Algebraic Operational Semantics and Occam

Yuri Gurevich[1,2]
Lawrence S. Moss[3]

Abstract

We generalize algebraic operational semantics from sequential languages to distributed, concurrent languages using Occam as an example. Elsewhere, we will discuss applications to the study of verification and transformation of programs.

1 Introduction

Computational processes involve change. Although this sounds like an empty slogan, semantical studies often downplay the dynamic aspects of computing; a sequential process is often modeled by a function which is merely a point in a large space. This approach is related to the study of continuously varying physical structures, where adding a dimension for time often doesn't make the mathematics that much more difficult. However, this trick doesn't always work as well in computer science. More importantly, stressing the static aspects of computation may lead one to mathematical side-issues removed from the main semantic issues.

The idea of **algebraic operational semantics** is that the dynamic and resource-bounded aspects of computation should be studied on their own terms. We seek a basic vocabulary to describe structures that change over time and are finite in the same way as real computers are finite. We are interested in questions such as: What structures are appropriate to model different programming languages? Which are appropriate to model operating systems? And so on. Once we have our dynamic (or evolving) structures (or algebras), we are interested in logics for reasoning about them and in complexity analysis of our computational models.

At the present time, we are still at the stage of modeling different programming languages. How can we represent best the resource-boundedness of real computation? What models do we need to adequately and efficiently reflect real programming languages? Answers to questions such as these give rise to a semantical approach which takes the dynamic and resource-bounded aspects of computation as central.

There have been three studies of programming languages in this framework, of Modula-2 [4], Smalltalk [1], and Prolog (including all of the non-logical operations that change the program) [2]. This paper extends the approach of dynamic structures to the case of distributed, concurrent computation. There are several new questions to be considered: What does it mean to have several parts of a computation active at the same time? How does one model the communicative aspects of distributed computation, without assuming the existence of a global clock, and without modeling the hardware of communication?

[1]Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor, MI 48109–2122.
[2]The work of this author was partially supported by NSF grants DCR 85-03275 and CCR 89-04728.
[3]Mathematical Sciences Department, IBM T. J. Watson Research Center, Yorktown Heights, NY 10598.

For concreteness, we focus on the language Occam [8], and the reader need not have familiarity with the language to understand what we are doing. We propose a generalization of *evolving structures* to *distributed evolving structures*, and our main claim is that distributed evolving structures are a good vehicle for understanding the operational behavior of distributed and concurrent programs. In a different direction, we believe that distributed evolving structures work well as a pedagogical tool, too.

Our approach is somewhat different from the existing influential approaches, such as CCS [7], CSP [5], denotational semantics and algebraic semantics. We don't use transition systems taking one program to another; usually only our models evolve, not our programs. And we do not exploit uninterpreted atomic actions. In reality, atomic actions come in different forms and with different parameters. Much is gained by abstracting away those parameters, but much is lost, too.

In no way do we wish to imply that other approaches are misguided. On the contrary, we feel that it is useful to study semantics from several points of view.

1.1 Acknowledgments

We are grateful to Egon Börger, and to his students Davide Sangiorgi and Giovanni Resta, for their detailed and constructive comments on a previous draft of this paper. We also thank Padmanabhan Krishnan and Dalia Malki for many discussions concerning Occam and distributed computing.

2 Background on Evolving Structures

Sequential evolving structures were introduced in [3] and used in [1], [2], and [4] to give operational semantics for Modula-2, Smalltalk, and Prolog, respectively. They are abstract machines working in discrete linear time. Sequentiality means only that the time is discrete and linear; the machine may be parallel and even distributed. Later in this paper we introduce a class of nonsequential evolving structures. In this section, we define in an independent manner a very narrow class of sequential evolving structures $\mathcal{S}$ sufficient for our purposes in this paper.

States of $\mathcal{S}$ are many-sorted first-order structures of the same finite signature, with the same finite universes (sorts) and with fixed interpretations of some basic functions. (Such basic functions will be called static; the other basic functions will be called dynamic.) It is supposed that one of the universes is $\mathcal{BOOL} = \{\mathsf{true}, \mathsf{false}\}$ and therefore we will not take relations as basic objects. The boolean functions corresponding to the standard propositional connectives are static functions. For every universe U of $\mathcal{S}$, the equality function on U (of type $U \times U \to \mathcal{BOOL}$) is a static function of $\mathcal{S}$. Some states of $\mathcal{S}$ are designated to be the **initial states**. $\mathcal{S}$ has a finite number of **transition rules**, and each transition rule has the form

(1) If b then U_1 and U_2 and $\cdots$ and U_n

where each U_m is an **update** of the form

(2) $f(e_1, \ldots, e_j) := e_0$.

Here b is a boolean expression (the **guard**), f is a dynamic basic function, and each e_i is an expression of an appropriate type in the signature of $\mathcal{S}$. It is supposed that different updates of the same rule update different basic functions, and the guards of different rules are incompatible.

A **run** of $\mathcal{S}$ is a finite or infinite sequence $s_0, s_1, \ldots$, of states of $\mathcal{S}$ such that s_0 is an initial state of $\mathcal{S}$, and each s_{k+1} is obtained from s_k by means of some (unique) transition rule of the form (1). This means that b evaluates to true in s_k, and s_{k+1} is obtained from s_k by means of updates U_m: If U_m is $f(e_1, \ldots, e_j) := e_0$ and $e_0, \ldots, e_j$ evaluate in s_k to $a_0, \ldots, a_j$, respectively, then $f(a_1, \ldots, a_j) = a_0$ in s_{k+1}, that is, the value of f at $a_1, \ldots, a_j$ is **updated** to a_0. We do this for each U_m; otherwise s_{k+1} is identical to s_k. Notice that $\mathcal{S}$ is unable to exchange information with the outside world. More general evolving structures can be found in [1, 2, 3, and 4].

