PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT. © ICOT, 1984

# PROGRAMS AS EXECUTABLE PREDICATES

C.A.R. Hoare and A.W. Roscoe

Oxford University Computing Laboratory,
Programming Research Group,
8-11 Keble Road, Oxford OX2 8QD,
United Kingdom.

## ABSTRACT

A computer program is identified with the strongest predicate describing every relevant observation that can be made of the behaviour of a computer executing that program. A programming language is a subset of logical and mathematical notations, which is so restricted that products described in the language can be automatically implemented on a computer. The notations enjoy a number of elegant algebraic properties, which can be used for optimising program efficiency. This philosophy has been taken by INMOS Ltd. as a basis for the design of the occam programming language, a language which is implemented on networks of communicating processors (transputers), and so may contribute to the objectives of the fifth generation computer project. For simplicity we describe the types of observation required for a version of occam without timing or priority, and give a detailed semantics for a subset of occam.

## 1 INTRODUCTION

It is the aim of the natural scientist to discover mathematical theories, formally expressed as predicates describing the relevant observations that can be made of some physical system. A physical system is fully defined by the strongest predicate which describes it. Such predicates contain free variables, standing for values determined by observation, for example "a" for acceleration, "v" for velocity, "t" for time, etc.

The aim of the engineer is complementary to that of the scientist. He starts with a specification, formally expressible as a predicate describing the desired observable behaviour of a system or product not yet in existence. Then, with a limited set of tools and materials, within a limited timescale and budget, he must design and construct a product which meets that specification. The product is fully defined by the strongest specification which it meets.

For example, an electronic amplifier may be required to amplify its input voltage by a factor of ten. A condition of its correct working is that the input voltage must be held in the range 0 to 1 volt. Furthermore, a margin of error of up to one volt is allowed on the output. This informal specification may be formalised as a predicate, with free variables

$v_i$ standing for the ith input voltage
$v'_i$ standing for the ith output voltage.

Then the specification is

$$\forall i.(i \leqslant j \Rightarrow 0 \leqslant v_i \leqslant 1) \Rightarrow |v'_j - 10 \times v_j| \leqslant 1.$$

Table 1: Two amplifiers

|   | $v$ | $v'$ |
|---|---|---|
| 1 | .5 | 5 |
| 2 | .4 | 5 |
| 3 | .5 | 4 |
| 4 | .3 | 5 |
| 5 | .6 | 6 |
| 6 | .7 | 7 |

(a)

|   | $v$ | $v'$ |
|---|---|---|
| 1 | .5 | 5 |
| 2 | .4 | 5 |
| 3 | .5 | 4 |
| 4 | 1.3 | 13 |
| 5 | .6 | 6 |
| 6 | .7 | 997 |

(b)

Table 1 (a) and (b) show the first six observations made of two different amplifiers. The first observation made of each amplifier shows it working with perfect accuracy at the mid point of its range. The second observation is only just within the margin of tolerance. On the third observation the amplifier reveals its "non-determinism": it does not always give the same output voltage for the same input voltage. On the fourth observation something goes wrong. In the case of 4(a) it is the amplifier that has gone wrong, because the five volt output is outside the permitted margin of error. Even if every subsequent observation is satisfactory, this product has not met its specification, and should be returned to its maker. In the case of 5(b), it is the observer who is at fault in supplying an excessive input voltage of 1.3. As a result, the amplifier breaks, and its subsequent behaviour is entirely unconstrained: no matter what it does, it continues to meet its original specification. As a result, on the sixth observation, it is the observer who returns to meet his Maker.

The serious point of this example is to illustrate the usefulness of material implication ($\Rightarrow$) in a specification. The consequent of the implication describes the desired relationship between the inputs and the outputs of the system. The antecedent describes the assumptions which must be satisfied by the inputs of the system in order for it to continue working. If the assumptions are falsified, the product may break, and its subsequent (but not its previous) behaviour may be wholly arbitrary. Even if it seems for a while to work, it is completely worthless, unreliable and even dangerous.

A computer programmer is an engineer whose main materials are the notations and structures of his programming language. A program is a detailed specification of the behaviour of a computer executing that program. Consequently, a program can be abstractly identified with a predicate describing all relevant observations which may be made of this behaviour. This identification assigns a meaning to the program (Floyd 1967), and a semantics to the programming language in which it is expressed.

These philosophical remarks lead to the main thesis of this paper, namely that programs are executable predicates. The qualification "executable" is important: consider, for example, any predicate which is wholly unsatisfiable (e.g. the predicate false). This cannot correspond to a program. If it did, the behaviour of a computer executing that program would be wholly unobservable! Consequently, every observation of that behaviour would satisfy every specification! A product which satisfies every need is known as a miracle. Since such a product is in principle unobservable, philosophical considerations lead us to suppose that it does not exist. Certainly any notation in which such a miracle could be expressed would not be an executable programming language. There are also obvious practical reasons for ensuring that all predicates expressible as programs are in some sense computable. Later, we will give a precise definition of the notion of "executable predicate" for occam.

