,z in that order. If $s,t \in X^*$, then st denotes the concatenation of s and t (e.g. <abc><de> = <abcde>). If $s,t \in X^*$ then $s \leq t$ (s is a *prefix* of t) if there is some $u \in X^*$ with su = t.

1. Constructing the model Our semantic domain for occam is based on the failure-sets model for communicating processes. The following is a brief summary of its construction; much fuller descriptions and motivations can be found in [3,4,5,12]. The version described here is that of [5].

The failure-sets model has as its only primitives the set Σ of atomic communications between processes. Communications are events which occur when the participating processes agree to execute them. In themselves they have no direction - there is no inputting process or outputting process. Input and output are modelled at a higher level by varying the set of a process' possible communications (an outputting process will typically have one possible communication, while an inputting process will have many). Each process communicates with its *environment*. This might be some other process or processes, or it might be some external observer. No distinction is made between these cases. We will think of a process proceeding by accepting (i.e. communicating) symbols in Σ which are offered by the environment. Only one symbol can be communicated at a time, and only finitely many in any finite time.

A process is modelled as a pair. The first component is a relation between the possible *traces* of the process (the elements of Σ^* which are the possible sequences of communications of the process up to some time) and the sets of symbols which the process can refuse to respond to after the traces (*refusals*). A *failure* of a process is a pair (s,X) where the process can refuse to communicate when offered the set X by its environment after the trace s. The first component of our representation of a process is the set of all its possible failures.

The second component is a set of traces. It represents the set of traces on which the process may *diverge*, that is, engage in an infinite unbroken sequence of internal actions. When a process diverges, it not only never communicates with its environment again, but furthermore the environment can never detect that this is so. A diverging process is bad both practically and technically, so it is desirable to differentiate between it and a process which merely stops. (We can imagine the environment being able to detect the absence of internal activity in a process, perhaps via some light on its side. If, in such a state, it does not accept communications offered to it immediately, the environment can deduce that it never will.)

The sets of failures and divergences always satisfy the laws below (see [5]). If our representation of a given process P is $\langle F,D\rangle$ where $F \subseteq \Sigma^* \times \mathbb{P}(\Sigma)$ and $D \subseteq \Sigma^*$, then

N1) traces(P) $(= \{s \in \Sigma^* : (s,\emptyset) \in F\})$ is nonempty and prefix closed (i.e. traces(P) $\neq \emptyset$, and if $s \in$ traces(P) and $t \leq s$ then $t \in$ traces(P));

N2) if $(s,X) \in F$ and $Y \subseteq X$, then $(s,Y) \in F$;

N3) if $(s,X) \in F$ and $Y \cap \{a \in \Sigma : s\langle a\rangle \in \text{traces}(P)\} = \emptyset$, then $(s,X \cup Y) \in F$;

N4) if $(s,Y) \in F$ for each $Y \in \mathcal{P}(X)$, then $(s,X) \in F$;

N5) if $s \in D$ and $t \in \Sigma^*$, then $st \in D$;

N6) if $s \in D$ and $X \subseteq \Sigma$, then $(s,X) \in F$.

The failure-sets model N is defined to be the set of all pairs $\langle F,D\rangle$ satisfying these laws.

If $P \in N$ then f(P) will denote the first component of P, and d(P) the second component. There is a natural partial order on N given by $P \sqsubseteq P'$ if and only if $f(P) \supseteq f(P')$ and $d(P) \supseteq d(P')$. If $P \sqsubseteq P'$ then we can naturally think of P' as being more deterministic than P, for it has fewer possible actions. N is a complete semilattice with respect to $\sqsubseteq$; its minimal element is $\langle \Sigma^* \times \mathbb{P}(\Sigma), \Sigma^*\rangle$ (which represents the completely unpredictable process) and its maximal elements are the *deterministic* processes. These can neither diverge nor have any choice about whether or not to accept any communication.

This model is adequate for modelling the behaviour of programs written in a purely parallel version of CSP. All the operators in CSP translate naturally to continuous functions over N. It is well suited to reasoning about the nondeterminism which arises from distributed systems and to reasoning about deadlock. Axioms N5 and N6 above correspond to the assumption that once it becomes possible for a process to diverge we do not care about its subsequent behaviour. In other words divergence is something to be avoided at all costs. The inclusion of these laws makes for considerable technical simplification at what does not seem to be a very great cost. Since the model has well-defined close links with behaviour, it is a good medium for expressing many correctness properties of processes.

Models whose only primitives are communications can be adequate for giving denotational

semantics to purely parallel languages because the only way in which one of two separate parts of a program can influence the behaviour of the other is via communication. One part of an occam program can influence another in two ways. Firstly, it can communicate along channels with its parallel partners. Secondly, it can, by assignment to common variables, influence the behaviour of its successors. Any mathematical model for occam will have to be able to deal with both these methods.

One process can communicate with another at any time before it terminates, but can only pass on its final state when it terminates successfully. (Since the sharing of variables by parallel processes is not permitted, its intermediate states cannot directly affect another process.) Successful termination has previously been modelled in purely parallel models (for example in [4,7]) by the communication of some special symbol: usually $\surd$ (pronounced "tick"). Thus all successful terminations looked the same.

Perhaps the most obvious way of letting a process pass on its final state is to have not one but many $\surd$s - one for each possible final state. If this solution were adopted then a large proportion of our alphabet of "communications" would consist of these $\surd$s. Noting that all elements of Σ have precisely the same status in the construction of N, it does seem rather unnatural to include final states in this way. Besides, there are several specific problems which arise from this treatment.

Firstly the degree of refinement required to correctly model ordinary communication seems inappropriate for the passing on of final states. It is natural to assume that if more than one final state is possible after some trace then the choice of which one is passed on is nondeterministic (i.e. outside the control of the environment). However the model as adapted above would contain elements which offer the environment a choice of final states. This would correspond to the environment controlling the internal workings of the process to a most unlikely degree.

Secondly, if the number of possible states were infinite, there would be problems in defining a continuous sequential composition operator. When a process could terminate in infinitely many different states after some trace, the "hiding" of termination by sequential composition would yield unbounded nondeterminism.

Finally, in a model where termination plays a more important role than before, the technical complexities caused by allowing nonfinal $\surd$s in traces are unacceptable (as well as being unnatural).

The solution we adopt is similar but avoids these difficulties. First, we remove $\surd$ from the traces of processes (which thus only contain "real" communications). A single symbol $\surd$ remains in the alphabet used for refusal sets, indicating that a process can refuse to terminate successfully. The second component is expanded. Instead of just recording the possible divergences, it now also records the possible states which result from successful termination. It becomes a function from Σ^* to $\wp(S) \cup \{\bot\}$, where S is the space of final states and $\bot$ represents possible divergence. Thus each process is now a pair $\langle F,T\rangle$, where $F \subseteq \Sigma^* \times \wp(\Sigma \cup \{\surd\})$ $(\surd \notin \Sigma)$ and $T:\Sigma^* \to \wp(S) \cup \{\bot\}$. Our interpretation of

the behaviour of a process $P = \langle F,T\rangle$ is as follows.

(i) F (the failures of P) lists all possible traces of the process, together with all sets of communications which it can refuse on each trace. (So that if a set of communications which is *not* a current refusal set is offered to the process, then it *must* accept some element.)

(ii) One of the possible elements of the refusal sets is $\surd$ - this indicates that the process may fail to terminate successfully (even though there may be some final states possible for the given trace). Thus it is possible to discriminate between a process which will always terminate successfully and one which may nondeterministically deadlock *or* terminate successfully. Termination *must* take place (if desired by the environment) only when the set $\{\surd\}$ cannot be refused.

(iii) Termination *can* take place on any trace s for which T(s) is a nonempty set of states. When T(s) contains more than one element the choice of which final state occurs is nondeterministic. (T will be referred to as the termination component of P.)

(iv) If $T(s) = \perp$, then the process is considered to be broken. We allow the possibility that it might diverge or do anything else at all.

For a given alphabet Σ and a set S of final states, the **space** Q of all processes is thus the set of all pairs $P = \langle F,T\rangle$ $(F \subseteq (\Sigma^* \times \mathbb{P}(\Sigma \cup \{\surd\}))$, $T:\Sigma^* \to \mathbb{P}(S) \cup \{\perp\})$ which satisfy the following eight laws.

F1) traces(P) $(= \{s \in \Sigma^* : (s,\emptyset) \in F\})$ is nonempty and prefix closed;

F2) $(s,X) \in F \;\&\; Y \subseteq X \;\Rightarrow\; (s,Y) \in F$;

F3) $(s,X) \in F \;\&\; Y \subseteq \{a \in \Sigma : s\langle a\rangle \notin \text{traces}(P)\} \;\Rightarrow\; (s,X \cup Y) \in F$;

F4) if $(s,Y) \in F$ for each $Y \in \mathbb{P}(X)$, then $(s,X) \in F$;

T1) $T(s) \neq \emptyset \;\Rightarrow\; s \in \text{traces}(P)$;

T2) $(s,X) \in F \;\&\; T(s) = \emptyset \;\Rightarrow\; (s,X \cup \{\surd\}) \in F$;

T3) $T(s) = \perp \;\&\; t \in \Sigma^* \;\Rightarrow\; T(st) = \perp$;

T4) $T(s) = \perp \;\&\; X \subseteq \Sigma \cup \{\surd\} \;\Rightarrow\; (s,X) \in F$.