3 An Example from Occam

Consider the following example of a program P of Occam:

```
PAR
   c! max(i,j)
   d! max(x,y)
   SEQ
     c? a
     d? b
     e! min(a,b)
```

Here is the intended meaning of this program: P consists of three processes running in parallel: `c! max(i,j)`, `d! max(x,y)` and the `SEQ` process. The first of these has two given numbers i and j. It computes the maximum and sends it over channel c to the `SEQ` process. The second computes the maximum of x and y and sends it over d to the `SEQ` process. The `SEQ` process receives these maxima, in order, and sends the minimum over channel e to the outside world. There are a few important remarks on the timing that we should make. First the processes `c! max(u,v)`and `d! max(x,y)` work in parallel, and they are not finished until their output is received on the other end of the appropriate channel. The `SEQ` process is written to always accept input on c before d. That means that the process `d! max(x,y)` might well be ready to output *before* `c! max(u,v)`, but communication over channel d cannot take place until communication over channel c has finished.

We also should make a remark on the variables used in P. The `SEQ` process calls its inputs a and b, but it might as well have called them anything else, including i and j, or x and y. There is a syntactic restriction in Occam which insures that no two children of a `PAR` process change the same variable. In principle, the programmer may always use different identifiers for different variables.

Lest the reader think this example too simple, we mention a few ways in which it could be made more interesting. One way would be to replace the processes `c! max(i,j)`and `d! max(x,y)` by more complicated processes, each of which accepts two inputs from the outside world. In addition, one could encase the entire process in a large `WHILE` loop. In this way we obtain a process which accepts four infinite input streams, and computes a new stream. The code here would be as follows:

```
PAR
     WHILE TRUE
        SEQ
          in1? i
          in2? j
          c! max(i,j)
     WHILE TRUE
        SEQ
          in3? x
          1n4? y
          d! max(x,y)
```

```
WHILE TRUE
    SEQ
     c? a
     d? b
     e! min(a,b)
```

More interestingly, we could take the original program and change the way the third process works. As it stands, it accepts the inputs in a fixed order. A better use of parallel resources is obtained if the third process is able to accept the inputs *as they become ready.* This is available in Occam, by using the ALT construction. We shall consider a process which uses ALT in Section 7.

3.1 A Distributed Evolving Structure for P

We interpret this program P by a **distributed evolving structure** M. The definition of such a structure is exactly the same as that of a sequential evolving structure. (But the definition of a run will be different.) So M has several universes, and it comes with a set of transition rules. We should mention that the transition rules of this structure M are specially tailored to P. Later we show how to give one overall set of rules, which applies to *any* program of the language. In this way, all distributed evolving structures for Occam have the same transition rules.

One novelty is the interpretation of the transition rules, as embodied in the definition of a run. We expand on this point below.

Two of the universes of M are standard: $\mathcal{BOOL}$ and $\mathcal{INT}$. These two come with all of the standard operations. Another universe of M is $\mathcal{TREE}$, the program considered as a syntactic tree. It is a labeled digraph, pictured in Figure 1.

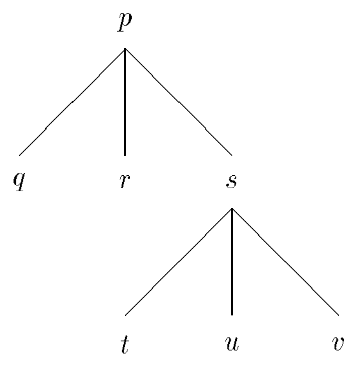


Figure 1: The Universe $\mathcal{TREE}$ of M

Each node corresponds to a line of P. For example, p corresponds to the PAR, and v corresponds to the line e! min(a,b). (In Section 6, we relax this condition to allow nodes to correspond to smaller syntactic units, such as expressions.) On the universe $\mathcal{TREE}$ we have the natural interpretations of the function symbols parent, first-child, last-child, and next-sibling. These interpretations are partial functions.

In addition, there is a universe $\mathcal{MODE}$ of **modes**. As an informal explanation of our modes, here is the description of what happens to a "typical" process p_0. Before p_0 starts, it is in dormant mode; i.e., mode(p_0) = dormant. When it becomes active for whatever reason, p_0 assumes the starting mode. It remains in this mode for exactly one moment, and then it proceeds to working mode. It is in this mode while processes which depend on it are computing. When and if the work is completed, p_0 enters reporting mode. Typically, p_0 reports to its parent or to the next sibling process. After reporting, a process becomes dormant until the next time it is needed.

Further, there is a dynamic function val which takes a node and a relevant variable and returns an integer value. For example, the variables relevant to q are i and j. The section val_q of val is never updated on any variable except i and j. We assume that at the beginning of a run, $\mathsf{val}_q(i)$ and $\mathsf{val}_q(j)$ are undefined. In this case, we write, e.g., $\mathsf{val}_q(i) = \mathsf{undef}$. (We assume that there is an extra element undef in $\mathcal{INT}$.) This assumption concerning undefined values at the beginning of a run holds for all of the nodes except the root. So when a node gets started by its parent, the child's section val is updated to match part of the parent's.

We introduce a new command, $\mathsf{output}_v(e)$ where e is an expression. (See rule (7) below.) The point is that in contrast to output to a different part of the program, there is no "assignee" to receive the output value. The command above has the same status for us as an update. That is, it may appear in transition rules. However, it does not result in any updates of dynamic functions. We might mention also that if this program P were put into into a larger program containing a recipient of the minimum on channel e, then the transition rule corresponding to the output line of the program would not mention this new command at all. Instead, it would be similar to the rules for internal communication in Section 4 below.