The design of a programming notation requires a preliminary selection of what are the relevant observable phenomena, and a choice of free variables to denote them. A meaning must then be given to the primitive components of the language, and to the operators which compose programs from smaller subprograms. Ideally, these operators should have pleasant algebraic properties, which permit proof of the identity of two programs whenever they are indistinguishable by observation. The achievement of these ideals is far from easy: so the language introduced in the next section for illustrative purposes has been kept very simple.

## 2 A SIMPLE PROGRAMMING LANGUAGE

The first and simplest predicate which is expressible in our simple programming language is the predicate "true". If this is the strongest specification of a product, then there is no constraint whatever on the behaviour or misbehaviour of the product. The only customer who is certain to be satisfied with this product is one who would be satisfied by anything. Thus the program "true" is the most useless of all products, just as a tautology is the most useless of scientific theories.

Now the most useless of computer programs is one that immediately goes into an infinite loop or recursion. Such a program is clearly broken or unstable, and can satisfy only the most undemanding customer. Thus we identify the infinitely looping program with the predicate "true". This may be a controversial decision; but in practice the ascription of a meaning to a divergent process is arbitrary, because no programmer will ever deliberately want to write a program which runs any risk of looping forever. Since the fact that a program is looping infinitely is not finitely observable, no observation of such a program can be completed in a finite time. Thus no observer can exclude any observation of such a program. This is the reason for its identification with "true".

### 2.1 Nondeterminism

The first and simplest operator of our programming language is disjunction. If P and Q are programs, the program $(P \vee Q)$ behaves either like P or like Q. There is no way of controlling or predicting the choice between P and Q; the choice is arbitrary or nondeterministic. All that is known is that each observation of $(P \vee Q)$ must be an observation of P or of Q or of both.

The algebraic properties of disjunction are very familiar: it is idempotent, symmetric, associative, etc. Furthermore, it is distributive (through disjunction)

$$P \vee (Q \vee R) \quad = \quad (P \vee Q) \vee (P \vee R)$$

and strict in the sense that

$$P \vee \text{true} \quad = \quad \text{true} \vee P \quad = \text{true}.$$

This means that if either P or Q may break then so may $(P \vee Q)$. To an engineer, a product that may break is as bad as one that does, because you can never rely on it.

### 2.2 Assignment

Let x be a list of distinct variables, and let e be a list of the same number of expressions, and let P(x) be a program describing the behaviour of a process as a function of the initial values of x. Then we define

$$(e \rightarrowtail x \longrightarrow P(x)) \quad \hat{=} \quad P(e)$$

i.e., the result of simultaneously substituting each variable in the list x by the corresponding expression in the list e, making sure that free variables of e remain free after substitution. We assume for simplicity that all expressions of e are defined for all values of the variables they contain. e may thus be regarded as a total function of these variables.

The predicate $(e \succ\!\!- x \longrightarrow P(x))$ describes the behaviour of a process which first simultaneously assigns the values of e to the variables x and then behaves like P(x). The initial assignment is an internal action, and is therefore wholly unobservable. In more conventional programming notation this would be written

$$x := e;\ \ P(x).$$

The reason for the infix notation $\succ\!\!- x \longrightarrow$ is that this permits elegant expression of algebraic properties. Considered as an infix operator between arbitrary formulae, $\succ\!\!- x \longrightarrow$ is associative with unit x:

$$(x \succ\!\!- x \longrightarrow P) = P$$

$$(e \succ\!\!- x \longrightarrow x) = e$$

$$(e \succ\!\!- x \longrightarrow (f \succ\!\!- x \longrightarrow P)) = ((e \succ\!\!- x \longrightarrow f) \succ\!\!- x \longrightarrow P).$$

Furthermore, if y is a list of variables, none of which occur in the list x,

$$(e \succ\!\!- x \longrightarrow P) = (e,y \succ\!\!- x,y \longrightarrow P).$$

These laws are sufficient to reduce nested substitutions to a single simultaneous substitution for a list containing all the variables involved. Such reductions lead to greater symmetry and abstractness in the appearance of predicates.

Assignment also distributes over $\vee$:

$$(e \succ\!\!- x \longrightarrow (P \vee Q)) = (e \succ\!\!- x \longrightarrow P) \vee (e \succ\!\!- x \longrightarrow Q).$$

2.3 Conditional

Let b be a propositional formula, i.e., a single expression which for all values of its free variables yields either "true" or "false". Let P and Q be programs. Define

$$P \triangleleft b \triangleright Q \;=\; ((b \wedge P) \vee (\neg b \wedge Q)).$$

This is a process that behaves like P if b is initially true and like Q if b is initially false. The conventional programming notation for a conditional is

<u>if</u> b <u>then</u> P <u>else</u> Q.