In the above s,t range over Σ^* and X,Y range over $\mathbb{P}(\Sigma \cup \{\surd\})$.

These laws are just the natural extension of the laws governing N to the revised structure.

If $P = \langle F,T\rangle$, then define f(P) = F and t(P) = T. We will adopt the conventions that $A \subseteq \perp$ and $A \cup \perp = \perp$ for all $A \subseteq S$, and that $\sigma \in \perp$ for all $\sigma \in S$.

The new model clearly has a great deal in common with the old one. On the assumption that S has no important partial order of its own, Q has a natural partial order:

$$P \sqsubseteq P' \;\Leftrightarrow\; f(P) \supseteq f(P') \;\&\; \forall s \in \Sigma^*.\ t(P)s \supseteq t(P')s.$$

$P \sqsubseteq P'$ can again be interpreted as meaning that P' is more deterministic than P. With respect to "$\sqsubseteq$", Q is a complete semilattice whose minimal element is $\langle \Sigma^* \times \mathcal{P}(\Sigma \cup \{\checkmark\}), T \rangle$ (denoted $\perp_Q$), where $T_\perp(s) = \perp$ for all $s \in \Sigma^*$. $\perp_Q$ is the completely unpredictable process; it may diverge immediately. The maximal elements of Q are again the deterministic processes, which are divergence free and never have any internal decisions to make. A process P is deterministic if and only if it satisfies

$$(s,X) \in f(P) \Rightarrow X \cap \{a \in \Sigma : s\langle a \rangle \in traces(P)\} = \emptyset$$

and $t(P)s \neq \emptyset \Rightarrow (s,\{\checkmark\}) \notin f(P)$ & $t(P)s$ is a singleton set.

The assumption that all sets of final states are finite corresponds closely to an assumption of bounded nondeterminism. As remarked earlier, this is necessary to make sequential composition easy to deal with. Of course, if the set S of states is finite, this assumption is vacuous.

There is no concept of time in our model. Thus the occam timing constructs (NOW and WAIT) cannot be modelled directly. The other main feature lacking is an analogy of prioritised ALT: there is no way of telling from our model that a process would rather communicate "a" than "b" (say). It seems to be necessary to have a model which includes time before one can handle priority in a fully satisfactory way. One could handle time in the present model by adding a "clock" process to every system which processes communicate with when they want to know the time. Unfortunately this solution does not prove to be fully satisfactory, and forces untidy semantic definitions which are, in some sense, at the wrong level of abstraction. We therefore omit timing and priority from our language, and pose the problem of the introduction of time as a topic for future research.

2. Denotational semantics In this section we see how the model we have constructed can be used to give a natural denotational semantics to a large subset of occam. Having constructed what is essentially a hybrid model, one might expect to be able to adapt work on purely parallel and sequential languages. This does indeed turn out to be the case, as there are few parts of the language which make demands on both aspects of the model.

In the previous section we discussed an abstract set S consisting of states. Nothing was assumed about S except (tacitly) that it did not use the space of processes in its definition (for then we would have required a recursive domain definition) and that it did not carry an important partial order with it. In devising our space of machine states we need to bear in mind the role they play in our model: passing on information from one occam process to its successor. The only way one occam process can influence its successors is by modifying (though assignment or input) the values of variables: it cannot change the binding of identifiers in any other way. Thus our states will be "stores" - functions from locations to storable values. A separate *environment* will be used to map identifiers to locations, constants, channels, procedures and so on.

This distinction between environment and state is a familiar idea in denotational semantics; the way the present model works makes it natural to adopt the same distinction here. The management of environments and states relative to sequential languages is well understood. In translating the idea to occam there is only one place where difficulties arise: we must decide how environments and states behave under the parallel operator. Parallel occam processes do not use shared variables for communication. They can only use global variables in a very restricted way: *either* only one process can use a given variable normally *or* all processes can read (but not write to) it. This idea corresponds to giving each parallel process a distinct portion of the state and reserving the rest as read only for the life of the parallel command. The state will be reconstructed at the end of a parallel construct by the "distributed termination" property of occam processes - a PAR construct can only terminate when each of its components can terminate (and thus yield its own component of the final state).

In order to construct our model we also need to know the structure of the alphabet of communications between processes. In occam each communication has two components - an element (or word value, the same as a storable value) and a channel. We will thus postulate the existence of sets CHAN and $\mathbb{B}$ of (respectively) channels and elements. The alphabet of communications is then $\Sigma = \{\chi.\beta : \chi \in \text{CHAN}, \beta \in \mathbb{B}\}$. Note that we make no distinction between "input" and "output" communications. The fact that $\mathbb{B}$ is in practice finite is useful to us, since the semantics of *hiding* (necessary for the correct definition of PAR) are much easier in this case. We will therefore assume that $\mathbb{B}$ is finite. If χ is any channel then $\chi.\mathbb{B}$ will denote the set $\{\chi.\beta : \beta \in \mathbb{B}\}$.

In the conclusion we will indicate how the semantics can be adapted to cope with an infinite $\mathbb{B}$, and also describe a way in which the model Q might be altered to take advantage of the special structure of the occam alphabet.

Our language The version of occam used in this paper differs slightly from the occam of [10]. We have already remarked that we will omit timing and priority. We also omit certain non-central features of occam such as those involving configuration. This particular omission is justified by the argument that since the logical behaviour of a program is independent of its configuration, so also should be its semantics. Minor omissions in the cause of simplicity are BYTE subscription, vector operations (slices) and vectors of constants. An additional atomic process STOP has been added, and the semantics of IF has been altered, to be consistent with the latest releases of occam.

The chief change we make in occam syntax is to insist that parallel processes declare which global channels and variables they want to use. Given the restrictions placed on their use this seems good practice. These restrictions (particularly where components of vectors are concerned) would not otherwise be syntactically checkable. Problems arise with the associativity of PAR (one of the most desirable algebraic laws) if these declarations are not made. Any truly parallel implementation of PAR will need to be able to determine the allocation of global variables and channels at compile-time.

For ease of presentation occam syntax has been linearised in this paper. For example we write $SEQ(P_1,P_2,\ldots,P_n)$ instead of

SEQ
  P_1
  P_2
  :
  P_n

A detailed formal syntax of occam is given in the occam programming manual [10]. Rather than duplicate the definitions of the subsidiary syntactic domains, we give instead a summary of the differences between the standard occam versions and those of this paper. We give each syntactic domain a name and a notation for its typical element.

Expressions ($e \in Exp$) The syntax of these is the same as in [10], except for the omission of NOW and BYTE.

Vector operations are omitted.

Declarations ($\Delta \in Decl$) The syntax of declarations is the same except for the omission of BYTE subscription and vectors of constants.

Formal parameter lists ($\Phi \in Form$) These are the same as in [10].

Guards ($g \in Guard$) There are no WAIT guards. For brevity all guards are assumed to contain a boolean expression (which could of course be TRUE).

The following are new syntactic domains.

Actual parameter lists ($A \in Act$) It is convenient to have a name for these objects, which are just lists of expressions.

Parallel declarations ($U \in PD$) We insist that a parallel process should declare which global channels it intends to use, dividing them into three categories.
OWNCHAN means that the channel(s) are for internal use by the process.
INCHAN means that the channel(s) are to be used by the process for inputting.
OUTCHAN means that the channel(s) are to be used by the process for outputting.
We also insist that the process declares which global channels it wants to assign to.

parallel.declaration = USING({*claim*})

claim = OWNCHAN *chan* {, *chan*}
| INCHAN *chan* {, *chan*}
| OUTCHAN *chan* {, *chan*}
| VAR *var* {, *var*}

Processes (or programs) ($P \in Proc$) The definition of our syntactic domain of processes is given below. The only differences from the domain defined in [10] are those which have already been described.

process = STOP
| SKIP
| *variable* := *expression*
| *channel* ?*variable*
| *channel* ? ANY
| *channel* ! *expression*
| *channel* ! ANY
| *identifier* (*actual.parameter.list*)
| SEQ({*process*})
| PAR({*parallel.declaration* : *process*})
| ALT({*guarded.process*})
| IF({*conditional*})
| SEQ *replicator process*
| PAR *replicator parallel.declaration* : *process*
| ALT *replicator guarded.process*
| IF *replicator conditional*
| WHILE *expression process*
| *declaration* : *process*

replicator = *identifier* = [*expression* FOR *expression*]

guarded.process = *guard process*
| ALT({*guarded.process*})
| ALT *replicator guarded.process*

conditional = *expression process*
| IF({*conditional*})
| IF *replicator conditional*

Semantic domains The semantic domains whose existence we postulate are the following.