Remark on notation We suppress the mode function in the following way. Instead of writing, for example, $\mathsf{mode}(p) = \mathsf{working}$, we write p is working. Instead of $\mathsf{mode}(p) := \mathsf{dormant}$, we write p changes to dormant. The purpose of these conventions is to make the rules easier to read and also to clarify the difference between changes of mode and changes of value.

4 Transition Rules

In this section, we present transition rules for the dynamic structure M.

Remark Our transition rules reflect a complete prohibition of shared variables: Different children of the SEQ process may as well live on different computers and maintain there the relevant variables. This is an extreme point of view and, viewed as an interpreter, our model is inefficient in handling variables: All transition rules apply "locally" on $\mathcal{TREE}$. For example, there is no rules which immediately transfers the value of a from from t to v; this value must first be passed to u. But this point of view is consistent with Occam. One can take a position of extreme distributivity and argue that sharing of variables belongs to optimization. We do not take a strong ideological stand. Our rules reflect one possible intuition about Occam. It is not difficult to change the rules and allow sharing of variables between processes.

(1) If p is starting and q, r, and s are dormant,
then p changes to working,
q changes to starting, $\mathsf{val}_q(i) := \mathsf{val}_p(i)$, $\mathsf{val}_q(j) := \mathsf{val}_p(j)$
r changes to starting,$\mathsf{val}_r(x) := \mathsf{val}_p(x)$, $\mathsf{val}_r(y) := \mathsf{val}_p(y)$
and s changes to starting.

(2) If s is starting, and t is dormant,
then s changes to working, and t changes to starting.

(3) If q is starting and t is starting,
then $\mathsf{val}_t(a) := max(\mathsf{val}_q(i), \mathsf{val}_q(j))$,
q changes to reporting, and t changes to reporting.

(4) If t is reporting, and u is dormant,
then u changes to starting, and $\mathsf{val}_u(a) := \mathsf{val}_t(a)$,
t changes to dormant, and $\mathsf{val}_t(a) :=$ undef.

(5) If r is starting, and u is starting,
then $\mathsf{val}_u(b) := max(\mathsf{val}_r(x), \mathsf{val}_r(y))$,
r changes to reporting, and u changes to reporting.

(6) If u is reporting, and v is dormant,
then v changes to starting, $\mathsf{val}_v(a) : \mathsf{val}_u(a)$, $\mathsf{val}_v(b) := \mathsf{val}_u(b)$,
u changes to dormant, $\mathsf{val}_u(a) :=$ undef, $\mathsf{val}_u(b) :=$ undef.

(7) If v is starting,
then $\mathsf{output}_v(min(\mathsf{val}_v(x), \mathsf{val}_v(y)))$, and v changes to reporting.

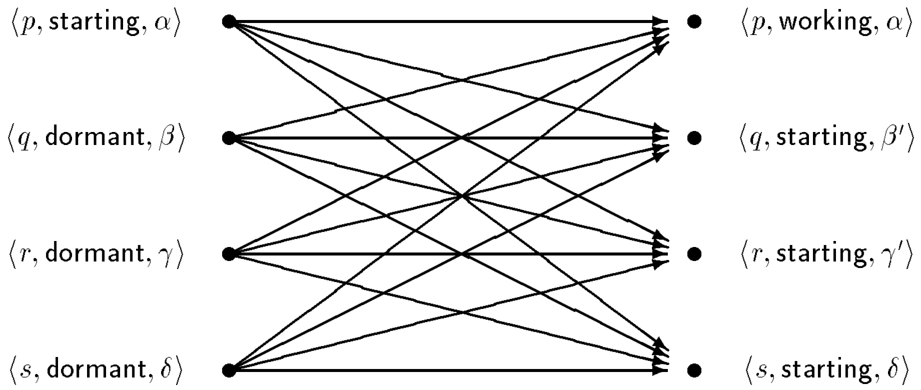
(8) If v is reporting, and s is working,
then s changes to reporting,
v changes to dormant, $\mathsf{val}_v(a) :=$ undef, $\mathsf{val}_v(b) :=$ undef.

(9) If p is working, and q, r, and s are reporting,
then p changes to reporting,
q changes to dormant, $\mathsf{val}_q(i) :=$ undef, $\mathsf{val}_q(j) :=$ undef
r changes to dormant, $\mathsf{val}_r(x) :=$ undef, $\mathsf{val}_r(y) :=$ undef
s changes to dormant, $\mathsf{val}_s(a) :=$ undef, and $\mathsf{val}_s(b) :=$ undef.

Note that whenever a process assumes the dormant mode, all of its variables become undefined. In the interests of readability and brevity, we henceforth adopt the convention that "p changes to dormant" is an abbreviation for "p changes to dormant and for all $x \in \mathsf{var}(p)$, $x :=$ undef."

5 Runs Of Distributed Evolving Structures

We mentioned above that the definition of a run of a distributed evolving structure is going to be different than that of a sequential evolving structure. In the sequential case, each state s contains all the information about *the entire process* at an instant of time. In our case, we don't have global states. So each of the processes in the tree will have an evolution of its own. As a result of this local approach, another difference arises. In the sequential case, every state s other than the initial state has an immediate predecessor, say t. This predecessor state t is fully responsible for the transition of the process to state s. In the distributed case, an transition may require more than one cause. The clear example of this is PAR. A PAR process is able to relinquish control only when all of its children have finished. In order to represent this, our graph structures for runs will



$\langle p, \mathsf{starting}, \alpha\rangle$
$\langle q, \mathsf{dormant}, \beta\rangle$
$\langle r, \mathsf{dormant}, \gamma\rangle$
$\langle s, \mathsf{dormant}, \delta\rangle$
$\langle p, \mathsf{working}, \alpha\rangle$
$\langle q, \mathsf{starting}, \beta'\rangle$
$\langle r, \mathsf{starting}, \gamma'\rangle$
$\langle s, \mathsf{starting}, \delta\rangle$

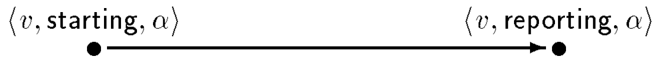
Figure 2: A Transition Via Rule (1)

be more complicated than simple chains. They will be labeled digraphs which are composed of **transitions**.

