DENOTATIONAL SEMANTICS FOR occam 2

by

M. H. Goldsmith
A. W. Roscoe
B. G. O. Scott

Technical Monograph PRG-108
ISBN 0-902928-85-6

June 1993

Oxford University Computing Laboratory
Programming Research Group
11 Keble Road
Oxford OX1 3QD
England

Oxford University Computing Laboratory
Programming Research Group
11 Keble Road
Oxford OX1 3QD
England

Electronic mail: Brian.Scott@comlab.ox.ac.uk

Denotational Semantics for occam 2

M. H. Goldsmith*
A. W. Roscoe
B. G. O. Scott

Abstract

This paper gives an untimed denotational semantics for the concurrent programming language occam 2. *It draws heavily on the semantics for a large subset of proto-*occam *[26], but addresses the complete extended language (to the extent that the model allows). The semantic domain used is a 'failures/divergences' model, modified to allow machine states to be properly dealt with. This means that issues of fairness and priority are not addressed.*

*Formal Systems (Europe) Ltd., 3 Alfred Street, Oxford, OX1 4EH

1 Introduction

The occam programming language [19] was designed with the philosophy of eliminating unnecessary complexities, thus keeping the language simple and elegant. This paper is a successor to the denotational semantics for an early version of the language, given in [26]. The aim of that paper was to show how traditional denotational semantic techniques could be readily adapted to this new type of language, and to provide a basis for subsequent formal work about and using occam. The semantics of that paper have been used, directly or indirectly, in almost all of the large amount of formal work which has been done since in the area, for example the development of a congruent algebraic semantics [16], the development of the occam Transformation System [11] and its applications, and Barrett's work on a hierarchy of congruent operational semantics [4]. The current extension is intended both to bring the earlier work in line with developments in the language and to form part of an international standardisation effort for 'occam and its use in verified systems'.

The main difficulties in developing a useful theory of occam, arising from the fact that it is a concurrent language, were overcome in the earlier paper. Further developments of the language have resulted in a new version, occam 2 [20], augmented by the addition of data-types (including floating-point numbers and multi-dimensional arrays), type-coercion, channel protocols, side-effect-free functions and a CASE construct. In addition, the separation rules governing the use of variables and channels, particularly array elements, have been made more restrictive with the aim of allowing them to be mechanically checked. Another contribution towards this end is a rule outlawing *aliasing*: at no point in a program can the same item be legitimately referred to by two different names.

The correctly typed use of types and channels is well understood, and checkable by static analysis; since only type-correct programs will be considered, the new declaration forms will have minimal impact on the semantics. The behaviour of floating-point arithmetic operations according to the specified standard [17] has been formalised and investigated [3]; that appropriate operations exist over a suitable space and behave in the desired manner will be taken as given in what follows. Value-preserving type-coercions cause no problem, requiring only the transfer of the item between parts of the value domain, and representation-preserving changes (RETYPES) are defined to be, in general, implementation dependent.

The new expression and process constructors have to be included in the semantics, but have been defined in a way which means that a minimal amount of extra machinery is required. The anti-aliasing and separation rules make the management of the environment somewhat more complex, as will be seen below.

Within this paper, considerations of brevity have led to the decision to leave some definitions of standard or straightforward auxiliary functions to the reader. The functions treated in this way, along with a brief indication of their purpose, are

- $\mathcal{O}$ — used to give meaning to (syntactic) operators on expressions;
- $\mathcal{N}$ — used to give meaning to literals;
- *convert* — used to shift values between parts of the value domain;
- *round* — used to round-off values then shift them between parts of the value domain;
- *trunc* — used to truncate values then shift them between parts of the value domain;
- *minval* — used to return the smallest element of a given type;

- *maxval* — used to return the largest element of a given type;
- *disjoint* — used to check whether the left hand sides of a multiple assignment are disjoint;
- *inchans* — used to isolate the input channels from a parallel declaration;
- *outchans* — used to isolate the output channels from a parallel declaration;
- *ownchans* — used to isolate the internal channels from a parallel declaration;
- *addrs* — used to isolate the mutable variables from a parallel declaration;
- $\mathcal{W}_V$ — used to allocate channels to processes defined in parallel;
- *newchan* — used to provide the information necessary for newly-declared channels;
- *newtim* — used to provide the information necessary for newly-declared timers;
- *newport* — used to provide the information necessary for newly-declared ports.

Care has been taken to ensure that this paper is not only applicable to particular implementations of occam 2, but remains valid for all probable implementations. Maintenance of such generality restricts the assumptions which can be made regarding the structure of the store, and for this reason it has not been possible to give definitions of those auxiliary semantic functions whose definitions depend on the store. Given an implementation of the store, the definition of each of the functions is straightforward. The functions whose definitions are necessarily omitted are

- *lookup* — used to read the contents of a variable from the store;
- *update* — used to update the contents of a variable in the store;
- $\oplus$ — used to combine the information contained in two stores;
- $\downarrow$ — used to ignore part of the information conveyed by a store;
- *contents* — used to calculate the addresses accessed while evaluating an expression;
- *abode* — used to calculate where the contents of a variable reside within the store;
- *startaddr* — used to calculate the position of an array component within the store;
- $\mathcal{W}_L$ — used to allocate store to processes defined in parallel;
- *new* — used to provide an area of store to hold the contents of a variable;
- *markaddr* — used to keep note of the utilisation of areas of the store;
- *restrict* — used to prevent variables altering within the scope of a value abbreviation.

The first part of the paper is concerned with the construction of a suitable model; the resultant model shares many characteristics with that used in [26], but requires richer value domains to reflect the greater expressive power of occam 2. An explicit alphabet for processes has been incorporated into the model, and the structure of the refusal sets in the model has been altered (recording channel names and not communications) to obtain a more natural congruence with the language. The second part of the paper uses the model to give a denotational semantics to occam 2, in the style of [22, 26, 29, 30].

2 Construction of the model

Throughout this paper, $\mathcal{P}(X)$ will denote the full powerset of X (the set of all subsets of X), while $\wp(X)$ 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, with $\langle\rangle$ denoting the empty sequence and $\langle a, b, \ldots, z\rangle$ denoting the sequence containing $a, b, \ldots, z$ in that order. If $s, t \in X^*$, st denotes the concatenation of s and t (e.g., $\langle a, b, c\rangle\langle d, e\rangle = \langle a, b, c, d, e\rangle$), and $s \leq t$ (s is a *prefix* of t) if there is some $u \in X^*$ with $su = t$.

As in [26], the semantic domain for occam is based on the failures/divergences model for communicating processes. This has now effectively become the standard semantic model for studying and applying Untimed CSP. Included below is a summary of the laws which must be satisfied by a process; the interested reader is referred to [6, 7, 8, 25] for descriptions and motivations concerning the construction of the model. The version used here is that of [8]; recent additions to deal with the possibilities of unbounded nondeterminism are not directly relevant to this paper. It is, however, proposed to describe in a future paper how the infinite traces model for CSP can be modified in order to deal with certain fairness concepts for occam.

The sets of failures and divergences of a CSP process satisfy the laws below (see [8]). If a process P (complete with the set αP of events in which it can participate) has representation (F, D), where $F \subseteq (\alpha P)^* \times \mathcal{P}(\alpha P)$ and $D \subseteq (\alpha P)^*$, then

N1) $traces(P)\ [= \{s \in (\alpha P)^* \mid (s, \emptyset) \in F\}]$ 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 $Y \cap \{a \in \alpha P \mid s\langle a\rangle \in traces(P)\} = \emptyset$, then $(s, X \cup Y) \in F$

N4) if $(s, Y) \in F$ for each $Y \in \wp(X)$, then $(s, X) \in F$

N5) if $s \in D$ and $t \in (\alpha P)^*$, then $st \in D$

N6) if $s \in D$ and $X \subseteq \alpha P$, then $(s, X) \in F$

The failures/divergences model $\mathcal{N}$ is defined to be the set of all pairs (F, D) satisfying these laws.

If $P \in \mathcal{N}$, $f(P)$ will denote the first component of P, and $d(P)$ the second. There is a natural partial order on $\mathcal{N}$ 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 P' can naturally be thought of as being more deterministic than P, for it has fewer possible actions. $\mathcal{N}$ is a complete semilattice with respect to $\sqsubseteq$; its minimal element is $((\alpha P)^* \times \mathcal{P}(\alpha P), (\alpha P)^*)$ (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.

The above model is adequate to represent the behaviour of programs written in CSP, with all the CSP operators translating naturally to continuous functions over $\mathcal{N}$. It is well suited to reasoning about the nondeterminism which arises from distributed systems, and to reasoning about deadlock. Axioms N5 and N6 correspond to the assumption that following the possibility of divergence, subsequent behaviour is irrelevant. Hence divergence is something to be avoided at all costs. The inclusion of these laws makes for considerable technical simplification at what does not appear 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.

Purely parallel languages can be given an adequate denotational semantics by models whose only primitives are communications, since one part of a program can only influence a disjoint part by communication. However, in occam, one part of a 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 incorporate both these methods.

The first step in the construction of a model for occam is to provide the alphabet for an occam process P. Communication over occam channels is directional, and so it is no longer enough merely to provide the set αP (which will now be the set $\{\chi.\beta \mid \chi \in CHAN \wedge \beta \in \overline{\chi}\}$ where $\overline{\chi}$, given a more specific description on page 11, is the set of values communicable over the channel χ and hence determined by the protocol associated with χ) of atomic communications. In addition to αP, the direction of all the channels which the process could theoretically use is useful, as are the areas of store which can be written to and those which can be read from. The most convenient means of storing this information is to add two components to a process description. The first component, representing the channels and denoted C, will be a partial function from $CHAN$ (the set of channels) to $DIRECTION \times PROT$ (the cartesian product of the set $\{in, out\}$ representing channel direction, and the set of denotations of possible protocols of an occam channel). The second component, representing the accessibility of portions of the store and denoted L, will be a partial function from $ADDR$ (the set of addresses in the store) to $ACCESS$ (the set $\{ro, rw\}$). A process can read from but cannot write to areas of the store with addresses mapped to ro, while it can read from or write to areas whose addresses are mapped to rw.

The treatment of communication will be similar to that of [26], except that it is necessary to take account of the way in which occam processes communicate over channels. Since the handshaking that occurs in occam is tied to the channel, not to the value which passes along that channel, it does not make sense to say that a process can refuse to communicate some communications over a channel but not others[1]. Therefore the alphabet used for communication is separated from the one used for refusal sets, consisting of the channel names alone plus $\checkmark$, which is used to denote refusal of termination (see below). Since the set of channels used in any particular program is finite, in order to avoid the complexities of axiom N4 above, it will be assumed that for any process $\mathrm{dom}(C)$, the set of channels which can be used, is finite. To avoid any errors being caused by a process not being able to claim the space in the store which it requires, it will be assumed that the set $ADDR$ of store addresses is infinite. Despite this, any given process will use only a finite number of store addresses.

A process which can accept at least one value on an input channel must be able to accept them all. In general, a process would be expected to specify precisely the value it is outputting; it is possible for nondeterministic choice and ALTs to make more than one value possible, but never an infinite number. This is important since it means that, when two processes are put in parallel and the communications which pass between them are hidden, no infinite branching (which leads to unbounded nondeterminism) arises. It will be assumed that no non-divergent process ever has an infinite number of possible outputs over χ after a given trace, since the presence of finite branching means that an infinite number of different outputs are only possible on traces where it is also possible for the process to engage in an infinite sequence of internal actions – in other words to diverge.

[1] This statement applies even to inputs over variant protocols. Indeed this statement is particularly true, in some sense, of communications over channels with these protocols. See the discussion on page 27 when the semantics of input over channels with variant protocol is discussed in detail.

In occam, parallel processes can read from a shared area of store provided that none of the processes can write to that area. Otherwise, at most one process can have access to an area at any point in time. This restriction can be checked by examining the second alphabet component (the partial function L) of a process description.

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.) In purely parallel models (for example in [7, 12]) successful termination has been modelled by the communication of some special symbol: usually ✓. 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 ✓s – one for each possible final state. If this solution were adopted then a large proportion of the alphabet of 'communications' would consist of these ✓s. A number of problems would arise if all of these different ✓s were simply included in the traces and refusal sets.

Two of these problems are the same as previously addressed when considering communication over channels. Firstly, it would not be appropriate to have a process which offered a choice of which state it terminated in – when a process terminates, it terminates and gives the final state it happens to be in. The solution to this is a single termination symbol ✓ for use in refusal sets. (This symbol is necessary since it provides the only means of distinguishing between a process that always terminates and one which may nondeterministically choose to do nothing at all.) The second problem would arise if the set of states were infinite and it were possible to have a process that could choose from an infinite set of states to terminate in. Since sequential composition hides the value of the state, infinite branching and hence unbounded nondeterminism would result. In fact this situation cannot arise in occam without the immediate possibility of divergence, because of the finite branching properties discussed above. Thus it will be assumed that a process which cannot diverge on a given trace has only a finite number of states in which it can then terminate. This condition is trivially satisfied in any language where the set of all states for a given program is finite.

Finally, in a model where termination plays a more important rôle than before, the technical complexities introduced by allowing non-terminal ✓s in traces are unacceptable (as well as being unnatural).

The first two problems could be solved by treating ✓ as though it were a channel; however the difference between termination and communication and the final problem make the following more attractive. First, remove ✓ from the traces of processes (hence leaving only 'real' communications). A single symbol ✓ remains in the alphabet used for refusal sets, indicating that a process can refuse to terminate successfully. The second component is expanded. Instead of merely recording the possible divergences, the possible final states (contents of locations which can be altered by the process) from successful termination after any trace are recorded. It becomes a function from $(\alpha P)^*$ to $\wp(S) \cup \{\bot\}$, where S is the space of final states and $\bot$ represents possible divergence. Further information about S will be given as the details of the model are filled in; all that is necessary is that S contains enough information to enable processes to pass on the values of variables still within scope to their successors.

Thus each process P is now represented by a quadruple $\langle F, T, C, L\rangle$, with components $F \subseteq (\alpha P)^* \times \mathcal{P}(\mathrm{dom}(C) \cup \{\checkmark\})$, $T : (\alpha P)^* \longrightarrow \wp(S) \cup \{\bot\}$, $C : CHAN \rightarrowtail\!\!\!\!\!\!\rightarrow \{in, out\} \times PROT$, and $L : ADDR \rightarrowtail\!\!\!\!\!\!\rightarrow \{rw, rw\}$. The interpretation of the behaviour of a process P, represented by $\langle F, T, C, L\rangle$, is as follows.

(i) F (the failures of P) lists all possible traces of the process, together with all sets of channels

on which, after the trace, the process can refuse to communicate once it has *stabilised*. (Hence if, after some trace, the environment offers to communicate over a set of channels which is *not* a refusal set the process *must*, once internal activity has ceased, accept some element.) A process is said to have stabilised if internal activity has ceased and no further activity will commence until communication with an external source occurs. The notion of stability is necessary since without it there is no guarantee that the failures following a given trace will remain invariant.

(ii) One of the possible elements of the refusal sets is $\checkmark$ – 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 only when the set $\{\checkmark\}$ cannot be refused.

(iii) Termination *can* take place on any trace s for which $T(s)$ is a nonempty set of states. Although $T(s)$ may consist of more than one element, previous limitations mean it may only consist of a finite number. 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 $T(s) = \bot$, then the process is considered to be broken. The process might diverge or do anything else at all.

(v) C lists the abstract channels which may be used by the process, and the direction of each such channel. Within the component the structure of communications which may occur along each abstract channel the process can use are also recorded.

(vi) The access which a process has to areas of the store is recorded by L. A process can read from but cannot write to areas of the store with addresses mapped to ro. This permits more than one parallel process to safely use the same piece of store. A process can read from or write to areas of the store with addresses mapped to rw. In such a situation, it is not safe for another process in parallel to read from or write to the area of store with this address. As time progresses, and parallel constructs terminate or are created, the store is repartitioned.

With αP, $CHAN$ and $ADDR$ as described above, and a set S of final states, the space $\mathcal{Q}$ of all processes P is thus represented by the set of all quadruples $\langle F, T, C, L\rangle$ which satisfy the following ten laws.

F1) $\mathit{traces}(P)\,[= \{s \in (\alpha P)^* \mid (s, \emptyset) \in F\}]$ is nonempty and prefix closed
F2) $(s, X) \in F \wedge Y \subseteq X \Rightarrow (s, Y) \in F$
F3) $(s, X) \in F \wedge Y \cap \{\chi \mid \exists \beta \in \overline{\chi} \bullet s\langle\chi.\beta\rangle \in \mathit{traces}(P)\} = \emptyset \Rightarrow (s, X \cup Y) \in F$

T1) $(s, X) \in F \wedge T(s) = \emptyset \Rightarrow (s, X \cup \{\checkmark\}) \in F$
T2) $T(s) = \bot \wedge t \in (\alpha P)^* \Rightarrow T(st) = \bot$
T3) $T(s) = \bot \wedge X \subseteq \mathrm{dom}(C) \cup \{\checkmark\} \Rightarrow (s, X) \in F$
T4) $T(s) \neq \emptyset \wedge X \subseteq \mathrm{dom}(C) \Rightarrow (s, X) \in F$

C1) $\{\chi \mid \exists \beta \in \overline{\chi};\ s \in (\alpha P)^* \bullet s\langle\chi.\beta\rangle \in \mathit{traces}(P)\} \subseteq \mathrm{dom}(C)$
C2) $s\langle\chi.\beta\rangle \in \mathit{traces}(P) \wedge (\exists p \in PROT \bullet C(\chi) = (in, p)) \Rightarrow \{s\langle\chi.\gamma\rangle \mid \gamma \in \overline{\chi}\} \subseteq \mathit{traces}(P)$
C3) $T(s) \neq \bot \wedge (\exists p \in PROT \bullet C(\chi) = (out, p)) \Rightarrow \{s\langle\chi.\gamma\rangle \mid \gamma \in \overline{\chi}\} \cap \mathit{traces}(P)$ is finite

In the above s ranges over $(\alpha P)^*$, χ ranges over $CHAN$, X and Y range over $\mathcal{P}(\mathrm{dom}(C) \cup \{\checkmark\})$.

These laws are just the natural extension of the laws governing $\mathcal{N}$ to the revised structure; if P is represented by $\langle F, T, C, L\rangle$, then define $f(P) = F$, $t(P) = T$, $c(P) = C$, and $l(P) = L$. In what follows, $B \subseteq \perp$, $B \cup \perp = \perp$ for all $B \subseteq S$ and $\sigma \in \perp$ for all $\sigma \in S$. Moreover, $\perp \oplus \sigma = \perp$ and $\sigma \oplus \perp = \perp$ for all $\sigma \in S$ and $\perp\!\downarrow A = \perp$ for all $A \in \wp(ADDR)$.

The new model clearly has a great deal in common with the old one. Because of the different set of communications possible for processes with different alphabets, it is not generally useful to compare processes with different C or L components. However, if P and P' are processes with $c(P) = c(P')$ and $l(P) = l(P')$, then a comparison can be meaningful. On the assumption that S has no important partial order of its own, if $\mathcal{Q}_{\langle C,L\rangle}$ is the set of processes P with $c(P) = C$ and $l(P) = L$, $\mathcal{Q}_{\langle C,L\rangle}$ has a natural partial order;

$$P \sqsubseteq P' \quad \Leftrightarrow \quad f(P) \supseteq f(P') \wedge \forall s \in traces(P') \bullet t(P)s \supseteq t(P')s$$

$P \sqsubseteq P'$ can be interpreted as meaning that P' is more deterministic than P. With respect to $\sqsubseteq$, $\mathcal{Q}_{\langle C,L\rangle}$ is a complete semilattice whose minimal element is $\langle F_\perp, T_\perp, C, L\rangle$ (denoted $\perp_{\langle C,L\rangle}$), where $F_\perp = (\alpha P)^* \times \mathcal{P}(\mathrm{dom}(C) \cup \{\checkmark\})$, and $T_\perp(s) = \perp$ for all $s \in (\alpha P)^*$. $\perp_{\langle C,L\rangle}$ is the completely unpredictable process with the appropriate alphabet; it may diverge immediately. The maximal elements of $\mathcal{Q}_{\langle C,L\rangle}$ are 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

$$(s, X) \in f(P) \Rightarrow X \cap \{\chi \in \mathrm{dom}(c(P)) \mid \exists \beta \in \overline{\chi} \bullet s\langle \chi.\beta\rangle \in traces(P)\} = \emptyset$$

and

$$(s, \{\chi\}) \notin f(P) \wedge (\exists p \in PROT \bullet c(P)\chi = (out, p)) \Rightarrow \{\beta \in \overline{\chi} \mid s\langle \chi.\beta\rangle \in traces(P)\} \text{ is a singleton set}$$

and

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

The assumptions that all sets of final states are finite, and that following any trace of a non-divergent process only a finite number of communications on any output channel are possible, correspond closely to an assumption of bounded determinism. Of course, if the set S of states associated with any given area is finite, or if the set of communicable values for every channel is finite, this assumption is vacuous.

Limitations of the model There is no concept of time in the model. Thus the occam 2 timing constructs (`TIMER` and `AFTER`) cannot be modelled directly, either for use in a process or as an `ALT` guard. The mechanisms for dealing with time are now well understood for CSP, and their application to occam will form the subject of a separate paper; here input over a timer is identified with the nondeterministic assignment of an unspecified 'random' `INT` to the designated variable, delayed input within a process is identified with `SKIP` and a branch of an `ALT` guarded by a delayed input is not followed (such branches are included as a 'timeout' to allow recovery from erroneous behaviour and following them would result in the possibly unacceptable omission of parts of the process).

Another main feature lacking is an analogue of priority; there is no way of telling from the model that a process would rather communicate 'a' than 'b' (say). An operational semantics for the purely parallel aspects of occam exists [4], and it is possible that a denotational semantics along the same line could be developed; however, for the sake of simplicity, any treatment of

priority will be omitted, either between processes or among ALT guards, and PRI PAR will be identified with PAR, and PRI ALT with ALT.

The net effect of these omissions is to give a semantics to a program which is less deterministic than one which took into account the timing and priority information: some execution sequences which are apparently allowed by the semantics will in fact be excluded by a correct implementation. This means that the correctness of a large part of an implementation can be judged against this semantics, since any implementation which gives a value to a process which is not a refinement of that given here is certainly incorrect; what then remains is to check that it is a refinement allowed when the effects of priority and real-time behaviour (which are, in any case, to some extent implementation dependent) are taken into account.

Similarly, any effect of placement of processes on processors has not been taken into account. PLACED PAR is identified with PAR, and the command PROCESSOR and its associated expression are ignored. This has no effect on the semantics of well-behaved programs, but may provide a different interpretation of error and divergence in a network than expected.

The placement of channels, timers or variables at an absolute location in store is entirely dependent on the particular implementation of the store and is thus not considered. Hence the command PLACE...AT and its associated expression are ignored.

Finally, while giving a denotational semantics to the language, output over ports is identified with SKIP and input over ports with the assignment of an unspecified 'random' value to the designated variable. The motivation behind this decision centres around the fact that only communications with external devices may take place over ports, and the modifications (complete with their inherent complexities) necessary to give a more accurate interpretation do not appear justified.

Additions to the language Within this paper, an extension of occam 2 will be included. On defining processes in parallel, the option of prefixing *each* of the processes with a *Parallel Declaration* ($U \in PD$) will be given. When processes are defined in parallel, it is necessary to set up local alphabets for each individual process. Normally within occam 2 the necessary information comes from syntactic analysis of the parallel processes. Here the information can come in two forms. Firstly, if parallel declarations do not accompany the processes, it is possible by syntactic analysis of the parallel processes to determine consistently which channels are for input, which for output and which are internal. It can also be determined to which global variables each process intends to assign. Secondly, if each parallel process carries with it a parallel declaration, these declarations provide a concrete indication of the intended channel use of each of the processes, splitting the global channels each process intends to use 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.

For simplicity[2] it is insisted that if a process carries with it a parallel declaration, the declaration must also mention to which global variables the process intends to be able to assign.

$$U ::= \texttt{USING}\ (\texttt{OWNCHAN}\ \{chan\}, \texttt{INCHAN}\ \{chan\}, \texttt{OUTCHAN}\ \{chan\}, \texttt{VAR}\ \{var\})$$

At first glance it may appear that the inclusion of parallel declarations is superfluous. This is not the case since if a channel which is currently in scope is not mentioned by any of a collection of

[2]Although the issue of mentioning variables explicitly is orthogonal to that of mentioning channels explicitly, the resultant exponential increase in the number of clauses necessary to deal with parallel composition makes the inclusion of both options independently undesirable.

processes defined in parallel, syntactic analysis of the processes cannot be guaranteed to allocate the channel to a particular component.

The need for parallel declarations comes about from the decision to use as the semantic domain a derivative of the failures/divergences model for CSP, with parallel composition and hiding, rather than a derivative of a model for CCS, with parallel composition and restriction.

The inclusion of parallel declarations means undeclared channels are given a more natural meaning, and promotes conventions which are, in any case, good programming practice.

3 Denotational semantics

In this section, the model previously constructed will be used to give a natural denotational semantics to the whole of the language occam 2, up to the restrictions discussed at the end of the previous section. 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.

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

SEQ
  P_1
  P_2
  ⋮
  P_n

In the previous section an abstract space S of final states was introduced. In devising the space of machine states it is necessary to bear in mind the rôle which states play in the model: passing on information from one occam process to its successor. The only way one occam process can influence its successors is by modifying (through assignment or input) the values of variables: it cannot change the binding of identifiers outside its own text in any other way. In order to protect against a process altering the contents of locations from which it is only permitted to read, and to avoid unwanted distinctions between processes, final states will only record the contents of locations which may be written to. Thus the states will resemble the restriction of the 'store' - a function from locations to storable values (see below) - to the writable locations of the process.

Throughout this paper the store will be thought of as a map from addresses (the identifiers of atomic pieces of store) to atomic pieces of information. Because any process is capable of accessing only a limited set of variables, the store associated with any process will be a partial function. Reference to variables involves the use of multiple addresses, the number being dependent on the particular implementation of the store under consideration, and for this reason the decision has been taken to abstract away from addresses where possible and use 'locations' - pairs containing the starting address of the variable's contents in the store and the type of the variable (from which, given the implementation, it is possible to calculate how many addresses are used to refer to the contents of the variable). This abstraction limits the retyping which can be modelled but is necessary if maximal generality is to be maintained.

Because of the implementation dependent nature of the store, all access to it will be via the auxiliary functions *lookup* and *update* and the operators $\oplus$ and $\downarrow$.

$$lookup : S \longrightarrow LOC_{\uparrow} \longrightarrow \mathcal{V}_{\uparrow}$$

lookup takes in a store and an area of store which houses a variable and returns the contents of the variable in the store. Use of the domain $\mathcal{V}_{\uparrow}$ (the domain formed by lifting each element of $\mathcal{V}$ above the new element $\perp_{\mathcal{V}}$ while maintaining the internal order of $\mathcal{V}$) allows $\perp_{\mathcal{V}}$ to be returned if the variable has not been initialised. *lookup* is strict in each of its arguments.

$$update : S \longrightarrow LOC_{\uparrow} \longrightarrow \mathcal{V}_{\uparrow} \longrightarrow S$$

update takes in a store, an area of store which houses a variable and a storable value. The result is the supplied store, modified to map the given area of store to the storable value. Within the language it is possible to pass an array prefixed by an integer (indicating how many elements of the array should be considered) along a channel. The presence of this option means that the function *update* must allow assignment to part of an array, not corrupting the information held in the remainder of the array while doing so. *update* is strict in each of its arguments.

