

# Occam: Specification and Compiler Correctness

## Part I: The Primary Model

Egon Börger<sup>a</sup> and Igor Đurdanović<sup>b</sup> and Dean Rosenzweig<sup>c</sup>

<sup>a</sup>Dipartimento di Informatica, Università di Pisa, Cso Italia 40, I-56100 Pisa  
email: boerger@di.unipi.it

<sup>b</sup>University Paderborn, FB 17 – Informatik, Warburgerstr. 100, D-33098 Paderborn  
email: igor@uni-paderborn.de

<sup>c</sup>University of Zagreb, FSB, Salajeva 5, 41000 Zagreb, Croatia  
email: dean@math.hr

We develop several simple operational models of Occam at different levels of abstraction, and relate them by relative correctness proofs, aiming at a transparent mathematical correctness proof for a general compilation scheme of Occam programs on the Transputer. Starting from a *primary* truly concurrent model of the language, we refine its salient concurrent features — communication, parallelism and alternation — to an abstract notion of processor, running a queue of processes, still close to the abstraction level of atomic Occam commands. The specification is effected within the framework of *evolving algebras* of Gurevich, relying on the theory of concurrency developed recently within that framework by Glavan and Rosenzweig. The model lends itself naturally to refinement down to the abstraction level of Transputer Instruction Set architecture, foreseen for a sequel to this paper.

Keyword Codes: C.1.0; C.2.4; D.3.1

Keywords: Processor Architectures, General, Distributed Systems; Software, Formal Definitions and Theory

### 1. INTRODUCTION

Gurevich [9] has introduced *evolving algebras* into semantics in order to study the dynamic and resource-bounded aspects of computation on their own terms. The concept has turned out to be a remarkably successful tool for formal specification of complex systems through hierarchies of abstraction levels, stepwise refined. The reader might look e.g. at simple but precise modeling of full fledged programming languages such as C [10] and Prolog [3] provided by evolving algebras at various levels of abstraction. In particular in [4] a formal specification of the Warren Abstract Machine has been derived — refining stepwise the formal Prolog specification of [3] — and used to prove the correctness of a general compilation scheme for Prolog programs on the WAM. Here we use the Glavan–Rosenzweig concurrency theory (developed in the meantime within the framework of

evolving algebras [7]) which allows us to extend the methodology to a mathematical correctness proof for a general compilation scheme of Occam programs on the Transputer [13], [14], [19], wrt a truly concurrent model of the language.

To justify fully the ultimate correctness claim, we start from a *primary, high level, truly concurrent* operational semantics for Occam. The model is ‘primary’ in the sense that it is intended to capture directly, in a mathematical form, the intuitive programmer’s view of the language and its dynamics.

This is not to say that we would accept any particular implementation as being a definition of the language. It is the other way round:

... unless there is a prior, generally accepted mathematical definition of a language at hand, who is to say whether a proposed implementation is *correct*? [18, p. 2]

If the development of a primary model releases us from any obligation of proof — it however places us under a (much more severe) obligation to abstract into mathematical form the central common ideas underlying current implementations and verbal descriptions. Here the challenge is that of *adequacy*, rather than correctness. Thus the model has to be transparent; the central common ideas should be recognizable by inspection, so to say. This implies in particular that basic concepts have to be expressed directly, without encoding, taking the objects of the language as abstract entities, such as they appear there.

The definition of the primary Occam model  $Occam_0$  is presented in Section 2. It is based on flowcharts of Occam programs. This allows to relegate the standard (sequential and syntax-directed) part of control to the graph structure<sup>1</sup>; it thus lets the dynamics of Occam’s *distributed features* stand out explicitly in the rules through which we make these features mathematically precise. The rules govern evolution of *processes*, which we like to view as represented by agents or *daemons* walking around the graph, each carrying along his own environment. Due to the PAR-construct the daemons may be created and deleted, and they move in a truly concurrent way, independently from each other (unless they synchronize by communication), each at its own pace, with its own notion of time.