A **transition via rule (1)** is a complete bipartite directed graph whose sources and targets are as in Figure 2. The function α is an arbitrary function from the variables i, j, x, and y to $\mathcal{INT}$. Similarly, β, γ, and δ are arbitrary functions from $\{i,j\}$, $\{x,y\}$, and $\{a,b\}$ to $\mathcal{INT}$. We require that β' be the restriction of α to $\{i,j\}$, and similarly for γ'. These conditions are immediate from rule (1) itself. Note that because the functions on the left are arbitrary, there are many possible transitions via rule (1).

The picture is a graphic representation of a transition that involves four processes. The idea is that the nodes on the left give a *cause* of this transition. We do not intend that the transition takes a fixed amount of time. Moreover, we don't suppose that different processes "live" in the same time.

Here is a second example: A **transition via rule (7)** is a graph



$\langle v, \mathsf{starting}, \alpha\rangle$
$\langle v, \mathsf{reporting}, \alpha\rangle$

Note that there is no change on the third component of the labels. This is because no clause of rule (7), not even the command $\mathsf{output}_v(min(\mathsf{val}_v(x), \mathsf{val}_v(y)))$ results in any update of any variable.

There are similar definitions of transitions via all of the other rules.

We call the possible labels involving a node p the **states** of p. More precisely, a state of p is a triple consisting of p, some mode m, and some section α of the function val. The variables in the domain of the section are determined by a static analysis of the particular process p Note that Nat hand; we shall say more about this in the next section.

Definition Let M be a distributed evolving structure. A run of M is a directed graph $G = \langle V, E\rangle$ whose vertices are labeled by states of processes, and such that

(1) For every node p of $\mathcal{TREE}$, the vertices of G labeled $\langle p, m, \alpha\rangle$ (for some m and α) form a chain under E. We refer to those states as $p_0, p_1, \ldots$, and we call this sequence the **stages of** p. This sequence may be either finite or infinite.

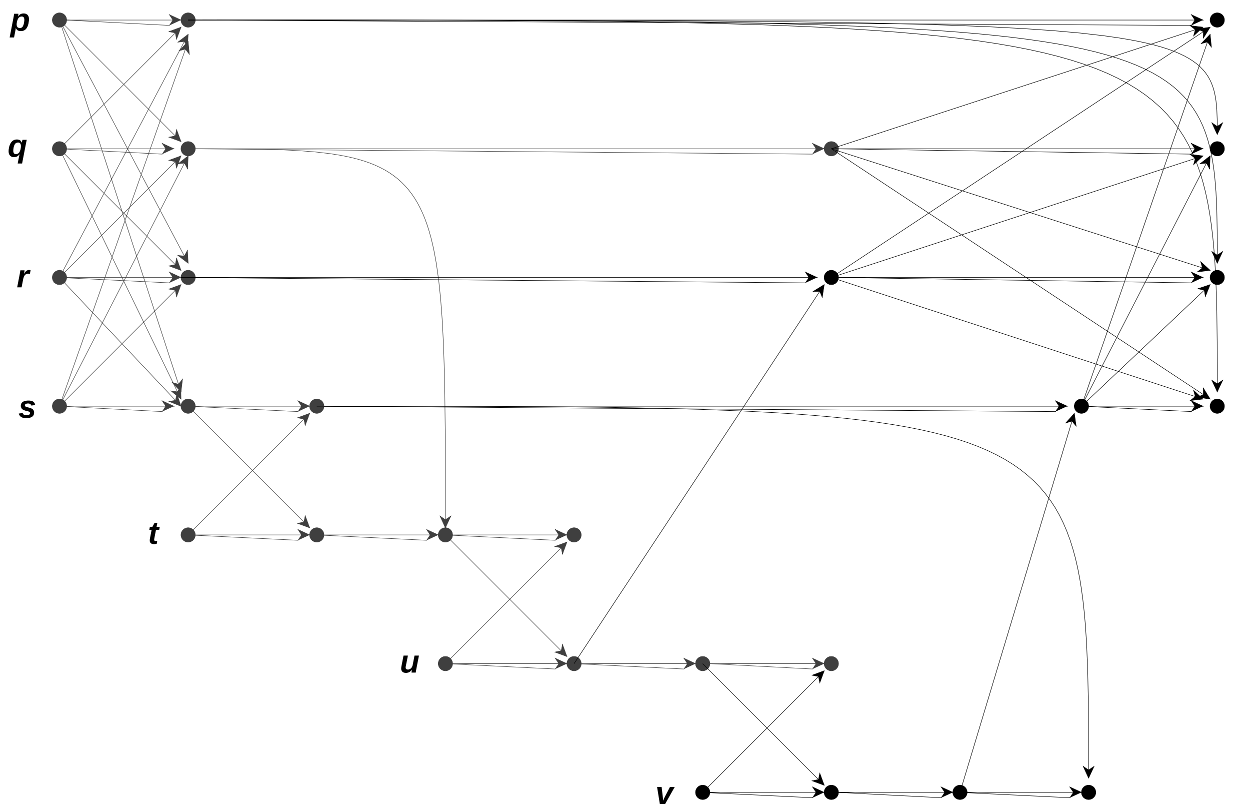


Figure 3: A Run of M. The labels are not shown. The horizontal chains are the evolutions of the nodes.

(2) There exists a partition of the set E of edges of G such that each piece of the partition is a transition via one of the transition rules.