$\beta \in B$ the (finite) domain of basic storable values or elements. We assume that each element can be identified with some positive or negative integer.

$\chi \in$ CHAN the domain of channels.

$\lambda \in$ LOC the domain of locations in store.

p,x $\in$ IDE the (syntactic) domain of identifiers.

There is no need in this work to suppose that any of the above domains is partially ordered or contains a "bottom" element. We will, however, need to deal with errors. Given any semantic domain X, we will denote by X^+ the domain $X \cup \{error\}$. If X is partially ordered then "error" will be incomparable with the other elements of X^+. Given a domain X, X^V will denote the domain of vectors of elements of X. We will regard an element of X^V as being a function from $\{0,1,\ldots,n-1\}$ to X for some non-negative integer n (the vector's length). If X has a partial order then vectors of the same length are ordered component-wise, vectors of different lengths being incomparable.

Given this notation we can construct a few semantic domains.

$$S = LOC \to \mathbb{B}^+$$

S is the domain of machine states. Each location is mapped either to a storable value or to error (perhaps indicating that the location is uninitialised).

$$ENV = (IDE \to D^+) \times LSTATUS \times CSTATUS$$

$$D = LOC + \mathbb{B} + CHAN + NP + LOC^V + CHAN^V$$

$$LSTATUS = LOC \to \{\underline{x}, \underline{u}, \underline{r/w}, \underline{ro}\}$$

$$CSTATUS = CHAN \to \{\underline{x}, \underline{u}, \underline{ud}, \underline{in}, \underline{out}\}$$

ENV is the domain of environments. The first component of each environment is a function from identifiers to denotable values (plus error). A denotable value ($\delta \in D$) is either a single location (corresponding to a non-vector variable) or an element (corresponding to a constant) or a channel or a named process (the domain NP will be defined later, when it is used) or a vector of locations (corresponding to a vector of variables) or a vector of channels. The second component of each environment gives the status of each location.

- $\underline{x}$ means that the location is not in the environment's range;
- $\underline{u}$ means that it is in the range but is unused;
- $\underline{r/w}$ means that it is in range and is in normal use *(read/write)*;
- $\underline{ro}$ means that it is in the range but has "read only" status.

The third component of each environment gives the status of each channel.

- $\underline{x}$ means that the channel is not in the environment's range;
- $\underline{u}$ means that it is in the range but is unused;
- $\underline{ud}$ means that it is in use but has not been assigned a direction;
- $\underline{in}$ means that it is in use as an input channel;
- $\underline{out}$ means that it is in use as an output channel.

If $\rho \in ENV$ then ρ_i ($i \in \{1,2,3\}$) will denote its ith component (so that $\rho = \langle \rho_1, \rho_2, \rho_3 \rangle$). If $x \in IDE$ then $\rho[\![x]\!]$ will mean $\rho_1[\![x]\!]$; similarly $\rho[\lambda]$ will mean $\rho_2[\lambda]$ ($\lambda \in LOC$) and $\rho[\chi]$ will mean $\rho_3[\chi]$ ($\chi \in CHAN$). If $x \in IDE$ and $\delta \in D^+$ then $\rho[\delta/x]$ will denote the environment which is the same as ρ except for mapping x to δ. The corresponding interpretations will be put on $\rho[r/\lambda]$, $\rho[r/\chi]$ and $\sigma[\beta/\lambda]$.

- $A \in Q$ the domain of processes is constructed as in the previous section, using **S** as the set of states and $\Sigma = \{\chi.\beta : \chi \in CHAN, \beta \in \mathbb{B}\}$ as the alphabet.

A few further semantic domains will be defined later, when they are required.

The semantics We will only give detailed definitions of the "higher level" semantic functions required. The other ones are all fairly standard and should not prove too hard for the diligent reader to define. The main semantic functions are listed below.

$$\mathcal{C} : \text{Proc} \to \text{ENV} \to \text{S} \to \text{Q}$$

This the main semantic function. Given a program segment, an environment and a state it yields an element of Q. In the definitions below all execution errors map a process to the minimal **element** $\perp_Q$ of Q (from the point in its communication history where the error arises). Thus an erroneous process is identified with a diverging one. We are thus allowing that erroneous processes might do anything. There is no reason why more sophisticated semantics could not be devised which allowed for a certain amount of error recovery, perhaps by introducing extra elements into the semantic domain. Our present approach has the advantage of simplicity, though.

$$\mathcal{F} : \text{Proc} \to \text{ENV} \to \text{S} \to \mathcal{P}(\Sigma^* \times \mathcal{P}(\Sigma \cup \{\checkmark\}))$$

$$\mathcal{T} : \text{Proc} \to \text{ENV} \to \text{S} \to (\Sigma^* \to p(\text{S}) \cup \{\perp\})$$

These two functions pick out the failure and termination components of $\mathcal{C}[\![P]\!]\rho\sigma$, so that $\mathcal{C}[\![P]\!]\rho\sigma = \langle \mathcal{F}[\![P]\!]\rho\sigma, \mathcal{T}[\![P]\!]\rho\sigma \rangle$.

$$\mathcal{D} : \text{Decl} \to \text{ENV} \to \text{ENV}^+$$

This function carries out the modifications to the environment caused by declarations. When an error occurs the value produced is "error".

$$\mathcal{E} : \text{Exp} \to \text{ENV} \to \text{S} \to \mathbb{B}^+$$

This function evaluates expressions as elements of $\mathbb{B}$. The error element results when something goes wrong (e.g. vector subscript out of bounds). Its definition is completely standard and is omitted.

$$\text{lv} : \text{Exp} \to \text{ENV} \to \text{S} \to \text{LOC}^+$$

$$\text{cv} : \text{Exp} \to \text{ENV} \to \text{S} \to \text{CHAN}^+$$

These are important auxiliary functions which help reduce the number of cases in higher level definitions. lv and cv produce (respectively) the values as locations and channels of expressions which are meant to denote them. They take value "error" when an expression cannot be given the relevant interpretation. lv is defined below, the definition of cv being very similar.

$$\text{lv}[\![x]\!]\rho\sigma = \rho[\![x]\!] \quad \text{if } \rho[\![x]\!] \in \text{LOC};$$

$$\text{lv}[\![x[e]]\!]\rho\sigma = \rho[\![x]\!](\mathcal{E}[\![e]\!]\rho\sigma) \quad \text{if } \rho[\![x]\!] \in \text{LOC}^{\text{V}} \text{ and } \mathcal{E}[\![e]\!]\rho\sigma \in \text{dom}(\rho[\![x]\!]);$$

$$\text{lv}[\![e]\!]\rho\sigma = \text{error} \quad \text{otherwise.}$$

A few more specialised semantic functions will be defined later, when required.

We will now concentrate on the definition of $\mathcal{C}$, the main semantic function. Each of the clauses is given a brief explanation. Many of the operators used are very similar to ones used over the failure-sets model in giving semantics to CSP. The construction of these is explained in detail in [4,5]. Many of the clauses contain one or more conditions "*provided* ....." which exclude error conditions. When these conditions are not met the value of the clause is always $\perp_Q$. Several clauses are split into separate

definitions of $\mathcal{F}$ and $\mathcal{T}$.

$\mathcal{F}[\![\text{STOP}]\!]\rho\sigma = \{(\langle\rangle, X) : X \subseteq \Sigma \cup \{\surd\}\}$

$\mathcal{T}[\![\text{STOP}]\!]\rho\sigma(s) = \emptyset \qquad$ (for all $s \in \Sigma^*$)

STOP never communicates or terminates. It just refuses everything offered to it.

$\mathcal{C}[\![\text{SKIP}]\!]\rho\sigma = \text{skip}_\sigma$, where

$$f(\text{skip}_\sigma) = \{(\langle\rangle, X) : \surd \notin X\}$$
$$t(\text{skip}_\sigma)s = \{\sigma\} \quad \text{if } s = \langle\rangle$$
$$= \emptyset \quad \text{otherwise.}$$

skip_σ is the element of Q which never communicates, but which must immediately terminate successfully in final state σ.

$\mathcal{C}[\![e_1 := e_2]\!]\rho\sigma = \text{skip}_{\sigma'}$, where *provided*

$\sigma' = \sigma[\mathcal{E}[\![e_2]\!]\rho\sigma / \text{lv}[\![e_1]\!]\rho\sigma]$ $\quad \mathcal{E}[\![e_2]\!]\rho\sigma \neq \text{error}$, $\text{lv}[\![e_1]\!]\rho\sigma \neq \text{error}$, *and* $\rho[\text{lv}[\![e_1]\!]\rho\sigma] = \underline{r/w}$

This process also terminates immediately, but modifies its final state to take account of the assignment.