As the semantics are developed, and the set of mutable areas (set of addresses whose contents can be altered) changes, the need to deal with stores with different areas will become apparent. If σ_1 and σ_2 are two stores, they can be combined using $\oplus$.

$$\oplus : S \times S \longrightarrow S$$

$\sigma_1 \oplus \sigma_2$ joins two stores together, allowing σ_2 to influence the result on addresses they have in common.

The importance of restricting stores has already been emphasised; $\downarrow$ will be used for this purpose.

$$\downarrow : S \times \wp(ADDR) \longrightarrow S$$

$\sigma \downarrow A$ returns the store which is formed by picking the contents of σ indexed by addresses contained in A.

Each of the above rely heavily on the particular implementation of the store under consideration, and consequently their definitions are not included.

A separate *environment* will be used to map identifiers to locations, constant values, channels, procedures, protocols and so on.

This distinction between environment and store is a familiar idea in denotational semantics; the way the present model is constructed means it is again appropriate here. The management of environments and stores relative to sequential languages is well understood, and as in [26] the only potential problem is with handling the store within a PAR construct. As previously described, parallel occam processes do not use shared variables for communication. Global variables may only be used in a very restricted way: *either* one process can use a given variable normally *or* all processes can read from the variable but none can write to it. This idea corresponds to giving each parallel process a distinct portion of the store and reserving the remainder as read only for the duration of the parallel command. The state will be constructed 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 give a denotational semantics to occam processes, it is necessary to know the structure of the alphabet of communications between processes. As noted earlier, each communication will have two components – a vessel along which the value passes (an abstract channel of some protocol), and the value to be communicated (in the case of a non-variant protocol, a collection of storable values; in the case of a variant protocol, a collection of storable values prefixed by a tag).

Given that a process P uses the channels $\{\chi_0, \chi_1, \ldots, \chi_n\}$, the set of possible communications along channels which the process can undertake is, as was previously mentioned on page 4, $\alpha P = \{\chi_i.\beta \mid i \in \{1, \ldots, n\} \wedge \beta \in \overline{\chi_i}\}$. (Throughout this paper, $\overline{\chi}$ will be used as shorthand for $\overline{p}$ – the set of values communicable over an occam channel with protocol p – where p is the protocol associated with χ in the current scope. The definition of $\overline{p}$ for an arbitrary protocol p will be given later, on page 16, when the form in which protocols appear in the environment has been described.) Note that no distinction is made between 'input' and 'output' communications,

since the relevant information already appears in the alphabet components of the process and would be confusing to duplicate.

Processes ($P \in Proc$) The definition of the syntactic domain of processes, based on [20], is included in the appendix. All the constructs of the occam 2 language are present, but the restrictions previously discussed mean that certain constructs may be given a less deterministic meaning in the semantic domain than expected.

3.1 Semantic domains

The existence of the following semantic domains is supposed:

$v \in \mathcal{V}$ — domain of storable values. This contains elements of all data types and arrays thereof.

$u \in TIMER$ — domain of timers. An abstract set of tokens allowing identification of timers.

$\varpi \in PORT$ — domain of ports. An abstract set of tokens allowing identification of ports.

$\chi \in CHAN$ — domain of abstract channels, as referred to in the previous section.

$\alpha \in ADDR$ — domain of addresses in store, as referred to in the previous section.

$\iota \in TAG$ — domain of tags. An abstract set of tokens.

$x \in IDE$ — (syntactic) domain of identifiers.

[For the language under consideration, $\mathcal{V}$ has the form $(\overline{\texttt{BOOL}})^{\%} + (\overline{\texttt{BYTE}})^{\%} + (\overline{\texttt{INT}})^{\%} + (\overline{\texttt{INT16}})^{\%} + (\overline{\texttt{INT32}})^{\%} + (\overline{\texttt{INT64}})^{\%} + (\overline{\texttt{REAL32}})^{\%} + (\overline{\texttt{REAL64}})^{\%}$ where $\bar{t}$ is the set of elements of type t (e.g., $\overline{\texttt{BOOL}} = \{\underline{true}, \underline{false}\}$) and $X^{\%}$, as defined below, is the domain of arrays with components drawn from X.]

There is no need in this work to suppose that any of the above domains is partially ordered or contains a 'bottom' element. It will, however, be necessary to deal with errors. Given any semantic domain X, the domain $X \cup \{error\}$ will be denoted X^+. If X is partially ordered, then *error* will be incomparable with the other non-bottom elements of X^+.

For any semantic domain X, the domain of arrays each of whose components is drawn from X, $X^{\%}$, is of the form $\biguplus_{s \in \mathbb{N}^*} X \uparrow s$ where $X \uparrow \langle\rangle = X$ and $X \uparrow \langle n \rangle s = \langle X \uparrow s \rangle^n$. Use of this seemingly complicated definition ensures that for every dimension of the array, each of the components is of the same type. When the domain $X^{\%}$ is used, it will often be useful to be able to extract the set of elements contained within a particular instance of the domain. This will be done with the aid of the auxiliary function *elements* which for $x \in X$ is defined

$$\begin{aligned} \mathit{elements}\ \langle\rangle &= \emptyset \\ \mathit{elements}\ (x : xs) &= \{x\} \cup (\mathit{elements}\ xs) \\ \mathit{elements}\ (xs : xss) &= (\mathit{elements}\ xs) \cup (\mathit{elements}\ xss) \end{aligned}$$

Given a domain X, X^* will denote the domain consisting of sequences of zero or more elements of X, and $X^{\&}$ will denote the domain consisting of sequences of one or more elements of X. Notationally, elements of X^* and $X^{\&}$ will be regarded as partial functions from $\mathbb{N}$ to X. If X has a partial order, in X^* and $X^{\&}$ sequences of the same length are ordered component-wise, with sequences of different lengths being incomparable.

In the domain $(X^*)_\uparrow$, formed by lifting each element of X^* above the new element $\bot$ while maintaining the internal order of X^*, concatenation of sequences is strict in both arguments (i.e., $\bot \, s = \bot$ and $s \bot = \bot$ for any sequence s).

If $s \in X^*$, $s[n]$ $(0 \leq n < \#s)$ denotes the n th element of s, and $s[n \ldots m]$ $(0 \leq n, n \leq m, m \leq \#s)$ denotes the sequence containing the n th to $(m - 1)$ th elements of s inclusively.

In order to give a denotational semantics to the language, several specific semantic domains need to be constructed. This is done using the above notation.

$$LOC = ADDR \times TYP$$

LOC is the domain of locations, used to provide access to blocks of store representing variables. A location is a pair – the first component contains the starting address of the block and the second component contains the type of variable stored in the block. From this information (and the particular implementation of the store under consideration) it is possible to determine which addresses make up the required block. Within the definition of LOC the domain TYP of types of variables has been used. The definition of this domain (which consists of a set of tokens), along with those of the other domains concerned with type information, are included next.

$$\begin{aligned} TYP &= \mathbb{N}^* \times (\{\texttt{BOOL}\} \cup INTEGER \cup REAL) \\ INTEGER &= \{\texttt{BYTE}, \texttt{INT}, \texttt{INT16}, \texttt{INT32}, \texttt{INT64}\} \\ REAL &= \{\texttt{REAL32}, \texttt{REAL64}\} \end{aligned}$$

TYP is intended to provide the type of variables. Within the definition primitive and array types must be catered for, and to facilitate later work it is beneficial to have separate subcomponents for each family of types. For simplicity the syntactic names of occam types (e.g., BOOL, BYTE, INT) have been used as the abstract tokens. Arrays are represented by a natural number sequence recording the dimensions (e.g., $[1][2][3]$ INT is represented by $(\langle 1, 2, 3 \rangle, \texttt{INT})$). For any type t, the set of elements of type t will be denoted $\overline{t}$. Care must be taken to make this distinction, since $\overline{\texttt{INT}}$ (a set of values) differs considerably from INT (simply a token).

$$\begin{aligned} CTYP1 &= TAG \; DATA^* \\ CTYP2 &= DATA^{\&} \\ DATA &= TYP + (INTEGER :: TYP) \end{aligned}$$

$CTYP1$ and $CTYP2$ are the domains of communicable types. They contain the types which a communication along a channel may have, and hence must deal with variable length arrays and communications on channels with variant or sequential protocols. In order to facilitate the construction of the domain of channel protocols, one domain ($CTYP1$) has been used to deal with tagged types and another ($CTYP2$) has been used to deal with untagged types.

$$PTYP = (\mathbb{N} \cup \{\bot\})^* \times (\{\texttt{BOOL}\} \cup INTEGER \cup REAL)$$

$PTYP$ is the domain of parameter types which will be used in procedures and functions. The difference between this domain and the one used to record variable types is that if a parameter is an array, it need not explicitly give the size of any of its dimensions. To deal with this, an extra token $\bot$ has been included in the set of possible array dimensions.

$$PROT = \{\texttt{ANY}\} \cup CTYP1^{\&} \cup CTYP2$$

$PROT$ is the domain of channel protocols. As expected the definition draws heavily on that of communicable types, but it is also necessary to take account of the anarchic protocol ANY. Communication on a channel with non-variant protocol must always consist of the same sequence

of types, but communication on a channel with variant protocol may consist of any one of a number of tags followed by its corresponding (possibly empty) sequence of types. The decision to provide two domains of communicable types allows this restriction to be clearly conveyed.

$$ENV = (IDE \longrightarrow D^+) \times VSTATUS \times LSTATUS$$

ENV is the domain of environments. The constituent domains are defined below, along with a brief explanation as to their purpose. The use of D^+ allows identifiers which have not been declared in the scope to map to *error*.

If $\rho \in ENV$ then ρ_I, ρ_V and ρ_L will denote the first, second and third components respectively (so that $\rho = \langle \rho_I, \rho_V, \rho_L \rangle$). If $x \in IDE$ then $\rho[\![x]\!]$ will mean $\rho_I[\![x]\!]$; similarly $\rho[\![\chi]\!]$ will mean $\rho_V[\![\chi]\!]$ ($\chi \in CHAN$) and $\rho[\![\alpha]\!]$ will mean $\rho_L[\![\alpha]\!]$ ($\alpha \in ADDR$). 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. Corresponding interpretations will be put on $\rho[r/\chi]$ and $\rho[r/\alpha]$.

$$VSTATUS = CHAN \rightarrowtail (PROT \times \{\underline{f}, \underline{u}, \underline{i}, \underline{o}\})$$

VSTATUS is the domain of functions intimating the status of those abstract channels which can be used within the current scope. Abstract channels not in the domain of *VSTATUS* cannot be used within the current scope; this protects against parallel processes attempting to claim the same abstract channel for different and unconnected purposes. In any particular function, each such abstract channel is mapped to a pair, the first component indicating the protocol of the occam channel currently associated with the abstract channel and the second component a token depending on the use of the abstract channel.

- $\underline{f}$ means that the channel has not been assigned to a particular identifier within the current scope, but is free to be associated with one if required.
- $\underline{u}$ means that the channel has been assigned to a particular identifier within the current scope, but has not had its direction determined.
- $\underline{i}$ means that the channel has been assigned to a particular identifier within the current scope, and is restricted to participating in only input communications.
- $\underline{o}$ means that the channel has been assigned to a particular identifier within the current scope, and is restricted to participating in only output communications.

$$LSTATUS = ADDR \rightarrowtail \{\underline{f}, \underline{r}, \underline{u}\}$$

LSTATUS is the domain of functions intimating storage use. Addresses not in the domain of *LSTATUS* cannot be used within the current scope; this affords protection against parallel processes using the store in an unacceptable way. A typical member maps each address accessible within the current scope to a token, the choice depending on the status of the address.

- $\underline{f}$ means that within the current scope the address is not associated with a variable, but can if necessary be associated with a 'read-only' or 'read/write' variable.
- $\underline{r}$ means that within the current scope the address is associated with a global 'read-only' variable or a variable which is the subject of a value abbreviation. The variable associated with the address can be read from but not written to.

$\underline{u}$ means that within the current scope the address is associated with a variable having unrestricted use. That is, the variable associated with the address can be written to as well as read from.

The domains $VSTATUS$ and $LSTATUS$ contain the information necessary to form the alphabet components of a process description.

$$D = LOC + (\underline{a} \times \wp(ADDR) \times LOC) + \mathcal{V} + TIMER^{\%} + (TYP \times PORT^{\%}) + PROT + CHAN^{\%} + NP + TAG$$

D is the domain of denotable values, storing the information associated with an identifier which will not change. Each component deals with a particular genre of object which an identifier could denote.

LOC deals with variables. As previously mentioned, given the starting address and type of a variable, one can uniquely determine the area of store in which its current value can be found. The variable may be of primitive or array type.

$\underline{a} \times \wp(ADDR) \times LOC$ deals with variables abbreviated within the current scope. The only access to such a variable which is allowed is a further abbreviation of a disjoint part of the variable. Verifying that further abbreviations refer to disjoint parts is made possible by keeping a record, in $\wp(ADDR)$, of the addresses associated with the variable whose contents are currently abbreviated.

$\mathcal{V}$ deals with constants. In occam the type of a constant, again of primitive or array type, can be uniquely determined from its value and hence type information need not be included here.

$TIMER^{\%}$ deals with timers, mapping each such item to an abstract timer. Although with the current interpretation of timing it would be sufficient to map every timer to the same token, this is unnatural and would lead to problems if timing were to be modelled.

$TYP \times PORT^{\%}$ deals with ports. As with timers the fact that it would be sufficient to map each port to the same token is ignored. TYP records the intended type of each of the ports.

$PROT$ deals with named protocols, the form of the protocol being stored for future examination.

$CHAN^{\%}$ deals with channels. The protocol associated with an occam channel can be extracted by examination of the function $VSTATUS$ and hence protocol information need not be included here.

NP deals with named procedures and functions. Its precise form will be defined later (on page 48) when it is required.

TAG deals with the tags necessary for channels of variant protocol.

Within the above definition of D, domains of the form $X^{\%}$ (arrays whose components are drawn from the set X, as described on page 12) appear. The use of such domains is necessary since when arrays of channels, timers or ports are declared, one abstract channel, timer or port must be associated with each individual component of the array.

In the course of defining semantic functions a few further semantic domains will be used. Definitions will be given as the semantic domains appear.

Communicable values Having described the semantic domains above, it is now possible to give a concise definition of $\overline{\vartheta}$ for each instance of ϑ.

For $\vartheta \in PROT$, $\overline{\vartheta}$ is to be the set of communicable values over a channel of protocol ϑ; for $\vartheta \in TYP$, $\overline{\vartheta}$ is to be the set of elements of type ϑ [3]. Hence:

$$
\begin{aligned}
\overline{\langle v_0 \ \ldots \ v_{n-1} \ \iota_n \rangle} &= \overline{\langle v_0 \ \ldots \ v_{n-1} \rangle} \cup \{\iota_n\} \\
\overline{\langle v_0 \ \ldots \ v_{n-1} \ (\iota_n \ w_n) \rangle} &= \overline{\langle v_0 \ \ldots \ v_{n-1} \rangle} \cup (\{\iota_n\} \times \overline{w_n}) \\
\overline{\langle p_0 \ \ldots \ p_n \rangle} &= \overline{p_0} \times \cdots \times \overline{p_n} \\
\overline{s :: t} &= \bigcup_{k=0}^{maxval\, s} \{k\} \times (\overline{t})^k \\
\overline{((n)ns.\, \tau)} &= \left(\overline{(ns, \tau)}\right)^n \\
\overline{\texttt{BOOL}} &= \{\underline{true}, \underline{false}\} \\
\overline{s} &= \{n \mid minval\, s \leqslant n \leqslant maxval\, s\} \\
\overline{f} &= \text{the set of values expressible in the real type } f \\
\overline{\texttt{ANY}} &= \bigcup_{k=1}^{maxval\, \texttt{INT}} (\overline{\texttt{BYTE}})^k
\end{aligned}
$$

[Here τ stands for an arbitrary primitive type, t stands for an arbitrary primitive or array type, s stands for an integer type (including BYTE), and *minval* and *maxval* respectively denote the smallest and largest values expressible in a given type. An arbitrary element of the domain $DATA$ is denoted by p_j, an arbitrary sequence of elements of $DATA$ by w_j, and an arbitrary sequence of elements of $DATA$ prefixed by a tag by v_j. The symbols ι_j denote elements of the domain of abstract tags (TAG).]

3.2 The semantics

Detailed descriptions of several of the necessary semantic functions will be given, with most attention being paid to the 'higher level' semantic functions. The remainder are fairly standard, and should not prove too taxing for the reader to define. The main semantic functions are listed below.

$$\mathcal{C} : Proc \longrightarrow ENV \longrightarrow S \longrightarrow Q$$

This is the function used to give a semantic meaning to an occam process. Given a program segment, an environment and a store it yields an element of Q. In the definitions below, all execution errors map a process to the minimal element with appropriate alphabet (e.g., an erroneous process P – with environment ρ (yielding the alphabet components C and L) and store σ – is mapped to $\perp_{(C,L)}$) from the point in its communication history where the error arises. 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

[3] Since $PROT$ incorporates TYP as a specific case, by defining the operator over $PROT$ the (expected) meaning over TYP is automatically inherited.

domain. The present approach, however, has the advantage of simplicity. The identification of errors with the bottom element uses the strictness of the model and semantic operators to give a severe if elegant view of errors. Just as understanding the implications of an execution error in one component of a parallel network on the behaviour of the whole is more complex than in a sequential one, so the introduction of detailed error-handling into the naturally-occurring domain and the semantics is more trouble than is warranted in most circumstances, though it could be done by introducing extra error elements throughout the domain.

$$\begin{array}{lcl} \mathcal{C}_1 & : & Proc \longrightarrow ENV \longrightarrow S \longrightarrow \mathcal{P}((\alpha P)^* \times \mathcal{P}(\mathrm{dom}(C) \cup \{\checkmark\})) \\ \mathcal{C}_2 & : & Proc \longrightarrow ENV \longrightarrow S \longrightarrow ((\alpha P)^* \longrightarrow \wp(S) \cup \{\bot\}) \\ \phi & : & ENV \longrightarrow (CHAN \rightarrowtail \{in, out\} \times PROT) \\ \varphi & : & ENV \longrightarrow (ADDR \rightarrowtail \{ro, rw\}) \end{array}$$

These functions calculate the individual components of $\mathcal{C}[\![P]\!]\rho\sigma$, so that

$$\mathcal{C}[\![P]\!]\rho\sigma = \langle \mathcal{C}_1[\![P]\!]\rho\sigma, \mathcal{C}_2[\![P]\!]\rho\sigma, \phi\,\rho, \varphi\,\rho \rangle.$$

The third and fourth components of $\mathcal{C}[\![P]\!]\rho\sigma$ do not depend on the state σ; passing around additional arguments on which functions do not depend is unnatural and for this reason (as well as considerations of clarity) the decision was taken not to supply σ as an argument to either ϕ or φ.

$$\mathcal{D} : Decl \longrightarrow ENV \longrightarrow ENV_\perp$$

This function carries out the modifications to the environment caused by declarations. When an error occurs, the value produced is the bottom environment $\perp$. In what follows an arbitrary member of *Decl* will be denoted Δ.

$$\mathcal{A} : Spec \longrightarrow ENV \longrightarrow S \longrightarrow ENV_\perp$$

In occam it is possible to use specifications (abbreviations and retyping) as well as declarations within a process. This function carries out the modifications to the environment caused by specifications. Erroneous specifications lead to $\perp$ being returned. An arbitrary member of *Spec* will be denoted Θ.

$$\begin{array}{lcl} cv & : & (CHAN^{\%} + TIMER^{\%} + (TYP \times PORT^{\%})) \longrightarrow \\ & & (\overline{\mathrm{INT}} + (\overline{\mathrm{INT}} \times \overline{\mathrm{INT}})) \longrightarrow (CHAN^{\%} + TIMER^{\%} + (TYP \times PORT^{\%}))^+ \\ lv & : & LOC \longrightarrow (\overline{\mathrm{INT}} + (\overline{\mathrm{INT}} \times \overline{\mathrm{INT}})) \longrightarrow LOC^+ \end{array}$$

These are important auxiliary functions which simplify the definitions of the main semantic functions by extracting components of arrays. *cv* returns the portion of its supplied array of channels, timers or ports referred to by its second argument; *lv* returns the portion of its supplied location referred to by its second argument. The definitions should help to clarify the store model being used and the distinction between locations (pairs of types and starting addresses) and addresses.

The definition of the function cv is:

$$\begin{aligned}
cv\ xs\ n &= xs[n] \\
&\quad \text{if } xs \in CHAN^{\%} \wedge 0 \leq n < \#xs \\
cv\ xs\ n &= xs[n] \\
&\quad \text{if } xs \in TIMER^{\%} \wedge 0 \leq n < \#xs \\
cv\ (\tau, xs)\ n &= (\tau, xs[n]) \\
&\quad \text{if } (\tau, xs) \in TYP \times PORT^{\%} \wedge 0 \leq n < \#xs \\
cv\ xs\ (n, m) &= xs[n \ldots n+m] \\
&\quad \text{if } xs \in CHAN^{\%} \wedge 0 \leq n \wedge 0 \leq m \wedge n+m \leq \#xs \\
cv\ xs\ (n, m) &= xs[n \ldots n+m] \\
&\quad \text{if } xs \in TIMER^{\%} \wedge 0 \leq n \wedge 0 \leq m \wedge n+m \leq \#xs \\
cv\ (\tau, xs)\ (n, m) &= (\tau, xs[n \ldots n+m]) \\
&\quad \text{if } (\tau, xs) \in TYP \times PORT^{\%} \wedge 0 \leq n \wedge 0 \leq m \wedge n+m \leq \#xs \\
cv\ y\ z &= error \\
&\quad \text{otherwise}
\end{aligned}$$

While that of lv is:

$$\begin{aligned}
lv\ (\alpha, (\langle z \rangle xs, \tau))\ n &= (\alpha', (xs, \tau)) \\
&\quad \text{where } startaddr\ (\alpha, (\langle z \rangle xs, \tau))\ n = \alpha' \\
&\quad \text{if } 0 \leq n < z \\
lv\ (\alpha, (\langle z \rangle xs, \tau))\ (n, m) &= (\alpha', (\langle m \rangle xs, \tau)) \\
&\quad \text{where } startaddr\ (\alpha, (\langle z \rangle xs, \tau))\ n = \alpha' \\
&\quad \text{if } 0 \leq n \wedge 0 \leq m \wedge n+m \leq z \\
lv\ \lambda\ y &= error \\
&\quad \text{otherwise}
\end{aligned}$$

In the definition of lv the auxiliary function *startaddr* has been used. This function takes in a store location (which is currently associated with an array) and an index and returns the starting address of the array component with the given index. Its definition is implementation dependent and hence is not included.

$$\mathcal{E} : Exp \longrightarrow ENV \longrightarrow S \longrightarrow E_l$$

Within the definition of this function the domain E has been used. E is the domain of expressible values and has the form

$$E = LOC + (\underline{a} \times \wp(ADDR) \times LOC) + CHAN^{\%} + TIMER^{\%} + (TYP \times PORT^{\%}) + \mathcal{V}^{\&}$$

It is necessary to include $\mathcal{V}^{\&}$ as a component of E in preference to $\mathcal{V}$ since value processes in occam are not restricted to returning a single expression.

The purpose of this function is to evaluate the natural value of expressions (e.g., if the expression refers to a location, the location and not its contents is returned) with the aim of minimising the amount of work which must be repeated to deal with expressions as *l-values* and *r-values*. The result of the function is an element of E_l, the domain derived from E by the addition of an extra element $\perp_{\mathcal{E}}$ which will represent all 'errors', both those detectable at run-time and those which result in non-termination. The domain is ordered with $\perp_{\mathcal{E}}$ below each of the

elements of E. Use of this domain allows all expression evaluation errors to be mapped to the same element, while allowing refinements to accommodate a certain amount of error recovery. One possible refinement would be the use of the domain $(E^+)_\perp$, with immediately detectable errors (e.g., division by zero) being mapped to *error*, and all other errors (e.g., a looping function call) being mapped to $\perp_\mathcal{E}$.

$$\begin{array}{lcl}
\mathcal{E}_V & : & Exp \longrightarrow ENV \longrightarrow S \longrightarrow V_\perp \\
\mathcal{E}_{V^*} & : & Exp \longrightarrow ENV \longrightarrow S \longrightarrow (V^*)_\perp \\
\mathcal{E}_L & : & Exp \longrightarrow ENV \longrightarrow S \longrightarrow LOC_\perp \\
\mathcal{E}_C & : & Exp \longrightarrow ENV \longrightarrow S \longrightarrow (CHAN + TIMER + (TYP \times PORT))_\perp
\end{array}$$

These functions evaluate an expression and return an element of a particular form. The target domain of each of the functions is $X_\perp$ for some X; in order to avoid a proliferation of 'bottom values', which would in turn lead to unwanted distinctions between processes, the bottom elements of each of the domains will be naturally identified with $\perp_\mathcal{E}$.

$\mathcal{E}_V$ takes in an expression, environment and store and returns a storable value (the *r-value* of the expression)[4]. If the expression evaluates to a storable value, it is returned; if it evaluates to a location, the contents of the location are returned; otherwise $\perp_\mathcal{E}$ is returned.

$\mathcal{E}_{V^*}$ takes in an expression, environment and store and returns a sequence of storable values (the sequence of *r-values* of the expression). For all non-erroneous expressions except value processes (VALOF) this sequence will have as its only element the result of $\mathcal{E}_V$; value processes, however, may have more than one storable value as their result and in this case the sequence of results of the value process is returned.

$\mathcal{E}_L$ takes in an expression, environment and store and returns a location (the *l-value* of the expression)[5]. If the expression refers to a slice or component of an array of variables, the starting address returned is that of the particular portion of the array referred to, and not necessarily the starting address of the array itself. If the expression does not evaluate to a mutable variable, $\perp_\mathcal{E}$ is returned.

$\mathcal{E}_C$ takes in an expression, environment and store and returns a channel, timer or port (the channel value or *c-value* of the expression). If the expression evaluates to a single channel, the abstract channel returned; if it evaluates to a single timer, the timer is returned; if it evaluates to a single port, the port along with its associated type is returned; otherwise $\perp_\mathcal{E}$ is returned.