Again, we make use of an infix notation ($\triangleleft b \triangleright$) because this permits elegant expression of algebraic properties such as idempotence, associativity and distributivity, e.g.

a) $(P \triangleleft b \triangleright P) = P$

b) $P \triangleleft b \triangleright (Q \triangleleft b \triangleright R) = (P \triangleleft b \triangleright Q) \triangleleft b \triangleright R$
$\qquad = P \triangleleft b \triangleright R$

c) $P \triangleleft b \triangleright (Q \vee R) = (P \triangleleft b \triangleright Q) \vee (P \triangleleft b \triangleright R)$

d) $(P \vee Q) \triangleleft b \triangleright R = (P \triangleleft b \triangleright R) \vee (Q \triangleleft b \triangleright R)$

e) $P \triangleleft b \triangleright (Q \triangleleft c \triangleright R) = (P \triangleleft b \triangleright Q) \triangleleft c \triangleright (P \triangleleft b \triangleright R)$

f) $(P \triangleleft b \triangleright Q) \triangleleft c \triangleright R = (P \triangleleft c \triangleright R) \triangleleft b \triangleright (Q \triangleleft c \triangleright R)$

g) $P \triangleleft b \triangleleft c \triangleright d \triangleright Q = (P \triangleleft b \triangleright Q) \triangleleft c \triangleright (P \triangleleft d \triangleright Q)$

h) $P \triangleleft \text{true} \triangleright Q = P$

i) $P \triangleleft \text{false} \triangleright Q = Q$

j) $\text{true} \triangleleft b \triangleright \text{false} = b$ .

These laws are simple tautologies. In fact, as is shown in (Hoare 1985), laws (a,b,e,...,j) provide a complete axiomatisation of the propositional calculus. This is because every propositional formula may be written in terms of the ternary operator $\triangleleft\ \triangleright$ and the logical constants "true" and "false", and because these laws allow us to reduce each formula to a normal form. If conditional expressions are similarly defined, they interact with assignment according to the law

$$(e \succ\!\!- x \longrightarrow P) \triangleleft b \triangleright (f \succ\!\!- x \longrightarrow P)$$
$$= ((e \triangleleft b \triangleright f) \succ\!\!- x \longrightarrow P).$$

2.4 Recursion

Let X be a predicate variable, and let F(X) be a formula constructed from X using any number of nested assignments, conditionals and disjunctions, then we introduce the notation

$$\mu X.F(X)$$

to describe the behaviour of a program which behaves like F(X), and whenever it encounters X, continues by behaving like $\mu X.F(X)$ again. Its meaning may be explained as a solution for X in the equation

$$X = F(X).$$

If there are several solutions, $\mu X.F(X)$ denotes the one which is the nondeterministic union of them all. Such solutions exist because all the operators in F are continuous in the sense that if $P_0, P_1, \ldots, P_n, \ldots$ are a sequence of programs such that we have $\forall i.(P_{i+1} \Rightarrow P_i)$, then

$$G(\forall i.P_i) = \forall i.G(P_i)$$

for all such operators G. We can thus define

$$\mu X.F(X) \triangleq \forall n \geqslant 0.\ F^n(\text{true})$$
$$\text{where } F^0(X) = X$$
$$\text{and } F^{n+1}(X) = F(F^n(X)) \quad \text{for each } n \geqslant 0.$$

Recursion is itself a continuous operator, so the use of nested recursions such as

$$\mu X.F(\mu Y.G(X,Y))$$

is permissible. Continuity is of great importance if we are to define processes recursively. If $\neg$ (negation) were in our programming language, the notation $\mu X.\neg X$ would be meaningless. Throughout the rest of this paper we will take care to ensure that all the operators we define are continuous.

### 2.5 Example

Let Y be a predicate variable. Let x,y, q,r be variables taking natural number values. Then

$$y > 0 \Rightarrow ((x \div y), (x \underline{\text{mod}}\, y) \succ\!\!- q, r \longrightarrow Y)$$

describes the behaviour of a process which first assigns to q the quotient of x and y, and to r the remainder of the division, and then behaves like Y. However, if the value of y is zero, then the process can do anything at all. (We assume that the expressions $x \div y$ and $x\, \underline{\text{mod}}\, y$ are still defined, to comply with our earlier assumption. However the above program does not have to compute these values.)

The following is a different way of writing the same predicate

$$(0, x \succ\!\!- q, r \longrightarrow Z), \quad \text{where}$$

$$Z = {}_{\mu}X.((q+1, r-y \succ\!\!- q, r \longrightarrow X) \triangleleft r \geqslant y \triangleright Y).$$

This method is very suggestive of a program that computes the quotient and remainder by the method of successive subtraction. In more conventional notations, the program would be written

```
q := 0;  r := x;
while r ⩾ y do
  begin  q := q + 1;  r := r - y  end;
Y
```

This is an example of the insight required to understand how programs can be regarded as executable predicates.

## 3 OCCAM

Occam is a language designed for the description of processes which communicate on named channels with their environment while they are running. We can therefore record and observe at any time the sequence of communications in which the process has engaged. Let us use the name "tr" to stand for this observation. "tr" is a sequence which for each process starts empty. Every time the process communicates some message m on a channel with name c, the pair (m.c) is appended to the end of "tr", giving $tr^\frown\langle m.c\rangle$.