$\mathcal{F}[\![e_1?e_2]\!]\rho\sigma = \{(\langle\rangle, X) : X \cap \chi.\mathbb{B} = \emptyset\} \cup \{(\langle\chi.\beta\rangle, X) : \surd \notin X \ \&\ \beta \in \mathbb{B}\}$

$\mathcal{T}[\![e_1?e_2]\!]\rho\sigma(s) = \{\sigma[\beta/\lambda]\} \quad \text{if } s = \langle\chi.\beta\rangle \quad (\beta \in \mathbb{B})$
$= \emptyset \quad \text{otherwise}$

where $\chi = \text{cv}[\![e_1]\!]\rho\sigma$ and $\lambda = \text{lv}[\![e_2]\!]\rho\sigma$ *provided* $\text{cv}[\![e_1]\!]\rho\sigma \neq \text{error}$, $\text{lv}[\![e_2]\!]\rho\sigma \neq \text{error}$, $\rho[\lambda] = \underline{r/w}$ *and* $\rho[\chi] = \underline{in}$.

On its first step this process is prepared to communicate anything on channel χ (it cannot refuse any element of $\chi.\mathbb{B}$). After communicating $\chi.\beta$ ($\beta \in \mathbb{B}$) it terminates with β substituted for λ.

$\mathcal{F}[\![e?\text{ANY}]\!]\rho\sigma = \{(\langle\rangle, X) : X \cap \chi.\mathbb{B} = \emptyset\} \cup \{(\langle\chi.\beta\rangle, X) : \surd \notin X\}$

$\mathcal{T}[\![e?\text{ANY}]\!]\rho\sigma(s) = \{\sigma\} \quad \text{if } s = \langle\chi.\beta\rangle \quad (\beta \in \mathbb{B})$
$= \emptyset \quad \text{otherwise}$

where $\chi = \text{cv}[\![e]\!]\rho\sigma$ *provided* $\text{cv}[\![e]\!]\rho\sigma \neq \text{error}$ *and* $\rho[\chi] = \underline{in}$.

This process is the same as the previous one except that it terminates with unchanged state.

$\mathcal{F}[\![e_1!e_2]\!]\rho\sigma = \{(\langle\rangle, X) : \chi.\beta \notin X\} \cup \{(\langle\chi.\beta\rangle, X) : \surd \notin X\}$

$\mathcal{T}[\![e_1!e_2]\!]\rho\sigma(s) = \{\sigma\} \quad \text{if } s = \langle\chi.\beta\rangle$
$= \emptyset \quad \text{otherwise}$

where $\beta = \mathcal{E}[\![e_2]\!]\rho\sigma$ and $\chi = \text{cv}[\![e_1]\!]\rho\sigma$ *provided* $\mathcal{E}[\![e_2]\!]\rho\sigma \neq \text{error}$, $\text{cv}[\![e_1]\!]\rho\sigma \neq \text{error}$ *and* $\rho[\chi] = \underline{out}$.

This process communicates the value of the expression e_2 along the output channel denoted by e_1, and then terminates in an unchanged state.

$$\mathcal{F}[\![e!ANY]\!]\rho\sigma = \{(<>,X) : \chi.\beta \notin X\} \cup \{(<\chi.\beta>,X) : \beta \in \mathbb{B} \ \& \ \checkmark \notin X\}$$

$$\mathcal{T}[\![e!ANY]\!]\rho\sigma(s) = \{\sigma\} \quad \text{if } s = <\chi.\beta> \quad (\beta \in \mathbb{B})$$
$$= \emptyset \quad \text{otherwise}$$

where $\chi = cv[\![e]\!]\rho\sigma$ *provided* $cv[\![e]\!]\rho\sigma \neq$ error *and* $\rho[\chi] =$ *out*.

This process communicates any value at all along the output channel denoted by e. The choice of value is nondeterministic. (Note that the process can refuse any *proper* subset of $\chi.\mathbb{B}$.) It then terminates with unchanged state. *This definition would need to be changed if* $\mathbb{B}$ *were infinite, in order to satisfy axiom F4.*

Thus each primitive process has a simple interpretation in the model.

$$\mathcal{C}[\![SEQ(P_1,P_2,\ldots,P_n)]\!]\rho\sigma = skip_\sigma \quad \text{if } n = 0, \text{ and otherwise}$$
$$= \underline{seq}(\mathcal{C}[\![P_1]\!]\rho\sigma, \mathcal{C}[\![SEQ(P_2,\ldots,P_n)]\!]\rho).$$

Here $\underline{seq}$ is the function from $Q \times (S \to Q)$ to Q which is defined

$$f(\underline{seq}(A,B)) = \{(s,X) : (s,X \cup \{\checkmark\}) \in f(A)\}$$
$$\cup \{(su,X) : \exists\sigma'.\ \sigma' \in t(A)s \ \& \ (u,X) \in f(B\sigma')\}$$
$$\cup \{(su,X) : t(A)s = \bot\}$$

$$t(\underline{seq}(A,B))s = \bot \text{ if } t(A)s = \bot, \text{ or if } s = uv \text{ where } \sigma' \in t(A)u \text{ and } t(B\sigma')v = \bot$$
$$= \bigcup\{t(B\sigma')v : \exists u,v,\sigma'.\ s = uv \ \& \ \sigma' \in t(A)u\} \quad \text{otherwise.}$$

If n = 0 then $SEQ(P_1,P_2,\ldots,P_n)$ behaves exactly like SKIP (terminating immediately in an unaltered state). Otherwise process P_1 is run until it terminates successfully, the final state of P_1 being given as the initial state to $SEQ(P_2,\ldots,P_n)$. Note that P_1 cannot refuse a set X of communications unless it can refuse $X \cup \{\checkmark\}$; otherwise it would be able to terminate (invisibly) and let $SEQ(P_2,\ldots,P_n)$ take over.

The semantics of ALT are made rather difficult by the presence of SKIP guards. This is because the two types of guard work in quite different ways. We need an extra semantic function which tells us whether any SKIP guard is ready. (In this section b,c and e will respectively denote boolean, channel and other expressions. G will be a typical guarded process.)

$$\mathcal{R}[\![b \,\&\, c?e \ P]\!]\rho\sigma = \underline{false}$$

$$\mathcal{R}[\![b \,\&\, SKIP \ P]\!]\rho\sigma = \underline{true} \quad \text{if } \mathcal{E}[\![b]\!]\rho\sigma \text{ is "true"}$$
$$= \underline{false} \quad \text{if } \mathcal{E}[\![b]\!]\rho\sigma \text{ is "false"}$$
$$= \text{error} \quad \text{otherwise}$$

$$\mathcal{R}[\![ALT(G_1,..,G_n)]\!]\rho\sigma = \bigvee_{i=1}^{n} \mathcal{R}[\![G_i]\!]\rho\sigma \quad \text{where } \bigvee \text{ is the error-strict version of the usual boolean "or".}$$

The clause for replicator guarded processes is similar.

It is convenient to extend the domain of $\mathcal{C}$ to include all guarded processes.

$$\mathcal{C}[\![b \,\&\, c?e\ P]\!]\rho\sigma = \text{stop} \qquad \text{if } \mathcal{E}[\![b]\!]\rho\sigma \text{ is "false"}$$
$$= \mathcal{C}[\![SEQ(c?e,P)]\!]\rho\sigma \qquad \text{if } \mathcal{E}[\![b]\!]\rho\sigma \text{ is "true"}$$

$$\mathcal{C}[\![b \,\&\, SKIP\ P]\!]\rho\sigma = \text{stop} \qquad \text{if } \mathcal{E}[\![b]\!]\rho\sigma \text{ is "false"}$$
$$= \mathcal{C}[\![P]\!]\rho\sigma \qquad \text{if } \mathcal{E}[\![b]\!]\rho\sigma \text{ is "true"}$$

where stop = $\mathcal{C}[\![STOP]\!]\rho\sigma$ (this value is independent of ρ and σ)

provided in each case that $\mathcal{E}[\![b]\!]\rho\sigma$ *can be regarded as a boolean.*

$$\mathcal{F}[\![ALT(G_1,\ldots,G_n)]\!]\rho\sigma = \{(<>,X) : \exists i.\ \mathcal{R}[\![G_i]\!]\rho\sigma \ \&\ (<>,X) \in \mathcal{F}[\![G_i]\!]\rho\sigma\}$$
$$\cup \{(<>,X) : \forall i.\ \neg\mathcal{R}[\![G_i]\!]\rho\sigma \ \&\ \forall i.(<>,X) \in \mathcal{F}[\![G_i]\!]\rho\sigma\}$$
$$\cup \{(<>,X) : \exists i.\ \mathcal{T}[\![G_i]\!]\rho\sigma(<>) = \bot\}$$
$$\cup \{(s,X) : s \neq <> \ \&\ \exists i.(s,X) \in \mathcal{F}[\![G_i]\!]\rho\sigma\}$$

$$\mathcal{T}[\![ALT(G_1,\ldots,G_n)]\!]\rho\sigma(s) = \bigcup\{\mathcal{T}[\![G_i]\!]\rho\sigma(s) : i \in \{1,\ldots,n\}\}$$

If any SKIP guard is ready then the process may choose (invisibly) to behave like the corresponding guarded process. If no SKIP guard is ready then the process must wait for something to be communicated to **it** along one of the channels of the c?e guards. Note that if none of its boolean expressions evaluates to "true", then $ALT(G_1,\ldots,G_n)$ is equivalent to STOP.