The functions are all obtained by suitable coercions of $\mathcal{E}$.

$$\mathcal{E}_V[\![e]\!]\rho\sigma = \begin{cases} (\mathcal{E}[\![e]\!]\rho\sigma)[0] & \text{if } \mathcal{E}[\![e]\!]\rho\sigma \in V^* \wedge \#(\mathcal{E}[\![e]\!]\rho\sigma) = 1 \\ lookup\ \sigma\ (\mathcal{E}[\![e]\!]\rho\sigma) & \text{if } \mathcal{E}[\![e]\!]\rho\sigma \in LOC \wedge \mathcal{E}[\![e]\!]\rho\sigma = (\alpha, t) \wedge \rho[\![\alpha]\!] \in \{\underline{r}, \underline{u}\} \\ \perp_\mathcal{E} & \text{otherwise} \end{cases}$$

$$\mathcal{E}_{V^*}[\![e]\!]\rho\sigma = \begin{cases} \mathcal{E}[\![e]\!]\rho\sigma & \text{if } \mathcal{E}[\![e]\!]\rho\sigma \in V^* \\ \langle lookup\ \sigma\ (\mathcal{E}[\![e]\!]\rho\sigma)\rangle & \text{if } \mathcal{E}[\![e]\!]\rho\sigma \in LOC \wedge \mathcal{E}[\![e]\!]\rho\sigma = (\alpha, t) \wedge \rho[\![\alpha]\!] \in \{\underline{r}, \underline{u}\} \\ \perp_\mathcal{E} & \text{otherwise} \end{cases}$$

$$\mathcal{E}_L[\![e]\!]\rho\sigma = \begin{cases} \mathcal{E}[\![e]\!]\rho\sigma & \text{if } \mathcal{E}[\![e]\!]\rho\sigma \in LOC \wedge \mathcal{E}[\![e]\!]\rho\sigma = (\alpha, t) \wedge \rho[\![\alpha]\!] = \underline{u} \\ \perp_\mathcal{E} & \text{otherwise} \end{cases}$$

[4]This function is often referred to as $\mathcal{R}$ in the literature. Within this paper the existence of channels means that the use of $\mathcal{R}$ is not particularly suitable.

[5]This function is often referred to as $\mathcal{L}$ in the literature. Again the existence of channels within this paper limits the suitability of the use of $\mathcal{L}$.

$$\mathcal{E}_C[\![e]\!]\rho\sigma = \begin{cases} \mathcal{E}[\![e]\!]\rho\sigma & \text{if } \mathcal{E}[\![e]\!]\rho\sigma \in CHAN \wedge \mathcal{E}[\![e]\!]\rho\sigma = \chi \wedge \rho[\![\chi]\!] \in \{(p,\underline{u}),(p,\underline{i}),(p,\underline{o})\} \\ \mathcal{E}[\![e]\!]\rho\sigma & \text{if } \mathcal{E}[\![e]\!]\rho\sigma \in TIMER \\ \mathcal{E}[\![e]\!]\rho\sigma & \text{if } \mathcal{E}[\![e]\!]\rho\sigma \in (TYP \times PORT) \\ \perp_{\mathcal{E}} & \text{otherwise} \end{cases}$$

The next step will be to concentrate on the four main semantic functions in turn. Each of the clauses will be given a brief explanation.

(1) THE FUNCTION $\mathcal{E}$ Several 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 $\perp_{\mathcal{E}}$. Throughout the definition of $\mathcal{E}$, $(\ldots)_\perp$ has been used in place of the more usual $(\ldots)$. The operator $(\ldots)_\perp$ is a strict sequence constructor (returning $\perp_{\mathcal{E}}$ if any of its arguments are $\perp_{\mathcal{E}}$ but otherwise agreeing with $(\ldots)$) and is used in order to maintain strictness when considering erroneous expressions. Within the definition of $\mathcal{E}$, when operators are applied, the auxiliary function $\mathcal{O}$ appears. This function, which is assumed to produce results strict in each of their arguments, translates a syntactic operator to the relevant function; its definition is standard and hence omitted. The definition of $\mathcal{N}$, an auxiliary function evaluating literals, is also standard and omitted for brevity.

$$\begin{aligned} \mathcal{E}[\![op\ e]\!]\rho\sigma &= (\mathcal{O}[\![op]\!](v))_\perp \\ &\quad \text{where } \mathcal{E}_V[\![e]\!]\rho\sigma = v \end{aligned}$$

Applying a monadic operator to an expression first involves evaluation of the expression. Once this has been done, the operator is applied. The assumption of type-correct programs means that the result of the evaluation of the expression is guaranteed to be within the domain of the operator.

$$\begin{aligned} \mathcal{E}[\![e_1 \text{ AND } e_2]\!]\rho\sigma &= \begin{cases} \langle \underline{false} \rangle & \text{if } \mathcal{E}_V[\![e_1]\!]\rho\sigma = \underline{false} \\ (\mathcal{E}_V[\![e_2]\!]\rho\sigma)_\perp & \text{otherwise} \end{cases} \\ &\quad \text{provided } \mathcal{E}_V[\![e_1]\!]\rho\sigma \in \overline{\text{BOOL}} \end{aligned}$$

The boolean operator AND is not strict in its right argument. For this reason it is necessary to include explicitly the clause dealing with its evaluation.

$$\begin{aligned} \mathcal{E}[\![e_1 \text{ OR } e_2]\!]\rho\sigma &= \begin{cases} \langle \underline{true} \rangle & \text{if } \mathcal{E}_V[\![e_1]\!]\rho\sigma = \underline{true} \\ (\mathcal{E}_V[\![e_2]\!]\rho\sigma)_\perp & \text{otherwise} \end{cases} \\ &\quad \text{provided } \mathcal{E}_V[\![e_1]\!]\rho\sigma \in \overline{\text{BOOL}} \end{aligned}$$

The boolean operator OR is not strict in its right argument. Again it is necessary to include explicitly the clause dealing with its evaluation.

$$\begin{aligned} \mathcal{E}[\![e_1\ op\ e_2]\!]\rho\sigma &= (\mathcal{O}[\![op]\!](v_1, v_2))_\perp \\ &\quad \text{where } \mathcal{E}_V[\![e_1]\!]\rho\sigma = v_1 \wedge \mathcal{E}_V[\![e_2]\!]\rho\sigma = v_2 \end{aligned}$$

When applying a dyadic operator to two values, the first step is the evaluation of both of the arguments. Next the operator is applied. As before the type-correctness assumption guarantees a pair of values within the domain of the operator.

$$\mathcal{E}[\![\tau\ e]\!]\rho\sigma = \langle convert\ \tau\ (\mathcal{E}_V[\![e]\!]\rho\sigma) \rangle_\perp$$

Since only type correct programs are considered, it is not necessary to include a collection of rules summarising acceptable conversions between types. The only step which must be carried out is transferring the evaluated expression to the correct part of $\mathcal{V}$. This task is achieved by the auxiliary function *convert*, the definition of which is not difficult and hence left for the reader. For any type τ, the function *convert* τ is strict.

$$\mathcal{E}\llbracket \tau \text{ ROUND } e \rrbracket \rho\sigma \quad = \quad \langle \mathit{round}\ \tau\ (\mathcal{E}_V \llbracket e \rrbracket \rho\sigma) \rangle_\perp$$

Type conversion can be coupled with rounding. The behaviour is similar to that of the type conversion described above, except that the auxiliary function must now act on the result in a different way. The function *round* (again left for the reader to define) is assumed to carry out this task. For any type τ, *round* τ is strict.

$$\mathcal{E}\llbracket \tau \text{ TRUNC } e \rrbracket \rho\sigma \quad = \quad \langle \mathit{trunc}\ \tau\ (\mathcal{E}_V \llbracket e \rrbracket \rho\sigma) \rangle_\perp$$

A third form of type conversion exists in occam. Instead of rounding an answer, it is possible to insist instead on truncation. The only semantic difference is the form of the auxiliary function; the auxiliary function *trunc* (again left for the reader to define) is assumed to carry out truncation. For any type τ, *trunc* τ is strict.

$$\mathcal{E}\llbracket x \rrbracket \rho\sigma = \begin{cases} \rho\llbracket x \rrbracket & \text{if } \rho\llbracket x \rrbracket \in LOC + (\underline{a} \times \wp(ADDR) \times LOC) + \\ & \qquad CHAN^{\%} + TIMER^{\%} + (TYP \times PORT^{\%}) \\ \langle \rho\llbracket x \rrbracket \rangle_\perp & \text{if } \rho\llbracket x \rrbracket \in \mathcal{V} \\ \perp_\mathcal{E} & \text{otherwise} \end{cases}$$

Evaluation of an identifier depends on its current use. If the identifier represents a location (of an abbreviated or unabbreviated variable), a channel (or array thereof), a timer (or array thereof), a port (or array thereof), the information contained in the environment is returned; if the identifier represents a constant, the sequence containing the constant as its only element is returned; otherwise $\perp_\mathcal{E}$ is returned.

$$\mathcal{E}\llbracket e_1[e_2] \rrbracket \rho\sigma \quad = \quad \begin{cases} \langle vs[0][n] \rangle_\perp & \text{if } \mathcal{E}\llbracket e_1 \rrbracket \rho\sigma \in \mathcal{V}^{\&} \wedge \mathcal{E}\llbracket e_1 \rrbracket \rho\sigma = vs \wedge 0 \le n < \#(vs[0]) \\ lv\ \lambda\ n & \text{if } \mathcal{E}\llbracket e_1 \rrbracket \rho\sigma \in LOC \wedge \mathcal{E}\llbracket e_1 \rrbracket \rho\sigma = \lambda \wedge lv\ \lambda\ n \neq \mathit{error} \\ (\underline{a}, A, lv\ \lambda\ n) & \text{if } \mathcal{E}\llbracket e_1 \rrbracket \rho\sigma \in (\underline{a} \times \wp(ADDR) \times LOC) \wedge \\ & \qquad \mathcal{E}\llbracket e_1 \rrbracket \rho\sigma = (\underline{a}, A, \lambda) \wedge lv\ \lambda\ n \neq \mathit{error} \\ cv\ x\ n & \text{if } \mathcal{E}\llbracket e_1 \rrbracket \rho\sigma \in CHAN^{\%} + TIMER^{\%} + \\ & \qquad\qquad (TYP \times PORT^{\%}) \wedge \\ & \qquad \mathcal{E}\llbracket e_1 \rrbracket \rho\sigma = x \wedge cv\ x\ n \neq \mathit{error} \\ \perp_\mathcal{E} & \text{otherwise} \end{cases}$$

where $\mathcal{E}_V \llbracket e_2 \rrbracket \rho\sigma = n$

provided $\mathcal{E}_V \llbracket e_2 \rrbracket \rho\sigma \in \overline{\text{INT}}$

In the case of an indexed expression, the index is first evaluated. Once this has been done, the remainder of the expression is evaluated and the relevant action (dependent on the form of the evaluated expression) is taken. The cases of indexed locations and indexed arrays of channels, timers or ports are dealt with by the auxiliary functions *lv* and *cv* defined previously.

$$\mathcal{E}\llbracket e \text{ FROM } e_1 \text{ FOR } e_2 \rrbracket \rho\sigma = \begin{cases} (vs[0][n \ldots n+m])_\perp \\ \quad \text{if } \mathcal{E}\llbracket e \rrbracket \rho\sigma \in \mathcal{V}^* \wedge \mathcal{E}\llbracket e \rrbracket \rho\sigma = vs \wedge \\ \quad\quad 0 \le n \wedge 0 \le m \wedge n+m \le \#(vs[0]) \\ lv\ \lambda\,(n,m) \\ \quad \text{if } \mathcal{E}\llbracket e \rrbracket \rho\sigma \in LOC \wedge \mathcal{E}\llbracket e \rrbracket \rho\sigma = \lambda \wedge \\ \quad\quad lv\ \lambda\,(n,m) \neq error \\ (\underline{a}, A, lv\ \lambda\,(n,m)) \\ \quad \text{if } \mathcal{E}\llbracket e \rrbracket \rho\sigma \in (\underline{a} \times \wp(ADDR) \times LOC) \wedge \\ \quad\quad \mathcal{E}\llbracket e \rrbracket \rho\sigma = (\underline{a}, A, \lambda) \wedge lv\ \lambda\,(n,m) \neq error \\ cv\ x\ (n,m) \\ \quad \text{if } \mathcal{E}\llbracket e \rrbracket \rho\sigma \in CHAN^{\%} + TIMER^{\%} + \\ \quad\quad\quad (TYP \times PORT^{\%}) \wedge \\ \quad\quad \mathcal{E}\llbracket e \rrbracket \rho\sigma = x \wedge cv\ x\ (n,m) \neq error \\ \perp_{\mathcal{E}} \quad \text{otherwise} \end{cases}$$

$$\text{where } \mathcal{E}_V\llbracket e_1 \rrbracket \rho\sigma = n \wedge \mathcal{E}_V\llbracket e_2 \rrbracket \rho\sigma = m$$

$$\text{provided } \mathcal{E}_V\llbracket e_1 \rrbracket \rho\sigma \in \overline{\text{INT}} \wedge \mathcal{E}_V\llbracket e_2 \rrbracket \rho\sigma \in \overline{\text{INT}}$$

Array slices are treated in the expected way, again relying of the functionality of the previously defined auxiliary functions lv and cv.

$$\mathcal{E}\llbracket v \rrbracket \rho\sigma = \langle \mathcal{N}\llbracket v \rrbracket \rangle_\perp$$

Within an expression it is possible to include literals. Such expressions are treated in the obvious way. The syntax of occam is such that the type of any literal is unambiguous.

$$\mathcal{E}\llbracket [e_1, e_2, \ldots, e_n] \rrbracket \rho\sigma = (\langle v_1, v_2, \ldots, v_n \rangle_\perp)_\perp$$
$$\text{where } \mathcal{E}_V\llbracket e_1 \rrbracket \rho\sigma = v_1 \wedge \mathcal{E}_V\llbracket e_2 \rrbracket \rho\sigma = v_2 \wedge \cdots \wedge \mathcal{E}_V\llbracket e_n \rrbracket \rho\sigma = v_n$$

Evaluating tables involves evaluating each element of the table. If any of the constituent expressions evaluate to $\perp_{\mathcal{E}}$, it is necessary to return $\perp_{\mathcal{E}}$; otherwise the table of evaluated expressions is returned.

$$\mathcal{E}\llbracket \text{MOSTPOS } \tau \rrbracket \rho\sigma = \langle maxval\ \tau \rangle_\perp$$
$$\mathcal{E}\llbracket \text{MOSTNEG } \tau \rrbracket \rho\sigma = \langle minval\ \tau \rangle_\perp$$

The two clauses above give respectively the largest and smallest elements of any integer type. The functions necessary to calculate such elements already exist and have thus been used directly.

$$\mathcal{E}\llbracket \text{VALOF } P \text{ RESULT } e_1, \ldots, e_n \rrbracket \rho\sigma = vs_1\, vs_2 \ldots vs_n$$
$$\text{where } \mathcal{C}_2\llbracket P \rrbracket \rho\sigma\langle\rangle = \{\sigma'\} \wedge \sigma \oplus \sigma' = \sigma'' \wedge$$
$$\mathcal{E}_{V^*}\llbracket e_1 \rrbracket \rho\sigma'' = vs_1 \wedge \cdots \wedge \mathcal{E}_{V^*}\llbracket e_n \rrbracket \rho\sigma'' = vs_n$$
$$\text{provided } \mathcal{C}_2\llbracket P \rrbracket \rho\sigma\langle\rangle \in \wp(S) \wedge \#(\mathcal{C}_2\llbracket P \rrbracket \rho\sigma\langle\rangle) = 1 \wedge$$
$$(n = 1 \vee \forall i \in \{1, \ldots, n\} \bullet \#(\mathcal{E}_{V^*}\llbracket e_i \rrbracket \rho\sigma'') = 1)$$

A VALOF command is evaluated by first calculating the termination states resulting from the body of the command with empty trace. It is sufficient to consider only such states since the restrictions

on occam processes force the body of a VALOF command to be a deterministic, sequential and non-communicating process. If precisely one termination state results, the required expressions are evaluated in the store updated with the information contained in the termination state. A VALOF command is restricted in what it can return: *either* it can return a sequence of one or more values each consisting of a single expression or it can return precisely one value consisting of a sequence of one or more expressions. Checking that this condition is satisfied is a necessary feature of the clause. Notice that a VALOF command attempting to return an erroneous expression is dealt with by the strictness of concatenation of sequences previously described.

$$\begin{aligned}\mathcal{E}[\![\Delta : e]\!]\rho\sigma &= \mathcal{E}[\![e]\!](\mathcal{D}[\![\Delta]\!]\rho)\sigma \\ &\quad \text{provided } \mathcal{D}[\![\Delta]\!]\rho \neq \perp\end{aligned}$$

If a declaration occurs before a (valof) expression, the necessary updating of the environment must take place before the expression is evaluated.

$$\begin{aligned}\mathcal{E}[\![\Theta : e]\!]\rho\sigma &= \mathcal{E}[\![e]\!](\mathcal{A}[\![\Theta]\!]\rho\sigma)\sigma \\ &\quad \text{provided } \mathcal{A}[\![\Theta]\!]\rho\sigma \neq \perp\end{aligned}$$

When a specification (an abbreviation or retyping) precedes a (valof) expression, the environment is updated as appropriate before the expression is evaluated.

The only clause left to define is that concerning functions. This is dependent on the way in which named functions are stored in the environment by declarations, and is consequently delayed until the function $\mathcal{D}$ has been defined.

(2) THE FUNCTION $\mathcal{C}$ This is the semantic function which gives meaning to an occam process. Many of the operators used are similar to those of [26], or those used in giving a semantics to CSP over the failures/divergences model. The construction of several of the operators is explained in detail in [7, 8]. Several 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 the bottom element with appropriate alphabet. For brevity, when no confusion as to the intended alphabet could arise, the bottom process with appropriate alphabet will be referred to as $\perp_Q$. The third and fourth components of the semantic function (ϕ and φ) are global, and suitable definitions for them are thus given first.

$$\phi\,\rho\chi = \begin{cases} (in, p) & \text{if } \rho[\![\chi]\!] = (p, \underline{i}) \\ (out, p) & \text{if } \rho[\![\chi]\!] = (p, \underline{o}) \\ \textit{undefined} & \text{otherwise} \end{cases}$$

$$\varphi\,\rho\alpha = \begin{cases} ro & \text{if } \rho[\![\alpha]\!] = \underline{r} \\ rw & \text{if } \rho[\![\alpha]\!] = \underline{u} \\ \textit{undefined} & \text{otherwise} \end{cases}$$

Below are given the semantic clauses of the language. Several clauses are split into separate definitions of $\mathcal{C}_1$ and $\mathcal{C}_2$ to aid clarity.

$$\begin{aligned}\mathcal{C}_1[\![\text{STOP}]\!]\rho\sigma &= \{(\langle\rangle, X) \mid X \subseteq (\text{dom}(\phi\,\rho) \cup \{\checkmark\})\} \\ \mathcal{C}_2[\![\text{STOP}]\!]\rho\sigma s &= \emptyset\end{aligned}$$

$stop_{\rho\sigma}$ will be used as an abbreviation for $\mathcal{C}[\![\texttt{STOP}]\!]\rho\sigma$. STOP never communicates or terminates. It merely refuses everything offered to it.

$$\mathcal{C}_1[\![\texttt{SKIP}]\!]\rho\sigma = \{(\langle\rangle, X) \mid X \subseteq \mathrm{dom}(\phi\rho)\}$$

$$\mathcal{C}_2[\![\texttt{SKIP}]\!]\rho\sigma s = \begin{cases} \{\sigma \downarrow \{\alpha \mid \varphi\rho\alpha = rw\}\} & \text{if } s = \langle\rangle \\ \emptyset & \text{otherwise} \end{cases}$$

$skip_{\rho\sigma}$ will be used as an abbreviation for $\mathcal{C}[\![\texttt{SKIP}]\!]\rho\sigma$. SKIP never communicates, but must terminate leaving the contents of all mutable addresses unchanged.

$$\begin{aligned} \mathcal{C}[\![e_1 := e_2]\!]\rho\sigma = \; & skip_{\rho\sigma'} \\ & \text{where } \mathcal{E}_V[\![e_2]\!]\rho\sigma = v \wedge \mathcal{E}_L[\![e_1]\!]\rho\sigma = \lambda \wedge \sigma' = update\ \sigma\ \lambda\ v \\ & \text{provided } \mathcal{E}_V[\![e_2]\!]\rho\sigma \neq \perp_{\mathcal{E}} \wedge \mathcal{E}_L[\![e_1]\!]\rho\sigma \neq \perp_{\mathcal{E}} \end{aligned}$$

This process also terminates without communicating, but modifies the termination state to take account of the assignment. In order to maintain strictness it is necessary to ensure that the right hand side is not erroneous and that the left hand side refers to a mutable variable.

$$\mathcal{C}_1[\![e_1, \ldots, e_n := f_1, \ldots, f_m]\!]\rho\sigma = \{(\langle\rangle, X) \mid X \subseteq \mathrm{dom}(\phi\rho)\}$$

$$\begin{aligned} \mathcal{C}_2[\![e_1, \ldots, e_n := f_1, \ldots, f_m]\!]\rho\sigma s = \; & \begin{cases} \{\sigma_n \downarrow \{\alpha \mid \varphi\rho\alpha = rw\}\} & \text{if } s = \langle\rangle \\ \emptyset & \text{otherwise} \end{cases} \\ & \text{where } \mathcal{E}_L[\![e_1]\!]\rho\sigma = \lambda_1 \wedge \cdots \wedge \mathcal{E}_L[\![e_n]\!]\rho\sigma = \lambda_n \wedge \\ & \quad \mathcal{E}_{V\&}[\![f_1]\!]\rho\sigma = vs_1 \wedge \cdots \wedge \mathcal{E}_{V\&}[\![f_m]\!]\rho\sigma = vs_m \wedge \\ & \quad vs = vs_1\, vs_2 \ldots vs_m \wedge \\ & \quad \sigma_1 = update\ \sigma\ \lambda_1\ vs[0] \wedge \cdots \wedge \\ & \quad \sigma_n = update\ \sigma_{n-1}\ \lambda_n\ vs[n-1] \\ & \text{provided } \mathcal{E}_L[\![e_1]\!]\rho\sigma \neq \perp_{\mathcal{E}} \wedge \cdots \wedge \mathcal{E}_L[\![e_n]\!]\rho\sigma \neq \perp_{\mathcal{E}} \wedge \\ & \quad \mathcal{E}_V[\![f_1]\!]\rho\sigma \neq \perp_{\mathcal{E}} \wedge \cdots \wedge \mathcal{E}_V[\![f_m]\!]\rho\sigma \neq \perp_{\mathcal{E}} \wedge \\ & \quad (m = 1 \vee m = n) \wedge disjoint(e_1, \ldots, e_n) \end{aligned}$$

Multiple assignments are possible in the language. The right hand sides are all calculated before any updating of the left hand sides occurs. The process terminates without communicating, modifying the final state to take account of each of the assignments. It is necessary to ensure that the left hand sides of a multiple assignment are disjoint, a task achieved by the auxiliary function *disjoint*; the definition of this function is straightforward but verbose and hence not included. The disjointness rules of occam (together with the severe anti-aliasing laws) mean that, once all the right hand sides have been calculated, the order in which the assignments are done is unimportant. An arbitrary decision has been taken to carry out the assignments from left to right.

$$\mathcal{C}[\![c!oe]\!]\rho\sigma = \begin{cases} \mathcal{J}^{CO}[\![c!oe]\!]\rho\sigma & \text{if } \mathcal{E}_C[\![c]\!]\rho\sigma \in CHAN \\ \mathcal{J}^{PO}[\![c!oe]\!]\rho\sigma & \text{if } \mathcal{E}_C[\![c]\!]\rho\sigma \in TYP \times PORT \\ \perp_{\mathcal{Q}} & \text{otherwise} \end{cases}$$

The way in which a process outputting an output expression (oe) behaves depends on the destination of the expression. Hence the above clause uses two auxiliary semantic functions, one to deal with output over a channel and the other to deal with output to a port.

$$\begin{aligned}
\mathcal{J}^{CO}{}_1[\![c!\alpha e]\!]\rho\sigma &= \{(\langle\rangle, X) \mid X \subseteq (\mathrm{dom}(\phi\,\rho) \cup \{\checkmark\} - \{\chi\})\} \cup \\
&\quad \{(\langle\chi,\beta\rangle, X) \mid X \subseteq \mathrm{dom}(\phi\,\rho)\} \\
\mathcal{J}^{CO}{}_2[\![e!\alpha e]\!]\rho\sigma s &= \begin{cases} \{\sigma \downarrow \{\alpha \mid \varphi\,\rho\alpha = \mathrm{rw}\}\} & \text{if } s = \langle\chi,\beta\rangle \\ \emptyset & \text{otherwise} \end{cases} \\
&\quad \text{where } \mathcal{E}_C[\![c]\!]\rho\sigma = \chi \wedge \mathcal{U}[\![\alpha e]\!]\rho\sigma\chi = \beta \\
&\quad \text{provided } \mathcal{U}[\![\alpha e]\!]\rho\sigma(\mathcal{E}_C[\![c]\!]\rho\sigma) \neq \perp_{\mathcal{E}} \wedge \\
&\qquad \exists\, p \in PROT \bullet \phi\,\rho(\mathcal{E}_C[\![c]\!]\rho\sigma) = (out, p)
\end{aligned}$$

Output over a channel proceeds by passing the communicable value along the channel then terminating in an unchanged state. In such a situation it is necessary to verify that the channel is suitable for output within the current scope and that the output expression is not erroneous. The semantic function $\mathcal{U}$ evaluates communicable values.

$$\mathcal{U}[\![\alpha e]\!]\rho\sigma\chi = \begin{cases} \mathcal{U}_T[\![\alpha e]\!]\rho\sigma & \text{if } \exists\, p \in CTYP1^{\&} \bullet \rho[\![\chi]\!] = (p, \underline{o}) \\ \mathcal{U}_S[\![\alpha e]\!]\rho\sigma & \text{if } \exists\, p \in \{\mathrm{ANY}\} \cup CTYP2 \bullet \rho[\![\chi]\!] = (p, \underline{o}) \\ \perp_{\mathcal{E}} & \text{otherwise} \end{cases}$$

Auxiliary semantic functions ($\mathcal{U}_T$ and $\mathcal{U}_S$) have been used to split the evaluation of communicable values into two parts; $\mathcal{U}_T$ deals with the case of channels with tagged protocols and $\mathcal{U}_S$ deals with the case of channels with simple or sequential protocols. Propagation of erroneous expressions in the desired manner is aided by use of the strict tupling constructor $(\ldots)_\perp$.

$$\begin{aligned}
\mathcal{U}_T[\![x]\!]\rho\sigma &= \rho[\![x]\!] \\
&\quad \text{if } \rho[\![x]\!] \in TAG \\
\mathcal{U}_T[\![x\,;\alpha e]\!]\rho\sigma &= (\mathcal{U}_T[\![x]\!]\rho\sigma, \mathcal{U}_S[\![\alpha e]\!]\rho\sigma)_\perp \\
\mathcal{U}_T[\![\alpha e]\!]\rho\sigma &= \perp_{\mathcal{E}} \\
&\quad \text{otherwise}
\end{aligned}$$

Once the tag has been removed from a communicable value over a channel with variant protocol, the remainder of the communicable value may be dealt with as if it were a communicable value over a channel with simple or sequential protocol.

$$\begin{aligned}
\mathcal{U}_S[\![e]\!]\rho\sigma &= \mathcal{E}_V[\![e]\!]\rho\sigma \\
\mathcal{U}_S[\![e_1 :: e_2]\!]\rho\sigma &= (\beta_1, \beta_2)_\perp \\
&\quad \text{where } \mathcal{E}_V[\![e_2]\!]\rho\sigma = v \wedge \mathcal{E}_V[\![e_1]\!]\rho\sigma = \beta_1 \wedge v[0 \ldots \beta_1] = \beta_2 \\
&\quad \text{if } 0 \leq \mathcal{E}_V[\![e_1]\!]\rho\sigma \leq \#(\mathcal{E}_V[\![e_2]\!]\rho\sigma) \\
\mathcal{U}_S[\![e_1\,;e_2\,;\ldots;e_n]\!]\rho\sigma &= (\beta_1, \beta_2, \ldots, \beta_n)_\perp \\
&\quad \text{where } \mathcal{U}_S[\![e_1]\!]\rho\sigma = \beta_1 \wedge \mathcal{U}_S[\![e_2]\!]\rho\sigma = \beta_2 \wedge \cdots \wedge \\
&\qquad \mathcal{U}_S[\![e_n]\!]\rho\sigma = \beta_n \\
\mathcal{U}_S[\![\alpha e]\!]\rho\sigma &= \perp_{\mathcal{E}} \\
&\quad \text{otherwise}
\end{aligned}$$

Communicable values over a channel with simple or sequential protocol can be variable length arrays. When this is the case there is no reason to pass on any part of the array outwith that specified by the length prefixing the array.

$$\mathcal{J}^{PO}{}_1\llbracket c!\alpha e\rrbracket\rho\sigma = \{(\langle\rangle, X) \mid X \subseteq \mathrm{dom}(\phi\,\rho)\}$$

$$\mathcal{J}^{PO}{}_2\llbracket c!\alpha e\rrbracket\rho\sigma s = \begin{cases} \{\sigma \downarrow \{\alpha \mid \varphi\,\rho\alpha = rw\}\} & \text{if } s = \langle\rangle \\ \emptyset & \text{otherwise} \end{cases}$$

provided $\mathcal{E}_V\llbracket \alpha e\rrbracket\rho\sigma \neq \perp_{\mathcal{E}}$

Output to a port is identified with SKIP within the current model; strictness, however, must be ensured.

$$\mathcal{C}\llbracket c?ie\rrbracket\rho\sigma = \begin{cases} \mathcal{J}^{TI}\llbracket c?ie\rrbracket\rho\sigma & \text{if } \mathcal{E}_C\llbracket c\rrbracket\rho\sigma \in TIMER \\ \mathcal{J}^{PI}\llbracket c?ie\rrbracket\rho\sigma & \text{if } \mathcal{E}_C\llbracket c\rrbracket\rho\sigma \in TYP \times PORT \\ \mathcal{J}^{CI}\llbracket c?ie\rrbracket\rho\sigma & \text{if } \mathcal{E}_C\llbracket c\rrbracket\rho\sigma \in CHAN \\ \perp_{\mathcal{Q}} & \text{otherwise} \end{cases}$$

The above clause relies on the fact that all occam types are finite, since without such an assumption it would not be possible to provide a *finite* set of termination states in the second component of the process description. As with output, the behaviour of a process which inputs an input expression (ie) depends on the source of the input; auxiliary semantic functions deal with each of the cases.