(3) $\mathsf{root}_0 = \langle \mathsf{root}, \mathsf{starting}, \alpha \rangle$ for some α. For all $p \neq \mathsf{root}$, the mode represented in p_0 is $\mathsf{dormant}$, and the section α represented is such that for all variables $x \in \mathsf{var}(p)$, $\alpha(x) = \mathsf{undef}$.

Usually, we are interested in **maximal runs**, those runs which cannot be extended further.

A complete run of M is shown in Figure 3. The labels on the nodes have been left off. The reader can easily supply labels and then check that the resulting labeled digraph is is indeed a run according to our definitions. The main point of the verification is a partition of the edges into sets corresponding to the transition rules.

6 A Single Set of Transition Rules for Arbitrary Programs

The set of transition rules for P given above obviously works only for the program P itself. In this section, we show how to get one set which suffices *for all programs.* Then every program Q of the language can be associated with a distributed evolving structure M_Q. The main difference between different M_Q would be in their process trees. The transition rules would be entirely the same.

In this section, we present a set of transition rules for Occam. The rules we have chosen embody our intuitions about how the language works. However, we know that there are other possible sets of transition rules, so we also discuss the choices that we have made. Our central commitment in this paper is not to any particular set of transition rules, but rather to a particular style of doing semantics.

6.1 Our Treatment of Variable Updates

Every semantics of distributed computing must face the issue that sometimes variables are maintained locally, and sometimes they are shared by processes. Our semantics for Occam models the intuition that there are no shared variables. (But see also our reservations on this at the beginning of Section 4.) Furthermore, we treat process nodes of type `SEQ` or `PAR` as autonomous sequential processes. Note that each such node changes no variables whatsoever. However, a node must exchange information with processes that it depends on or which depend on it. So for each node p of a process tree, we shall have a function $\mathsf{var}(p)$ which specifies all of the variables needed by p or any process which depends on p. We skip the definition of which processes depend on a given process p. The nontrivial point is that if p is a child of a `SEQ` process then the younger siblings of p depend on p.

If the type of p is `EXPR` or `BOOL EXPR`, then in addition to (explicit) variables, $\mathsf{var}(p)$ contains a dynamic distinguished element (an impicit variable), expr. It is intended to name the value of the expression in the right-hand side.

6.2 Transition Rules, Transitions, and Runs

We make an important change from our previous example in Section 3. The statements of our transition rules will involve parameters (such as p, q, and r) ranging over the nodes of the $\mathcal{TREE}$ universe of a structure for Occam. We understand such rules as being universally quantified with respect to nodes. In this way, we obtain a *finite* set of transition rules for Occam. Together with the definition of a run, this is the basis of our proposal to generalize evolving algebras. The next subsections present transition rules, grouped according to construct, along with some discussion.

A **run** will once again be a certain labeled digraph. The possible labels are again triples $\langle p, m, \alpha \rangle$ consisting of a node p of the $\mathcal{TREE}$, a mode m , and a state α of the section val_p, the function val. (Here val_p is a finite function defined on $\mathsf{var}(p)$.) A **transition** (according to one of our rules) is a complete bipartite digraph. The node information on the labels of the sources and the targets is the same. (Note though, that for some of our rules, e.g., the rules for `PAR` below, it is not the case that all of the nodes involved are named explicitly. Instead, there is a quantification. Of course, a transition corresponding to such a transition rule must contain information about some node p of type `PAR` and *all* of its children.) We shall not write down a formal definition of a transition according to a rule; it is a labeled digraph whose labels come from the rule in the natural way.

Now the definition of a run is exactly as before.

6.3 SEQ

(10) If $(\mathsf{type}(p) = \mathtt{SEQ}$ or $\mathsf{type}(p) = \mathtt{IF}$ or $\mathsf{type}(p) = \mathtt{WHILE}$ or $\mathsf{type}(p) = \mathtt{OUTPUT})$,
$q = \mathsf{first\text{-}child}(p)$, and p **is starting**,
then p **changes to working**, q **changes to starting**,
and for all $x \in \mathsf{var}(q)$, $\mathsf{val}_q(x) := \mathsf{val}_p(x)$.

(11) If $\mathsf{type}(p) = \mathtt{SEQ}$, $p = \mathsf{parent}(q)$, $r = \mathsf{next\text{-}sibling}(q)$,
p **is working**, q **is reporting**, and r **is dormant**,
then r **changes to starting**, and for all $x \in \mathsf{var}(r)$, $\mathsf{val}_r(x) := \mathsf{val}_q(x)$,
and q **changes to dormant**.

(Recall our convention that "q changes to dormant" means "q changes to dormant and for all $x \in \mathsf{var}(q)$, $x := \mathsf{undef}$.")

(12) If $\mathsf{type}(p) = \mathtt{SEQ}$, $p = \mathsf{parent}(q)$, $q = \mathsf{last\text{-}child}(p)$ and q **is reporting**,
then for all $x \in \mathsf{var}(q)$, $\mathsf{val}_p(x) := \mathsf{val}_q(x)$,
p **changes to reporting**, and q **changes to dormant**.

Concerning the last rule, we adopt the convention that when a process becomes dormant, all of its variables become undefined. The reader may wonder whether this is necessary. Surely, something like this is needed, since in illegal in Occam to refer to variables after the process has become dormant.

This is not strictly necessary, of course, but we adopt it anyway. Certainly it would be a mistake to assume that values of variable persist indefinitely, and we lose no expressive power by our assumption that the values are lost immediately.

It should be noted that none of the above rules tell when the parent SEQ becomes dormant. This is a matter between the SEQ process and *its* parent; if it has no parent, the rule of Section 6.10 applies.

6.4 PAR

(13) If $\mathsf{type}(p) = \mathtt{PAR}$ and p **is starting**, and for all children q of p, q **is dormant**,
then p **changes to working**, and for all children q of p, q **changes to starting**,
and for all $x \in \mathsf{var}(q)$, $\mathsf{val}_q(x) := \mathsf{val}_p(x)$.