To allow for the possibility of nested "IF"s we need to adopt a similar technique for conditionals: we define a semantic function to determine whether or not a given conditional is "ready". (In this section C will denote a typical conditional.)

$$\mathcal{J}[\![e\ P]\!]\rho\sigma = \underline{\text{true}} \qquad \text{if } \mathcal{E}[\![e]\!]\rho\sigma \text{ is "true"}$$
$$= \underline{\text{false}} \qquad \text{if } \mathcal{E}[\![e]\!]\rho\sigma \text{ is "false"}$$
$$= \text{error} \qquad \text{otherwise}$$

$$\mathcal{J}[\![IF(C_1,\ldots,C_n)]\!]\rho\sigma = \bigvee_{i=1}^{n} \mathcal{J}[\![C_i]\!]\rho\sigma \qquad \text{(once again } \vee \text{ is error-strict),}$$

the clause for replicators being similar.

$$\mathcal{C}[\![IF(C_1,\ldots,C_n)]\!]\rho\sigma = \text{stop} \qquad \text{if } n = 0,$$
$$= \mathcal{C}[\![P]\!]\rho\sigma \qquad \text{if } C_1 = e\ P \text{ and } \mathcal{E}[\![e]\!]\rho\sigma \text{ is "true",}$$
$$= \mathcal{C}[\![C_1]\!]\rho\sigma \qquad \text{if } C_1 \in Proc \text{ and } \mathcal{J}[\![C_1]\!]\rho\sigma = \underline{\text{true}},$$
$$= \mathcal{C}[\![IF(C_2,\ldots,C_n)]\!]\rho\sigma \quad \text{otherwise,}$$

provided $\mathcal{J}[\![IF(C_1,\ldots,C_n)]\!]\rho\sigma \neq \text{error}$.

The only form of recursion allowed in occam is the WHILE loop. The fact that the following definition works, essentially only depends on the fact that seq is continuous in its second argument.

$$\mathcal{C}[\![WHILE\ e\ P]\!]\rho\sigma = (\bigsqcup_{n=0} F^n(\bot_{S\to Q}))\sigma, \qquad \text{where } F : (S \to Q) \to (S \to Q) \text{ is the function defined}$$
$$F(B)\sigma' = \underline{\text{seq}}(\mathcal{C}[\![P]\!]\rho\sigma', B) \quad \text{if } \mathcal{E}[\![b]\!]\rho\sigma' \text{ is "true"}$$
$$= \text{skip}_{\sigma'} \qquad \text{if } \mathcal{E}[\![b]\!]\rho\sigma' \text{ is "false"}$$
$$= \bot_Q \qquad \text{otherwise.}$$

In fact it turns out that *all* our operators are continuous, so that there is no reason why these semantics should not be extended to more ambitious forms of recursion. (The failure-sets model has been used to reason about recursions through parallel and hiding operators [4,12].)

The parallel operator is understandably the most complicated to define. The first part of our definition shows how the parallel declarations are used to set up local environments and states for each individual process. The second part shows how the processes interact once they are running. Overall we have

$$\mathcal{C}[\![\text{PAR}(U_1{:}P_1,\ldots,U_n{:}P_n)]\!]\rho\sigma = \text{skip}_\sigma \quad \text{if } n=0, \text{ and otherwise}$$
$$= \|_{i=1}^{n} (\mathcal{C}[\![P_i]\!]\rho_i\sigma_i, X_i) \;/\; Y.$$

The processes are run in parallel($\|$) with their respective alphabets (X_i), environments (ρ_i) and states (σ_i). The communications local to the network are hidden (/Y). These items are all defined below.

The following basic semantic functions are not hard to define, but their definitions are omitted for brevity.

$$\text{inchans} : \text{PD} \to \text{ENV} \to \text{S} \to \wp(\text{CHAN})^+$$
$$\text{outchans} : \text{PD} \to \text{ENV} \to \text{S} \to \wp(\text{CHAN})^+$$
$$\text{ownchans} : \text{PD} \to \text{ENV} \to \text{S} \to \wp(\text{CHAN})^+$$
$$\text{locs} : \text{PD} \to \text{ENV} \to \text{S} \to \wp(\text{LOC})^+$$

These functions extract respectively the sets of input, output and internal channels and locations claimed by a parallel declaration (USING(.)). *To be declared as an input channel by* $\text{inchans}[\![U_i]\!]\rho\sigma$, χ *must have status* ud *or* in *in* ρ; *output channels must have status* ud *or* out; *internal channels must have status* ud; *locations must have status* r/w. *If an undirected (*ud*) channel of* ρ *is declared as an input (output) channel by one of the* U_i, *then it must be declared as an output (input) channel by another. In addition, the* U_i *must satisfy the following:*

$$\text{inchans}[\![U_i]\!]\rho\sigma \cap \text{ownchans}[\![U_j]\!]\rho\sigma = \emptyset$$
$$\text{outchans}[\![U_i]\!]\rho\sigma \cap \text{ownchans}[\![U_j]\!]\rho\sigma = \emptyset$$
$$\text{inchans}[\![U_i]\!]\rho\sigma \cap \text{outchans}[\![U_i]\!]\rho\sigma = \emptyset$$
$$\text{inchans}[\![U_i]\!]\rho\sigma \cap \text{inchans}[\![U_j]\!]\rho\sigma = \emptyset \quad \textit{whenever } i \neq j$$
$$\text{outchans}[\![U_i]\!]\rho\sigma \cap \text{outchans}[\![U_j]\!]\rho\sigma = \emptyset \quad \textit{whenever } i \neq j$$
$$\text{locs}[\![U_i]\!]\rho\sigma \cap \text{locs}[\![U_j]\!]\rho\sigma = \emptyset \quad \textit{whenever } i \neq j$$

We define $X_i = \bigcup\{\chi.\mathcal{B} : \chi \in \text{inchans}[\![U_i]\!]\rho\sigma \cup \text{outchans}[\![U_i]\!]\rho\sigma\}$
and $Y = \bigcup\{X_i \cap X_j : i \neq j\}$.

(Note that the above disjointness conditions, which enforce the separation of occam parallel processes, imply that $X_i \cap X_j \cap X_k = \emptyset$ *whenever* i,j,k *are all different.)*

The first component of each ρ_i is the same as that of ρ, and

$$\begin{aligned}
\rho_i[\lambda] &= \underline{r/w} && \text{if } \lambda \in \text{locs}[\![U_i]\!]\rho\sigma \\
&= \underline{ro} && \text{if } \rho[\lambda] \in \{\underline{ro}, \underline{r/w}\} \text{ and } \lambda \notin \bigcup_{j=1}^{n} \text{locs}[\![U_j]\!]\rho\sigma \\
&= \underline{x} && \text{if } \rho[\lambda] = \underline{x} \text{ or } \lambda \in \text{locs}[\![U_j]\!]\rho\sigma \text{ for some } i \neq j \\
&\in \{\underline{x}, \underline{u}\} && \text{if } \rho[\lambda] = \underline{u}, \text{ subject to } \rho_i[\lambda] = \underline{u} \Rightarrow \rho_j[\lambda] = \underline{x} \text{ whenever } i \neq j.
\end{aligned}$$

The last line says that a free location in ρ becomes either free in ρ_i or outside the domain of ρ_i, but that no such location becomes free in more than one ρ_i. If stores are infinite then we will assume that infinitely many free locations are allocated to each ρ_i.

$$\begin{aligned}
\rho_i[\chi] &= \underline{ud} && \text{if } \chi \in \text{ownchans}[\![U_i]\!]\rho\sigma \\
&= \underline{in} && \text{if } \chi \in \text{inchans}[\![U_i]\!]\rho\sigma \\
&= \underline{out} && \text{if } \chi \in \text{outchans}[\![U_i]\!]\rho\sigma \\
&\in \{\underline{x}, \underline{u}\} && \text{if } \rho[\chi] = \underline{u}, \text{ subject to } \rho_i[\chi] = \underline{u} \Rightarrow \rho_j[\chi] = \underline{x} \text{ whenever } i \neq j \\
&= \underline{x} && \text{otherwise.}
\end{aligned}$$

If the supply of channels is infinite then infinitely many are allocated to each ρ_i.