Since it is parallelism, communication and alternation we are interested in, we remain aloof from the exact syntax and evaluation of Occam expressions, assuming explicitly only some standard abstract properties. Due to space limits we also skip the datatypes of Occam, declarations and procedures, as they are not in any way characteristic for the language (and can be incorporated into the treatment in a standard way).

Note that communication in Occam is *synchronous*. In the primary model communication is effected in one blow, while waiting for a ready communicating partner is left implicit, as built into the evolving algebra execution mechanism [9]. Our primary model is thus a ‘purely high-level programmer’s view’, closer to the CSP background of Occam than to its Transputer implementation. An implementation-wise programmer may thus find one of the following refinements to be closer to his intuition; of course they become more complex, resembling the Transputer implementation.

Given the basic model, we develop a series of refinements which provide smooth provably correct transition to the Transputer Instruction Set architecture. In this Part I the

<sup>1</sup>The description in [11] is parse tree based. This creates a hierarchy of processes (“children reporting to parent”) and thereby introduces a control structure which is, in general, not present in program execution.

elaboration is confined to communication, parallelism and alternation, leaving Occam expressions, values and environments on the abstraction level of the language. At each step a simple proof of *correctness and completeness* wrt to the basic model is given. For the benefit of a reader interested in *models* rather than *proofs*, we postpone the latter to Section 4 which presupposes the theory of [7]. The refinements are developed in Section 3. Each refinement comes by refining the signature and/or the rules and by relating correspondent runs of the two algebras.

The first refinement step provides an implementation of channel communication which assigns to channels an active role (representing abstractly the external channels of the Transputer). The second refinement introduces an abstract notion of *processor* as a partition class of daemons sharing a store. For communication within a processor channels are optimized to ‘internal’ ones, under the assumption that both reader and writer environment are immediately available. Then we finally let the processors become sequential, running a queue of processes all of which then share the same timer. A sequential processor runs concurrently to other processors and external channels. The notion of *priority* and the completeness-preserving device of *time-slicing* are introduced on this level of refinement. The refinements are proved correct and complete under the usual Occam assumptions on usage of channels. Thus we prove:

**Main Theorem.** The sequential implementation  $Occam_s$  of  $Occam_0$  with time-slicing is correct and complete.

Due to space limits, for the notion of evolving algebras we refer to [9]. The specification of Occam semantics can nevertheless be understood because evolving algebra rules can easily be read as ‘pseudocode’ over abstract data. We only remind the reader that in the rules below, the updates are thought to be executed simultaneously. For comparison of the evolving algebra approach to SOS [16] see the introduction to [3].

## 2. OCCAM — A TRULY CONCURRENT MODEL

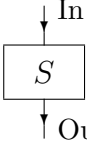
Our primary model of Occam is abstract, truly concurrent and machine-independent. Occam objects, expressions and their evaluation are represented abstractly. We concentrate on the fragment of Occam consisting of **assignment**, **time**, **stop**, **skip**, **seq**, **while**, **if**, **communication**, **alt** (including guarded alternation with time-delay) and **par**. The syntax-directed and sequential aspects of Occam are rendered trivial, by relying on the usual flowchart-scheme. The dynamics of Occam is represented by agents (or processes, or ‘daemons’, as we prefer to call them) walking around the flowchart, each carrying along his own environment, executing commands associated to the nodes.

### 2.1. Flowcharts, or Transition Diagrams of Occam Programs

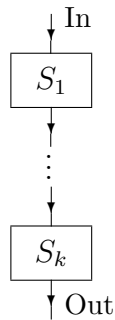
The flowcharts are given as directed graphs, with nodes taken  $i$  from a universe  $NODE$ , and the edges represented by a partial function  $next : NODE \times \mathbf{N} \rightarrow NODE$ . The nodes come decorated by *instructions*, which may be either atomic Occam statements (**assignment**, **time**, **input**, **output**, **skip**, **stop**), Occam booleans, or pseudoinstructions of form **end**, **alt**( $\vec{G}$ ) (where  $\vec{G}$  is a sequence of guards), **par**  $k$ ,  $k \in \mathbf{N}$ .