The first case to be dealt with is that of input from a timer. Within the current model, this is to be identified with the assignment of a random member of $\overline{\mathrm{INT}}$ to the variable.

$$\mathcal{J}^{TI}{}_1\llbracket c?ie\rrbracket\rho\sigma = \{(\langle\rangle, X) \mid X \subseteq \mathrm{dom}(\phi\,\rho)\}$$

$$\mathcal{J}^{TI}{}_2\llbracket c?ie\rrbracket\rho\sigma s = \begin{cases} \{(\mathit{update}\ \sigma\ (\mathcal{E}_L\llbracket ie\rrbracket\rho\sigma)\ z) \downarrow \{\alpha \mid \varphi\,\rho\alpha = rw\} \mid z \in \overline{\mathrm{INT}}\} & \text{if } s = \langle\rangle \\ \emptyset & \text{otherwise} \end{cases}$$

Next to be considered is input from a port. The decision was taken to model such actions by the assignment of a random value of appropriate type to the variable.

$$\mathcal{J}^{PI}{}_1\llbracket c?ie\rrbracket\rho\sigma = \{(\langle\rangle, X) \mid X \subseteq \mathrm{dom}(\phi\,\rho)\}$$

$$\mathcal{J}^{PI}{}_2\llbracket c?ie\rrbracket\rho\sigma s = \begin{cases} \{(\mathit{update}\ \sigma\ (\mathcal{E}_L\llbracket ie\rrbracket\rho\sigma)\ v) \downarrow \{\alpha \mid \varphi\,\rho\alpha = rw\} \mid v \in \bar{t}\} & \text{if } s = \langle\rangle \\ \emptyset & \text{otherwise} \end{cases}$$

where $\mathcal{E}_C\llbracket c\rrbracket\rho\sigma = (t, \varpi)$

Finally, input can be over a channel. Such an action proceeds by first accepting a value on the channel (it cannot refuse any communicable value) then terminating in the final state which results when the value inputted has been substituted into the current store. Errors result on an input over a channel if any variable mentioned cannot be written to, if the channel cannot be used for input, or if arrays are accessed out of bounds.

$$\mathcal{J}^{CI}{}_1\llbracket c?ie\rrbracket\rho\sigma = \{(\langle\rangle, X) \mid X \subseteq (\mathrm{dom}(\phi\,\rho) \cup \{\checkmark\} - \{\chi\})\} \cup \{(\langle\chi.\beta\rangle.\ X) \mid X \subseteq \mathrm{dom}(\phi\,\rho) \wedge \beta \in \overline{\chi}\} \cup \{(\langle\chi.\beta\rangle s, X) \mid X \subseteq (\mathrm{dom}(\phi\,\rho) \cup \{\checkmark\}) \wedge \beta \in \overline{\chi} \wedge \mathit{newstore}\llbracket ie\rrbracket\rho\sigma\beta = \perp_{\mathcal{S}}\}$$

$$\mathcal{J}^{CI}{}_2\llbracket c?ie\rrbracket\rho\sigma s = \begin{cases} \{(\mathit{newstore}\llbracket ie\rrbracket\rho\sigma\beta) \downarrow \{\alpha \mid \varphi\,\rho\alpha = rw\}\} & \text{if } s = \langle\chi.\beta\rangle \wedge \beta \in \overline{\chi} \wedge \mathit{newstore}\llbracket ie\rrbracket\rho\sigma\beta \neq \perp_{\mathcal{S}} \\ \perp & \text{if } s \geq \langle\chi.\beta\rangle \wedge \beta \in \overline{\chi} \wedge \mathit{newstore}\llbracket ie\rrbracket\rho\sigma\beta = \perp_{\mathcal{S}} \\ \emptyset & \text{otherwise} \end{cases}$$

where $\mathcal{E}_C\llbracket c\rrbracket\rho\sigma = \chi$

provided $\exists\, p \in PROT \bullet \phi\,\rho(\mathcal{E}_C\llbracket c\rrbracket\rho\sigma) = (in, p)$

Because a communicable value may not consist of an atomic item, it is not possible to alter the store directly with the function *update* already defined. This problem is overcome by the use of the auxiliary function *newstore*.

$$
\begin{aligned}
\mathit{newstore}\llbracket e \rrbracket \rho\sigma\beta &= \mathit{update}\ \sigma\ (\mathcal{E}_L\llbracket e \rrbracket \rho\sigma)\ \beta \\
\mathit{newstore}\llbracket e_1 :: e_2 \rrbracket \rho\sigma(\beta_1, \beta_2) &= \mathit{update}\ \sigma'\ (\mathcal{E}_L\llbracket e_2 \rrbracket \rho\sigma')\ \beta_2 \\
&\quad \text{where } \mathit{update}\ \sigma\ (\mathcal{E}_L\llbracket e_1 \rrbracket \rho\sigma)\ \beta_1 = \sigma' \\
\mathit{newstore}\llbracket e_1 ; \ldots ; e_n \rrbracket \rho\sigma(\beta_1, \ldots, \beta_n) &= \mathit{newstore}\llbracket e_2 ; \ldots ; e_n \rrbracket \rho\sigma'(\beta_2, \ldots, \beta_n) \\
&\quad \text{where } \sigma' = \mathit{newstore}\llbracket e_1 \rrbracket \rho\sigma\beta_1 \\
\mathit{newstore}\llbracket ie \rrbracket \rho\sigma\beta &= \bot_{\mathcal{S}} \\
&\quad \text{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}_1\llbracket c?\text{CASE } tag \rrbracket \rho\sigma &= \{(\langle\rangle, X) \mid X \subseteq (\mathrm{dom}(\phi\rho) \cup \{\checkmark\} - \{\chi\})\} \cup \\
&\quad \{(\langle \chi.\iota \rangle, X) \mid X \subseteq \mathrm{dom}(\phi\rho) \wedge \iota \in \overline{\chi} \wedge \rho\llbracket tag \rrbracket = \iota\} \cup \\
&\quad \{(\langle \chi.\iota\underline{\beta} \rangle s, X) \mid X \subseteq (\mathrm{dom}(\phi\rho) \cup \{\checkmark\}) \wedge \iota\underline{\beta} \in \overline{\chi} \wedge \rho\llbracket tag \rrbracket \neq \iota\} \\
\mathcal{C}_2\llbracket c?\text{CASE } tag \rrbracket \rho\sigma s &= \begin{cases} \{\sigma \downarrow \{\alpha \mid \varphi\rho\alpha = rw\}\} & \text{if } s = \langle \chi.\iota \rangle \wedge \iota \in \overline{\chi} \wedge \rho\llbracket tag \rrbracket = \iota \\ \bot & \text{if } s \geq \langle \chi.\iota\underline{\beta} \rangle \wedge \iota\underline{\beta} \in \overline{\chi} \wedge \rho\llbracket tag \rrbracket \neq \iota \\ \emptyset & \text{otherwise} \end{cases} \\
&\quad \text{where } \mathcal{E}_C\llbracket c \rrbracket \rho\sigma = \chi \\
&\quad \text{provided } \mathcal{E}_C\llbracket e \rrbracket \rho\sigma \in \mathit{CHAN} \wedge \rho\llbracket tag \rrbracket \in \mathit{TAG} \wedge \\
&\qquad \exists p \in \mathit{PROT} \bullet \phi\rho(\mathcal{E}_C\llbracket c \rrbracket \rho\sigma) = (in, p)
\end{aligned}
$$

In occam it is possible to construct a process which expects to receive input over a channel with variant protocol prefixed by a particular tag. Above is given the clause which deals with the case where the expected tag does not carry any data with it. If the tag received does not match the expected tag, the inputting process behaves like the bottom process from the communication onwards. In order to improve clarity, within the above clause $\chi.\iota\underline{\beta}$ has been used as an abbreviation for an arbitrary tuple of length greater than or equal to one whose first element is the tag ι.

$$
\begin{aligned}
\mathcal{C}_1\llbracket c?\text{CASE } tag ; ie \rrbracket \rho\sigma &= \{(\langle\rangle, X) \mid X \subseteq (\mathrm{dom}(\phi\rho) \cup \{\checkmark\} - \{\chi\})\} \cup \\
&\quad \{(\langle \chi.\iota\beta \rangle, X) \mid X \subseteq \mathrm{dom}(\phi\rho) \wedge \iota\beta \in \overline{\chi} \wedge \rho\llbracket tag \rrbracket = \iota\} \cup \\
&\quad \{(\langle \chi.\iota\beta \rangle s, X) \mid X \subseteq (\mathrm{dom}(\phi\rho) \cup \{\checkmark\}) \wedge \iota\beta \in \overline{\chi} \wedge \\
&\qquad \rho\llbracket tag \rrbracket = \iota \wedge \mathit{newstore}\llbracket ie \rrbracket \rho\sigma\beta = \bot_{\mathcal{S}}\} \cup \\
&\quad \{(\langle \chi.\iota\underline{\beta} \rangle s, X) \mid X \subseteq (\mathrm{dom}(\phi\rho) \cup \{\checkmark\}) \wedge \iota\underline{\beta} \in \overline{\chi} \wedge \rho\llbracket tag \rrbracket \neq \iota\} \\
\mathcal{C}_2\llbracket c?\text{CASE } tag ; ie \rrbracket \rho\sigma s &= \begin{cases} \{(\mathit{newstore}\llbracket ie \rrbracket \rho\sigma\beta) \downarrow \{\alpha \mid \varphi\rho\alpha = rw\}\} & \text{if } s = \langle \chi.\iota\beta \rangle \wedge \iota\beta \in \overline{\chi} \wedge \\ & \quad \rho\llbracket tag \rrbracket = \iota \wedge \mathit{newstore}\llbracket ie \rrbracket \rho\sigma\beta \neq \bot_{\mathcal{S}} \\ \bot & \text{if } s \geq \langle \chi.\iota\beta \rangle \wedge \iota\beta \in \overline{\chi} \wedge \\ & \quad \rho\llbracket tag \rrbracket = \iota \wedge \mathit{newstore}\llbracket ie \rrbracket \rho\sigma\beta = \bot_{\mathcal{S}} \\ \bot & \text{if } s \geq \langle \chi.\iota\underline{\beta} \rangle \wedge \iota\underline{\beta} \in \overline{\chi} \wedge \rho\llbracket tag \rrbracket \neq \iota \\ \emptyset & \text{otherwise} \end{cases} \\
&\quad \text{where } \mathcal{E}_C\llbracket c \rrbracket \rho\sigma = \chi \\
&\quad \text{provided } \mathcal{E}_C\llbracket c \rrbracket \rho\sigma \in \mathit{CHAN} \wedge \rho\llbracket tag \rrbracket \in \mathit{TAG} \wedge \\
&\qquad \exists p \in \mathit{PROT} \bullet \phi\rho(\mathcal{E}_C\llbracket c \rrbracket \rho\sigma) = (in, p)
\end{aligned}
$$

Above is given the clause which deals with the case where the expected tag carries data with it. If the tag received does not match the expected tag, the subsequent behaviour is again equivalent to the bottom process. $\chi.\iota\underline{\beta}$ has again been used as an abbreviation for an arbitrary tuple containing ι as its first element and within the clause a second abbreviation appears; $\chi.\iota\beta$ is used as an abbreviation for an arbitrary tuple of length greater than one whose first element is the tag ι.

$$\mathcal{C}\left[\!\left[\text{SEQ}\,(P_1, P_2, \ldots, P_n)\right]\!\right]\rho\sigma = \begin{cases} \mathit{skip}_{\rho\sigma} & \text{if } n = 0 \\ \underline{\mathit{seq}}\,(\mathcal{C}\left[\!\left[P_1\right]\!\right]\rho\sigma)\,(\mathcal{C}\left[\!\left[\text{SEQ}\,(P_2, \ldots, P_n)\right]\!\right]\rho)\,\sigma & \text{otherwise} \end{cases}$$

Here $\underline{\mathit{seq}}$ is the function which takes arguments of type $\mathcal{Q}$, $S \longrightarrow \mathcal{Q}$ and S and returns an element of $\mathcal{Q}$. Notice that the third argument (of type S) is necessary since it is used in order to provide the correct contents of 'read-only' variables with which to calculate the process description. The function $\underline{\mathit{seq}}$ is defined

$$\begin{aligned} f(\underline{\mathit{seq}}\ A\ B\ \sigma) \;=\; & \{(s, X) \mid (s, X \cup \{\checkmark\}) \in f(A)\} \cup \\ & \{(su, X) \mid \exists\, \sigma' \bullet \sigma' \in t(A)s \wedge (u, X) \in f(B(\sigma \oplus \sigma'))\} \cup \\ & \{(su, X) \mid t(A)s = \bot \wedge (s, X) \in f(A)s\} \end{aligned}$$

$$t(\underline{\mathit{seq}}\ A\ B\ \sigma)s \;=\; \begin{cases} \bigcup\{t(B(\sigma \oplus \sigma'))v \mid \exists\, u \bullet s = uv \wedge \sigma' \in t(A)u\} \\ \qquad \text{if } (s, \emptyset) \in f(\underline{\mathit{seq}}\ A\ B\ \sigma) \wedge t(A)s \neq \bot \wedge \\ \qquad\quad \nexists u, v, \sigma' \bullet s = uv \wedge \sigma' \in t(A)u \wedge t(B(\sigma \oplus \sigma'))v = \bot \\ \bot \qquad \text{if } t(A)s = \bot \vee \\ \qquad\quad \exists\, u, v, \sigma' \bullet s = uv \wedge \sigma' \in t(A)u \wedge t(B(\sigma \oplus \sigma'))v = \bot \\ \emptyset \qquad \text{otherwise} \end{cases}$$

If $n = 0$ then SEQ $(P_1, P_2, \ldots, P_n)$ behaves exactly like SKIP (terminating, without communicating, in an unaltered state). Otherwise process P_1 is run until it terminates successfully, the initial store of SEQ $(P_2, \ldots, P_n)$ being formed by updating the initial store of P_1 with the information contained in the final state of P_1. Note that P_1 cannot refuse a set X of channels unless it can refuse $X \cup \{\checkmark\}$; otherwise it would be able to terminate (invisibly) and pass control to SEQ $(P_2, \ldots, P_n)$.

$$\begin{aligned} \mathcal{C}\left[\!\left[\text{SEQ}\ x = e_1 \text{ FOR } e_2\ P\right]\!\right]\rho\sigma \;=\; & \underline{\mathit{rseq}}\,(\beta_1, \beta_2, x)\,(\mathcal{C}\left[\!\left[P\right]\!\right])\,\rho\sigma \\ & \text{where } \mathcal{E}_V\left[\!\left[e_1\right]\!\right]\rho\sigma = \beta_1 \wedge \mathcal{E}_V\left[\!\left[e_2\right]\!\right]\rho\sigma = \beta_2 \\ & \text{provided } \mathcal{E}_V\left[\!\left[e_1\right]\!\right]\rho\sigma \in \overline{\text{INT}} \wedge \mathcal{E}_V\left[\!\left[e_2\right]\!\right]\rho\sigma \in \overline{\text{INT}} \end{aligned}$$

Here $\underline{\mathit{rseq}}$ is the function which takes arguments of type $\overline{\text{INT}} \times \overline{\text{INT}} \times \mathit{IDE}$, $\mathit{ENV} \longrightarrow S \longrightarrow \mathcal{Q}$, ENV and S and returns an element of $\mathcal{Q}$, and is defined

$$\underline{\mathit{rseq}}\,(\beta_1, \beta_2, x)\ D\ \rho\sigma = \begin{cases} \mathit{skip}_{\rho\sigma} & \text{if } \beta_2 = 0 \\ \underline{\mathit{seq}}\,(D(\rho[\beta_1/x])\sigma)\,(\underline{\mathit{rseq}}\,(\beta_1 + 1, \beta_2 - 1, x)\ D\ \rho)\,\sigma & \\ \qquad \text{if } 0 \leq \beta_1 \leq \text{MOSTPOS INT} \wedge 0 < \beta_2 & \\ \bot_{\mathcal{Q}} & \text{otherwise} \end{cases}$$

This process carries out a replicated sequential command. The number of the iteration currently being executed is recorded with the help of an auxiliary variable, stored in the environment as a constant in order to avoid it being used in an unacceptable way during an iteration.

$$\mathcal{C}[\![\texttt{IF}\ (C_1,\ldots,C_n)]\!]\rho\sigma = \begin{cases} stop_{\rho\sigma} & \text{if } n = 0 \\ \mathcal{C}_{IF}[\![C_1]\!]\rho\sigma & \text{if } n > 0 \wedge \mathcal{I}[\![C_1]\!]\rho\sigma = \underline{true} \\ \mathcal{C}[\![\texttt{IF}\ (C_2,\ldots,C_n)]\!]\rho\sigma & \text{otherwise} \end{cases}$$
$$\text{provided } \mathcal{I}[\![\texttt{IF}\ (C_1,\ldots,C_n)]\!]\rho\sigma \neq \perp_{\mathcal{E}}$$

Care must be taken to ensure that if, on traversing the branches of the process in order, an erroneous guard is encountered before a guard which evaluates to *true* the process is mapped to the bottom element ($\perp_{\mathcal{Q}}$).

$$\begin{aligned} \mathcal{C}_{IF}[\![b\ P]\!]\rho\sigma &= \mathcal{C}[\![P]\!]\rho\sigma \\ \mathcal{C}_{IF}[\![\texttt{IF}\ (C_1,\ldots,C_n)]\!]\rho\sigma &= \mathcal{C}[\![\texttt{IF}\ (C_1,\ldots,C_n)]\!]\rho\sigma \\ \mathcal{C}_{IF}[\![\texttt{IF}\ x = e_1\ \texttt{FOR}\ e_2\ C]\!]\rho\sigma &= \mathcal{C}[\![\texttt{IF}\ x = e_1\ \texttt{FOR}\ e_2\ C]\!]\rho\sigma \\ \mathcal{C}_{IF}[\![\Delta : C]\!]\rho\sigma &= \mathcal{C}_{IF}[\![C]\!](\mathcal{D}[\![\Delta]\!]\rho)\sigma \\ &\quad \text{provided } \mathcal{D}[\![\Delta]\!]\rho \neq \perp \\ \mathcal{C}_{IF}[\![\Theta : C]\!]\rho\sigma &= \mathcal{C}_{IF}[\![C]\!](\mathcal{A}[\![\Theta]\!]\rho\sigma)\sigma \\ &\quad \text{provided } \mathcal{A}[\![\Theta]\!]\rho\sigma \neq \perp \end{aligned}$$

Giving a meaning to a branch of a conditional closely resembles giving a meaning to a process; the only difference is the boolean guard at the innermost level which must be dropped.

The auxiliary function $\mathcal{I}$ is necessary in order to allow for the possibility of nested 'IF's. The function returns a value indicating whether a given branch should be followed, taking account of the status of any nested guards.

When nested replicated conditionals appear, a much clearer description results from the decision to explicitly substitute the index value into the body of the conditional. Distinct environments for each index value could, if desired, be used to avoid this substitution, but their inclusion was not thought necessary.

$$\mathcal{I}[\![b\ P]\!]\rho\sigma = \mathcal{E}_V[\![b]\!]\rho\sigma$$

$$\mathcal{I}[\![\texttt{IF}\ (C_1,\ldots,C_n)]\!]\rho\sigma = \begin{cases} \underline{false} & \text{if } n = 0 \\ \underline{true} & \text{if } n > 0 \wedge \mathcal{I}[\![C_1]\!]\rho\sigma = \underline{true} \\ \mathcal{I}[\![\texttt{IF}\ (C_2,\ldots,C_n)]\!]\rho\sigma & \text{if } n > 0 \wedge \mathcal{I}[\![C_1]\!]\rho\sigma = \underline{false} \\ \perp_{\mathcal{E}} & \text{otherwise} \end{cases}$$

$$\begin{aligned} \mathcal{I}[\![\texttt{IF}\ x = e_1\ \texttt{FOR}\ e_2\ C]\!]\rho\sigma &= \mathcal{I}[\![\texttt{IF}\ (C[\beta_1/x],\ldots,C[\beta_1+\beta_2-1/x])]\!]\rho\sigma \\ &\quad \text{where } \mathcal{E}[\![e_1]\!]\rho\sigma = \beta_1 \wedge \mathcal{E}[\![e_2]\!]\rho\sigma = \beta_2 \wedge \\ &\quad \text{if } \mathcal{E}[\![e_1]\!]\rho\sigma \in \overline{\texttt{INT}} \wedge \mathcal{E}[\![e_2]\!]\rho\sigma \in \overline{\texttt{INT}} \wedge \\ &\qquad \mathcal{E}[\![e_2]\!]\rho\sigma \geq 0 \wedge \\ &\qquad \mathcal{E}[\![e_1]\!]\rho\sigma + \mathcal{E}[\![e_2]\!]\rho\sigma - 1 \leq \texttt{MOSTPOS INT} \\ \mathcal{I}[\![\Delta : C]\!]\rho\sigma &= \mathcal{I}[\![C]\!](\mathcal{D}[\![\Delta]\!]\rho)\sigma \\ &\quad \text{if } \mathcal{D}[\![\Delta]\!]\rho \neq \perp \\ \mathcal{I}[\![\Theta : C]\!]\rho\sigma &= \mathcal{I}[\![C]\!](\mathcal{A}[\![\Theta]\!]\rho\sigma)\sigma \\ &\quad \text{if } \mathcal{A}[\![\Theta]\!]\rho\sigma \neq \perp \\ \mathcal{I}[\![C]\!]\rho\sigma &= \perp_{\mathcal{E}} \\ &\quad \text{otherwise} \end{aligned}$$

A conditional with zero branches behaves like STOP. If one or more branches are present, the branches are traversed in order, the body of the first branch whose guard evaluates to *true* being executed.

$$\begin{aligned} \mathcal{C}\llbracket \texttt{IF } x = e_1 \texttt{ FOR } e_2\ C \rrbracket \rho\sigma \;=\; & \underline{rcond}\ (\beta_1, \beta_2, x)\ \llbracket C \rrbracket \rho\sigma \\ & \text{where } \mathcal{E}_V \llbracket e_1 \rrbracket \rho\sigma = \beta_1 \wedge \mathcal{E}_V \llbracket e_2 \rrbracket \rho\sigma = \beta_2 \\ & \text{provided } \mathcal{E}_V \llbracket e_1 \rrbracket \rho\sigma \in \overline{\texttt{INT}} \wedge \mathcal{E}_V \llbracket e_2 \rrbracket \rho\sigma \in \overline{\texttt{INT}} \end{aligned}$$

Here *rcond* is the function defined

$$\underline{rcond}\ (\beta_1, \beta_2, x)\ \llbracket C \rrbracket \rho\sigma = \begin{cases} stop_{\rho\sigma} & \text{if } \beta_2 = 0 \\ C_{IF} \llbracket C \rrbracket (\rho[\beta_1/x])\sigma & \text{if } \mathcal{I} \llbracket C \rrbracket (\rho[\beta_1/x])\sigma = \underline{true} \wedge \\ & \quad 0 \le \beta_1 \le \texttt{MOSTPOS INT} \wedge 0 < \beta_2 \\ \underline{rcond}\ (\beta_1 + 1, \beta_2 - 1, x)\ \llbracket C \rrbracket \rho\sigma & \text{if } \mathcal{I} \llbracket C \rrbracket (\rho[\beta_1/x])\sigma = \underline{false} \wedge \\ & \quad 0 \le \beta_1 \le \texttt{MOSTPOS INT} \wedge 0 < \beta_2 \\ \perp_{\mathcal{Q}} & \text{otherwise} \end{cases}$$

This process carries out a replicated conditional command. Again an auxiliary constant stored in the environment records the number of the current iteration. A replicated conditional re-evaluates the guard(s) on each iteration; the body is carried out with the value of the index equal to the smallest integer which makes the guard evaluate to *true*.

$$\begin{aligned} \mathcal{C}\llbracket c\texttt{?AFTER } e \rrbracket \rho\sigma \;=\; & skip_{\rho\sigma} \\ & \text{provided } \mathcal{E}_V \llbracket e \rrbracket \rho\sigma \in \overline{\texttt{INT}} \wedge \mathcal{E}_C \llbracket c \rrbracket \rho\sigma \in TIMER \end{aligned}$$