(14) If $\mathsf{type}(p) = \mathtt{PAR}$, p **is working**, and for all children q of p, q **is reporting**,
then p **changes to reporting**,
for all children q of p, q **changes to dormant**,
and for all $x \in \mathsf{var}(q)$, $\mathsf{val}_p(x) := \mathsf{val}_q(x)$.

Notice here that the children of a PAR process work at their own rates, so they may assume the reporting mode independently. After each of the children assumes this mode, the parent assumes the reporting mode.

We should mention that it certainly is possible to insist that the children report and become dormant in some pre-arranged order, or that the parent keeps track of the progress of all the children in an explicit way. The formalism of evolving structures does not forbid this interpretation of PAR, but it doesn't suggest it either.

6.5 SKIP and STOP.

SKIP has exactly one rule:

(15) If $\mathsf{type}(p) = \mathtt{SKIP}$ and p **is starting**,
then p **changes to reporting**.

In contrast, STOP has no transition rules whatsoever. If it should happen to get started by a parent or older sibling, it never evolves.

6.6 EXPR and BOOL EXPR

Our treatment of these syntactic types is brief, since we are much more interested in this paper in the treatment of control in distributed computing.

We stipulate that the sequential structures corresponding to nodes of types EXPR and BOOL EXPR contain a dynamic distinguished element expr. As its name suggests, expr is either an integer or a boolean, depending on p. We also assume that the domain of the function var defined on these nodes contains expr.

Let p be a node of type EXPR or BOOL EXPR. The transition rules insure that when p evolves to starting, it acquires values of variables from its parent. Next, p assumes working mode, and its children (if any) change to starting; those children correspond to subexpressions. Eventually, p evolves to reporting mode. In reporting mode, $\mathsf{val}_p(\mathsf{expr})$ is the value of the expression corresponding to p, with the given values of the variables.

It is straightforward to write transition rules which accomplish this, and we omit the details.

6.7 ASSIGNMENT

We next turn to the rules for assignment statements. Syntactically, we assume that a node p corresponding to an assignment statement has exactly two children, one of type ASSIGNEE, and the other is of type EXPR or BOOL EXPR. Only the second evolves, and the var set of the first is a singleton. We should mention that when $\mathsf{type}(p) = \mathtt{ASSIGNMENT}$, $\mathsf{var}(p)$ might contain more variables than those used in the corresponding expression; this happens when p is the child of a SEQ process.

(16) If $\mathsf{type}(p) = \mathtt{ASSIGNMENT}$, $q = \mathsf{last\text{-}child}(p)$, p **is starting**, and q **is dormant**,
then p **changes to working**, q **changes to starting**,
and for all $x \in \mathsf{var}(q)$, $\mathsf{val}_q(x) := \mathsf{val}_p(x)$.

(17) If $\mathsf{type}(p) = \mathtt{ASSIGNMENT}$, $q = \mathsf{first\text{-}child}(p)$, $r = \mathsf{last\text{-}child}(p)$,
p **is working**, and r **is reporting**,
then p **changes to reporting**, for all $x \in \mathsf{var}(q)$, $\mathsf{val}_p(x) := \mathsf{val}_r(\mathsf{expr})$,
and r **changes to dormant**.

6.8 IF and WHILE

The rules for these two constructions are self-explanatory. A node corresponding to a WHILE process has two children, the first of which corresponds to a BOOL EXPR, and the second to an arbitrary process. IF may have many children. The children of a node of type IF alternate between BOOL EXPRs and processes. The expressions are evaluated, in order, until one of them evaluates to true. Then the immediately following process is executed. If none of the children evalates to true, then the overall conditional becomes reporting.

Before stating these rules, we remind the reader that rule (10) describes what nodes of type `IF` and `WHILE` do when they start.

(18) If (type(p) = `IF` or type(p) = `WHILE`), p = parent(q), type(q) = `BOOL EXPR`, r = next-sibling(q), p is working, q is reporting, r is dormant, and val$_q$(expr) = true,
then q changes to dormant, r changes to starting, and for all $x \in$ var(r), val$_r(x)$:= val$_p(x)$.

(19) If type(p) = `IF`, p = parent(q), type(q) = `BOOL EXPR`, r = next-sibling(next-sibling(q)), p is working, q is reporting, and val$_q$(expr) = false,
then q changes to dormant, r changes to starting, and for all $x \in$ var(r), val$_r(x)$:= val$_p(x)$.

(20) If type(p) = `IF`, p = parent(q), type(q) = `BOOL EXPR`, next-sibling(q) = last-child(p), p is working, q is reporting, and val$_q$(expr) = false,
then q changes to dormant, and p changes to reporting.

(21) If type(p) = `IF`, p = parent(r), type(r) $\neq$ `BOOL EXPR`, p is working, and r is reporting,
then r changes to dormant, p changes to reporting, and for all $x \in$ var(r), val$_p(x)$:= val$_r(x)$.

(22) If type(p) = `WHILE`, q = first-child(p), r = last-child(p), p is working, q is dormant, and r is reporting,
then p changes to starting, for all $x \in$ var(r), val$_p(x)$:= val$_r(x)$, and r changes to dormant.

(23) If type(p) = `WHILE`, q = first-child(p), p is working, q is reporting, and val$_q$(expr) = false,
then p changes to reporting, and q changes to dormant.

6.9 INPUT and OUTPUT

A node p of type `OUTPUT` has a unique child of type `EXPR` or `BOOL EXPR`. A node p of type `INPUT` has a unique child. The type of the child is of type `ASSIGNEE`, and its var set is a singleton. Furthermore, the $\mathcal{TREE}$ universe has a primitive relation channel(p, q) with the property that if channel(p, q), then type(p) = `OUTPUT`, type(q) = `INPUT`, and the same channel is associated with p and q.