Occam is also used for the description of sequential processes, which operate on the initial values of their variables x,y,..., and on termination pass on the final values. We will denote the final values by adding a ✓ to the variable name:

$$x^{\checkmark},\ y^{\checkmark},\ \ldots\ldots\ .$$

The undecorated variable will stand for the initial value. We assume that both initial and final values are observable.

We also assume that we can observe if a process is terminated, or if it is held up waiting for an external communication. If it is neither then an observer will wait until it comes into one of these states (i.e., finishes its internal computation). We therefore introduce a variable "st" (status) which takes as values "waiting" or "terminated". There is a third possibility, however. A process might never reach one of these two states however long its observer waited, because it was looping and performing an unbroken infinite sequence of internal actions. We say that such a process is diverging, and introduce a third value, "divergent", which "st" may take.

Finally, when the process is waiting, we assume we can observe on which channels the environment of the process is attempting (unsuccessfully) to communicate with it. This set of channels is known as a refusal of the process, and is denoted by the variable "ref".

With each process, we associate the alphabet of names which can be used in the description of that process. Ignoring the problem of subscripted names, these can be determined by the context of the declarations within which the process is written. For the purpose of illustration, we will assume that the names are

| | |
|---|---|
| x,y | variables |
| b | input channel |
| c | output channel. |

### 3.1 Atomic processes

The simplest of all processes in occam is STOP. This process never inputs, never outputs, and never terminates. The trace of its activity is forever empty. The predicate which describes its behaviour is very simple

$$STOP \triangleq st = waiting \wedge tr = \langle\rangle$$

where <> denotes the empty trace. Note that this predicate says nothing about the final values of the variables. This is because the process never terminates, so there are no final values. It also says nothing about refusals, because STOP refuses communication on any subset of the channels connected to it. There is no need to say anything about an observation that can take any value whatsoever. STOP is thus a process which forever thinks it is waiting for external communication, but never actually accepts one.

The next simplest process is SKIP, which communicates nothing, but terminates successfully with the final values of all variables equal to their intial values. Since it does not communicate, it never waits, and its trace remains empty.

$$SKIP \triangleq tr = \langle\rangle \wedge st = terminated$$
$$\wedge\ x = x^{\checkmark} \wedge\ y = y^{\checkmark}.$$

Nothing is said about the refusals here since SKIP can never be "waiting".

Let e be an expression in occam; then the assignment is simply defined

$$x := e \;\triangleq\; (e \succ\!\!- x \longrightarrow \mathrm{SKIP})$$

as may be verified by carrying out the substitution specified on the right hand side. In general, if P is an arbitrary predicate describing the behaviour of a process, then

$$x := e;\; P \;\triangleq\; (e \succ\!\!- x \longrightarrow P).$$

The next simple process in occam is output of the value of an expression e on an output channel c. There are two observations which can be made of this process, one before the output and one after. Before the output occurs, the trace is empty, the process is waiting, and it refuses to communicate on any channel other than c. After the output, the trace contains the single communication c.e, the process terminates, and the final values of all variables are equal to the initial values.

$$\begin{aligned} c!e \;\triangleq\; & (st = \mathrm{waiting} \wedge c \notin ref) \triangleleft tr = \langle\rangle \triangleright \\ & (tr_0 = c.e \wedge (tr' \succ\!\!- tr \longrightarrow \mathrm{SKIP})) \end{aligned}$$

where $tr_0$ is the first communication recorded in tr and tr' is the rest of tr after removal of the first communication. Thus

$$tr = \langle tr_0\rangle^\frown tr'$$

whenever $tr \neq \langle\rangle$.

The behaviour of input is very similar, except that the value input is not known in advance; it is equal to the final value of the relevant variable.

$$\begin{aligned} b?x \;\triangleq\; & (st = \mathrm{waiting} \wedge b \notin ref) \triangleleft tr = \langle\rangle \triangleright \\ & (\mathrm{chan}(tr_0) = b \;\wedge \\ & \;\mathrm{cont}(tr_0), tr' \succ\!\!- x, tr \longrightarrow \mathrm{SKIP}) \end{aligned}$$

where chan and cont are functions giving respectively the channel and message components of a communication.

As in the case of assignment, both of these definitions generalise to arbitrary sequels:

$$\begin{aligned} c!e;\; P \;\triangleq\; & (st = \mathrm{waiting} \wedge c \notin ref) \triangleleft tr = \langle\rangle \triangleright \\ & (tr_0 = c.e \wedge (tr' \succ\!\!- tr \longrightarrow P)) \\ b?x;\; P \;\triangleq\; & (st = \mathrm{waiting} \wedge b \notin ref) \triangleleft tr = \langle\rangle \triangleright \\ & (\mathrm{chan}(tr_0) = b \;\wedge \\ & \;\mathrm{cont}(tr_0), tr' \succ\!\!- x, tr \longrightarrow P). \end{aligned}$$

### 3.2 Simple constructors

We have now described each of the atomic processes of occam in terms of predicates describing their behaviour. We now turn to methods of composing these simple processes into processes with more elaborate behaviour. The simplest constructor is the conditional, which we will write in the manner introduced in the previous section. (The usual version of occam uses a tabulated notation

```
IF
  b
   P
  b'
   P'
   .
   :
```

but it is easy to translate either sort of conditional to the other.)