Input delay is equated with SKIP in the model. It is, however, necessary to check that the expression is suitable and that the timer is not erroneous by dint of the value of subscripts. The acceptability of equating input delay with SKIP is due to the fact that the presence of AFTER within a process does not determine the behaviour of the process (e.g., influence which branch of execution is followed); it merely alters the timing characteristics (which are not dealt with by the model).

$$\mathcal{C}\llbracket \texttt{WHILE } b\ P \rrbracket \rho\sigma = (\bigsqcup_{n=0}^{\infty} F^n(\perp_{(S \longrightarrow \mathcal{Q}_{(C,L)})}))\sigma$$

where $F : (S \longrightarrow \mathcal{Q}_{(C,L)}) \longrightarrow (S \longrightarrow \mathcal{Q}_{(C,L)})$ is the function defined

$$F(B)\sigma' = \begin{cases} \underline{seq}\ (\mathcal{C}\llbracket P \rrbracket \rho\sigma')\ B\ \sigma' & \text{if } \mathcal{E}_V \llbracket b \rrbracket \rho\sigma' = \underline{true} \\ skip_{\rho\sigma'} & \text{if } \mathcal{E}_V \llbracket b \rrbracket \rho\sigma' = \underline{false} \\ \perp_{\mathcal{Q}_{(C,L)}} & \text{otherwise} \end{cases}$$

This is the only form of recursion allowed in occam, and the functionality of the above definition depends only on the continuity of *seq*. In fact *all* the operators used are continuous.

$$\mathcal{C}\llbracket \texttt{CASE } e\ (C_1, \ldots, C_n) \rrbracket \rho\sigma \;=\; \begin{cases} stop_{\rho\sigma} & \text{if } n = 0 \\ \underline{case}\ (\mathcal{E}_V \llbracket e \rrbracket \rho\sigma)\ \emptyset\ (\mathcal{C}\llbracket \texttt{STOP} \rrbracket \rho)\ \rho\ \llbracket C_1, \ldots, C_n \rrbracket \rho\sigma & \text{otherwise} \end{cases}$$

Here *case* is the function defined

$$\underline{case}\ \beta\ A\ B\ \varrho\ [\![C_1, C_2, \ldots, C_n]\!]\rho\sigma = \begin{cases} B\sigma & \text{if } n = 0 \wedge \beta \neq \perp_{\mathcal{E}} \\ \underline{case}\ \beta\ (A \cup \{\text{ELSE}\})\ (\mathcal{C}[\![P']\!]\rho)\ \varrho\ [\![C_2, \ldots, C_n]\!]\rho\sigma & \\ \quad \text{if } n > 0 \wedge C_1 = \text{ELSE } P' \wedge & \\ \quad\quad \text{ELSE} \notin A \wedge \beta \notin A & \\ \underline{case}\ \beta\ (A \cup \{\text{ELSE}\})\ B\ \varrho\ [\![C_2, \ldots, C_n]\!]\rho\sigma & \\ \quad \text{if } n > 0 \wedge C_1 = \text{ELSE } P' \wedge & \\ \quad\quad \text{ELSE} \notin A \wedge \beta \in A & \\ \underline{case}\ \beta\ (A \cup \{\mathcal{E}_V[\![e_1]\!]\rho\sigma, \ldots, \mathcal{E}_V[\![e_m]\!]\rho\sigma\}) & \\ \quad\quad (\mathcal{C}[\![P']\!]\rho)\ \varrho\ [\![C_2, \ldots, C_n]\!]\rho\sigma & \\ \quad \text{if } n > 0 \wedge C_1 = (e_1, \ldots, e_m)\ P' \wedge & \\ \quad\quad \mathcal{E}_V[\![e_1]\!]\rho\sigma \notin A \wedge \cdots \wedge \mathcal{E}_V[\![e_m]\!]\rho\sigma \notin A \wedge & \\ \quad\quad \mathit{fit}\ \beta\ [\![C_1]\!]\rho\sigma = \underline{true} & \\ \underline{case}\ \beta\ (A \cup \{\mathcal{E}_V[\![e_1]\!]\rho\sigma, \ldots, \mathcal{E}_V[\![e_m]\!]\rho\sigma\}) & \\ \quad\quad B\ \varrho\ [\![C_2, \ldots, C_n]\!]\rho\sigma & \\ \quad \text{if } n > 0 \wedge C_1 = (e_1, \ldots, e_m)\ P' \wedge & \\ \quad\quad \mathcal{E}_V[\![e_1]\!]\rho\sigma \notin A \wedge \cdots \wedge \mathcal{E}_V[\![e_m]\!]\rho\sigma \notin A \wedge & \\ \quad\quad \mathit{fit}\ \beta\ [\![C_1]\!]\rho\sigma = \underline{false} & \\ \underline{case}\ \beta\ A\ B\ \varrho\ [\![C, C_2, \ldots, C_n]\!](\mathcal{D}[\![\Delta]\!]\rho)\sigma & \\ \quad \text{if } n > 0 \wedge C_1 = \Delta : C \wedge \mathcal{D}[\![\Delta]\!]\rho \neq \perp & \\ \underline{case}\ \beta\ A\ B\ \varrho\ [\![C, C_2, \ldots, C_n]\!](\mathcal{A}[\![\Theta]\!]\rho\sigma)\sigma & \\ \quad \text{if } n > 0 \wedge C_1 = \Theta : C \wedge \mathcal{A}[\![\Theta]\!]\rho\sigma \neq \perp & \\ \perp_{\mathcal{Q}} \quad \text{otherwise} & \end{cases}$$

It is necessary to supply the initial environment (ϱ) as an argument to *case* since declarations or specifications prefixing one branch of the CASE command must not be allowed to influence later branches. When it has been decided that a particular branch should not be followed, not only is the process contained within the branch dropped, but the initial environment is also reinstated. Care must be taken to ensure that *case* is strict – even if the expression of a CASE command is erroneous, a branch with an erroneous guard must not be followed (although both are mapped to $\perp_{\mathcal{E}}$ by $\mathcal{E}_V$). While traversing the branches of the CASE command, a note of the values which have already appeared as guards must be kept since it is not permitted for the same value to prefix more than one branch. A branch prefixed by an ELSE guard requires special consideration, since its body may not influence the behaviour of the process.

Within the definition of *case* an auxiliary function *fit* appears. The purpose of this function is to determine whether a given branch of the CASE construct should be followed.

$$\mathit{fit}\ \beta\ [\![C]\!]\rho\sigma \quad = \quad \begin{cases} \underline{true} & \text{if } C = e\ P \wedge \mathcal{E}_V[\![e]\!]\rho\sigma \neq \perp_{\mathcal{E}} \wedge \mathcal{E}_V[\![e]\!]\rho\sigma = \beta \\ \underline{false} & \text{if } C = e\ P \wedge \mathcal{E}_V[\![e]\!]\rho\sigma \neq \perp_{\mathcal{E}} \wedge \mathcal{E}_V[\![e]\!]\rho\sigma \neq \beta \\ \bigvee_{i=1}^{n} \mathit{fit}\ \beta\ [\![e_i\ P]\!]\rho\sigma & \text{if } C = (e_1, \ldots, e_n)\ P \\ \perp_{\mathcal{E}} & \text{otherwise} \end{cases}$$

(where $\bigvee$ is strict)

This process carries out a CASE command. The (necessarily unique) choice whose guard matches the given expression is followed, subject to the restriction that a choice guarded by ELSE is followed only if each of the other guards evaluate to *false*. The guard ELSE matches any proper expression; in any given CASE command of an occam process, only one guard may be equal to

ELSE. A CASE command with no branches (or one where the given expression does not match any of the guards) is equivalent to STOP.

$$\mathcal{C}[\![\texttt{PRI ALT}\ (A_1, A_2, \ldots, A_n)]\!]\rho\sigma = \mathcal{C}[\![\texttt{ALT}\ (A_1, A_2, \ldots, A_n)]\!]\rho\sigma$$

As previously described, priority is not considered, and so a rather less deterministic meaning than expected may result.

$$\mathcal{C}[\![\texttt{PRI ALT}\ x = e_1\ \texttt{FOR}\ e_2\ A]\!]\rho\sigma = \mathcal{C}[\![\texttt{ALT}\ x = e_1\ \texttt{FOR}\ e_2\ A]\!]\rho\sigma$$

Priority is also ignored in replicated alternations.

It is necessary, however, to model alternations accurately. In occam, apart from CASE constructs or nested alternations, there are two types of guard (input guards and SKIP guards) which can appear within an alternation. The existence of SKIP guards provides a slight complication since they work differently from input guards. The neatest solution is to extend the domain of $\mathcal{C}$ to include all guarded processes and to invent an auxiliary semantic function, $\mathcal{R}$, which will serve a similar purpose to the one used for conditionals and tell whether any SKIP guard is ready. In what follows, G will represent a guard not containing a boolean (either SKIP or $c?e$ or $c?\texttt{AFTER}\ e$).

$$\mathcal{C}[\![G\ P]\!]\rho\sigma = \begin{cases} \mathcal{C}[\![P]\!]\rho\sigma & \text{if } G = \texttt{SKIP} \\ \mathcal{C}[\![\texttt{SEQ}\ (G, P)]\!]\rho\sigma & \text{if } G = c?e \\ stop_{\rho\sigma} & \text{if } G = c?\texttt{AFTER}\ e \wedge \\ & \quad \mathcal{E}_C[\![c]\!]\rho\sigma \in TIMER \wedge \mathcal{E}_V[\![e]\!]\rho\sigma \in \overline{\texttt{INT}} \end{cases}$$

$$\mathcal{C}[\![b\&G\ P]\!]\rho\sigma = \begin{cases} stop_{\rho\sigma} & \text{if } \mathcal{E}_V[\![b]\!]\rho\sigma = \underline{false} \\ \mathcal{C}[\![P]\!]\rho\sigma & \text{if } \mathcal{E}_V[\![b]\!]\rho\sigma = \underline{true} \wedge G = \texttt{SKIP} \\ \mathcal{C}[\![\texttt{SEQ}\ (G, P)]\!]\rho\sigma & \text{if } \mathcal{E}_V[\![b]\!]\rho\sigma = \underline{true} \wedge G = c?e \\ stop_{\rho\sigma} & \text{if } \mathcal{E}_V[\![b]\!]\rho\sigma = \underline{true} \wedge G = c?\texttt{AFTER}\ e \wedge \\ & \quad \mathcal{E}_C[\![c]\!]\rho\sigma \in TIMER \wedge \mathcal{E}_V[\![e]\!]\rho\sigma \in \overline{\texttt{INT}} \\ \perp_{\mathcal{Q}} & \text{otherwise} \end{cases}$$

$$\mathcal{C}[\![b\&\texttt{CASE}\ (T_1, \ldots, T_n)]\!]\rho\sigma = \begin{cases} stop_{\rho\sigma} & \text{if } \mathcal{E}_V[\![b]\!]\rho\sigma = \underline{false} \\ \mathcal{C}[\![\texttt{CASE}\ (T_1, \ldots, T_n)]\!]\rho\sigma & \text{if } \mathcal{E}_V[\![b]\!]\rho\sigma = \underline{true} \\ \perp_{\mathcal{Q}} & \text{otherwise} \end{cases}$$

Above is the extension which must be added to $\mathcal{C}$ to simplify alternations. Although not in itself complex, it provides a great simplification in what follows.

Notice that the behaviour of a branch guarded by a delayed input (with or without an accompanying boolean) is equivalent to STOP, providing the guard is not erroneous. This ties in with the decision that such branches should not under any circumstances be followed.

$$\begin{aligned} \mathcal{C}_1[\![\texttt{ALT}\ (A_1, \ldots, A_n)]\!]\rho\sigma = {} & \{(\langle\rangle, X) \mid \exists\, i \bullet \mathcal{R}[\![A_i]\!]\rho\sigma = \underline{true} \wedge (\langle\rangle, X) \in \mathcal{C}_1[\![A_i]\!]\rho\sigma\} \cup \\ & \{(\langle\rangle, X) \mid \forall\, i \bullet \mathcal{R}[\![A_i]\!]\rho\sigma = \underline{false} \wedge (\langle\rangle, X) \in \mathcal{C}_1[\![A_i]\!]\rho\sigma\} \cup \\ & \{(\langle\rangle, X) \mid \exists\, i \bullet \mathcal{C}_2[\![A_i]\!]\rho\sigma\langle\rangle = \perp\} \cup \\ & \{(s, X) \mid s \neq \langle\rangle \wedge \exists\, i \bullet (s, X) \in \mathcal{C}_1[\![A_i]\!]\rho\sigma\} \end{aligned}$$

$$\begin{aligned} \mathcal{C}_2[\![\texttt{ALT}\ (A_1, \ldots, A_n)]\!]\rho\sigma s = {} & \bigcup\{\mathcal{C}_2[\![A_i]\!]\rho\sigma s \mid i \in \{1, \ldots, n\}\} \\ & \text{provided } \mathcal{R}[\![\texttt{ALT}\ (A_1, \ldots, A_n)]\!]\rho\sigma \neq \perp_{\mathcal{E}} \end{aligned}$$

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 input guards. Note that if every alternative contains a boolean expression evaluating to *false*, then ALT $(A_1, \ldots, A_n)$ is equivalent to STOP.

$$\mathcal{R}[\![b\&G\ P]\!]\rho\sigma = \begin{cases} \underline{false} & \text{if } G = c?e \wedge \mathcal{E}_C[\![c]\!]\rho\sigma \in CHAN \\ \mathcal{E}_V[\![b]\!]\rho\sigma & \text{if } G = c?e \wedge \mathcal{E}_C[\![c]\!]\rho\sigma \in TIMER \\ \underline{false} & \text{if } G = c?\texttt{AFTER}\ e \wedge \mathcal{E}_C[\![c]\!]\rho\sigma \in TIMER \\ \mathcal{E}_V[\![b]\!]\rho\sigma & \text{if } G = c?e \wedge \mathcal{E}_C[\![c]\!]\rho\sigma \in TYP \times PORT \\ \mathcal{E}_V[\![b]\!]\rho\sigma & \text{if } G = \texttt{SKIP} \\ \perp_{\mathcal{E}} & \text{otherwise} \end{cases}$$
$$\text{if } \mathcal{E}_V[\![b]\!]\rho\sigma \in \overline{\texttt{BOOL}}$$

$$\mathcal{R}[\![c?e\ P]\!]\rho\sigma = \begin{cases} \underline{false} & \text{if } \mathcal{E}_C[\![c]\!]\rho\sigma \in CHAN \\ \underline{true} & \text{if } \mathcal{E}_C[\![c]\!]\rho\sigma \in TIMER \\ \underline{true} & \text{if } \mathcal{E}_C[\![c]\!]\rho\sigma \in TYP \times PORT \\ \perp_{\mathcal{E}} & \text{otherwise} \end{cases}$$

$$\mathcal{R}[\![c?\texttt{AFTER}\ e]\!]\rho\sigma = \underline{false} \quad \text{if } \mathcal{E}_C[\![c]\!]\rho\sigma \in TIMER$$

$$\mathcal{R}[\![\texttt{ALT}\ (A_1, \ldots, A_n)]\!]\rho\sigma = \bigvee_{i=1}^{n} \mathcal{R}[\![A_i]\!]\rho\sigma$$

$$\mathcal{R}[\![\texttt{PRI ALT}\ (A_1, \ldots, A_n)]\!]\rho\sigma = \bigvee_{i=1}^{n} \mathcal{R}[\![A_i]\!]\rho\sigma$$

$$\mathcal{R}[\![\texttt{ALT}\ x = e_1\ \texttt{FOR}\ e_2\ A]\!]\rho\sigma = \mathcal{R}[\![\texttt{ALT}\ (A[\beta_1/x], \ldots, A[\beta_1 + \beta_2 - 1/x])]\!]\rho\sigma$$
$$\text{where } \mathcal{E}_V[\![e_1]\!]\rho\sigma = \beta_1 \wedge \mathcal{E}_V[\![e_2]\!]\rho\sigma = \beta_2$$
$$\text{if } \mathcal{E}_V[\![e_1]\!]\rho\sigma \in \overline{\texttt{INT}} \wedge \mathcal{E}_V[\![e_2]\!]\rho\sigma \in \overline{\texttt{INT}} \wedge$$
$$\mathcal{E}_V[\![e_2]\!]\rho\sigma \geq 0 \wedge$$
$$\mathcal{E}_V[\![c_1]\!]\rho\sigma + \mathcal{E}_V[\![e_2]\!]\rho\sigma - 1 \leq \texttt{MOSTPOS INT}$$

$$\mathcal{R}[\![\texttt{PRI ALT}\ x = e_1\ \texttt{FOR}\ e_2\ A]\!]\rho\sigma = \mathcal{R}[\![\texttt{ALT}\ (A[\beta_1/x], \ldots, A[\beta_1 + \beta_2 - 1/x])]\!]\rho\sigma$$
$$\text{where } \mathcal{E}_V[\![e_1]\!]\rho\sigma = \beta_1 \wedge \mathcal{E}_V[\![e_2]\!]\rho\sigma = \beta_2$$
$$\text{if } \mathcal{E}_V[\![e_1]\!]\rho\sigma \in \overline{\texttt{INT}} \wedge \mathcal{E}_V[\![e_2]\!]\rho\sigma \in \overline{\texttt{INT}} \wedge$$
$$\mathcal{E}_V[\![e_2]\!]\rho\sigma \geq 0 \wedge$$
$$\mathcal{E}_V[\![e_1]\!]\rho\sigma + \mathcal{E}_V[\![e_2]\!]\rho\sigma - 1 \leq \texttt{MOSTPOS INT}$$

$$\mathcal{R}[\![c?\texttt{CASE}\ (T_1, \ldots, T_n)]\!]\rho\sigma = \underline{false} \quad \text{if } \mathcal{E}_C[\![c]\!]\rho\sigma \in CHAN$$

$$\mathcal{R}[\![b\&c?\texttt{CASE}\ (T_1, \ldots, T_n)]\!]\rho\sigma = \underline{false} \quad \text{if } \mathcal{E}_C[\![c]\!]\rho\sigma \in CHAN \wedge \mathcal{E}_V[\![b]\!]\rho\sigma \in \overline{\texttt{BOOL}}$$

$$\mathcal{R}[\![\Delta : A]\!]\rho\sigma = \mathcal{R}[\![A]\!](\mathcal{D}[\![\Delta]\!]\rho)\sigma \quad \text{if } \mathcal{D}[\![\Delta]\!]\rho \neq \perp$$

$$\mathcal{R}[\![\Theta : A]\!]\rho\sigma = \mathcal{R}[\![A]\!](\mathcal{A}[\![\Theta]\!]\rho\sigma)\sigma \quad \text{if } \mathcal{A}[\![\Theta]\!]\rho\sigma \neq \perp$$

$$\mathcal{R}[\![A]\!]\rho\sigma = \perp_{\mathcal{E}} \quad \text{otherwise}$$

(where $\bigvee$ is strict)

Any guard dependent on a future input cannot possibly be ready and so cannot take value *true*. However, when considering such guards prefixed by a boolean, it is necessary to maintain

strictness. This is achieved by explicitly checking for erroneous boolean expressions. Branches guarded by delayed input must not be followed; this is despite the fact that input from a timer is not visible and is necessary in order to prevent the premature exit from an alternative and the resultant curtailed behaviour. As a result of the above, the availability of recovery from a deadlocked process is removed, but its reinstatement in order to meet timing constraints remains a valid refinement.

Nested replicated alternations can occur. The readiness of a particular branch in such a situation can be calculated independently of the other branches, since the property of being ready cannot be influenced by the other branches. The substitution of the index value within the body of a nested replicated alternation is carried out directly in order to aid clarity; it would be possible to adapt the environment separately for each branch, but such action was not thought warranted in this instance.

$$\begin{aligned}\mathcal{C}\llbracket \texttt{ALT}\ x = e_1\ \texttt{FOR}\ e_2\ A\rrbracket\rho\sigma \;=\;& \underline{ralt}\ (\beta_1, \beta_2, x)\ \llbracket A\rrbracket\rho\sigma \\ & \text{where } \mathcal{E}_V\llbracket e_1\rrbracket\rho\sigma = \beta_1 \wedge \mathcal{E}_V\llbracket e_2\rrbracket\rho\sigma = \beta_2 \\ & \text{provided } \mathcal{E}_V\llbracket e_1\rrbracket\rho\sigma \in \overline{\texttt{INT}} \wedge \mathcal{E}_V\llbracket e_2\rrbracket\rho\sigma \in \overline{\texttt{INT}} \wedge \\ & \qquad \mathcal{E}_V\llbracket e_2\rrbracket\rho\sigma \geq 0 \wedge \\ & \qquad \mathcal{E}_V\llbracket e_1\rrbracket\rho\sigma + \mathcal{E}_V\llbracket e_2\rrbracket\rho\sigma - 1 \leq \texttt{MOSTPOS INT} \wedge \\ & \qquad \mathcal{R}\llbracket \texttt{ALT}\ (A[\beta_1/x], \ldots, A[\beta_1 + \beta_2 - 1/x])\rrbracket\rho\sigma \neq \bot_{\mathcal{E}}\end{aligned}$$

Here *ralt* is the function defined

$$\begin{aligned}f(\underline{ralt}\ (\beta_1, \beta_2, x)\ \llbracket A\rrbracket\rho\sigma) \;=\;& \{(\langle\rangle, X) \mid \exists\, i \in \{\beta_1, \ldots, \beta_1 + \beta_2 - 1\} \bullet \\ & \qquad \mathcal{R}\llbracket A\rrbracket(\rho[i/x])\sigma = \underline{true} \wedge (\langle\rangle, X) \in \mathcal{C}_1\llbracket A\rrbracket(\rho[i/x])\sigma\} \cup \\ & \{(\langle\rangle, X) \mid \forall\, i \in \{\beta_1, \ldots, \beta_1 + \beta_2 - 1\} \bullet \\ & \qquad \mathcal{R}\llbracket A\rrbracket(\rho[i/x])\sigma = \underline{false} \wedge (\langle\rangle, X) \in \mathcal{C}_1\llbracket A\rrbracket(\rho[i/x])\sigma\} \cup \\ & \{(\langle\rangle, X) \mid \exists\, i \in \{\beta_1, \ldots, \beta_1 + \beta_2 - 1\} \bullet \\ & \qquad \mathcal{C}_2\llbracket A\rrbracket(\rho[i/x])\sigma\langle\rangle = \bot\} \cup \\ & \{(s, X) \mid s \neq \langle\rangle \wedge \exists\, i \in \{\beta_1, \ldots, \beta_1 + \beta_2 - 1\} \bullet \\ & \qquad (s, X) \in \mathcal{C}_1\llbracket A\rrbracket(\rho[i/x])\sigma)\}\end{aligned}$$

$$t(\underline{ralt}\ (\beta_1, \beta_2, x)\ \llbracket A\rrbracket\rho\sigma)s \;=\; \bigcup\{\mathcal{C}_2\llbracket A\rrbracket(\rho[i/x])\sigma s \mid i \in \{\beta_1, \ldots, \beta_1 + \beta_2 - 1\}\}$$

The definition of a replicated alternative differs substantially from the other clauses concerned with replicated processes since it is necessary to consider all branches at once. (Unlike a conditional or sequence, the behaviour of a constituent part is not independent of those following it.) An auxiliary constant stored in the environment records the value of the index at any point, and care must be taken to ensure that the environment is correct throughout. Note that a replicated alternation in which any of the branches are indexed by a value greater than MOSTPOS INT is identified with the bottom process ($\bot_{\mathcal{Q}}$).

$$\begin{aligned}\mathcal{C}_1\llbracket c?\texttt{CASE}\ (T_1, \ldots, T_n)\rrbracket\rho\sigma \;=\;& \{(\langle\rangle, X) \mid X \subseteq (\mathrm{dom}(\phi\rho) \cup \{\checkmark\} - \{\chi\})\} \cup \\ & \{(\langle\chi.\iota\underline{\beta}\rangle s, X) \mid \iota\underline{\beta} \in \overline{\chi} \wedge \\ & \qquad (s, X) \in f(\underline{action}\ \iota\underline{\beta}\ \rho\ \llbracket T_1, \ldots, T_n\rrbracket\rho\sigma)\}\end{aligned}$$

$$\mathcal{C}_2\llbracket c?\texttt{CASE}\ (T_1, \ldots, T_n)\rrbracket\rho\sigma s \;=\; \begin{cases} t(\underline{action}\ \iota\underline{\beta}\ \rho\ \llbracket T_1, \ldots, T_n\rrbracket\rho\sigma)s' & \text{if } s = \langle\chi.\iota\underline{\beta}\rangle s' \wedge \iota\underline{\beta} \in \overline{\chi} \\ \emptyset & \text{otherwise}\end{cases}$$

$$\begin{aligned}& \text{where } \mathcal{E}_C\llbracket c\rrbracket\rho\sigma = \chi \\ & \text{provided } \mathcal{E}_C\llbracket c\rrbracket\rho\sigma \in CHAN \wedge \\ & \qquad \exists\, p \in PROT \bullet \phi\rho(\mathcal{E}_C\llbracket c\rrbracket\rho\sigma) = (in, p)\end{aligned}$$

In the above clause an auxiliary function *action* has been used. This function calculates the behaviour of the process after the tagged input. The function *action* requires the initial environment as an argument since declarations and specifications must not be allowed to have any influence outwith their scope. As with the clauses concerning input over channels with variant protocol, $\iota\underline{\beta}$ has been used as an abbreviation for an arbitrary value communicable over the channel.

$$\underline{action}\ \iota\underline{\beta}\ \varrho\ [\![T_1, T_2, \ldots, T_n]\!]\rho\sigma = \begin{cases} \bot_{\mathcal{Q}} & \text{if } n = 0 \\ \underline{action}\ \iota\underline{\beta}\ \varrho\ [\![T_2, \ldots, T_n]\!]\varrho\sigma & \\ \qquad \text{if } n > 0 \wedge T_1 = tag\ P \wedge \rho[\![tag]\!] \neq \iota & \\ \underline{action}\ \iota\underline{\beta}\ \varrho\ [\![T_2, \ldots, T_n]\!]\varrho\sigma & \\ \qquad \text{if } n > 0 \wedge T_1 = tag\ ; e\ P \wedge \rho[\![tag]\!] \neq \iota & \\ \mathcal{C}[\![P]\!]\rho\sigma \quad \text{if } n > 0 \wedge T_1 = tag\ P \wedge \rho[\![tag]\!] = \iota & \\ \mathcal{C}[\![P]\!]\rho(newstore[\![e]\!]\rho\sigma\underline{\beta}) & \\ \qquad \text{if } n > 0 \wedge T_1 = tag\ ; e\ P \wedge \rho[\![tag]\!] = \iota \wedge & \\ \qquad\quad newstore[\![e]\!]\rho\sigma\underline{\beta} \neq \bot_{\mathcal{S}} & \\ \underline{action}\ \iota\underline{\beta}\ \varrho\ [\![T, T_2, \ldots, T_n]\!](\bar{\mathcal{D}}[\![\Delta]\!]\rho)\sigma & \\ \qquad \text{if } n > 0 \wedge T_1 = \Delta : T \wedge \mathcal{D}[\![\Delta]\!]\rho \neq \bot & \\ \underline{action}\ \iota\underline{\beta}\ \varrho\ [\![T, T_2, \ldots, T_n]\!](\mathcal{A}[\![\Theta]\!]\rho\sigma)\sigma & \\ \qquad \text{if } n > 0 \wedge T_1 = \Theta : T \wedge \mathcal{A}[\![\Theta]\!]\rho\sigma \neq \bot & \\ \bot_{\mathcal{Q}} \quad \text{otherwise} & \end{cases}$$

Giving a semantic interpretation is made more complicated by the fact that the action depends on the input in a deterministic way. At first glance, it may appear that the easiest solution would be to store the input and then act as for a CASE command. This is unacceptable since it requires tags to be storable – an assumption better not to make if it can be avoided. When the input carries data with it the auxiliary function *newstore* is used in order to correctly deal with the case where the data is not an atomic value.

$$\begin{aligned} \mathcal{C}[\![\text{PRI PAR } (Q_1, \ldots, Q_n)]\!]\rho\sigma &= \mathcal{C}[\![\text{PAR } (Q_1, \ldots, Q_n)]\!]\rho\sigma \\ \mathcal{C}[\![\text{PRI PAR } x = e_1 \text{ FOR } e_2\ Q]\!]\rho\sigma &= \mathcal{C}[\![\text{PAR } x = e_1 \text{ FOR } e_2\ Q]\!]\rho\sigma \\ \mathcal{C}[\![\text{PLACED PAR } (Q_1, \ldots, Q_n)]\!]\rho\sigma &= \mathcal{C}[\![\text{PAR } (Q_1, \ldots, Q_n)]\!]\rho\sigma \\ \mathcal{C}[\![\text{PLACED PAR } x = e_1 \text{ FOR } e_2\ Q]\!]\rho\sigma &= \mathcal{C}[\![\text{PAR } x = e_1 \text{ FOR } e_2\ Q]\!]\rho\sigma \\ \mathcal{C}[\![\text{PROCESSOR } e\ Q]\!]\rho\sigma &= \mathcal{C}[\![Q]\!]\rho\sigma \\ & \quad \text{provided } \mathcal{E}_V[\![e]\!]\rho\sigma \in \overline{\text{INT}} \end{aligned}$$

Above is included a selection of clauses which result from the decision not to model priority and placement within the mathematical model. Care must be taken to maintain the desired strictness properties.

$$\begin{aligned} \mathcal{C}[\![\text{PLACE } x \text{ AT } e : P]\!]\rho\sigma &= \mathcal{C}[\![P]\!]\rho\sigma \\ & \quad \text{provided } \mathcal{E}_V[\![e]\!]\rho\sigma \in \overline{\text{INT}} \end{aligned}$$

The decision was taken not to model the allocation of channels, timers or variables to absolute locations in store. Provided the expression is not erroneous, the placement is ignored.