To describe this layout of Occam syntax into flowchart dynamically, we may allow also composite Occam programs as instructions. Then single steps of flowchart generation are

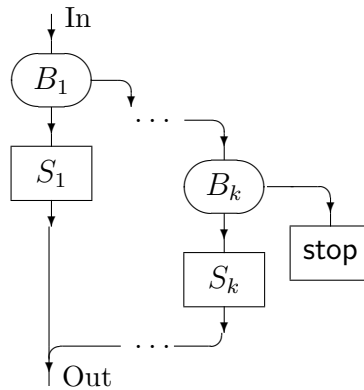
given by the following pictures (noting that nodes decorated by boolean tests are depicted as ovals).

Any configuration  gets transformed by applying successively the following steps.

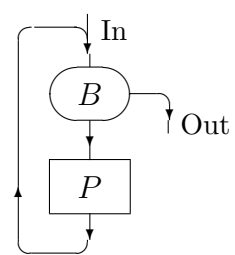
$S = \text{seq } S_1 \dots S_k$



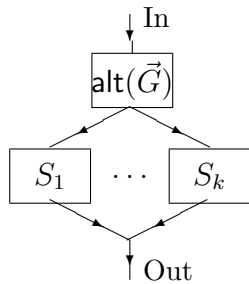
$S = \text{if } B_1 S_1 \dots B_k S_k$



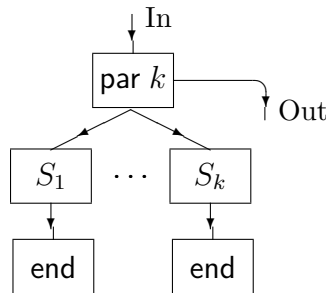
$S = \text{while } B P$



$S = \text{alt } G_1 S_1 \dots G_k S_k$



$S = \text{par } S_1 \dots S_k$



The flowcharts will be accessed using the abbreviations:  $next(n) \stackrel{\text{def}}{=} next(n, 0)$ ,  $yes(n) \stackrel{\text{def}}{=} next(n, 0)$ ,  $no(n) \stackrel{\text{def}}{=} next(n, 1)$  where the instruction decorating a node is given by a function  $cmd$  defined on  $NODE$ . We refer to the appendix for a simple formalization of flowchart generation in form of a ‘parsing’ evolving algebra<sup>2</sup>.

### 2.2. Dynamics of Occam

To represent Occam dynamics, we introduce a universe  $DAEMON$  of agents (processes), which walk around the graph carrying their own environments, and may also be sleeping, as given by:  $loc : DAEMON \rightarrow NODE$ ,  $env : DAEMON \rightarrow ENV$ ,  $mode : DAEMON \rightarrow \{running, sleeping\}$ .

The function  $loc$  represents the ‘program counter’ of the daemon, the place he is just visiting in the flowchart. The function  $env$  represents the current environment of the daemon. Since we want the dynamics of Occam’s *distributed* features — parallelism,

<sup>2</sup>In [15] slight variants of the above pictures are interpreted as compiling occam statements into field-programmable gate arrays.

communication, alternation — to stand out explicitly, we abstract from details of expression syntax and their evaluation in environments and stores by using abstract functions:  $bind : ID \times ENV \rightarrow VAR \cup CHANNEL$ ,  $eval : EXP \times ENV \rightarrow VAL$  which abstractly represent *binding* of identifiers in a given environment, and *evaluation* of expressions in a given environment.

In writing the rules, we will use the following abbreviations: ‘ $x$  does  $C$ ’, ‘proceed  $x$ ’ abbreviate, respectively, the condition ‘ $mode(x) = running \wedge cmd(loc(x)) = C$ ’ and update ‘ $loc(x) := next(loc(x))$ ’. We write  $\bar{c}$  instead of  $bind(c, e)$  where we suppose the environment  $e$  to be clear from the context.