The while loop in occam is also written in tabulated form

```
WHILE b
  P
```

but we will use the notation

$$b * P.$$

This may be defined by recursion

$$b * P \;\triangleq\; \mu X.((P;X) \triangleleft b \triangleright \mathrm{SKIP}).$$

This definition uses the sequential composition operator, which we have not yet defined in general. Nevertheless, the cases we have defined permit us to give our first example, a process which copies every message input on channel c by outputting it on channel b.

$$\begin{aligned} \mathrm{COPY} &= \mathrm{true} * (c?x;\; b!x) \\ &= \forall n.\, P_n \end{aligned}$$

where $P_0 = \mathrm{true}$

and $P_{n+1} = c?x;\; b!x;\; P_n$ .

By induction one can prove

$$\begin{aligned} P_n \Rightarrow (\#tr < 2n) \Rightarrow (st = \mathrm{waiting} \;\;\wedge & \\ ((tr\upharpoonright b = tr\upharpoonright c \wedge c \notin ref) \vee & \\ (\exists m.\, tr\upharpoonright c = (tr\upharpoonright b)^\frown\langle m\rangle \wedge b \notin ref)) & \end{aligned}$$

where $tr\upharpoonright b$ (tr restricted to b) is the sequence of messages whose passage along b is recorded in tr. For example $\langle c.1, b.5, c.2\rangle\upharpoonright c = \langle 1,2\rangle$. The implication above cannot be reversed, for when $n \geq 1$ the predicate on the right hand side above is not executable. This is because it allows traces (for example many of length $\geq 2n$) some of whose prefixes (initial subsequences) are not allowed. Clearly any process which executed a trace would previously have executed every prefix. We will return to this issue later.

The division example can also be written in our version of occam:

$$q := 0;\; r := x;$$

$$(r \geq y) * (q := q + 1;\; r := r - y).$$

Another induction easily proves this equal to

$$\begin{aligned} y > 0 \;\Rightarrow\; & tr = \langle\rangle \wedge st = \mathrm{terminated} \;\wedge \\ & x' = x \wedge y' = y \wedge q' = x \div y \wedge r' = x \;\underline{\mathrm{mod}}\; y. \end{aligned}$$

As a final example note that

$$\mathrm{true} * \mathrm{SKIP} \;=\; \mathrm{true}$$

so the looping process is indeed identified with "true" as stated earlier.

### 3.3 Algebraic laws

One of the main advantages of using mathematical predicates as computer programs is that it becomes possible to reason mathematically about programs and their specifications. To make this easy, it is very desirable that the programs should obey reasonably simple and memorable algebraic laws. These laws can be used to transform a clear and simple specification into a program which is more efficient, but is otherwise guaranteed to be correct with respect to its specification (i.e. implies its specification). So in this section we give the laws which govern the occam constructs which we met in the previous sections.

The laws governing the conditional are the same as those in section 2.3, with the addition of

$$(b!e;\ P) \triangleleft k \triangleright (b!f;\ Q)$$
$$=\ b!(e \triangleleft k \triangleright f);\ (P \triangleleft k \triangleright Q)$$

$$(c?x;\ P) \triangleleft k \triangleright (c?x;\ Q) \ =\ c?x;\ (P \triangleleft k \triangleright Q)$$
provided $x$ does not appear in $k$.

There are several laws governing programs of the form $x := e;\ P$:

$$x := e;\ \mathrm{STOP} \ =\ \mathrm{STOP}$$

$$x := e;\ c!f \ =\ c!(e \succ\!\!-\, x \longrightarrow f);\ x := e$$

$$x := e;\ c?y \ =\ c?y;\ x := e \quad \text{provided } x \neq y \text{ and } y \text{ does not appear in } e$$

$$x := e;\ c?x \ =\ c?x$$

$$x := e; (P \triangleleft b \triangleright Q) \ =\ ((x := e;\ P) \triangleleft e \succ\!\!-\, x \longrightarrow b \triangleright (x := e;\ Q)).$$

The * (WHILE) constructor is a fixed point of its defining equation:

$$b * P \ =\ (P;\ (b * P)) \triangleleft b \triangleright \mathrm{SKIP}.$$

The above laws are all readily proved from the earlier definition, with the exception of the final one, which needs the continuity of the as yet undefined operator ;. In addition, it is straightforward to show that each case of ; that we have met thus far is distributive (over ∨) in its second argument, but that b * P need not be.