$$\begin{aligned}
\sigma_i[\lambda] &= \sigma[\lambda] && \text{if } \lambda \notin \bigcup_{j \neq i} \text{locs}[\![U_j]\!]\rho\sigma \\
&= \text{error} && \text{otherwise.}
\end{aligned}$$

This completes the definitions of the local environments and states. The parallel operator (||) and hiding operator (/Y) defined below are derived from the CSP operators in [5]. (Below, $A_i \in Q$, $X_i \subseteq \Sigma$, $X = \bigcup_{i=1}^{n} X_i$.)

$$\begin{aligned}
f(\|_{i=1}^{n}(A_i,X_i)) = \{(s,S) : \exists Y_1,..,Y_n.\ & s \in X^* \ \&\ (s\lceil X_i, Y_i) \in f(A_i) \\
& \&\ S \cap (X \cup \{\surd\}) = \bigcup\{Y_i \cap (X_i \cup \{\surd\}) : i \in \{1,2,\ldots,n\}\}\} \\
\cup \{(su,S) : s \in X^* \ \&\ & s\lceil X_i \in \text{traces}(A_i) \ \&\ \exists i.t(A_i)(s\lceil X_i) = \bot\}
\end{aligned}$$

$$\begin{aligned}
t(\|_{i=1}^{n}(A_i,X_i))s &= \bot && \text{if there exists } u \leq s \text{ such that } u \in X^*, u\lceil X_i \in \text{traces}(A_i) \text{ for all } i \\
& && \text{and there is some } j \text{ with } t(A_j)(u\lceil X_j) = \bot, \\
&= \{\text{join}(\sigma_1,..,\sigma_n) : s \in X^* \ \&\ \sigma_i \in t(A_i)(s\lceil X_i)\} && \text{otherwise.}
\end{aligned}$$

Here, $s\lceil X$ is the restriction of trace s to the set X, so that

$$\begin{aligned}
\langle\rangle\lceil X &= \langle\rangle \qquad \text{and} \quad s\langle a\rangle\lceil X = s\lceil X && \text{if } a \notin X \\
&= (s\lceil X)\langle a\rangle && \text{if } a \in X.
\end{aligned}$$

Given states $\sigma_1,\ldots,\sigma_n$, $\text{join}(\sigma_1,\ldots,\sigma_n)$ is the state σ^* such that

$$\begin{aligned}
\sigma^*[\lambda] &= \sigma_i[\lambda] && \text{whenever } \sigma_i[\lambda] \neq \text{error for some } i \\
&= \text{error} && \text{otherwise.}
\end{aligned}$$

If σ_i and σ_j map λ to different non-error values the parallel combination is broken from the point of this error. The disjointness constraints of PAR *guarantee that this error cannot arise in occam.*

The parallel operator works by allowing each process to communicate only in its own dec-

lared alphabet, and only allowing a given communication to occur when each process whose alphabet it belongs to agrees. Termination can only take place when all processes agree, the final state being formed by joining together the states of the individual processes. As soon as one process breaks or diverges, the whole system is considered broken.

$f(A/Y) = \{(s/Y, X) : (s, X \cup Y) \in f(A)\} \cup \{(u, X) : \{s \in traces(A) : s/Y \leqslant u\} \text{ is infinite}\}$

$t(A/Y)s = \bot$ if $\{u \in traces(A) : u/Y \leqslant s\}$ is infinite,
$= \bigcup\{t(A)u : u/Y = s\}$ otherwise.

If $s \in \Sigma^*$ then s/Y is defined to be $s\lceil(\Sigma - Y)$.

The hiding operator is used to conceal the communications which are internal to the parallel system. The above definition is valid as a continuous function on Q when Y is finite (which is implied by the finiteness of $\mathbb{B}$). When $\mathbb{B}$ is infinite one can no longer separate the parallel (||) operator from hiding (/Y) (see Conclusions).

/Y transforms communications in Y into internal actions which occur automatically. Thus A/Y cannot refuse any set X unless A can refuse $X \cup Y$, as an (internal) Y action might bring the process into a state where it can accept an element of X.

This completes the definition of the parallel operator PAR.

Replicators allow us to construct (using SEQ, ALT, IF, PAR) processes from many similar small processes. Their semantics are of course closely related to those of the constructs they extend. As an example we will show how to deal with replicated SEQ.

$\mathbb{C}[\![SEQ\ x = [e_1\ FOR\ e_2]\ P]\!]\rho\sigma = \underline{rseq}(\beta_1, \beta_2, \lambda)(\mathbb{C}[\![P]\!]\rho')\sigma$

where $\lambda = new\rho$ $\rho' = \rho[\lambda/x][\underline{ro}/\lambda]$, *(see below for definition of* new*)*

$\beta_1 = \mathcal{E}[\![e_1]\!]\rho\sigma$, $\beta_2 = \mathcal{E}[\![e_2]\!]\rho\sigma$

and $\underline{rseq}(\beta_1, \beta_2, \lambda)B\sigma' = skip_{\sigma'}$ if $\beta_2 \leqslant 0$
$= \underline{seq}(B\sigma'[\beta_1/\lambda], \underline{rseq}(\beta_1 + 1, \beta_2 - 1, \lambda)B)$ otherwise,

provided none of newρ, $\mathcal{E}[\![e_1]\!]\rho\sigma$, $\mathcal{E}[\![e_2]\!]\rho\sigma$ *evaluate to error, and each of* $\beta_1, \beta_2 + 1, \ldots, \beta_1 + \beta_2 - 1$ *is in* $\mathbb{B}$.

Note that a replicator index is given the status of a read-only variable. This is because it may neither be assigned to nor used in constant definitions.

A declaration introduces a new identifier for the process which follows it. In the semantics this is achieved by modifying the environment.

$\mathbb{C}[\![\Delta : P]\!]\rho\sigma = \underline{prune}_\rho(\mathbb{C}[\![P]\!](\mathcal{D}[\![\Delta]\!]\rho)\sigma)$ *provided* $\mathcal{D}[\![\Delta]\!]\rho \neq$ error

where $\underline{prune}_\rho : Q \to Q$ is defined

$f(\underline{prune}_\rho(A)) = f(A)$

$t(\underline{prune}_\rho(A))s = \bot$ if $t(A)s = \bot$,
$= \{prune_\rho(\sigma') : \sigma' \in t(A)s\}$ otherwise;

and $\text{prune}_\rho(\sigma')[\lambda] = \sigma'[\lambda]$ if $\rho[\lambda] \in \{\underline{ro}, \underline{r/w}\}$

$= \text{error}$ otherwise.

The function $\underline{\text{prune}}_\rho$ makes sure that when $\Delta : P$ terminates it does not pass on information local to itself (the contents of locations corresponding to identifiers declared in Δ). The use of $\underline{\text{prune}}_\rho$ improves the set of algebraic laws satisfied by occam.

This completes the definition of $\mathcal{C}$ except for calls of named processes. Before this final clause can be defined we must see how named processes are stored in the environment by declarations. We must therefore see how the function $\mathcal{D}$ is defined. We assume the existence of the following semantic functions.

$\text{new}: \text{ENV} \to \text{LOC}^+$ $\qquad$ $\text{newvec}: \text{N} \to \text{ENV} \to (\text{LOC}^V)^+$

$\text{newchan}: \text{ENV} \to \text{CHAN}^+$ $\qquad$ $\text{newchanvec}: \text{N} \to \text{ENV} \to (\text{CHAN}^V)^+$

These functions have the job of producing locations or channels (either singly or in vectors) which are unused (i.e. have value $\underline{u}$) in some environment. They give result "error" if the required resource is not available. For example $\text{new}\rho$ is a location such that $\rho[\lambda] = \underline{u}$, and $\text{newchanvec}(3)\rho$ is a vector of three channels, each of which has value $\underline{u}$ in ρ.

$\mathcal{D}[\![\text{VAR x}]\!]\rho = \rho[\lambda/\text{x}][\underline{r/w}/\lambda]$ where $\lambda = \text{new}\,\rho$ *provided* $\text{new}\rho \neq \text{error}$.

$\mathcal{D}[\![\text{VAR x[e]}]\!]\rho = \rho[\nu/\text{x}][\underline{r/w}/\nu[0]] \ldots [\underline{r/w}/\nu[\beta-1]]$

where $\beta = \max(\mathcal{E}[\![\text{e}]\!]\rho\sigma_{err}, 0)$ $\quad (\sigma_{err}[\lambda] = \text{error}$ for all $\lambda \in \text{LOC})$

$\nu = \text{newvec}\,\beta\,\rho$ *provided* $\text{newvec}\beta\rho \neq \text{error}$ *and* $\mathcal{E}[\![\text{e}]\!]\rho\sigma_{err} \neq \text{error}$.

$\mathcal{D}[\![\text{CHAN c}]\!]\rho = \rho[\chi/\text{c}][\underline{ud}/\chi]$ where $\chi = \text{newchan}\,\rho$ *provided* $\text{newchan}\rho \neq \text{error}$.