The dynamics of sequential control in our base model is then given by the following rules for **Sequential control**:

$\boxed{\text{skip}(x)}$ <p><b>if</b> <math>x</math> does <b>skip</b> <b>then</b> proceed <math>x</math></p>	$\boxed{\text{stop}(x)}$ <p><b>if</b> <math>x</math> does <b>stop</b> <math>mode(x) := sleeping</math></p>
$\boxed{\text{ass}(x)}$ <p><b>if</b> <math>x</math> does <math>v := t</math>  <b>then</b> write <math>eval(t, env(x))</math> to <math>x</math> at <math>v</math>  proceed <math>x</math></p>	$\boxed{\text{time}(x)}$ <p><b>if</b> <math>x</math> does <b>TIME</b> ? <math>v</math>  <b>then</b> write <math>timer(x)</math> to <math>x</math> at <math>v</math>  proceed <math>x</math></p>
$\boxed{\text{if}(x, b)}$ <p><b>if</b> <math>x</math> does <math>b</math>  <b>thenif</b> <math>eval(b, env(x))</math> <b>then</b> <math>loc(x) := yes(loc(x))</math> <b>else</b> <math>loc(x) := no(loc(x))</math></p>	

As said above in the primary model for Occam we want to abstract from details of environment, binding of variables and channels and of expression evaluation. Therefore instead of updates of form  $eval(v, env(x)) := \alpha$  we use the intuitive verbal equivalent “write  $\alpha$  to  $x$  at  $v$ ”. The external function<sup>3</sup>  $timer : DAEMON \rightarrow \mathbf{N}$  indicates the local time of each daemon. For each  $x$ , the value  $timer(x)$  is supposed to grow.

**Communication** in Occam is via channels — here elements of an abstract domain *CHANNEL* — each of which requires exactly one reader and one writer for the communication to take place. After some hesitation we have chosen to start with what seems to be the prevailing viewpoint of Occam programmers, i.e. with instantaneous channel communication. This is reflected in our communication rule below: if one communication partner is not ready, then the rule is simply not applicable, and the ready partner is standing still. (We shall make this waiting more explicit in the next section, where we refine it to a more implementation-oriented view of channel.)

$\boxed{\text{com}(x, c, v; y, d, t)}$	<p><b>if</b> <math>x</math> does <math>c?v \wedge y</math> does <math>d!t \wedge \bar{c} = \bar{d}</math>  <b>then</b> write <math>eval(t, env(y))</math> to <math>x</math> at <math>v</math>, proceed <math>x</math>, proceed <math>y</math></p>
--	---

where  $\bar{c}, \bar{d}$  mean the binding of (channel) identifier  $c, d$  in  $env(x), env(y)$  respectively.

**Alternation:** In executing an alt-instruction the daemon can proceed if at least one of the guards is satisfied. A guard is either boolean or boolean plus time requirement or

<sup>3</sup>The notion of ‘external’ function is the evolving algebra way of describing an interface with the ‘outside world’ — its values are to be considered as *not* determined by our rules or initial state (cf. [9] for discussion), but might be subject to change due to actions of the environment.

boolean with an input request. This is reflected by the following rules. Instead of writing three rules which differ only in parts  $p_i$  of their guards and in updates  $u_i$ , we write one rule of form:

$$\begin{array}{c} \mathbf{if} \ p_1 \mid p_2 \mid p_3 \\ \quad \wedge \ \dots \\ \mathbf{then} \ u_1 \mid u_2 \mid u_3 \\ \quad \dots \end{array}$$

$$\boxed{\mathbf{alt\_com}(\vec{G}, i; x; y, d, t)} \mid \boxed{\mathbf{alt\_time}(\vec{G}, i; x)} \mid \boxed{\mathbf{alt\_skip}(\vec{G}, i; x)}$$