From here on we will place as much emphasis on an operator's laws as on its semantic definition. In (Hoare and Roscoe 1985) we will show how these laws can be shown to completely define the semantics of occam, because they can be used to transform every WHILE-free program into normal form.

### 3.4 Sequential composition

We have seen several cases of the sequential composition operator (;), and will now see how it is defined in general. (Once again, we choose the binary infix operator ; rather than the tabular SEQ form of standard occam.)

Sequential composition shares many of the properties of relational composition; it is associative, distributes through ∨, and has SKIP as its unit:

$$P;\ (Q;\ R) \ =\ (P;\ Q);\ R$$

$$P;\ (Q \vee R) \ =\ (P;\ Q) \vee (P;\ R)$$

$$(P \vee Q);\ R \ =\ (P;\ R) \vee (Q;\ R)$$

$$\mathrm{SKIP};\ P \ =\ P;\ \mathrm{SKIP} \ =\ P.$$

It has left zero true, and distributes to the right over $\triangleleft b \triangleright$:

$$\mathrm{true};\ P \ =\ \mathrm{true}$$

$$(P \triangleleft b \triangleright Q);\ R \ =\ (P;\ R) \triangleleft b \triangleright (Q;\ R).$$

It must also satisfy all the laws in the last section.

But there is another important property. As we have seen, in order for recursion to work properly it must be continuous. In other words it must distribute through ascending chains of predicates:

$$P; (\forall n. Q_n) \ =\ \forall n. (P;\ Q_n)$$
$$(\forall n. Q_n);\ P \ =\ \forall n. (Q_n;\ P)$$
whenever $\forall n. (Q_{n+1} \Rightarrow Q_n)$.

The required definition, somewhat more complex that those we have seen up to now, is

$$\begin{aligned} P;\ Q \ \triangleq\ & (P \wedge st \neq \mathrm{terminated}) \ \vee \\ & (\exists s, t, \check{x}, \check{y}.\ tr = s\text{^}t \ \wedge \\ & \qquad s, \check{x}, \check{y}, \mathrm{terminated} \succ\!\!-\, tr, x', y', st \longrightarrow P \\ & \qquad \wedge\ \ t, \check{x}, \check{y} \succ\!\!-\, tr, x, y \longrightarrow Q\ ) \ \ \vee \\ & \ \mathrm{divergent} \succ\!\!-\, st \longrightarrow P\ . \end{aligned}$$

The first clause says that any observation of P in which it has not terminated is also an observation of P;Q. The second clause deals with the case where P terminates (after trace s) and passes on the final values of its variables to Q. The final clause, necessary to make true a left zero, says that if P can diverge on trace tr then so can P;Q.

### 3.5 Further constructs

The two remaining occam constructors are a parallel constructor and an alternative constructor. We will again use a binary infix operator (||) rather that the standard PAR. The alphabets of processes combined in parallel must be disjoint except that any input channel of one process can be an output channel of the other. When this happens the channel connects the two processes, so that all communications on it are participated in by both processes and hidden from the environment. Note that the fact that alphabets are disjoint implies the absence of shared variables.

The parallel operator is symmetric, associative, has unit SKIP and zero true, and distributes over ∨.

$P \| Q = Q \| P$

$P \| (Q \| R) = (P \| Q) \| R$

$\mathrm{SKIP} \| P = P$ provided the process' alphabets are disjoint

$\mathrm{true} \| P = \mathrm{true}$

$P \| (Q \vee R) = (P \| Q) \vee (P \| R)$

The following are a few laws governing the relationship of the parallel operator with other constructs. Their applicability is often limited by the disjoint alphabets condition: the laws below are universally valid as transformations from left to right, but only conditionally valid in reverse.

$(x := e;\ P) \| Q \;=\; x := e;\ (P \| Q)$

$(c!e;\ P) \| (c?x;\ Q) \;=\; x := e;\ (P \| Q)$

$(b?y;\ P) \| (c?x;\ Q) \;=\; b?y;\ (P \| (c?x;\ Q))$
if b is not in the alphabet of the right hand side, but c is an output channel of the left hand side

$P \| (Q \triangleleft b \triangleright R) \;=\; (P \| Q) \triangleleft b \triangleright (P \| R)$

The definition of $\|$ as an operator on predicates, which is of the same order of difficulty as that of ;, is left as an exercise to the interested reader.

The constructor ALT (a form which we retain) takes as arguments not processes, but <u>guarded processes</u>. A <u>guard</u> g may take any of the forms

| | |
|---|---|
| SKIP | b & SKIP |
| c?x | b & c?x |
| c!e | b & c!e |

where b is a boolean expression. A guarded process has the form g P, where g is a guard and P is a process. The process

$\mathrm{ALT}(g_1 P_1, \ldots, g_n P_n)$

waits until one of the $g_i$ becomes ready, after which it behaves like that $P_i$, prefixed by the communication of $g_i$ if any.