Next the semantic definition for the parallel construct will be given. As was found in [26], this proves to be the most complicated to define. The first part of the definition shows how the

local environments required for each individual process are set up. The second part shows how the processes interact once they are running. Overall this gives

$$\mathcal{C}[\![\text{PAR}\ (Q_1,\ldots,Q_n)]\!]\rho\sigma = \begin{cases} skip_{\rho\sigma} & \text{if } n = 0 \\ (\sigma \|_{i=1}^{n} (\mathcal{C}[\![P_i]\!]\rho_i\sigma))/Y & \text{otherwise} \end{cases}$$

$$\text{where } P_i = \begin{cases} Q_i & \text{if } Q_i \in Proc \\ P & \text{if } Q_i = U_i : P \text{ with } U_i \in PD \text{ and } P \in Proc \end{cases}$$

The processes are run in parallel (||) with their respective environments (ρ_i). The communications local to the network are then hidden ($/Y$).

It was mentioned previously that the inclusion of parallel declarations was to be optional. When such declarations are not included, it is necessary to calculate the local environments by syntactic analysis. Because of this, calculation of local environments will be split into two cases.

Parallel declarations present Given that parallel declarations accompany each process, the first step is to use them to calculate the necessary information concerning store and channel use. This is done by the following semantic functions, the definitions of which are not difficult but are omitted for brevity.

$$\begin{aligned} inchans &: PD \longrightarrow ENV \longrightarrow S \longrightarrow \mathcal{P}(CHAN)^+ \\ outchans &: PD \longrightarrow ENV \longrightarrow S \longrightarrow \mathcal{P}(CHAN)^+ \\ ownchans &: PD \longrightarrow ENV \longrightarrow S \longrightarrow \mathcal{P}(CHAN)^+ \\ addrs &: PD \longrightarrow ENV \longrightarrow S \longrightarrow \mathcal{P}(ADDR)^+ \end{aligned}$$

To be declared as an input channel by $inchans[\![U_i]\!]\rho\sigma$, χ must have status $\underline{u}$ or $\underline{i}$ in ρ; output channels must have status $\underline{u}$ or $\underline{o}$ in ρ; internal channels must have status $\underline{u}$ in ρ. If an undirected ($\underline{u}$) channel of ρ is declared as an input (output) channel by one of the parallel declarations, then it must be declared as an output (input) channel by another. To be declared as an address of a variable which can be assigned to, α must have status $\underline{u}$ in ρ.

In addition, the parallel declarations accompanying a parallel construct must satisfy the following collection of equations:

$$\begin{aligned} inchans[\![U_i]\!]\rho\sigma \cap ownchans[\![U_j]\!]\rho\sigma &= \emptyset \\ outchans[\![U_i]\!]\rho\sigma \cap ownchans[\![U_j]\!]\rho\sigma &= \emptyset \\ inchans[\![U_i]\!]\rho\sigma \cap outchans[\![U_i]\!]\rho\sigma &= \emptyset \\ inchans[\![U_i]\!]\rho\sigma \cap inchans[\![U_j]\!]\rho\sigma &= \emptyset \quad \text{whenever } i \neq j \\ outchans[\![U_i]\!]\rho\sigma \cap outchans[\![U_j]\!]\rho\sigma &= \emptyset \quad \text{whenever } i \neq j \\ addrs[\![U_i]\!]\rho\sigma \cap addrs[\![U_j]\!]\rho\sigma &= \emptyset \quad \text{whenever } i \neq j \end{aligned}$$

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

$$\rho_i[\![\chi]\!] = \begin{cases} \underline{u} & \text{if } \chi \in ownchans[\![U_i]\!]\rho\sigma \\ \underline{i} & \text{if } \chi \in inchans[\![U_i]\!]\rho\sigma \\ \underline{o} & \text{if } \chi \in outchans[\![U_i]\!]\rho\sigma \\ \underline{f} \text{ or } undefined & \text{if } \rho[\![\chi]\!] = \underline{f} \text{ subject to} \\ & \quad \rho_i[\![\chi]\!] = \underline{f} \Rightarrow \rho_j[\![\chi]\!] = undefined \text{ whenever } i \neq j \\ undefined & \text{otherwise} \end{cases}$$

Since it has been assumed that there are an infinite number of abstract channels, it willbe assumed that an infinite number are allocated to each ρ_i. Thus problems will not manifest themselves when it becomes necessary, due to the presence of declarations, to allocate abstract channels to each process.

$$\rho_i[\![\alpha]\!] = \begin{cases} \underline{u} & \text{if } \alpha \in addrs[\![U_i]\!]\rho\sigma \\ \underline{r} & \text{if } \rho[\![\alpha]\!] \in \{\underline{r}, \underline{u}\} \wedge \alpha \notin \bigcup_{j=1}^{n} addrs[\![U_j]\!]\rho\sigma \\ undefined & \text{if } \exists j \neq i \bullet \alpha \in addrs[\![U_j]\!]\rho\sigma \\ \underline{f} \text{ or } undefined & \text{if } \rho[\![\alpha]\!] = \underline{f} \text{ subject to} \\ & \quad \rho_i[\![\alpha]\!] = \underline{f} \Rightarrow \rho_j[\![\alpha]\!] = undefined \text{ whenever } i \neq j \\ undefined & \text{otherwise} \end{cases}$$

An assumption of an infinite number of addresses within the store has already been made, and so it will be assumed that an infinite number of free addresses (chosen in such a way to ensure that each process can allocate suitable store space for each of its local variables) will be allocated to each ρ_i.

Parallel declarations absent Without parallel declarations, it is not possible to derive a suitable local environment for each process by consideration of the process alone. This is because it is impossible to tell from examination of a process whether its input and output channels are shared by exactly one other process defined in parallel. The solution is a pair of semantic functions ($\mathcal{W}_L$ and $\mathcal{W}_V$) which input *all* the parallel processes and return suitable store and channel allocations respectively.

$$\mathcal{W}_V : Proc^{\&} \longrightarrow ENV \longrightarrow S \longrightarrow VSTATUS^{\&}$$

$$\mathcal{W}_L : Proc^{\&} \longrightarrow ENV \longrightarrow S \longrightarrow LSTATUS^{\&}$$

Using these functions (the definitions of which, as intimated earlier, are not included) it is trivial to provide suitable local environments. The first component of each ρ_i is again the same as ρ. For the other components,

$$\rho_i[\![\chi]\!] = \omega_i \quad \text{where } \mathcal{W}_V[\![P_1, \ldots, P_n]\!]\rho\sigma = \langle \omega_1, \ldots, \omega_n \rangle$$

$$\rho_i[\![\alpha]\!] = \omega_i \quad \text{where } \mathcal{W}_L[\![P_1, \ldots, P_n]\!]\rho\sigma = \langle \omega_1, \ldots, \omega_n \rangle$$

This completes the definition of the local environments. The set of channels on which communications are to be hidden (Y) is found by examination of the local environments and is defined

$$Y = \{\chi \mid \exists i, j \bullet \rho_i[\![\chi]\!] = \underline{i} \wedge \rho_j[\![\chi]\!] = \underline{o}\}$$

The parallel operator ($||$) and hiding operator ($/Y$) defined below are derived from the

corresponding CSP operators of [8].

$$
\begin{aligned}
f(\sigma \|_{i=1}^{n} A_i) \;=\; & \{(s, X) \mid \exists s_1, \ldots, s_n, X_1, \ldots, X_n \bullet \\
& \quad s_1 = s \restriction \mathrm{dom}(c(A_1)) \wedge \cdots \wedge s_n = s \restriction \mathrm{dom}(c(A_n)) \wedge \\
& \quad (s_1, X_1) \in f(A_1) \wedge \cdots \wedge (s_n, X_n) \in f(A_n) \wedge X \subseteq X_1 \cup \cdots \cup X_n\} \cup \\
& \{(su, X) \mid X \subseteq (\textstyle\bigcup_{i=1}^{n} \mathrm{dom}(c(A_i))) \cup \{\surd\}) \wedge \\
& \quad s \restriction \mathrm{dom}(c(A_1)) \in \mathit{traces}(A_1) \wedge \cdots \wedge \\
& \quad s \restriction \mathrm{dom}(c(A_n)) \in \mathit{traces}(A_n) \wedge \exists i \bullet t(A_i)(s \restriction \mathrm{dom}(c(A_i))) = \bot\}
\end{aligned}
$$

$$
t(\sigma \|_{i=1}^{n} A_i)s \;=\; \begin{cases}
\bot & \text{if } s \in \mathit{traces}(\sigma \|_{i=1}^{n} A_i) \wedge \\
& \quad \exists u \leq s \bullet (\forall i \bullet u \restriction \mathrm{dom}(c(A_i)) \in \mathit{traces}(A_i)) \wedge \\
& \quad\quad (\exists i \bullet t(A_i)(u \restriction \mathrm{dom}(c(A_i))) = \bot) \\
\{(\sigma \oplus \sigma_1 \oplus \cdots \oplus \sigma_n) \downarrow \{\alpha \mid \exists i \bullet \alpha \in \mathrm{dom}(t(A_i))\} \mid & \\
\quad\quad \forall i \bullet \sigma_i \in t(A_i)(s \restriction \mathrm{dom}(c(A_i)))\} & \\
\text{otherwise} &
\end{cases}
$$

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

$$
\begin{aligned}
\langle\rangle \restriction X \;&=\; \langle\rangle \\
s\langle a\rangle \restriction X \;&=\; \begin{cases} s \restriction X & \text{if } a = \chi.\beta \wedge \chi \notin X \\ (s \restriction X)\langle a\rangle & \text{if } a = \chi.\beta \wedge \chi \in X. \end{cases}
\end{aligned}
$$

Note that within the definition of the final states of the process, the contents of all readable addresses (not solely those which are mutable addresses of one of the constituent processes) are retained. This is necessary in order to preserve the information pertaining to addresses not mentioned by any of the constituent processes.

The parallel operator works by allowing each process to communicate only in its own 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 the processes agree. The final states of a parallel process are calculated by collecting the final states from each of the constituent processes and combining them. Because it is not necessary to allow each mutable address to be altered by one of the constituent processes, it is necessary to include the initial state (σ) in the combination of states. If σ and σ_i differ on the value of the contents of an address, the contents of σ_i must be taken; if two σ_i differ on the value of the contents of an address, then either value can be taken, since by the disjointness rules this can never occur in occam. As soon as one process diverges, the whole system does.

$$
\begin{aligned}
f(A/Y) \;=\; & \{(s \restriction (\mathrm{dom}(c(A)) - Y), X) \mid (s, X \cup Y) \in f(A)\} \cup \\
& \{(s, X) \mid \{u \in \mathit{traces}(A) \mid u \restriction (\mathrm{dom}(c(A)) - Y) \leq s\} \text{ is infinite}\}
\end{aligned}
$$

$$
t(A/Y)s \;=\; \begin{cases}
\bot & \text{if } \{u \in \mathit{traces}(A) \mid u \restriction (\mathrm{dom}(c(A)) - Y) \leq s\} \text{ is infinite} \\
\bigcup\{(t(A)u) \downarrow \{\alpha \mid t(A/Y)\alpha = r w\} \mid u \restriction (\mathrm{dom}(c(A)) - Y) = s\} & \\
\text{otherwise} &
\end{cases}
$$

The hiding operator is used to conceal communications over channels which are internal to the parallel system. The above definition is continuous provided that only finitely many outputs can occur on any channel and that Y is finite. (The assumption of finite outputs over any channel has already been made; the finiteness of Y is a consequence of the assumption of any process using only a finite number of channels.)

The operator $/\,Y$ transforms communications over channels 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) action over a channel in Y may bring the process into a state where it can accept an element of X.

This completes the definition of the parallel operator PAR.

$$\mathcal{C}[\![\texttt{PAR } x = e_1 \texttt{ FOR } e_2\ Q]\!]\rho\sigma = \begin{cases} \mathit{skip}_{\rho\sigma} & \text{if } \beta_2 = 0 \\ (\sigma \|_{i=1}^{\beta_2} (\mathcal{C}[\![P]\!]\varrho_i\sigma))/Y & \text{otherwise} \end{cases}$$

$$\text{where } \mathcal{E}_V[\![e_1]\!]\rho\sigma = \beta_1 \wedge \mathcal{E}_V[\![e_2]\!]\rho\sigma = \beta_2 \wedge$$
$$\varrho_i = \rho_1[\beta_1 + i - 1/x] \wedge$$
$$P = \begin{cases} Q & \text{if } Q \in \mathit{Proc} \\ P & \text{if } Q = U : P \text{ with } U \in \mathit{PD} \text{ and } P \in \mathit{Proc} \end{cases}$$

$$\text{provided } \mathcal{E}_V[\![e_1]\!]\rho\sigma \in \overline{\texttt{INT}} \wedge \mathcal{E}_V[\![e_2]\!]\rho\sigma \in \overline{\texttt{INT}} \wedge$$
$$\mathcal{E}_V[\![e_2]\!]\rho\sigma \geq 0 \wedge$$
$$\mathcal{E}_V[\![e_1]\!]\rho\sigma + \mathcal{E}_V[\![e_2]\!]\rho\sigma - 1 \leq \texttt{MOSTPOS INT}$$

Once the framework to carry out a parallel command exists, replicated parallel commands offer no extra difficulty. Notice that each of the constituent processes is supplied with a different environment, taking account of the index value as well as the information provided by the parallel declarations or syntactic analysis.

$$\mathcal{C}[\![\Delta : P]\!]\rho\sigma = \mathcal{C}[\![P]\!](\mathcal{D}[\![\Delta]\!]\rho)\sigma$$
$$\text{provided } \mathcal{D}[\![\Delta]\!]\rho \in \mathit{ENV}$$

If a declaration precedes a process, the necessary changes to the environment are made before the process begins.

$$\mathcal{C}[\![\Theta : P]\!]\rho\sigma = \mathcal{C}[\![P]\!](\mathcal{A}[\![\Theta]\!]\rho\sigma)\sigma$$
$$\text{provided } \mathcal{A}[\![\Theta]\!]\rho\sigma \in \mathit{ENV}$$

The clause dealing with processes prefixed by a specification is exactly as expected, with the relevant changes to the environment being carried out before the process commences.

The only clause left to define is that concerning procedures. This is dependent on the way in which named procedures are stored in the environment by declarations, and is consequently delayed until the function $\mathcal{D}$ has been defined.

(3) THE FUNCTION $\mathcal{D}$ Within this section, the existence of the following semantic functions will be assumed. The definition of *new* is extremely implementation dependent; those of *newchan*, *newtim* and *newport* are verbose but not difficult. Hence none of the definitions are included.

$$\begin{aligned} \mathit{new} &: \mathit{TYP} \longrightarrow \mathit{ENV} \longrightarrow \mathit{LOC} \\ \mathit{newchan} &: \mathbb{N}^* \longrightarrow \mathit{PROT} \longrightarrow \mathit{ENV} \longrightarrow \mathit{CHAN}^{\$} \\ \mathit{newtim} &: \mathbb{N}^* \longrightarrow \wp(\mathit{TIMER}) \longrightarrow \mathit{ENV} \longrightarrow \mathit{TIMER}^{\$} \\ \mathit{newport} &: \mathbb{N}^* \longrightarrow \wp(\mathit{PORT}) \longrightarrow \mathit{TYP} \longrightarrow \mathit{ENV} \longrightarrow (\mathit{TYP} \times \mathit{PORT}^{\$}) \end{aligned}$$

The function *new* takes in a type and environment. It returns a location which may be used to store the current value of a variable of the supplied type. The location returned will be such that it refers to a contiguous area of store, all of the addresses of which had been mapped to $\underline{t}$ under the third component of the environment. Any such contiguous area of the store is suitable;

the particular allocation strategy used will depend on the implementation of the store under consideration.

The function *newchan* takes in a natural number sequence (indicating the array structure of the required set of channels), protocol (that intended for each channel of the array) and environment and returns a sequence. This sequence contains the information concerning the array of channels (the sequence of abstract channels used to represent the array) which must be stored in the first component of the environment. Each of the abstract channels returned will have been mapped to $\underline{f}$ under the second component of the environment, and any such abstract channels are suitable.

The function *newtim* takes in a natural number sequence (indicating the array structure of the required set of timers), collection of abstract timers and environment and returns a sequence of abstract timers; the structure of the sequence returned will match that suggested by the natural number sequence. None of the abstract timers returned will previously have been allocated, and none will be contained within the collection of abstract timers, but no further restrictions on those which can result need be imposed.

The function *newport* takes in a natural number sequence, collection of abstract ports, primitive or array type and environment and returns a pair consisting of a sequence of abstract ports (mirroring the desired array structure) and the type intended for each of the ports being declared. None of the tokens returned by the function will have been previously allocated, and none will be contained within the collection of abstract ports, but any such tokens will do.

In the clauses which follow it will often be necessary to evaluate expressions consisting solely of constants or operations thereon. The expression evaluation semantic function has already been defined and adequately carries out this task. However, before applying this function, a state is required regardless of whether it is ever examined. When a state is required, but cannot sensibly be accessed, σ_{err} will be used. This state maps every location to *error*, and hence any expression dependent on it evaluates to $\perp_{\mathcal{E}}$.

Below are given the semantic clauses, using the functions above as necessary. Several 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 $\perp$.

$$\mathcal{D}\left[\begin{matrix}[e_1]\ldots[e_n]\\ \tau\, x_1,\ldots,x_n\end{matrix}\right]\rho = \begin{cases}\rho & \text{if } m = 0\\ \varrho_m[\lambda_1/x_1]\ldots[\lambda_m/x_m] & \text{if } m > 0 \wedge \varrho_m \neq \perp\\ \perp & \text{otherwise}\end{cases}$$

where $\mathcal{E}_V[\![e_1]\!]\rho\sigma_{err} = \beta_1 \wedge \cdots \wedge \mathcal{E}_V[\![e_n]\!]\rho\sigma_{err} = \beta_n \wedge$
new $(\langle\beta_1,\ldots,\beta_n\rangle,\tau)\,\rho = \lambda_1 \wedge \cdots \wedge$
new $(\langle\beta_1,\ldots,\beta_n\rangle,\tau)\,\varrho_{m-1} = \lambda_m \wedge$
markaddr $\lambda_1\ \rho = \varrho_1 \wedge \cdots \wedge$ *markaddr* $\lambda_m\ \varrho_{m-1} = \varrho_m$

provided $\mathcal{E}_V[\![e_1]\!]\rho\sigma_{err} \in \overline{\text{INT}} \wedge \cdots \wedge \mathcal{E}_V[\![e_n]\!]\rho\sigma_{err} \in \overline{\text{INT}} \wedge$
$\tau \in (\{\text{BOOL}\} \cup \mathit{INTEGER} \cup \mathit{REAL})$

In the clause above the function *markaddr* has been used. This function takes in a location and an environment and returns the environment which results when all the addresses referred to by the location have been mapped to $\underline{u}$. The definition is dependent on the particular implementation of the store under consideration and is hence not included.

Declaration of variables involves allocation of suitable locations and relevant marking of the addresses which are to be used to store part of the values of the variables. Errors result if τ is not a primitive data type. The type of each of the variables being declared is calculated before any updating of the first component of the environment is done since otherwise array dimensions

containing the names of any of the variables being declared would not be given the correct treatment.

$$\mathcal{D}\llbracket [e_1]\ldots[e_n]\ \text{CHAN OF}\ [f_1]\ldots[f_{n'}]\ \tau\ x_1,\ldots,x_m\rrbracket\rho = \begin{cases}\rho & \text{if } m = 0\\ \varrho_m[ys_1/x_1]\ldots[ys_m/x_m] & \text{if } m > 0 \wedge \varrho_m \neq \bot\\ \bot & \text{otherwise}\end{cases}$$

where $\mathcal{E}_V\llbracket e_1\rrbracket\rho\sigma_{err} = \beta_1 \wedge \cdots \wedge \mathcal{E}_V\llbracket e_n\rrbracket\rho\sigma_{err} = \beta_n \wedge$
$\mathcal{E}_V\llbracket f_1\rrbracket\rho\sigma_{err} = \gamma_1 \wedge \cdots \wedge \mathcal{E}_V\llbracket f_{n'}\rrbracket\rho\sigma_{err} = \gamma_{n'} \wedge$
$\beta = \langle\beta_1,\ldots,\beta_n\rangle \wedge \gamma = \langle\gamma_1,\ldots,\gamma_{n'}\rangle \wedge$
newchan $\beta\,(\gamma,\tau)\,\rho = ys_1 \wedge \cdots \wedge$
newchan $\beta\,(\gamma,\tau)\,\varrho_{m-1} = ys_m \wedge$
markchans $(\gamma,\tau)\,ys_1\,\rho = \varrho_1 \wedge \cdots \wedge$
markchans $(\gamma,\tau)\,ys_m\,\varrho_{m-1} = \varrho_m$

provided $\mathcal{E}_V\llbracket e_1\rrbracket\rho\sigma_{err} \in \overline{\text{INT}} \wedge \cdots \wedge \mathcal{E}_V\llbracket e_n\rrbracket\rho\sigma_{err} \in \overline{\text{INT}} \wedge$
$\mathcal{E}_V\llbracket f_1\rrbracket\rho\sigma_{err} \in \overline{\text{INT}} \wedge \cdots \wedge \mathcal{E}_V\llbracket f_{n'}\rrbracket\rho\sigma_{err} \in \overline{\text{INT}} \wedge$
$\tau \in (\{\text{BOOL}\} \cup \mathit{INTEGER} \cup \mathit{REAL})$

$$\mathcal{D}\llbracket [e_1]\ldots[e_n]\ \text{CHAN OF}\ \tau_1 :: []\,[f_1]\ldots[f_{n'}]\ \tau_2\ x_1,\ldots,x_m\rrbracket\rho = \begin{cases}\rho & \text{if } m = 0\\ \varrho_m[ys_1/x_1]\ldots[ys_m/x_m] & \text{if } m > 0 \wedge \varrho_m \neq \bot\\ \bot & \text{otherwise}\end{cases}$$

where $\mathcal{E}_V\llbracket e_1\rrbracket\rho\sigma_{err} = \beta_1 \wedge \cdots \wedge \mathcal{E}_V\llbracket e_n\rrbracket\rho\sigma_{err} = \beta_n \wedge$
$\mathcal{E}_V\llbracket f_1\rrbracket\rho\sigma_{err} = \gamma_1 \wedge \cdots \wedge \mathcal{E}_V\llbracket f_{n'}\rrbracket\rho\sigma_{err} = \gamma_{n'} \wedge$
$\beta = \langle\beta_1,\ldots,\beta_n\rangle \wedge \gamma = \langle\gamma_1,\ldots,\gamma_{n'}\rangle \wedge$
newchan $\beta\,(\tau_1 :: (\gamma,\tau_2))\,\rho = ys_1 \wedge \cdots \wedge$
newchan $\beta\,(\tau_1 :: (\gamma,\tau_2))\,\varrho_{m-1} = ys_m \wedge$
markchans $(\tau_1 :: (\gamma,\tau_2))\,ys_1\,\rho = \varrho_1 \wedge \cdots \wedge$
markchans $(\tau_1 :: (\gamma,\tau_2))\,ys_m\,\varrho_{m-1} = \varrho_m$

provided $\mathcal{E}_V\llbracket e_1\rrbracket\rho\sigma_{err} \in \overline{\text{INT}} \wedge \cdots \wedge \mathcal{E}_V\llbracket e_n\rrbracket\rho\sigma_{err} \in \overline{\text{INT}} \wedge$
$\mathcal{E}_V\llbracket f_1\rrbracket\rho\sigma_{err} \in \overline{\text{INT}} \wedge \cdots \wedge \mathcal{E}_V\llbracket f_{n'}\rrbracket\rho\sigma_{err} \in \overline{\text{INT}} \wedge$
$\tau_1 \in \mathit{INTEGER} \wedge$
$\tau_2 \in (\{\text{BOOL}\} \cup \mathit{INTEGER} \cup \mathit{REAL})$

$$\mathcal{D}\llbracket [e_1]\ldots[e_n]\ \text{CHAN OF ANY}\ x_1,\ldots,x_m\rrbracket\rho = \begin{cases}\rho & \text{if } m = 0\\ \varrho_m[ys_1/x_1]\ldots[ys_m/x_m] & \text{if } m > 0 \wedge \varrho_m \neq \bot\\ \bot & \text{otherwise}\end{cases}$$

where $\mathcal{E}_V\llbracket e_1\rrbracket\rho\sigma_{err} = \beta_1 \wedge \cdots \wedge \mathcal{E}_V\llbracket e_n\rrbracket\rho\sigma_{err} = \beta_n \wedge$
$\beta = \langle\beta_1,\ldots,\beta_n\rangle \wedge$
newchan β ANY $\rho = ys_1 \wedge \cdots \wedge$
newchan β ANY $\varrho_{m-1} = ys_m \wedge$
markchans ANY $ys_1\,\rho = \varrho_1 \wedge \cdots \wedge$
markchans ANY $ys_m\,\varrho_{m-1} = \varrho_m$

provided $\mathcal{E}_V\llbracket e_1\rrbracket\rho\sigma_{err} \in \overline{\text{INT}} \wedge \cdots \wedge \mathcal{E}_V\llbracket e_n\rrbracket\rho\sigma_{err} \in \overline{\text{INT}}$

$$\mathcal{D}[\![[e_1]\ldots[e_n]\,\texttt{CHAN OF}\ x\ x_1,\ldots,x_m]\!]\rho = \begin{cases}\rho & \text{if } m=0\\ \varrho_m[ys_1/x_1]\ldots[ys_m/x_m] & \text{if } m>0 \wedge \varrho_m \neq \bot\\ \bot & \text{otherwise}\end{cases}$$

where $\mathcal{E}_V[\![e_1]\!]\rho\sigma_{err} = \beta_1 \wedge \cdots \wedge \mathcal{E}_V[\![e_n]\!]\rho\sigma_{err} = \beta_n \wedge$
$\beta = \langle\beta_1,\ldots,\beta_n\rangle \wedge$
newchan $\beta\ (\rho[\![x]\!])\ \rho = ys_1 \wedge \cdots \wedge$
newchan $\beta\ (\rho[\![x]\!])\ \varrho_{m-1} = ys_m \wedge$
markchans $(\rho[\![x]\!])\ ys_1\ \rho = \varrho_1 \wedge \cdots \wedge$
markchans $(\rho[\![x]\!])\ ys_m\ \varrho_{m-1} = \varrho_m$

provided $\mathcal{E}_V[\![e_1]\!]\rho\sigma_{err} \in \overline{\texttt{INT}} \wedge \cdots \wedge \mathcal{E}_V[\![e_n]\!]\rho\sigma_{err} \in \overline{\texttt{INT}} \wedge$
$\rho[\![x]\!] \in PROT$