**if**  $x$  does  $\mathbf{alt}(G_1, \dots, G_k)$   
 $\wedge G_i = b : c?v \mid G_i = b : \mathbf{TIME?AFTER} \ t \mid G_i = b : \mathbf{SKIP}$   
 $\wedge \mathit{eval}(b, \mathit{env}(x))$   
 $\wedge y$  does  $d!t \wedge \bar{c} = \bar{d} \mid \mathit{timer}(x) > \mathit{eval}(t, \mathit{env}(x)) \mid$   
**then**  
 $\mathit{loc}(x) := \mathit{next}(\mathit{loc}(x), i)$   
write  $\mathit{eval}(t, \mathit{env}(y))$  to  $x$  at  $v$ , proceed  $y \mid \mid$   
**where**  $\vec{G} = G_1, \dots, G_k$

**Parallelism:** A daemon executing  $\mathbf{par} \ k$  spawns  $k$  ‘child-daemons’ and goes to sleep at the next location. Child daemons will vanish when reaching an  $\mathbf{end}$ ; the *father* may wake up as soon as the countdown of children expected to terminate, as recorded by *count*, is through. Therefore in the rules for  $\mathbf{PAR}$  we use two additional functions: *father* :  $\mathbf{DAEMON} \rightarrow \mathbf{DAEMON}$ , *count* :  $\mathbf{DAEMON} \rightarrow \mathbf{Z}$ .

$\boxed{\mathbf{par}(x, k)}$ <p><b>if</b> <math>x</math> does <math>\mathbf{par} \ k</math>  <b>then</b> create <math>x_1 \dots x_k</math>  <math>\dots</math>  <math>\mathit{mode}(x_i) := \mathit{running}</math>  <math>\mathit{father}(x_i) := x</math>  <math>\mathit{loc}(x_i) := \mathit{next}(\mathit{loc}(x), i)</math>  <math>\mathit{env}(x_i) := \mathit{env}(x)</math>  <math>\dots</math>  <math>\mathit{count}(x) := k</math>  put <math>x</math> asleep at <math>\mathit{next}(\mathit{loc}(x))</math></p>	$\boxed{\mathbf{end}(x)}$ <p><b>if</b> <math>x</math> does <math>\mathbf{end}</math>  <b>then</b> <math>\mathit{count}(\mathit{father}(x)) - = 1</math>  delete <math>x</math></p> $\boxed{\mathbf{count}(x)}$ <p><b>if</b> <math>x</math> sleeps <math>\wedge \mathit{count}(x) = 0</math>  <b>then</b> <math>\mathit{count}(x) := -1</math>  wake up <math>x</math></p>
--	---

Here ‘ $x$  sleeps’ abbreviates  $\mathit{mode}(x) = \mathit{sleeping}$  and  $\mathit{count}(\mathit{father}(x)) - = 1$  stands for decrementing a distributed counter, as discussed in [7]. Different child-daemons can terminate independently. Since the effect of their execution consists in modifying the unique store (suppressed at this level of abstraction), no result needs to be explicitly communicated back to the father.

In the *initial state* we assume the main daemon, *Demiurge* (of undefined *father*), to be at *Begin* of code — the program will *terminate* when *Demiurge* arrives to its *End*. Note that all children of *Demiurge* will have vanished by that time. Any nonterminating state in which no rule is applicable means *deadlock*<sup>4</sup>. The initial state is thus completely

<sup>4</sup>If the guard of a rule depends on external function such as *timer*, this should be extended to the guard of the rule remaining false for all possible values of external function.

determined by the program. Talking about runs, we shall in the sequel tacitly understand runs started in such a static algebra.

Let us note the form that the usual assumptions on Occam programs take in our model.

**Channel Assumption.** Any two daemons, which in any two states would both input or both output on the same channel, are connected by a *father* chain.

**Shared Variables Assumption.** If a daemon changes the value of a variable (by assignment or input), then any other daemon which either uses or changes the value of that variable is connected to it by a *father* chain.