SKIP is always ready, and b & SKIP is ready if b evaluates to true. c?x and c!e become ready when the process' environment is willing to carry out the respective communication. The addition of a boolean to a communication guard puts an additional restriction on when it can become ready: b & c?x is only ready when b is true and the environment is willing to output on channel c. (There are no output guards in standard occam. We have added them because they improve the language's algebraic properties.)

The following are a few of the many algebraic properties that ALT enjoys. It is independent of the order of its arguments, and furthermore:

a) $\mathrm{ALT}() = \mathrm{STOP}$

b) $\mathrm{ALT}(G, G, G_1, \ldots, G_n) = \mathrm{ALT}(G, G_1, \ldots, G_n)$

c) $\mathrm{ALT}(g_1 P_1, \ldots, g_n P_n);\ Q = \mathrm{ALT}(g_1 P_1;Q, \ldots, g_n P_n;Q)$

d) $\mathrm{ALT}(b \,\&\, g\ P, G_1, \ldots, G_n) =$
$\qquad \mathrm{ALT}(g\ P, G_1, \ldots, G_n) \triangleleft b \triangleright \mathrm{ALT}(G_1, \ldots, G_n)$

e) $\mathrm{ALT}(g\ P) = g;P$ if g is SKIP, c?x or c!e

f) $(c?x;P) \| (d!e;Q) =$
$\qquad \mathrm{ALT}(c?x\ (P \| (d!e;Q)),\ d!e\ ((c?x;P) \| Q))$
provided c is not in the alphabet of the right hand side, and d is not in the alphabet of the left hand side.

g) $P \Rightarrow \mathrm{ALT}(\mathrm{SKIP}\ P, G_1, \ldots, G_n)$

Laws (a) and (d) allow us to give a formal definition of ALT only in the case of non-empty argument lists and guards with no boolean component. When none of the $g_i$ is SKIP and $n \neq 0$,

$$\mathrm{ALT}(g_1 P_1, \ldots, g_n P_n) \triangleq$$
$$(g_1;P_1 \wedge \ldots \wedge g_n;P_n) \triangleleft tr = \langle\rangle \triangleright (g_1;P_1 \vee \ldots \vee g_n;P_n)$$

and

$$\mathrm{ALT}(\mathrm{SKIP}\ P_1, \ldots, \mathrm{SKIP}\ P_n, g_1 Q_1, \ldots, g_m Q_m) \triangleq$$
$$P_1 \vee \ldots \vee P_n \vee (tr \neq \langle\rangle \wedge (g_1;Q_1 \vee \ldots \vee g_m;Q_m))$$

where the final disjunct is false if $m = 0$.

The final types of construct we will consider are declarations of variables and of channels. The effect of a channel declaration

chan c: P

is to add channel c, undirected, to the alphabet of P. Its direction (input or output) within P will be altered by parallel operators. A variable declaration is defined

$$\mathrm{var}\ x{:}P \triangleq (\exists x, x\checkmark . P \wedge$$
$$(\mathrm{divergent} \succ\!\!- st \rightarrow P \vee$$
$$st = \mathrm{waiting} \Rightarrow x = x\checkmark)$$

where x is included in the alphabet of P, and the second clause is omitted if x is not in the external alphabet. This definition would need to be changed if our language included procedures, because of the problems of static binding.

There are many algebraic laws satisfied by declaration, of which the following are a sample.

$\mathrm{var}\ x{:}\ P = P$ provided x does not appear free in P

$\mathrm{var}\ x{:}\ P = \mathrm{var}\ y{:}\ (y \succ\!\!- x \rightarrow P)$ provided y does not appear free in P

$\mathrm{var}\ x{:}\ (P;Q) = (\mathrm{var}\ x{:}\ P);Q$ provided x does not appear free in Q

### 3.6 Safety conditions

We have already observed that not all predicates are executable. For example the "miracle" false which satisfies all specifications, and the predicates in section 3.2 which allowed traces impossible by virtue of some of their prefixes being banned. In this section we will discover a number of "safety conditions" which a predicate must satisfy to be executable, and postulate that a predicate is executable if it satisfies these conditions.

Recall that each process is a predicate whose free variables are (at most) st, tr, ref, and x, $x^{\checkmark}$, for x drawn from some finite set of variables (part of the process' alphabet). For simplicity we will assume that the "messages" set over which x,$x^{\checkmark}$ range is finite. From now on $\underline{x}$ will represent all of the initial values of a process' variables, and $\underline{x}^{\checkmark}$ all of the final values.

Our first safety condition is "absence of miracles". It says that whatever initial values we give to a process' variables, there is some behaviour possible.

S1 $\forall \underline{x}. \exists st, tr, ref, \underline{x}^{\checkmark}. P$

The second condition deals with the problem we met in 3.2. It says that if $s^\frown t$ is a possible trace of P (for some particular initialisation of its variables), then so is s.

S2 $\exists st, ref, \underline{x}^{\checkmark}. (s^\frown t \succ\!\!- tr \longrightarrow P)$

$\Rightarrow \quad \exists st, ref, \underline{x}^{\checkmark}. (s \succ\!\!- tr \longrightarrow P)$

The third condition says that P can always refuse any subset of what it is already refusing. (If it is waiting, and refusing set ref, then it would also be refusing Y∩ref.)