In the four clauses above the function *markchans* has been used.

$$\begin{aligned} \textit{markchans}\ p\ \langle\rangle\ \rho &= \rho\\ \textit{markchans}\ p\ (\chi : xs)\ \rho &= \textit{markchans}\ p\ xs\ (\rho[(p,\underline{u})/\chi])\\ \textit{markchans}\ p\ (ys : yss)\ \rho &= \textit{markchans}\ p\ yss\ (\textit{markchans}\ p\ ys\ \rho)\end{aligned}$$

Declaration of channels involves two stages. First, suitable sequences of abstract channels are found for each identifier and marked in the environment with the protocol of the channels being declared and the token $\underline{u}$. Second, the environment which results is altered to associate each of the identifiers with its corresponding sequence of abstract channels.

$$\mathcal{D}[\![[e_1]\ldots[e_n]\ \texttt{TIMER}\ x_1,\ldots,x_m]\!]\rho = \begin{cases}\rho & \text{if } m=0\\ \rho[ys_1/x_1]\ldots[ys_m/x_m] & \text{if } m>0 \wedge \rho \neq \bot\\ \bot & \text{otherwise}\end{cases}$$

where $\mathcal{E}_V[\![e_1]\!]\rho\sigma_{err} = \beta_1 \wedge \cdots \wedge \mathcal{E}_V[\![e_n]\!]\rho\sigma_{err} = \beta_n \wedge$
$\beta = \langle\beta_1,\ldots,\beta_m\rangle \wedge$
newtim $\beta\ \emptyset\ \rho = ys_1 \wedge \cdots \wedge$ *newtim* $\beta\ A_{m-1}\ \rho = ys_m \wedge$
elements $ys_1 = A_1 \wedge \cdots \wedge$ *elements* $ys_{m-1} = A_{m-1}$

provided $\mathcal{E}_V[\![e_1]\!]\rho\sigma_{err} \in \overline{\texttt{INT}} \wedge \cdots \wedge \mathcal{E}_V[\![e_n]\!]\rho\sigma_{err} \in \overline{\texttt{INT}}$

Declaration of timers involves updating the environment in order to associate suitable abstract timers with each of the identifiers. Care must be taken avoid allocating the same abstract timer to more than one identifier being declared at the same time; *newtim* achieves this.

$$\mathcal{D}[\![[e_1]\ldots[e_n]\ \texttt{PORT OF}\ [f_1]\ldots[f_{n'}]\ \tau\ x_1,\ldots,x_m]\!]\rho = \begin{cases}\rho & \text{if } m=0\\ \rho[(t_1,ys_1)/x_1]\ldots[(t_m,ys_m)/x_m] & \text{if } m>0 \wedge \rho \neq \bot\\ \bot & \text{otherwise}\end{cases}$$

where $\mathcal{E}_V[\![e_1]\!]\rho\sigma_{err} = \beta_1 \wedge \cdots \wedge \mathcal{E}_V[\![e_m]\!]\rho\sigma_{err} = \beta_n \wedge$
$\mathcal{E}_V[\![f_1]\!]\rho\sigma_{err} = \gamma_1 \wedge \cdots \wedge \mathcal{E}_V[\![f_{n'}]\!]\rho\sigma_{err} = \gamma_{n'} \wedge$
$\beta = \langle\beta_1,\ldots,\beta_n\rangle \wedge \gamma = \langle\gamma_1,\ldots,\gamma_{n'}\rangle \wedge$
newport $\beta\ \emptyset\ (\gamma,\tau)\ \rho = (t_1, ys_1) \wedge \cdots \wedge$
newport $\beta\ A_{m-1}\ (\gamma,\tau)\ \rho = (t_m, ys_m) \wedge$
elements $ys_1 = A_1 \wedge \cdots \wedge$ *elements* $ys_{m-1} = A_{m-1}$

provided $\mathcal{E}_V[\![e_1]\!]\rho\sigma_{err} \in \overline{\texttt{INT}} \wedge \cdots \wedge \mathcal{E}_V[\![e_n]\!]\rho\sigma_{err} \in \overline{\texttt{INT}} \wedge$
$\mathcal{E}_V[\![f_1]\!]\rho\sigma_{err} \in \overline{\texttt{INT}} \wedge \cdots \wedge \mathcal{E}_V[\![f_{n'}]\!]\rho\sigma_{err} \in \overline{\texttt{INT}} \wedge$
$\tau \in (\{\texttt{BOOL}\} \cup INTEGER \cup REAL)$

Declaration of ports involves updating the environment in order to associate suitable abstract ports with each identifier. As with declaration of timers, allocating the same abstract ports to distinct identifiers must be avoided; *newport* deals with this situation.

$$\begin{aligned}\mathcal{D}\llbracket\texttt{PROTOCOL } x \texttt{ IS } [e_1]\ldots[e_n]\,\tau\rrbracket\rho \;=\; & \rho[(\langle\langle\beta_1,\ldots\beta_n\rangle,\tau\rangle)/x] \\ & \text{where } \mathcal{E}_V\llbracket e_1\rrbracket\rho\sigma_{err} = \beta_1 \wedge\cdots\wedge \mathcal{E}_V\llbracket e_n\rrbracket\rho\sigma_{err} = \beta_n \\ & \text{provided } \mathcal{E}_V\llbracket e_1\rrbracket\rho\sigma_{err} \in \overline{\texttt{INT}} \wedge\cdots\wedge \\ & \qquad \mathcal{E}_V\llbracket e_n\rrbracket\rho\sigma_{err} \in \overline{\texttt{INT}} \wedge \\ & \qquad \tau \in (\{\texttt{BOOL}\} \cup \mathit{INTEGER} \cup \mathit{REAL})\end{aligned}$$

Identifiers may be declared to represent named protocols in occam. Above the case of simple protocols not consisting of variable length arrays is dealt with.

$$\begin{aligned}\mathcal{D}\llbracket\texttt{PROTOCOL } x \texttt{ IS } \tau_1 :: [\,][e_1]\ldots[e_n]\,\tau_2\rrbracket\rho \;=\; & \rho[(\tau_1 :: (\langle\beta_1,\ldots\beta_n\rangle,\tau_2))/x] \\ & \text{where } \mathcal{E}_V\llbracket e_1\rrbracket\rho\sigma_{err} = \beta_1 \wedge\cdots\wedge \\ & \qquad \mathcal{E}_V\llbracket e_n\rrbracket\rho\sigma_{err} = \beta_n \\ & \text{provided } \mathcal{E}_V\llbracket e_1\rrbracket\rho\sigma_{err} \in \overline{\texttt{INT}} \wedge\cdots\wedge \\ & \qquad \mathcal{E}_V\llbracket e_n\rrbracket\rho\sigma_{err} \in \overline{\texttt{INT}} \wedge \\ & \qquad \tau_1 \in \mathit{INTEGER} \wedge \\ & \qquad \tau_2 \in (\{\texttt{BOOL}\} \cup \mathit{INTEGER} \cup \mathit{REAL})\end{aligned}$$

The extension to simple protocols consisting of variable length arrays poses no problems.

$$\begin{aligned}\mathcal{D}\llbracket\texttt{PROTOCOL } x \texttt{ IS } p_1 ; p_2 ; \ldots ; p_n\rrbracket\rho \;=\; & \rho[ys_1 \ldots ys_n/x] \\ & \text{where } ys_1 = (\mathcal{D}\llbracket\texttt{PROTOCOL } x \texttt{ IS } p_1\rrbracket\rho)\llbracket x\rrbracket \wedge\cdots\wedge \\ & \qquad ys_n = (\mathcal{D}\llbracket\texttt{PROTOCOL } x \texttt{ IS } p_n\rrbracket\rho)\llbracket x\rrbracket \\ & \text{provided } \mathcal{D}\llbracket\texttt{PROTOCOL } x \texttt{ IS } p_1\rrbracket\rho \neq \bot \wedge\cdots\wedge \\ & \qquad \mathcal{D}\llbracket\texttt{PROTOCOL } x \texttt{ IS } p_n\rrbracket\rho \neq \bot\end{aligned}$$

The most convenient method of storing a sequential protocol in the environment is to treat each part of the protocol separately and then concatenate the relevant contents of the resulting environments.

$$\begin{aligned}\mathcal{D}\llbracket\texttt{PROTOCOL } x \texttt{ IS CASE} \\ (tag_1 ; p_1), \ldots, (tag_n ; p_n)\rrbracket\rho \;=\; & \rho[\iota_1/tag_1]\ldots[\iota_n/tag_n][(\iota_1\ ys_1, \ldots, \iota_n\ ys_n)/x] \\ & \text{where } (\mathcal{D}\llbracket\texttt{PROTOCOL } x \texttt{ IS } p_1\rrbracket\rho)\llbracket x\rrbracket = ys_1 \wedge\cdots\wedge \\ & \qquad (\mathcal{D}\llbracket\texttt{PROTOCOL } x \texttt{ IS } p_n\rrbracket\rho)\llbracket x\rrbracket = ys_n \\ & \text{provided } \mathcal{D}\llbracket\texttt{PROTOCOL } x \texttt{ IS } p_1\rrbracket\rho \neq \bot \wedge\cdots\wedge \\ & \qquad \mathcal{D}\llbracket\texttt{PROTOCOL } x \texttt{ IS } p_n\rrbracket\rho \neq \bot\end{aligned}$$

Care must be taken when storing a variant protocol in the environment. The first step is the allocation of a disjoint set of tags to the identifiers which appear at the start of each option of the protocol. The particular choice of tags is unimportant, and for this reason a sequence (without repetitions) of the elements of TAG ($\langle\iota_1, \iota_2, \ldots, \iota_n, \ldots\rangle$) has been assumed. Once tags have been allocated, the next step is to calculate the protocols corresponding to each clause of the variant protocol. Finally, the necessary changes to the environment are combined. The tags which prefix each clause of a variant protocol declaration must be distinct; this is a consequence of the rules of occam.

The only clauses left to define concern procedure or function declarations. While considering such clauses it will be assumed, for clarity, that the identifiers which appear as formal parameters of procedures or functions do not appear outwith their associated procedure or function. Such a restriction can be verified during type-checking and failure to satisfy it can, if necessary, be rectified by simple textual renaming of the identifiers. In giving a semantic definition to such objects, the aim is to model the effect of copying the procedure or function body into the process at the point of call, sequentially replacing each of the formal parameters by its corresponding actual parameter. The substitution required (both for value and for reference parameters) closely resembles that necessary to deal with the abbreviation present in occam, and for this reason the decision has been taken to mention explicitly the dependence on the semantic function which deals with abbreviation. Although slightly unconventional, it is felt that within the context of this paper the increased clarity warranted such action.

$$\mathcal{D}[\![\texttt{PROC}\ q(\Psi) = P]\!]\rho \;=\; \rho[\pi/q] \quad \text{where } \pi = \lambda\langle\varrho_V,\varrho_L\rangle \bullet \lambda\Upsilon \bullet \lambda\sigma' \bullet \mathcal{C}[\![P]\!](\mathcal{L}[\![\Upsilon]\!][\![\Psi]\!](\rho_I,\varrho_V,\varrho_L)\sigma')\sigma'$$

Above is the clause governing procedure declarations. Free variables appearing within the procedure body are statically bound to the name used in the procedure declaration. The binding of identifiers used for any procedure call is that of the environment at the point of declaration; the second and third environment components used, however, are those from the point of call, since this affords protection against illegal channel or variable use by parallel processes. Within the above clause the auxiliary function $\mathcal{L}$ appears. This function updates the environment in order to correctly associate the formal and actual parameters.

$$\mathcal{L}[\![v_1,\ldots,v_n]\!][\![\psi_1,\ldots,\psi_n]\!]\rho\sigma = \begin{cases} \rho & \text{if } n = 0 \\ \mathcal{L}[\![y_2,\ldots,y_m,v_2,\ldots,v_n]\!][\![\texttt{VAL}\ \varsigma\ x_2,\ldots,x_m,\psi_2,\ldots,\psi_n]\!](\mathcal{A}[\![\texttt{VAL}\ x_1\ \texttt{IS}\ y_1]\!]\rho\sigma)\sigma & \\ & \text{if } n > 0 \wedge \psi_1 = \texttt{VAL}\ \varsigma\ x_1,\ldots,x_m \wedge \\ & \quad v_1 = y_1,\ldots,y_m \wedge m > 1 \wedge \\ & \quad \mathcal{A}[\![\texttt{VAL}\ x_1\ \texttt{IS}\ y_1]\!]\rho\sigma \neq \bot \\ \mathcal{L}[\![v_2,\ldots,v_n]\!][\![\psi_2,\ldots,\psi_n]\!](\mathcal{A}[\![\texttt{VAL}\ x\ \texttt{IS}\ y]\!]\rho\sigma)\sigma & \\ & \text{if } n > 0 \wedge \psi_1 = \texttt{VAL}\ \varsigma\ x \wedge v_1 = y \wedge \\ & \quad \mathcal{A}[\![\texttt{VAL}\ x\ \texttt{IS}\ y]\!]\rho\sigma \neq \bot \\ \mathcal{L}[\![y_2,\ldots,y_m,v_2,\ldots,v_n]\!][\![\varsigma\ x_2,\ldots,x_m,\psi_2,\ldots,\psi_n]\!](\mathcal{A}[\![x_1\ \texttt{IS}\ y_1]\!]\rho\sigma)\sigma & \\ & \text{if } n > 0 \wedge \psi_1 = \varsigma\ x_1,\ldots,x_m \wedge \\ & \quad v_1 = y_1,\ldots,y_m \wedge m > 1 \wedge \\ & \quad \mathcal{A}[\![x_1\ \texttt{IS}\ y_1]\!]\rho\sigma \neq \bot \\ \mathcal{L}[\![v_2,\ldots,v_n]\!][\![\psi_2,\ldots,\psi_n]\!](\mathcal{A}[\![x\ \texttt{IS}\ y]\!]\rho\sigma)\sigma & \\ & \text{if } n > 0 \wedge \psi_1 = \varsigma\ x \wedge v_1 = y \wedge \\ & \quad \mathcal{A}[\![x\ \texttt{IS}\ y]\!]\rho\sigma \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

Within the function $\mathcal{L}$, errors are captured by the use of the semantic function $\mathcal{A}$.

$$\mathcal{D}[\![\tau_1,\ldots,\tau_m\ \texttt{FUNCTION}\ g(\Psi) = e]\!]\rho \;=\; \rho[\pi/g] \quad \text{where } \pi = \lambda\langle\varrho_V,\varrho_L\rangle \bullet \lambda\Upsilon \bullet \lambda\sigma' \bullet \mathcal{E}_{V^*}[\![e]\!](\mathcal{L}[\![\Upsilon]\!][\![\Psi]\!](\rho_I,\varrho_V,\varrho_L)\sigma')\sigma'$$

Declaring functions whose body consists of a VALOF command is very similar to procedure declaration; the only noticeable difference is that the semantic function $\mathcal{E}_{V\&}$ is applied instead of the semantic function $\mathcal{C}$.

$$\begin{aligned}\mathcal{D}[\![\tau_1,\ldots,\tau_m\ \texttt{FUNCTION}\\ g(\Psi)\ \texttt{IS}\ e_1, e_2,\ldots,e_n]\!]\rho \;=\; & \rho[\pi/g]\\ & \text{where } \pi = \lambda\langle\varrho_V,\varrho_L\rangle \bullet \lambda\Upsilon \bullet \lambda\sigma' \bullet\\ & \qquad (\mathcal{E}_{V\&}[\![e_1]\!](\mathcal{L}[\![\Upsilon]\!][\![\Psi]\!]\langle\rho_1,\varrho_V,\varrho_L\rangle\sigma')\sigma')\ldots\\ & \qquad\quad (\mathcal{E}_{V\&}[\![e_n]\!](\mathcal{L}[\![\Upsilon]\!][\![\Psi]\!]\langle\rho_1,\varrho_V,\varrho_L\rangle\sigma')\sigma')\end{aligned}$$

The declaration of functions whose body does not consist of a VALOF command proceeds in the expected way. The items to be returned need not be atomic expressions.

(4) THE FUNCTION A Several clauses contain one or more conditions 'provided ...' which exclude error conditions. As before failure to meet these conditions causes the result $\perp$ to be returned.

$$\begin{aligned}\mathcal{A}[\![\texttt{VAL}\ x\ \texttt{IS}\ e]\!]\rho\sigma \;=\; & (\textit{restrict}\ e\ \rho)[\beta/x]\\ & \text{where } \mathcal{E}_V[\![e]\!](\textit{augment}\ \rho)\sigma = \beta\\ & \text{provided } \mathcal{E}_V[\![e]\!](\textit{augment}\ \rho)\sigma \neq \perp_{\mathcal{E}} \wedge \textit{contents}\ e \cap \textit{varabbr}\ \rho = \emptyset\end{aligned}$$

Here *augment* ρ is the environment whose second and third components are identical to those of ρ and whose first component is formed by replacing each occurrence of $(\underline{a}, A, \lambda)$ by λ. Use of this environment allows value abbreviations to make reference to unabbreviated components of arrays even if the array is the subject of a current variable abbreviation. Collating the addresses which are the subject of a current variable abbreviation is the task of *varabbr*, defined

$$\textit{varabbr}\ \rho = \bigcup\{A \mid \exists\lambda \in LOC \bullet (\underline{a}, A, \lambda) \in \rho_1\}$$

The above clause deals with the abbreviation of an expression. On first glance it may appear that this situation should be covered by the semantic function $\mathcal{D}$, but the dependence on the state σ makes this impossible. Within the definition, the auxiliary functions *restrict* and *contents* have been used. The purpose of *restrict*, whose definition is not difficult but depends on the particular implementation of the store under consideration, is to mark all addresses of any variable upon whose value the expression depends with $\underline{r}$. This prevents the values of any such variables being altered within the scope of the abbreviation. The function *restrict* is strict. The function *contents*, again implementation dependent and hence not defined, returns all addresses which must be accessed in order to evaluate the expression. This facilitates verifying that no variable-abbreviated addresses are referred to within a value abbreviation. In a value abbreviation the new identifier is associated with a constant, the value of the constant being that of the expression at the point of the abbreviation (and maintained throughout the scope of the abbreviation by the restriction imposed on the environment by the function *restrict*).

$$\mathcal{A}[\![\texttt{VAL}\ \varsigma\ x\ \texttt{IS}\ e]\!]\rho\sigma = \mathcal{A}[\![\texttt{VAL}\ x\ \texttt{IS}\ e]\!]\rho\sigma$$

When considering only type correct programs, the inclusion of a specifier within a value abbreviation is superfluous. In order to give a semantic definition to the clause, the specifier is first omitted.

$$\mathcal{A}[\![x \text{ IS } y]\!]\rho\sigma = \begin{cases} \rho[(\alpha,t)/x][(\underline{a},A,(\alpha,t))/y] & \text{if } \rho[\![y]\!] \in LOC \wedge \rho[\![y]\!] = (\alpha,t) \wedge \\ & \quad \rho[\![\alpha]\!] \in \{\underline{r},\underline{u}\} \wedge \textit{abode } y = A \\ \bot & \text{otherwise} \end{cases}$$

This clause carries out the abbreviation of a variable. The new identifier is associated with the location holding the value of the variable to be abbreviated, and the environment is further updated to prevent the abbreviated variable being used within the scope of the abbreviation. Calculation of the addresses holding the contents of the variable is left to the auxiliary function *abode*; the definition of this function is implementation dependent and hence not included. Since the value remains at the same position in the store (although the name by which it is referred to is altered, its contents are left untouched), no difference in treatment is necessary regardless of whether it was permitted to write to the variable.

$$\mathcal{A}[\![x \text{ IS } e_1[e_2]]\!]\rho\sigma = \begin{cases} (\textit{restrict } e_2\ \varrho)[(\alpha,t)/x][(\underline{a},A,(\alpha,t))/y] & \\ \quad \text{if } \mathcal{E}[\![e_1[e_2]]\!]\rho\sigma \in LOC \wedge \mathcal{E}[\![e_1[e_2]]\!]\rho\sigma = (\alpha,t) \wedge & \\ \quad\quad \rho[\![\alpha]\!] \in \{\underline{r},\underline{u}\} \wedge \mathcal{A}[\![x \text{ IS } e_1]\!]\rho\sigma = \varrho & \\ (\textit{restrict } e_2\ \varrho)[(\alpha,t)/x][(\underline{a},A \cup B,(\alpha',t'))/y] & \\ \quad \text{if } \mathcal{E}[\![e_1[e_2]]\!]\rho\sigma \in (\underline{a} \times \wp(ADDR) \times LOC) \wedge & \\ \quad\quad \mathcal{E}[\![e_1[e_2]]\!]\rho\sigma = (\underline{a},B,(\alpha,t)) \wedge \rho[\![\alpha]\!] \in \{\underline{r},\underline{u}\} \wedge & \\ \quad\quad \rho[\![y]\!] = (\underline{a},B',(\alpha',t')) \wedge B = B' \wedge & \\ \quad\quad A \cap B = \emptyset \wedge \mathcal{A}[\![x \text{ IS } e_1]\!](\rho[(\alpha',t')/y])\sigma = \varrho & \\ \bot \quad \text{otherwise} & \end{cases}$$

$$\text{where } \textit{abode } (e_1[e_2]) = A \wedge \textit{name } e_1 = y$$

$$\text{provided } \mathcal{E}[\![e_1[e_2]]\!]\rho\sigma \in LOC + (\underline{a} \times \wp(ADDR) \times LOC)$$

When abbreviating a component of an array, there are several complications which need to be dealt with.

Firstly, it is necessary to ensure that any variables which appear in subscript expressions are prevented from altering within the scope of the abbreviation; for this purpose the function *restrict* is used.