Therefore they cannot be simultaneously *running*. These assumptions are in no way necessary for our algebra to run — it could accommodate other notions of concurrent programming. Under the above assumptions, however, it is easy to see that the above *Occam*<sub>0</sub>-algebra enjoys the following independence property (for the notion of independence defined in [7] which roughly speaking says that two rules are independent if none of them modifies what the other uses).

**Independence Property.** Any two simultaneously possible rule instances are independent unless they are of form  $\text{alt}_{\dots}(\vec{G}, i; x \dots)$  with the same parameters  $\vec{G}$  and  $x$ .

The model is thus truly concurrent. *Nondeterminism* in a strong sense, i.e. one not reducible to arbitrariness of interleaving of independent actions, is fully confined to alternation.

### 3. REFINING THE MODEL

In this section we start a sequence of refinements, intended to lead, in a sequel to this paper, to a mathematical model of the Transputer Instruction Set architecture. Every step of refinement comes together with a local proof of *correctness and completeness* wrt the preceding abstraction level—the local proofs shall compose to a mathematical correctness proof for a general compilation scheme of Occam programs on the Transputer.

In this section we effect three steps of refinement. In 3.1. we provide an *implementation* of the abstract programmer's view of channel communication and alternation; communication partners are permitted to arrive asynchronously to the point of synchronization — channels which thus become active. In 3.2. we introduce an abstract notion of *processor*, and the related distinction between *internal* and *external* channels. In 3.3. the algebra modeling a single processor becomes sequential, mimicking abstractly the Transputer implementation of parallelism by a *queue*. It is here that the notions of *priority* and *global timing* are introduced, as well as the related completeness-preserving mechanism of *time-slicing*.

#### 3.1. Descheduling Processes

*Communication.* If  $x$  does  $c?v$  with no input available or  $c!t$  with nobody listening, no rule is applicable to  $x$ . This reflects the abstract programmer's view of the previous section; other daemons proceed independently. However, such an  $x$  would have to be descheduled in a sequential implementation, in order to enable other daemons to execute. To smoothen the transition to sequential implementation, such a waiting daemon  $x$  will be 'descheduled' already under concurrency, by putting him explicitly to sleep. This will

have the pleasant consequence that a running daemon can indeed do something—has a rule applicable to it—go to sleep at least. Then deadlock will be equivalent to all daemons sleeping, without a way to wake anyone up.

In  $\text{Occam}_0$ , **com** (and **alt**) rules are *global*—they magically detect communication readiness of any daemons whatsoever, and the channels serve just to select the communication partners. In the new algebra  $\text{Occam}_1$ , the channels become more active—daemons will use them to announce their readiness to communicate, while the communication proper will be executed by the channels. Readiness to communicate through a channel is represented by functions:  $reader, writer : \text{CHANNEL} \rightarrow \text{DAEMON} \cup \{\text{nil}\}$ .

It is characteristic for this view of channel communication that (i) reader and writer can arrive independently to the point of synchronization, and (ii) the channel, once both reader and writer have arrived, decides independently when the communication is to take place. Therefore communication is now decomposed into three separate actions: input request, output request and firing of the channel.

A daemon wishing to communicate on a channel will indicate his wish by recording his identity at the channel, as well as the message he wants to send or the place where he wants to receive one, and go to sleep, waiting for the channel to pass the message. When a channel has both a reader and a writer recorded, it transmits the message, wakes up reader and writer, and clears his record. To model this mechanism, we need the following two functions:  $mssg : \text{CHANNEL} \rightarrow \text{VAL}$ ,  $place : \text{CHANNEL} \rightarrow \text{ID}$ .