$\mathcal{D}[\![\text{CHAN c[e]}]\!]\rho = \rho[\nu/\text{c}][\underline{ud}/\nu[0]] \ldots [\underline{ud}/\nu[\beta-1]]$

where $\beta = \max(\mathcal{E}[\![\text{e}]\!]\rho\sigma_{err}, 0)$ and $\nu = \text{newchanvec}\beta\rho$

provided $\mathcal{E}[\![\text{e}]\!]\rho\sigma_{err} \neq \text{error}$ *and* $\text{newchanvec}\beta\rho \neq \text{error}$.

$\mathcal{D}[\![\text{DEF x = e}]\!]\rho = \rho[\beta/\text{x}]$ where $\beta = \mathcal{E}[\![\text{e}]\!]\rho\sigma_{err}$ *provided* $\mathcal{E}[\![\text{e}]\!]\rho\sigma_{err} \neq \text{error}$.

The evaluation of expressions in the state σ_{err} forces them only to use identifiers representing constants (for accessing a location will lead to error).

The denotations of multiple declarations such as VAR x,y are easy generalisations of the above definitions.

$\mathcal{D}[\![\text{PROC p}(\Phi)]\!]\rho = \rho[\pi/\text{p}]$, where $\pi = \lambda\langle\rho'_2, \rho'_3\rangle.\lambda L.\mathcal{C}[\![P]\!]\langle S[\![\Phi]\!]\rho_1 L, \rho'_2, \rho'_3\rangle$

S substitutes the denotations listed in L ($\in D^*$) into the environment component ρ_1, using the identifiers listed in Φ. *If S fails (because of incorrect length or type) then a call of the procedure produces value* $\perp_Q$.

The bindings of identifiers used in the call of a procedure are the same as those at the point of declaration. However the state and the status part of the environment are as at the point of call. This last detail stops parallel processes accessing illegal variables via their procedures. The denotation of a procedure or named process is thus

$\pi \in NP = (LSTATUS \times CSTATUS) \to (D')^* \to S \to Q \quad (D' = LOC + \mathbb{B} + CHAN + LOC^V + CHAN^V)$.

We can now complete the definition of $\mathcal{C}$ by giving the clause for procedure calls.

$\mathcal{C}[\![p(A)]\!]\rho\sigma = \rho[\![p]\!](\rho_2, \rho_3)(\mathcal{L}[\![A]\!]\rho\sigma)\sigma$ *provided* $\mathcal{L}[\![A]\!]\rho\sigma \neq$ error.

Here $\mathcal{L}$: Act $\to$ ENV $\to$ S $\to (D'^*)^+$ is a semantic function which converts syntactic actual parameter lists into their denotations. If a particular parameter could represent either an element ($\mathbb{B}$) or a location, then the location is chosen. *If the evaluation of any element of A produces an error (or a named process) then the procedure call breaks* ($\perp_Q$).

This completes the definition of our denotational semantics for occam. It gives an interesting illustration of how the domain Q defined in the first section can be used to model concurrent languages. Our model (sometimes with minor modifications) can cope with many possible extensions to the language. These include more sophisticated value domains, recursive procedure definitions, and additional operators on processes. We will mention a few of the possible additional operators in the conclusion.

3. Applications of the semantics We now have a mapping from occam to a well-defined mathematical model which is closely related to the behaviour of processes. This section outlines a few uses to which this mapping might be put. No topic is covered in detail; we merely identify some promising areas for future work.

Our model's relationship with process behaviour makes it a natural framework for expressing correctness conditions on processes. Our semantics will then provide a basis for proving correctness. We should therefore look at the type of correctness condition that is expressible by reference to the model, techniques for manipulating these conditions within the model, and methods (formal, and perhaps informal) of proving them of programs.

The most natural types of correctness condition to consider are those which specify that every "behaviour" of a process must satisfy some predicate (related, perhaps, to the environment and initial state). There are essentially three different types of behaviour to consider for a process $\langle F,T\rangle \in Q$. The first consists of the process' *failures* ($(s,X) \in F$); the second consists of its (successful) *terminations* ((s,σ) with $\sigma \in T(s)$); the third consists of its *divergences* (s such that $T(s) = \perp$). The predicate of behaviour should tell us which of each sort of behaviour is acceptable.

Except for specifications which are approximations to final goals one will almost always demand that a process is *divergence-free* (i.e. no divergence is a correct behaviour). Divergence can arise in three distinct ways in our semantics: from a badly constructed WHILE loop; from a PAR construct which becomes livelocked (an infinite sequence of internal communications takes place without any communication with the environment); and from technical errors of diverse sorts (e.g. assigning to a channel identifier).

The first type can be eliminated by familiar methods such as making sure some loop variant is decreased. On its variant becoming zero one can allow a loop *either* to terminate

or to communicate. This is because an occam WHILE loop can be correct without ever terminating. Proving the absence of livelock in a parallel system will require consideration of the connection structure of that network. Freedom from livelock is established if (for example) no two processes are connected together, each of which has the ability to communicate infinitely without communicating with the external environment of the network. (Properties such as this can be rigorously specified and proved within Q.) A few simple results on the absence of livelock for specific types of network (simple linear arrangements, rectangular arrays and trees) can be found in [12]. Often one will expect the absence of divergence to be a corollary to some more specific correctness proof; sometimes, however, one might find that its proof is a necessary first step.

A second type of behaviour which we will usually wish to eliminate is *deadlock*. A process deadlocks when it comes into a state where it can do nothing at all (neither communicate nor terminate successfully). Deadlock after trace s is represented by the failure $(s, \Sigma \cup \{\surd\})$. The usual cause of deadlock will be a badly designed parallel network, where it is possible for the individual processes to reach a state where they cannot agree which communications to perform. Perhaps the best known example of a potentially deadlocked system is *the five dining philosophers* (this was studied relative to the failure-sets model in [5]).

Once again, a proof of deadlock-freedom might be a corollary to some more specific result (such as congruence with some sequential program). On the other hand it is sometimes useful to be able to prove it in isolation. The simple way in which deadlock is represented (as the failure $(s, \Sigma \cup \{\surd\})$) makes our model a natural vehicle for studying it. Proofs will depend on the form of the network of connections between processes, and will often be graph-theoretic. A few simple applications of the failure-sets model to this problem will be found in [5,12]. There it is shown, for example, that if the interconnection graph of a network is a tree, then the whole network is free of deadlock if it is locally free of deadlock (in a strictly defined sense). The phenomenon of distributed termination will considerably complicate the connection graphs in occam. It should be possible, though, to separate off the issue of termination, and so only consider the graphs of ordinary communications.

In general our model gives us a rich language for specifying correctness conditions. Specifying which traces and terminations are allowable corresponds to *safety* or *partial correctness*, while divergences and refusal sets enable us to specify *liveness* or *total correctness*. An important topic for future research must be discovering a good formal syntax for useful correctness conditions for occam programs.

Another important topic will be research into formal methods for proving these properties of programs. Our semantics will provide the foundations for justifying such methods (proving them sound and perhaps complete). There are several promising approaches, a few of which are summarised below.

One can regard every program as a logical formula describing the strongest predicate

satisfied by all the observations one can make of it. If a program's specification is also a logical formula, describing all allowable observations, then proving a program P meets specification S simply requires the proof of the formula $P \Rightarrow S$. In [9], we show how every occam program may be identified with a predicate describing its traces, refusals, status (waiting, terminated or divergent) and the values (initial and final) of its free variables. The way this was done was based very closely on the semantics presented in this paper, and essentially provides an encoding of our model Q in the predicate calculus. It should be possible to develop this approach into a very useful formal system for proving properties of occam programs.

Our semantics induce a natural congruence on occam: programs P,Q are equivalent if and only if $\mathbb{C}[\![P]\!]\rho\sigma = \mathbb{C}[\![Q]\!]\rho\sigma$ for all ρ,σ. Two programs are thus equivalent if they behave identically under all circumstances. It is convenient to strengthen this congruence a little: we can reasonably regard P and Q as equivalent if $\mathbb{C}[\![P]\!]\rho\sigma = \mathbb{C}[\![Q]\!]\rho\sigma$ for all ρ,σ such that ρ has infinitely many free locations and channels, and $\sigma[\lambda] = \text{error}$ whenever $\rho[\lambda] \in \{\underline{u},\underline{x}\}$. These restrictions prevent "naturally equivalent" programs being distinguished for technical reasons. Henceforth, if $P,Q \in \text{Proc}$, then $P = Q$ will mean that P is congruent to Q in this sense. (This relation is indeed a congruence in the sense that if we replace any subprogram P of Q by $P' = P$ to get Q', then $Q = Q'$.)

The following theorem lists a small selection of the algebraic laws which can be proved relating the constructs of occam.