Secondly, it is the name of the identifier (and not merely a component of it) which must be marked as abbreviated within the environment; extracting the name of an identifier from an expression can be achieved by use of the function *name*, which (assuming that x represents an identifier) is defined

$$\begin{aligned} \textit{name } x &= x \\ \textit{name } (e_1[e_2]) &= \textit{name } e_1 \\ \textit{name } [e \text{ FROM } e_1 \text{ FOR } e_2] &= \textit{name } e \end{aligned}$$

Thirdly, a list of the addresses whose contents have been abbreviated must be kept in order to ensure that further abbreviations refer to disjoint parts of the array. The function *abode* which appears within the clause dealing with variable abbreviation can be used to adequately carry out the technical aspect of this task; the required list of addresses is precisely the union of those previously the subject of an abbreviation (and hence already stored in the environment) plus those picked out by *abode*.

$$\mathcal{A}[\![x \text{ IS } [e \text{ FROM } e_1 \text{ FOR } e_2]]\!]\rho\sigma = \begin{cases} (\textit{restrict } e_1\ (\textit{restrict } e_2\ \rho))[(\alpha,t)/x][(\underline{a},A,(\alpha,t))/y] \\ \quad \text{if } \mathcal{E}[\![[e \text{ FROM } e_1 \text{ FOR } e_2]]\!]\rho\sigma \in LOC \wedge \\ \quad\ \mathcal{E}[\![[e \text{ FROM } e_1 \text{ FOR } e_2]]\!]\rho\sigma = (\alpha,t) \wedge \\ \quad\ \rho[\![\alpha]\!] \in \{\underline{r},\underline{u}\} \wedge \mathcal{A}[\![x \text{ IS } e]\!]\rho\sigma = \varrho \\ (\textit{restrict } e_1\ (\textit{restrict } e_2\ \rho))[(\alpha,t)/x][(\underline{a},A \cup B,(\alpha',t'))/y] \\ \quad \text{if } \mathcal{E}[\![[e \text{ FROM } e_1 \text{ FOR } e_2]]\!]\rho\sigma \in (\underline{a} \times \wp(ADDR) \times LOC) \wedge \\ \quad\ \mathcal{E}[\![[e \text{ FROM } e_1 \text{ FOR } e_2]]\!]\rho\sigma = (\underline{a},B,(\alpha,t)) \wedge \\ \quad\ \rho[\![\alpha]\!] \in \{\underline{r},\underline{u}\} \wedge \rho[\![y]\!] = (\underline{a},B',(\alpha',t')) \wedge B = B' \wedge \\ \quad\ A \cap B = \emptyset \wedge \mathcal{A}[\![x \text{ IS } e]\!](\rho[(\alpha',t')/y])\sigma = \varrho \\ \bot \text{ otherwise} \end{cases}$$

where *abode* $([e \text{ FROM } e_1 \text{ FOR } e_2]) = A \wedge$ *name* $e = y$

provided $\mathcal{E}[\![[e \text{ FROM } e_1 \text{ FOR } e_2]]\!]\rho\sigma \in LOC + (\underline{a} \times \wp(ADDR) \times LOC)$

Slices of an array may also be abbreviated. Again the clause makes use of the functions *restrict*, *name* and *abode* to prevent variables appearing in subscripts altering, to find the identifier associated with the component of an array and to calculate the addresses holding the contents of the slice being abbreviated.

$$\mathcal{A}[\![\varsigma\ x \text{ IS } y]\!]\rho\sigma = \mathcal{A}[\![x \text{ IS } y]\!]\rho\sigma$$

$$\mathcal{A}[\![\varsigma\ x \text{ IS } e_1[e_2]]\!]\rho\sigma = \mathcal{A}[\![x \text{ IS } e_1[e_2]]\!]\rho\sigma$$

$$\mathcal{A}[\![\varsigma\ x \text{ IS } [e \text{ FROM } e_1 \text{ FOR } e_2]]\!]\rho\sigma = \mathcal{A}[\![x \text{ IS } [e \text{ FROM } e_1 \text{ FOR } e_2]]\!]\rho\sigma$$

As with value abbreviations, the inclusion of a specifier within a variable abbreviations is redundant.

$$\mathcal{A}[\![\text{VAL } [f_1]\ldots[f_n]\ \tau\ x \text{ RETYPES } e]\!]\rho\sigma = \begin{cases} (\textit{restrict } e\ \rho)[(\textit{lookup } \sigma'\ (\alpha,(\langle\gamma_1,\ldots,\gamma_n\rangle,\tau)))/x] \\ \qquad \text{if } \mathcal{E}_V[\![e]\!]\rho\sigma = \beta \wedge \beta \in \overline{(\langle\beta_1\ldots\beta_m\rangle,\tau')} \wedge \\ \qquad\ \tau' = \tau \wedge \beta_1 \times\cdots\times \beta_m = \gamma_1 \times\cdots\times\gamma_n \wedge \\ \qquad\ \textit{new } (\langle\beta_1,\ldots,\beta_m\rangle,\tau')\ \rho = (\alpha,t) \wedge \\ \qquad\ \sigma' = \textit{update } \sigma\ (\alpha,t)\ \beta \\ \bot \qquad \text{otherwise} \end{cases}$$

where $\mathcal{E}_V[\![f_1]\!]\rho\sigma = \gamma_1 \wedge\cdots\wedge \mathcal{E}_V[\![f_n]\!]\rho\sigma = \gamma_n$

provided $\mathcal{E}_V[\![f_1]\!]\rho\sigma \in \overline{\text{INT}} \wedge\cdots\wedge \mathcal{E}_V[\![f_n]\!]\rho\sigma \in \overline{\text{INT}} \wedge$
$\mathcal{E}_V[\![e]\!]\rho\sigma \neq \bot_{\mathcal{E}}$

The only expression retyping which is to be modelled is the reshaping of arrays. Producing a suitable clause for the semantic function is made more difficult by the detailed checking which must be carried out to verify that a given retype makes sense. The extra detail and possible ambiguity which allowing unquantified dimensions ([]) would introduce was not considered warranted for the very restricted form of retyping modelled, and so it is insisted that every array dimension contains an integer expression.

$$\mathcal{A}\llbracket [f_1]\ldots[f_n]\,\tau \\ \quad x\ \texttt{RETYPES}\ y\rrbracket\rho\sigma = \begin{cases} \rho[(\alpha,(\langle\gamma_1,\ldots,\gamma_n\rangle,\tau))/x][(\underline{a},A,(\alpha,\langle\beta_1,\ldots,\beta_m\rangle\tau'))/y] \\ \quad \text{if } \rho\llbracket y\rrbracket \in LOC \wedge \rho\llbracket y\rrbracket = (\alpha,(\langle\beta_1,\ldots,\beta_m\rangle,\tau')) \wedge \\ \quad\quad \rho\llbracket\alpha\rrbracket \in \{\underline{r},\underline{u}\} \wedge \tau' = \tau \wedge \beta_1 \times\cdots\times \beta_m = \gamma_1 \times\cdots\times\gamma_n \\ \bot \quad \text{otherwise} \end{cases}$$

where $\mathcal{E}_V\llbracket f_1\rrbracket\rho\sigma = \gamma_1 \wedge\cdots\wedge \mathcal{E}_V\llbracket f_n\rrbracket\rho\sigma = \gamma_n \wedge \textit{abode}\ y = A$

provided $\mathcal{E}_V\llbracket f_1\rrbracket\rho\sigma \in \overline{\texttt{INT}} \wedge\cdots\wedge \mathcal{E}_V\llbracket f_n\rrbracket\rho\sigma \in \overline{\texttt{INT}}$

$$\mathcal{A}\llbracket [f_1]\ldots[f_n]\,\tau \\ \quad x\ \texttt{RETYPES}\ e_1[e_2]\rrbracket\rho\sigma = \begin{cases} \varrho[(\alpha,(\langle\gamma_1,\ldots,\gamma_n\rangle,\tau))/x] \\ \quad \text{if } \varrho\llbracket x\rrbracket = (\alpha,(\langle\beta_1,\ldots,\beta_m\rangle,\tau')) \wedge \\ \quad\quad \tau' = \tau \wedge \beta_1 \times\cdots\times \beta_m = \gamma_1 \times\cdots\times\gamma_n \\ \bot \quad \text{otherwise} \end{cases}$$

where $\mathcal{E}_V\llbracket f_1\rrbracket\rho\sigma = \gamma_1 \wedge\cdots\wedge \mathcal{E}_V\llbracket f_n\rrbracket\rho\sigma = \gamma_n \wedge$
$\mathcal{A}\llbracket x\ \texttt{IS}\ e_1[e_2]\rrbracket\rho\sigma = \varrho$

provided $\mathcal{E}_V\llbracket f_1\rrbracket\rho\sigma \in \overline{\texttt{INT}} \wedge\cdots\wedge \mathcal{E}_V\llbracket f_n\rrbracket\rho\sigma \in \overline{\texttt{INT}} \wedge$
$\mathcal{A}\llbracket x\ \texttt{IS}\ e_1[e_2]\rrbracket\rho\sigma \neq \bot$

$$\mathcal{A}\llbracket [f_1]\ldots[f_n]\,\tau\ x\ \texttt{RETYPES} \\ \quad [e\ \texttt{FROM}\ e_1\ \texttt{FOR}\ e_2]\rrbracket\rho\sigma = \begin{cases} \varrho[(\alpha,(\langle\gamma_1\ldots\gamma_m\rangle,\tau))/x] \\ \quad \text{if } \varrho\llbracket x\rrbracket = (\alpha,(\langle\beta_1\ldots\beta_m\rangle,\tau')) \wedge \\ \quad\quad \tau' = \tau \wedge \beta_1 \times\cdots\times \beta_m = \gamma_1 \times\cdots\times\gamma_n \\ \bot \quad \text{otherwise} \end{cases}$$

where $\mathcal{E}_V\llbracket f_1\rrbracket\rho\sigma = \gamma_1 \wedge\cdots\wedge \mathcal{E}_V\llbracket f_n\rrbracket\rho\sigma = \gamma_n \wedge$
$\mathcal{A}\llbracket x\ \texttt{IS}\ [e\ \texttt{FROM}\ e_1\ \texttt{FOR}\ e_2]\rrbracket\rho\sigma = \varrho$

provided $\mathcal{E}_V\llbracket f_1\rrbracket\rho\sigma \in \overline{\texttt{INT}} \wedge\cdots\wedge \mathcal{E}_V\llbracket f_n\rrbracket\rho\sigma \in \overline{\texttt{INT}} \wedge$
$\mathcal{A}\llbracket x\ \texttt{IS}\ [e\ \texttt{FROM}\ e_1\ \texttt{FOR}\ e_2]\rrbracket\rho\sigma \neq \bot$

As previously explained, the implementation dependent nature of RETYPES means that a generic semantic definition cannot be given. However, for one particular use of the constructor (reshaping of arrays), a semantic definition can be useful. Because of the similarity between retyping and abbreviation, the semantic definition given for retyping draws heavily on that of abbreviation. As with expression retyping, it is insisted that every array dimension contains an integer expression.

(5) FUNCTION AND PROCEDURE CALLS Having given the clauses for procedure and function declarations, the form of the domain *NP* is known. *NP* consists of two distinct parts, one to deal with procedures and the other to deal with functions:

$$NP = NPROC + NFUNCT$$

$$\begin{aligned} NPROC &= (VSTATUS \times LSTATUS) \longrightarrow Exp^* \longrightarrow S \longrightarrow Q \\ NFUNCT &= (VSTATUS \times LSTATUS) \longrightarrow Exp^* \longrightarrow S \longrightarrow (\mathcal{V}^{\mathcal{A}})_\perp \end{aligned}$$

It is beneficial to make a clear distinction between the two components of the domain since it allows one to determine in which way to act immediately given a particular identifier.

With the semantic function $\mathcal{D}$ having been defined, it is now possible to complete the definitions of $\mathcal{C}$ and $\mathcal{E}$ by giving the clauses for procedure and function calls respectively.

$$\mathcal{C}[\![q(T)]\!]\rho\sigma = \rho[\![q]\!]\langle \rho_V, \rho_L \rangle [\![T]\!]\sigma \quad \text{provided } \rho[\![q]\!] \in NPROC$$

Having carefully designed the information to be stored in the environment when dealing with procedure declarations, the semantic clause to deal with procedure calls does not provide any difficulties. All that is necessary is to supply the relevant arguments (second and third environment components from the point of call, actual parameters of the call and current store).

$$\mathcal{E}[\![q(T)]\!]\rho\sigma = \rho[\![q]\!]\langle \rho_V, \rho_L \rangle [\![T]\!]\sigma \quad \text{provided } \rho[\![q]\!] \in NFUNCT$$

As expected, the treatment of function calls is very similar to that of procedure calls; the only difference is the part of the domain NP in which the information associated with the identifier lies. There is no difference in the treatment of function calls where the function body consists of a VALOF command and those where the function returns an expression list; the difference in behaviour is dealt with by the clauses of $\mathcal{D}$.

This completes the definition of the main semantic functions. A denotational semantics has now been given for occam (as far as the model allows). The semantics gives an interesting illustration of how the domain $\mathcal{Q}$ defined in the first part of the paper can be used to model concurrent languages, by means of reference to a specific example.

Throughout the construction of the model, simplifications which could be made due either to the structure of occam or to decisions not to attempt to model particular aspects of the language were not made. This, along with the power of the failures/divergences model on which it was based, means that the 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.

It is also possible to refine the model used in order to provide finer distinctions between processes. Within the semantic function $\mathcal{E}$, for instance, no attempt has been made to differentiate between detectable and non-detectable errors. If such distinctions are desirable, the form of the domain of expressible values, E, can be changed (along with relevant changes to a small number of clauses of the semantic functions $\mathcal{E}$ and $\mathcal{D}$) without necessitating major alterations to the other clauses.

4 Conclusions

4.1 The structure of the semantics

In the course of constructing a mathematical model and calculating the denotational semantics, many decisions were made. It is useful to examine some of these, incorporating a brief study of how certain of the restrictions imposed might have been relaxed.

Most of the major decisions arose during the construction of the mathematical model; once the model is fixed, the semantics of most constructs are determined by their rôles in the language. Several factors 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 language; it should be as simple, elegant and understandable as possible.

It is possible to do without 'states' in the model. Variables can be replaced by suitable processes which run in parallel with the 'main' process for the duration of their scope; reading from and writing to variables is then equivalent to communicating with the relevant process. Because one can use a 'purely parallel' semantic model, the task of constructing a suitable model is greatly simplified. The definitions of the semantic functions, however, become more complex and the balance does not seem right. Many of the advantages gained from similar experience with other languages are lost; in particular, the semantics of a 'purely sequential' occam fragment no longer bears much resemblance to a relation on states. Nevertheless, on programs without free variables, this approach would lead to a semantics congruent to that given in this paper.

The model $\mathcal{Q}$ was devised with occam processes in mind. One of the restrictions of occam is that it is not possible for a process to include 'selective inputs' (for example a process cannot be prepared to input any even integer on channel χ but refuse all other inputs over that channel). If such behaviour were permitted then it would not be sufficient to include solely channel names in the refusal sets of a process. A semantics for occam which were capable of modelling processes including 'selective inputs' (and which was congruent to that given) could be constructed using $(\alpha P)^* \times \mathcal{P}(\alpha P \cup \{\surd\})$ rather than $(\alpha P)^* \times \mathcal{P}(CHAN \cup \{\surd\})$ to represent the failures of P.

The techniques employed in this paper to incorporate the model $\mathcal{N}$ into $\mathcal{Q}$ would work for other alternatives to the *failures* model. A number of examples of this are currently under examination, including models for Timed CSP.

Once the main model was decided, the majority of the semantic definitions were fixed. Some decisions, however, were still made. An example of such a decision was the way in which communications over ports and timers were modelled.

4.2 Applications

There has already been much successful work based around the clean mathematical theory underlying occam. The language was designed with the need for a clean semantics in mind, and all of this work has taken advantage, directly or indirectly, of the semantics. The threads of existing formal work include the following:

- The denotational semantics of proto-occam [26] were used to derive a congruent algebraic semantics [16], which subsequently became the basis of the occam Transformation System [11]. This in turn was used in a variety of ways, most notably in the design of the T800 Floating-Point Unit [18].

- The relationships of the semantics of occam and CSP have been exploited several times, including investigations of deadlock [28] and fault-tolerance [5]. The existence of a model-checking program for CSP [10] is thus exploitable rather directly on occam systems.
- occam has been used by several researchers, for example [24, 9], as the input to systems which compile programs to silicon.
- Major use has been made within INMOS of occam as a language for describing and designing VLSI. A variety of formal techniques have been used in connection with this work by INMOS and by Formal Systems (Europe), as described, for example, in [27] and [21].
- One major component of the previous item, and entirely attributable to the clean semantics of occam, was the development of "ghosting" tools which permit the automatic transformation of an occam program to one which computes symbolic representations of the values in chosen types.

Work is currently taking place on a number of topics, including:

- Translation between CSP and occam. It will be possible to discover subsets of the two notations where there is a close correspondence, allowing translation either way, and both at the Timed and Untimed levels. Transformational techniques should help in getting the source language into the correct form for this automated translation.
- It is hoped to exploit the above in a methodology for developing real-time occam processes via Timed CSP.
- Wood [31] is developing a refinement calculus similar to that of Morgan's [23] for languages akin to occam, using the semantic models presented in this paper.

Acknowledgements

Thanks must be given to Bryan Scattergood whose advice helped clear up several misunderstandings concerning technicalities of the language. Thanks are also due to INMOS Limited for the provision of the syntax from which the appendix was adapted. This work was carried out with support from INMOS Limited, ESPRIT Project 2701: PUMA, the US Office of Naval Research and SERC.

A Syntactic summary

This appendix contains a syntactic summary of the language considered in this paper. It is closely related to the occam 2 language summary of [20], but as previously mentioned is augmented to allow parallel declarations in the language. The syntactic objects are listed in alphabetical order.

Within the syntactic summary, *op* appears. This is the set of operators on expressions which are available within the language; differences between implementations mean that its form cannot be described more concretely.

Following each syntactic object, the means by which the object is referred to during the course of the paper is included within parenthesis. Throughout the paper the same symbol is used to refer to several different classes of object; more precise distinctions than those present were not deemed necessary and their omission led to more consise descriptions without causing the introduction of any ambiguity.

actual (Y) =
element
| *expression*

alternation (P) =
ALT ($alternative_1, \ldots, alternative_n$)
| ALT *identifier* =
expression$_1$ FOR *expression*$_2$ *alternative*
| PRI ALT ($alternative_1, \ldots, alternative_n$)
| PRI ALT *identifier* =
expression$_1$ FOR *expression*$_2$ *alternative*

alternative (A) =
input process
| *boolean* & *input process*
| *boolean* & SKIP *process*
| *alternation*
| *channel* ? CASE ($variant_1, \ldots, variant_n$)
| *boolean* &
channel ? CASE ($variant_1, \ldots, variant_n$)
| *declaration* : *alternative*
| *specification* : *alternative*

boolean (b) =
expression

byte (e or f) =
'character'

channel (c) =
element

choice (C) =
boolean process
| *conditional*
| *declaration* : *choice*
| *specification* : *choice*

conditional (*P*) =
IF (*choice*$_1$, ..., *choice*$_n$)
| IF *identifier* =
expression$_1$ FOR *expression*$_2$ *choice*

declaration (Δ) =
[*expression*$_1$]...[*expression*$_n$] *primitive.type*
identifier$_1$, ..., *identifier*$_m$
| [*expression*$_1$]...[*expression*$_n$] CHAN OF
[*expression*$_1$]...[*expression*$_{n'}$] *primitive.type*
identifier$_1$, ..., *identifier*$_m$
| [*expression*$_1$]...[*expression*$_n$] CHAN OF
primitive.type$_1$ :: [ ][*expression*$_1$]
...[*expression*$_{n'}$]
primitive.type$_2$ *identifier*$_1$, ..., *identifier*$_m$
| [*expression*$_1$]...[*expression*$_n$] CHAN OF
ANY *identifier*$_1$, ..., *identifier*$_m$
| [*expression*$_1$]...[*expression*$_n$] CHAN OF
identifier identifier$_1$, ..., *identifier*$_m$
| [*expression*$_1$]...[*expression*$_n$] TIMER
identifier$_1$, ..., *identifier*$_m$
| [*expression*$_1$]...[*expression*$_n$] PORT OF
[*expression*$_1$]...[*expression*$_{n'}$] *primitive.type*
identifier$_1$, ..., *identifier*$_m$
| PROTOCOL *identifier* IS
protocol$_1$; ...; *protocol*$_n$
| PROTOCOL *identifier* IS
CASE (*tag*$_1$; *protocol*$_1$), ..., (*tag*$_n$; *protocol*$_n$)
| PROC *procedure.name*(*formal*$_1$,
..., *formal*$_n$) = *process*
| *primitive.type*$_1$, ... *primitive.type*$_n$ FUNCTION
function.name(*formal*$_1$, ..., *formal*$_n$) = *valof*
| *primitive.type*$_1$, ... *primitive.type*$_n$ FUNCTION
function.name(*formal*$_1$, ..., *formal*$_n$) IS
expression$_1$, ..., *expression*$_m$

digit (*0* or ...or *9*) =
0 | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9*

element (*e* or *f*) =
identifier
| *element* [*expression*]
| [*element* FROM *expression*$_1$ FOR *expression*$_2$]

exponent (*e* or *f*) =
+*digit*$_1$ ... *digit*$_n$
| −*digit*$_1$ ... *digit*$_n$

expression (*e* or *f*) =
op expression
| *expression*$_1$ AND *expression*$_2$
| *expression*$_1$ OR *expression*$_2$
| *expression*$_1$ *op expression*$_2$
| MOSTPOS *primitive.type*
| MOSTNEG *primitive.type*
| *primitive.type argument*
| *primitive.type* ROUND *argument*
| *primitive.type* TRUNC *argument*
| *argument*

formal (Ψ) =
specifier identifier$_1$, ..., *identifier*$_n$
| VAL *specifier identifier*$_1$, ..., *identifier*$_n$

function.name (*g*) =
identifier

hex.digit (*0* or ...or *F*) =
digit | *A* | *B* | *C* | *D* | *E* | *F*

input (*P*) =
channel ? *input.expression*$_1$;
...; *input.expression*$_n$
| *channel* ? CASE *tag*
| *channel* ? CASE *tag* ; *input.expression*$_1$;
...; *input.expression*$_n$
| *timer* ? *variable*
| *timer* ? AFTER *expression*
| *port* ? *variable*

input.expression (*ie*) =
variable
| *variable* :: *variable*

integer (*e* or *f*) =
digit$_1$ ... *digit*$_n$
| #*hex.digit*$_1$ ... *hex.digit*$_n$

loop (*P*) =
WHILE *boolean process*

operand (*e* or *f*) =
TRUE
| FALSE
| *integer*
| *byte*
| *integer*(*primitive.type*)
| *byte*(*primitive.type*)
| *real*(*primitive.type*)
| *string*
| *elcment*
| *table*
| (*expression*)
| (*valof*)
| *function.name*(*expression*$_1$, ..., *expression*$_n$)

option (*O*) =
(*expression*$_1$, ..., *expression*$_n$) *process*
| ELSE *process*
| *declaration* : *option*
| *specification* : *option*

output.expression (*oe*) =
expression
| *expression*$_1$ :: *expression*$_2$

parallel (*P*) =
PAR (*process*$_1$, ..., *process*$_n$)
| PAR *identifier* =
expression$_1$ FOR *expression*$_2$ *process*
| PAR (*parallel.declaration*$_1$: *process*$_1$,
..., *parallel.declaration*$_n$: *process*$_n$)
| PAR *identifier* =
expression$_1$ FOR *expression*$_2$
parallel.declaration : *process*
| PRI PAR (*process*$_1$, ..., *process*$_n$)
| PRI PAR *identifier* =
expression$_1$ FOR *expression*$_2$ *process*
| PRI PAR (*parallel.declaration*$_1$: *process*$_1$,
..., *parallel.declaration*$_n$: *process*$_n$)
| PRI PAR *identifier* =
expression$_1$ FOR *expression*$_2$
parallel.declaration : *process*
| *placed.parallel*

parallel.declaration (*U*) =
USING (OWNCHAN *channel*$_1$, ..., *channel*$_n$,
INCHAN *channel*$_1$, ..., *channel*$_n$,
OUTCHAN *channel*$_1$, ..., *channel*$_n$,
VAR *variable*$_1$, ..., *variable*$_n$)

placed.parallel (P) =
PLACED PAR ($placed.parallel_1$, ..., $placed.parallel_n$)
| PLACED PAR *identifier* =
$expression_1$ FOR $expression_2$ *placed.parallel*
| PROCESSOR *expression process*
| PROCESSOR *expression*
parallel.declaration : *process*

port (c) = *element*

primitive.type (τ) =
BOOL
| BYTE
| INT
| INT16
| INT32
| INT64
| REAL32
| REAL64

procedure.name (q) = *identifier*

process (P) =
SKIP
| STOP
| PLACE *identifier* AT *expression* : *process*
| *variable* := *expression*
| $variable_1, \ldots, variable_n$:=
$expression_1, \ldots, expression_m$
| *channel* ! $output.expression_1$;
..., $output.expression_n$
| *channel* ! *tag*
| *channel* ! *tag* ; $output.expression_1$;
..., $output.expression_n$
| *port* ! *expression*
| *channel* ? CASE ($variant_1, \ldots, variant_n$)
| *input*
| *sequence*
| *conditional*
| *selection*
| *loop*
| *parallel*
| *alternation*
| *procedure.name*($actual_1, \ldots, actual_n$)
| *declaration* : *process*
| *specification* : *process*

protocol	(p)	=	$[expression_1] \ldots [expression_n]\ primitive.type$
		|	$primitive.type_1 :: [\,][expression_1]$
			$\ldots [expression_n]\ primitive.type_2$
real	$(e \text{ or } f)$	=	$digit_1 \ldots digit_m.digit_1 \ldots digit_n$
		|	$digit_1 \ldots digit_m.digit_1 \ldots digit_n$ E *exponent*
selection	(P)	=	CASE *expression* $(option_1, \ldots, option_n)$
sequence	(P)	=	SEQ $(process_1, \ldots, process_n)$
		|	SEQ *identifier* =
			$expression_1$ FOR $expression_2$ *process*
specification	(Θ)	=	*specifier identifier* IS *element*
		|	*identifier* IS *element*
		|	VAL *specifier identifier* IS *expression*
		|	VAL *identifier* IS *expression*
		|	*type identifier* RETYPES *element*
		|	VAL *type identifier* RETYPES *expression*
specifier	(ς)	=	*primitive.type*
		|	[ ] *specifier*
		|	$[e]$ *specifier*
string	$(e \text{ or } f)$	=	$[byte_1, \ldots, byte_n]$
table	$(e \text{ or } f)$	=	$[expression_1, \ldots, expression_n]$
		|	*table*[*expression*]
		|	[*table* FROM $expression_1$ FOR $expression_2$]
tag	(tag)	=	*identifier*
timer	(c)	=	*element*
type	$(e_1 \ldots e_n\ \tau)$	=	*primitive.type*
		|	[*expression*] *type*
valof	$(e \text{ or } f)$	=	VALOF *process*
			RESULT $expression_1, \ldots, expression_n$
		|	*declaration* : *valof*
		|	*specification* : *valof*
variable	$(e \text{ or } f)$	=	*element*

variant (T) =
| *tag process*
| *tag* ; *input.expression*$_1$;
... ; *input.expression*$_n$ *process*
| *declaration* : *variant*
| *specification* : *variant*

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] Barrett, G., *Formal methods applied to a floating point number system*, Tech. Report PRG-58, Oxford University Programming Research Group, 1987.

[4] Barrett, G., *The semantics and implementation of* occam, D.Phil. thesis, Oxford University, 1989.

[5] Brock, N.A., and Jackson, D.M., *Formal Verification of a Fault Tolerant Computer.*

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

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

[8] Brookes, S.D., and Roscoe, A.W., *An improved failures model for communicating processes*, Carnegie-Mellon Tech. Report 1984.

[9] Brunvand, E.L., and Starkey, M., *An integrated environment for the design and simulation of self-timed systems*, in VLSI 91, 1991, pp4a.2.1-4a.2.10.

[10] Formal Systems (Europe) Ltd., *Failures Divergence Refinement: Users Manual and Tutorial*, 1992.

[11] Goldsmith, M.H, *The Oxford Occam Transformation System (Version 0.1) (Draft user documentation)*, PRG, Oxford.

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

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

[14] Hoare, C.A.R., *Communicating sequential processes*, Prentice-Hall International, 1985.

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

[16] Hoare, C.A.R., and Roscoe, A.W., *The laws of* occam *programming*, Tech. Report PRG-53, Oxford University Programming Research Group, 1986.

[17] *IEEE Standard for Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985, New York, 1985.

[18] INMOS Ltd., *Communicating Process Architectures*, Prentice-Hall International, 1988.

[19] INMOS Ltd., *The* occam *Programming Manual*, Prentice-Hall International, 1984.

[20] INMOS Ltd., *The* occam 2 *Reference Manual*, Prentice-Hall International, 1988.

[21] May, D., Barrett, G., and Shepherd, D., *Designing chips that work*, in Mechanized Reasoning and Hardware Design, Prentice-Hall International, 1992.

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

[23] Morgan, C.C., *Programming from Specifications*, Prentice-Hall International, 1990.

[24] Page, I., and Luk, W., *Compiling* occam *into Field Programmable Gate Arrays*, FPGAs, Proceedings of International Workshop on Field Programmable Logic and Applications, Oxford, 1991, pp271-283.

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

[26] Roscoe, A.W., *Denotational semantics for* occam, Springer-Verlag LNCS 197, 1985.

[27] Roscoe, A.W., occam *in the specification and verification of microprocessors*, in Mechanized Reasoning and Hardware Design, Prentice-Hall International, 1992.

[28] Roscoe, A.W., *Routing messages through networks: an exercise in deadlock avoidance*, Proceedings of OUGTM 7, Grenoble, 1987.

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

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

[31] Wood, K.R., *A Pragmatic Basis for the Formal Development of Distributed Systems*, Proc. of 7th International Workshop on Software Specification and Design, IEEE Computer Society Press (to appear), 1993.

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