(24) If $\mathsf{type}(p) = \texttt{INPUT}$ and p is starting,
then p changes to ready.

(It turns out that it is not necessary for INPUT processes to assume a working mode.)

(25) If $\mathsf{type}(p) = \texttt{OUTPUT}$, $q = \mathsf{first\text{-}child}(p)$, and p is starting,
then p changes to working, q changes to starting,
and for all $x \in \mathsf{var}(q)$, $\mathsf{val}_q(x) := \mathsf{val}_p(x)$.

(26) If $\mathsf{type}(p) = \texttt{INPUT}$, $q = \mathsf{first\text{-}child}(p)$, $\neg(\exists r)\mathsf{channel}(r, p)$ and p is ready,
then for all $x \in \mathsf{var}(q)$, $\mathsf{input}_p(x)$, and p changes to reporting.

In Setion 4 we discussed the command $\mathsf{output}_p(e)$, where e is an expression. The command $\mathsf{input}_p(x)$ is a dual command, but unlike output, it does involve an update. It means that the value of the variable x is updated to any element of the appropriated domain. Note that $\mathsf{type}(q) = \texttt{ASSIGNEE}$, so $\mathsf{var}(q)$ is a singleton. In contrast, p might be a child of a SEQ, and therefore $\mathsf{var}(p)$ might contain many other variables. This is why we must mention q in this rule.

(27) If $\mathsf{type}(p) = \texttt{OUTPUT}$ and $\neg(\exists q)\mathsf{channel}(p, q)$, and p is ready,
then $\mathsf{output}_p(\mathsf{val}_p(\mathsf{expr}))$, and p changes to reporting.

Finally, we come to the rule that actually takes care of internal communication :

(28) If $\mathsf{type}(p) = \texttt{OUTPUT}$, $\mathsf{type}(q) = \texttt{INPUT}$, $r = \mathsf{first\text{-}child}(q)$,
$\mathsf{channel}(p, q)$, and p and q areready,
then for all $x \in \mathsf{var}(r)$, $\mathsf{val}_p(x) := \mathsf{val}_q(\mathsf{expr})$,
p changes to reporting, and q changes to reporting.

6.10 How the Root Becomes dma

(29) If $p = \mathsf{root}$ and p is reporting,
then p changes to dormant.

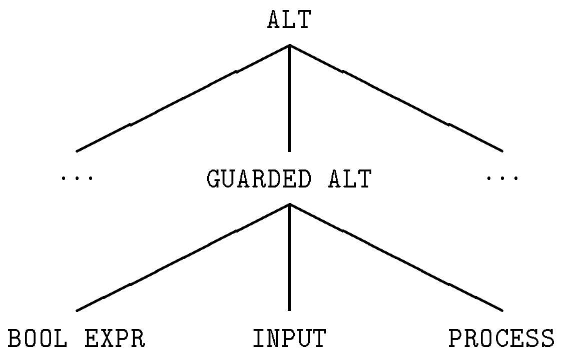


Figure 5: The Syntax of `ALT`.

7 `ALT` and `PRI ALT`

The following example illustrates the `ALT` construction of Occam:

```
WHILE TRUE
   ALT
         c? a
           e! b
         d? a
           e! b
```

`ALT` allows the input to be received from whichever channel is ready first. So this process accepts inputs on channels *c* and *d in whatever order they come*, and sends them out on channel *e*. It is not assumed in Occam that `ALT` behaves fairly. The question arises as to what happens when inputs arrive simultaneously. There are two mechanisms to do this in Occam.

The first takes takes inputs in a non-deterministic fashion, and the second takes the input corresponding to the channel that was mentioned first. `ALT` and its variation `PRI ALT` are essential in order to make full use of the capabilities afforded by parallel computing. It turns out that the implementation of `ALT` is rather tricky; we do not base the semantics on the details of the standard implementation of Occam on transputers.

We interleave the rules of `ALT` with explanations. The Occam Tutorial [8] holds that "Because of this power, and because it is unlike anything in conventional programming languages, `ALT` is far-and-away the most difficult of the occam constructions to explain and to understand." We feel that an algebraic operational treatment might help people to grasp the `ALT` construction.

The children of an `ALT` or `PRI ALT` node are of type `GUARDED ALT`. A node of type `GUARDED ALT` has three children, the first of type `BOOL EXPR`, the second of type `INPUT`, and the third an arbitrary node of type `SEQ` or `PAR`, or one of the other types. It is customary to delete the `BOOL EXPR` node when the expression is `true`. (Sometimes there is no need for the `INPUT` node. In that case, a special `SKIP` node is used instead. For simplicity, we ignore this possibility.)

(30) If (type(p) = ALT or type(p) = PRI ALT) and p is starting,
then p changes to administrating, and for all children q of p, q changes to starting,
and for all children q of p, first-child(q)changes to starting,
and for all $x \in$ var(first-child(q)), $\mathsf{val}_{\mathsf{first\text{-}child}(q)}(x) := \mathsf{val}_p(x)$.

A node p of type ALT or PRI ALT starts and then goes immediately to administrating mode. At some later time, p may assume the working mode. Our discussion of rule (34) contains an explanation of why the new mode administrating is needed. In the starting mode, a GUARDED ALT node starts its first child, which is of type BOOL EXPR. When the BOOL EXPR node reports, there are two cases, depending on whether the INPUT has a corresponding OUTPUT in the $\mathcal{TREE}$, or whether it is an INPUT from the outside.

(31) If type(q) = GUARDED ALT, r = first-child(q), s = second-child(q), channel(t, s),
q is starting, t is ready, r is reporting, and val_r(expr) = true,
then q changes to ready, and r changes to dormant.