Theorem The following equivalences all hold between occam programs.

a) $SEQ(P_1,P_2,\ldots,P_n) = SEQ(P_1,SEQ(P_2,\ldots,P_n)) = SEQ(SEQ(P_1,\ldots,P_{n-1}),P_n)$

b) $SEQ(P,SKIP) = SEQ(SKIP,P) = P$

c) $SEQ(ALT(G_1,\ldots,G_n),P) = ALT(G_1;P,\ldots,G_n;P)$
where the guarded processes $G_i;P$ are defined by induction on the structure of G:
$(g\ P');P = g\ SEQ(P',P)$
$(ALT(G'_1,\ldots,G'_m));P = ALT(G'_1;P,\ldots,G'_m;P)$

d) $WHILE\ e\ P = IF(e\ SEQ(P,WHILE\ e\ P),\ true\ SKIP)$

e) $PAR(U_1{:}P_1,\ldots,U_n{:}P_n) = PAR(U_1{:}P_1,U^*{:}PAR(U_2{:}P_2,\ldots,U_n{:}P_n))$
where U* is the *union* of $U_2,\ldots,U_n$, in other words the parallel declaration which claims all the variables claimed by any of $U_2,\ldots,U_n$; claims as OWNCHANs all OWNCHANs of $U_2,\ldots,U_n$, as well as all channels declared both as an INCHAN and as an OUTCHAN among $U_2,..,U_n$; claims as INCHANs all other INCHANs of $U_2,\ldots,U_n$; and as OUTCHANs all other OUTCHANs of $U_2,..,U_n$.

f) $PAR(U_1{:}P_1,U_2{:}P_2) = PAR(U_2{:}P_2,U_1{:}P_1)$

In [8], this list will be much extended. We will in fact show there that there are enough algebraic laws to completely characterise the semantics of occam, as presented in the present paper. We will construct a *normal form* for WHILE-free programs (every such program is equivalent to one in normal form, but no two distinct normal form programs

are semantically equivalent), and give enough laws to transform every such program into normal form. We are therefore able to decide whether any pair of WHILE-free programs are equivalent. The postulate that the value of any program is determined by those of its *finite syntactic approximations* then gives us an infinitary rule for deciding the equivalence of arbitrary programs. (This postulate is a theorem in the denotational semantics of the present paper.) This *algebraic semantics* is closely related to similar work on a purely parallel version of CSP in [3].

Several authors (for example in [1,2] and in [15]) have given Hoare-style axiom systems for similar languages (usually the CSP of [6]). It should be possible to construct such a system for occam. Our semantics will allow the formal analysis of such a system.

Denotational semantics should provide a standard by which to judge implementations of a language. Proving an implementation correct (with respect to our semantics) will probably require one or more intermediate levels of abstraction as bridges. A typical intermediate level would be an *operational* semantics. In [5] there is an operational semantics for CSP which is provably congruent to the failure-set semantics.

4. Conclusions In building our mathematical model and constructing the denotational semantics we have made many decisions. In this section we will discuss a few of these, and see how a few of the restrictions we have imposed might be relaxed.

Most of the major decisions come when constructing the mathematical model: once this is fixed, the semantics of most constructs are determined by their roles in the language. There are several factors which influence the choice of model. It should be at the right level of abstraction; it should have sufficient power to specify desired correctness properties; it should be able to cope with all the constructs of the given language; and it should be as simple, elegant and understandable as possible.

It is possible to do without "states" in the model. Variables could be replaced by parallel processes, created at their points of declaration, with which the "main" processes communicate when they want to read or assign. Because one can use a "purely parallel" semantic model, this approach simplifies model building. On the other hand the semantic definitions become more complex, and the level of abstraction seems wrong. One loses many of the advantages gained from the similarity with the semantics of ordinary languages; in particular, the semantics of a "purely sequential" occam fragment will no longer bear much resemblence to a relation on states. Nevertheless, this approach should lead to a semantics congruent to ours on programs without free variables.

The model Q was devised as a *general purpose* model for communicating processes with state. It is slightly too "rich" for occam, in that an occam process' refusals are really sets of channels, rather that sets of individual communications. The consequence of this is that Q contains rather more types of behaviour than are actually describable in occam (for example a process which is prepared to input any integer *greater than 6* on channel c). One can give a semantics for occam, congruent with our own, into a model

where a failure is an element of $\Sigma^* \times (\wp(CHAN) \cup \{\surd\})$ rather than $\Sigma^* \times (\wp(\Sigma) \cup \{\surd\})$. This fact is illustrated by the predicate logic model in [9]. Of course, if occam were extended to include "selective inputs" of the type indicated above, then the reduced model would no longer be sufficiently detailed.

It seems likely that the techniques employed in this paper to combine the models N and $\wp(S \times S)$ into Q would work for other pairs. For example one could simplify the model by using the *traces* model [7] in place of the failures model N. However the semantics would only be simplified in their "purely parallel" aspects; we would suffer by losing the ability to reason about total correctness properties. If the object language contained jumps, there would be few problems in adapting our model for *continuation* semantics.

In this paper we have been guilty of omitting part of the language, as well as making other simplifying assumptions. The ones which made the construction of the model easier were omission of *timing* and *priority*, and the restriction of the set $\mathbb{B}$ to be finite. To rectify the first two will require future research, probably into timed models for communicating processes. While the final assumption is perfectly realistic, it is unfortunate from a theoretical point of view. The main problem which arises when $\mathbb{B}$ is infinite is that the hiding operator (/Y), used in the semantics of PAR, can yield unbounded nondeterminism (and so discontinuity). However the restricted syntax of occam, for example the separation rules for channels under PAR, means that this unbounded nondeterminism cannot actually occur. If one were prepared to make the model Q more technical, perhaps by discriminating between input and output communications and restricting the circumstances under which they could occur or be refused, one could construct a satisfactory model for occam with infinite $\mathbb{B}$.

On the other hand our model can cope easily with certain features not present in occam, such as general recursion, more elaborate types, output guards in ALTs, and multiple (simultaneous) assignments. The two final members of this list are useful theoretical additions to occam (see [8]).

Once the main model was decided, the majority of the semantic definitions seemed to follow naturally. Nevertheless there were still a few decisions made, both consciously and unconsciously. An example of such a decision arises in the hiding of internal communication. In this paper the hiding has been incorporated into the definition of the parallel operator: we hide the internal communications of each network as it is constructed. An alternative would have been to hide the channels at their points of declaration. These two alternatives should give identical results for programs with no free variables that are used for internal communication. The version we have chosen does, however, have better algebraic properties. For example, under suitable restrictions on U_1, U_2,

$$SEQ(x := e,\ PAR(U_1{:}P_1,\ U_2{:}P_2)) \ =\ PAR(U_1{:}SEQ(c!e,P_1),\ U_2{:}SEQ(c?x,P_2)),$$

a law which would not hold under the alternative interpretation of hiding.

Acknowledgements I would like to thank Tony Hoare, whose work on occam inspired this research, and who has made many valuable suggestions. I would also like to thank Steve Brookes and everyone else who has commented on the various versions of this work.

References

[1] Apt, K.R., Formal justification of a proof system for communicating sequential processes, JACM Vol. 30, No. 1 (Jan 1983) pp197-216.

[2] Apt, K.R., Francez, N., and de Roever, W.P., A proof system for communicating sequential processes, Trans. Prog. Lang. Syst. 2, 3 (July 1980) pp359 - 385.

[3] Brookes, S.D., A model for communicating sequential processes, D.Phil. thesis, Oxford University, 1983.

[4] Brookes, S.D., Hoare, C.A.R., and Roscoe, A.W., A theory of communicating sequential processes, JACM 31, 3 (July 1984) pp560 - 599.

[5] Brookes, S.D., and Roscoe, A.W., An improved failures model for communicating processes, Carnegie - Mellon Tech. Report 1984. (Appears in an abbreviated form in this volume.)

[6] Hoare, C.A.R., Communicating sequential processes, CACM 21, 8 (August 1978) pp666 - 676.

[7] Hoare, C.A.R., A model for communicating sequential processes, Tech. Report PRG-22, Oxford University Programming Research Group, 1981.

[8] Hoare, C.A.R., and Roscoe, A.W., The laws of occam programming, in preparation.

[9] Hoare, C.A.R., and Roscoe, A.W., Programs as executable predicates, in Proceedings of FGCS 84, North-Holland 1984.

[10] INMOS Ltd., The occam programming manual, Prentice - Hall International, 1984.

[11] Milne, R.E., and Strachey, C., A theory of programming language semantics, Chapman Hall, London, and Wiley, New York, 1976.

[12] Roscoe, A.W., A mathematical theory of communicating processes, D.Phil. thesis, Oxford University, 1982.

[13] Stoy, J.E., Denotational semantics, MIT Press, 1977.

[14] Tennent, R.D., Principles of programming languages, Prentice - Hall International, 1981.

[15] Zhou Chaochen, The consistency of the calculus of total correctness for communicating processes, Tech. Report PRG - 26, Oxford University Programming Research Group, 1982.