S3 $P \Rightarrow (Y \cap ref) \succ\!\!- ref \longrightarrow P$

The fourth condition says that if P is waiting and refusing set ref, then it will also refuse ref∪{c}, where c is any channel on which it can never communicate after its current trace.

S4 $P \wedge \neg \exists m, ref, st, \underline{x}^{\checkmark}. (tr^\frown \langle c.m \rangle \succ\!\!- tr \longrightarrow P)$

$\Rightarrow \quad ref \cup \{c\} \succ\!\!- ref \longrightarrow P$

The final three conditions tell us about processes in the three statuses. A waiting process tells us nothing about the final values of its variables; a terminated process tells us nothing about refusals; a diverging process is identified with true.

S5 $\underline{y}, waiting \succ\!\!- \underline{x}^{\checkmark}, st \longrightarrow P \Leftrightarrow waiting \succ\!\!- st \longrightarrow P$

S6 $X, terminated \succ\!\!- ref, st \longrightarrow P \Leftrightarrow terminated \succ\!\!- st \longrightarrow P$

S7 $divergent \succ\!\!- st \longrightarrow P$

$\Rightarrow \forall s, st, ref, \underline{x}^{\checkmark}. (tr^\frown s \succ\!\!- tr \longrightarrow P)$

We define a predicate to be executable if and only if it satisfies S1-7. (Of course, a different language would require different safety conditions.) The predicate defined by each occam program is executable in this sense. The class of executable predicates has several interesting properties, but we do not have space to discuss them here. It has much in common with the semantic model for occam given in (Roscoe 1985).

## 4 CONCLUSIONS

This paper has made the claim that a computer program can be identified with the strongest predicate describing all relevant observations of a computer executing the program. The claim is illustrated by the formal definitions of the notations of a simplified version of occam. We hope that it will also be justified by its promised practical benefits for the specification and development of reliable programs. Some reasons for optimism on this point are discussed in (Hoare 1984): possible methods are described for achieving

$$P \Rightarrow S$$

where P is a program and S is a specification.

A second aim of this paper has been to demonstrate the elegance of the occam language (INMOS Ltd, 1984). We have shown how its constructs can be given simple but expressive semantics in our formalism, and how there are many useful laws relating occam terms. The approach in this paper is one of several complementary ways of giving a semantics to occam. A rather more standard denotational semantics is given in (Roscoe 1984). In (Hoare and Roscoe 1984) we will construct a normal form for finite (WHILE-free) occam programs. Two programs are semantically equivalent if and only if they have the same normal form. Because we can transform every finite program to normal form using algebraic laws, these laws completely characterise the semantics of occam. (An infinitary rule is used to deal with non-finite programs.)

## ACKNOWLEDGEMENTS

The work in this paper owes much to the influence of E.W. Dijkstra (Dijkstra 1976), D.S. Scott (Scott 1981), R. Milner (Milner 1980) and E.C.R. Hehner (Hehner 1984). Our construction of a mathematical model for concurrent systems owes much to collaborations with S.D. Brookes (Brookes et al, 1984; Brookes and Roscoe, 1984) and E.R. Olderog (Olderog and Hoare 1984).

## REFERENCES

Brookes, S.D., Hoare, C.A.R., and Roscoe, A.W. A Theory of Communicating Sequential Processes. Journal of the ACM (to appear, July 1984).

Brookes, S.D. and Roscoe, A.W. An Improved Failures Model for Communicating Processes. To appear in Procedings of NSF/SERC workshop on concurrency, Springer LNCS 1984,

Dijkstra, E.W. A Discipline of Programming Prentice Hall, 1976.

Floyd, R.W. Assigning Meanings to Programs. Proc. AMS Symposia in Applied Mathematics 1967.

Hehner, E.C.R. Predicative Programming. Commun ACM 27,2 pp134-151 (February 1984).

Hoare, C.A.R. Programs are Predicates. To appear in Phil Proc R Soc London A, 1984.

Hoare, C.A.R. A Couple of Novelties in the Propositional Calculus. Zeit. für Math. Logik (to appear 1985).

Hoare, C.A.R. and Roscoe, A.W. The Laws of Occam Programming. To appear 1985.

INMOS Ltd. The Occam Programming Manual. Prentice-Hall 1984.

Milner, A.J.R.G. A Calculus of Communicating Systems. Springer LNCS 92, 1980.

Olderog, E.-R. and Hoare, C.A.R. Specification-Oriented Semantics for Communicating Processes. Technical monograph PRG-37, Oxford University Computing Laboratory, 1984.

Roscoe, A.W. Denotational Semantics for Occam. To appear in Proc. NSF/SERC workshop on concurrency, Springer LNCS 1984.

Scott, D.S. Lecture Notes on a Mathematical Theory of Computation. Technical monograph PRG-19, Oxford University Computing Laboratory 1981.