(32) If type(q) = GUARDED ALT, r = first-child(q), s = second-child(q),
$\neg(\exists t)$channel(t, s), q is starting, r is reporting, and val_r(expr) = true,
then q changes to ready, and r changes to dormant.

Compare (31) and (32). In (31), we made sure that a source is ready. In (32), we can't do the same. This reflects the intuition that the source of information, the outside world, is supposed be ready.

Eventually, one or more of the GUARDED ALT children may become ready. (It is of course possible that none of the inputs become ready, and then the overall ALT is stuck. This is as it should be. One can use timing commands to avoid this possibility. For brevity, we do not treat real-time aspects of Occam in this paper, but they are definitely amenable to treatment by algebraic operational semantics.) It is up to the parent to select one child to proceed. The method of selection depends on whether the parent is of type ALT or PRI ALT. In the case of PRI ALT, the older child is chosen. To account for this we assume that $\mathcal{TREE}$ has a relation older than. This relation holds if the two nodes have the same parent and the first is an older sibling of the second.

(33) If type(p) = PRI ALT and p is administrating, p = parent(q), q is ready,
and $\neg(\exists q')(q'$ is older than q and q' is ready),
then p and q change to working.

(34) If type(p) = ALT and p is administrating, p = parent(q), and q is ready,
then p and q change to working.

Rule (34) is unusual for us; it is non-deterministic. Our definition of a run demands a *partition into transitions*. Since rule (34) changes the mode of p to working, it insures that when p is administrating and more than one child is ready, then only one ready child evolves to working.

Before going on, we remark that if a GUARDED ALT never becomes ready, or if it becomes ready but is not chosen, then at some point it must become dormant again. This issue will be addressed below. (See rule (37) below.)

(35) If $\mathsf{type}(q) = \texttt{GUARDED ALT}$, $s = \mathsf{second\text{-}child}(p)$, $t = \mathsf{last\text{-}child}(p)$,
q is working, and s and t are dormant,
then s changes to starting.

The reason that (35) demands that t be dormant is that the next rule sets s to dormant and keeps q working. So if we didn't mention t in (35), s would start again after it received input.

(36) If $\mathsf{type}(q) = \texttt{GUARDED ALT}$, $s = \mathsf{second\text{-}child}(q)$, $t = \mathsf{last\text{-}child}(p)$,
$u = \mathsf{first\text{-}child}(s)$, q is working, and s is reporting,
then for all $x \in \mathsf{var}(u)$, $\mathsf{val}_t(x) := \mathsf{val}_u(x)$,
for all $x \in \mathsf{var}(t) - \mathsf{var}(u)$, $\mathsf{val}_t(x) := \mathsf{val}_q(x)$,
t changes to starting, and s changes to dormant.

This rule describes how a GUARDED ALT starts its process child. Note that $\mathsf{type}(u) = \texttt{ASSIGNEE}$, so $\mathsf{var}(u)$ is a singleton, say $\{x\}$. This rule insures that the value of x which was just input is passed to the process child t, In addition, t receives values of all other variables from q.

We come to the rule which tells when an ALT or PRI ALT process p evolves to reporting mode. Of course, it is necessary for the process child t of the chosen GUARDED ALT child q to have assumed reporting mode. But it is also necessary that all of the BOOL EXPR grandchildren of p be reporting. (These were started when the children of p were starting.) Since Occam has no interrupt mechanism, we permit the evaluation of the boolean expressions to run their courses.

(37) If ($\mathsf{type}(p) = \texttt{ALT}$ or $\mathsf{type}(p) = \texttt{PRI ALT}$), $p = \mathsf{parent}(q)$, $t = \mathsf{last\text{-}child}(q)$,
p is working, t is reporting,
and for all children q' of p, $\mathsf{first\text{-}child}(q')$ is reporting,
then p changes to reporting, for all $x \in \mathsf{var}(q)$, $\mathsf{val}_p(x) := \mathsf{val}_t(x)$,
and for all children q' of p, q' and $\mathsf{first\text{-}child}(q')$ change to dormant.

8 Conclusion

The main goal of this paper has been to generalize algebraic operational semantics to distributed programming languages, using Occam as an example. Although we did work out the semantics of a large fragment of Occam, we are not wed to all the details of our formalization; what we believe in is the strength of the method. This work led us to formalization of several key ideas for distributed computing: Individual small sequential parts of a distributed structure working independently, the existence of a modest collection of transition rules, of a relatively simple character, sufficient for all programs, etc. This supports our intuition that abstract machines are useful mathematical models of programming languages like Occam. Elsewhere we will discuss applications of this study.

9 References

[1] Blakley, R., Ph. D. Thesis, University of Michigan. (In preparation).

[2] Börger, E., A Logical Operational Semantics for Full Prolog, these Proceedings.

[3] Gurevich, Y., Logic and the Challenge of Computer Science. In **Trends in Theoretical Computer Science** (E. Börger, ed.), Computer Science Press, 1988, 1–57.

[4] Gurevich, Y. and J. M. Morris, Algebraic Operational Semantics and Modula-2. In **Proceedings, Logik in der Informatik**, Springer LNCS, vol. 329, pp. 81-101.

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

[6] Hoare, C. A. R. and A. W. Roscoe, The Laws of Occam Programming, Oxford University Computing Laboratory Technical Monograph PRG–53, 1986. Also appears in Theoretical Computer Science **60** (1988), pp. 177–229.

[7] Milner, R., **A Calculus of Communicating Systems**, Springer LNCS vol. 92, 1980.

[8] Pountain, D., **A Tutorial Introduction to OCCAM Programming**, INMOS Ltd, 1987.

[9] Roscoe, A. W., Denotational Semantics for Occam, in S. D. Brookes, et al (eds.), **Seminar on Concurrency** Springer LNCS 197, 1985, 306–329.