Since in the rules for the ALT construct, we have to distinguish ‘ordinary’ input from that under alternative, we will also use a function:  $c\_mode : \text{DAEMON} \rightarrow \{\text{input}, \text{alt\_sleep}, \text{alt\_running}\}$  which takes value *input* for ordinary communication. In the rules of communication we will use the following abbreviations:

$$\begin{aligned} \text{ready } C &\stackrel{\text{def}}{=} reader(C) \neq \text{nil} \wedge writer(C) \neq \text{nil} \\ \text{idle } C &\stackrel{\text{def}}{=} reader(C) = \text{nil} \wedge writer(C) = \text{nil} \\ \text{clear } C &\stackrel{\text{def}}{=} reader(C) := \text{nil}, writer(C) := \text{nil} \end{aligned}$$

This corresponds to the description of ‘external channels’ in [19].

$$\begin{array}{ll} \boxed{\text{in}(x, c, v)} & \boxed{\text{out}(x, c, t)} \\ \text{if } x \text{ does } c?v & \text{if } x \text{ does } c!t \\ \text{then put } x \text{ asleep at } next(loc(x)) & \text{then put } x \text{ asleep at } next(loc(x)) \\ \quad reader(\bar{c}) := x & \quad writer(\bar{c}) := x \\ \quad place(\bar{c}) := v & \quad mssg(\bar{c}) := eval(t, env(x)) \\ \quad c\_mode(x) := \text{input} & \end{array}$$

$$\begin{array}{l} \boxed{\text{chan}(C)} \\ \text{if ready } C \wedge c\_mode(reader(C)) = \text{input} \\ \text{then write } mssg(C) \text{ to } reader(C) \text{ at } place(C) \\ \quad \text{wakeup } reader(C), \text{ wakeup } writer(C), \text{ clear } C \end{array}$$

Intuitively  $\text{com}(x, c, v; y, d, t)$  gets implemented as  $(\text{in}(x, c, v) \mid \text{out}(y, d, t)) \text{chan}(\bar{c})$  (see 4.1 for detailed arguments)<sup>5</sup>;  $\mid$  here indicates *independence*, including arbitrariness of

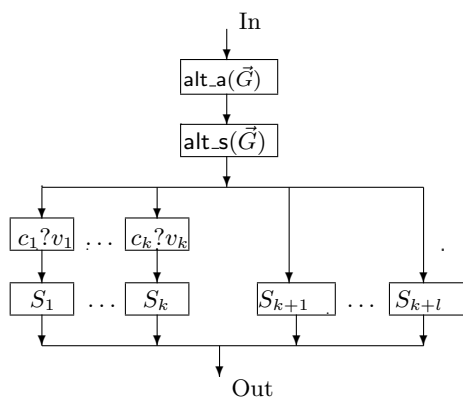
<sup>5</sup>Note that there is no need to reset  $c\_mode(reader(C))$  in the  $\text{chan}(C)$  rule because before its next use it will be updated to the appropriate value.



sequencing, rather than simultaneity (cf. [7]). Note that the Independence Property is preserved, and that it is the Channel Assumption which allows `in` and `out` to execute without asking whether *reader* resp. *writer* is *nil*.

In this implementation of alternation, the daemon doing `alt` will announce, for each of the input guards  $c_i?v_i$  allowed by their boolean conditions, its readiness to select it once the corresponding input is ready. This is done by ‘enabling’ the channel—setting its reader to oneself. Also the smallest among the time requirements `TIME ? AFTER  $t_j$`  allowed by their boolean guard is recorded into  $min\_time(x)$ , to be checked against the daemon’s current time. If none of the inputs is ready and none of the time requirements is satisfied yet, the selecting daemon goes to sleep (and records the fact by setting its ‘communication mode’ to *alt\_sleep*); it can be waken up by any relevant channel getting ready for communication (`chan_wakeup` rule) or by the daemon’s time having grown beyond  $min\_time$  (`time_wakeup` rule). If, when enabling, the daemon finds at least one input to be ready or the time requirement to be satisfied — the latter is immediately true in case of the empty requirement, denoted by `SKIP` — he proceeds immediately to selection and sets  $c\_mode$  to *alt\_running*.

Selection is then done among alternatives with ready communication or satisfied time requirement (disabling all channels involved<sup>6</sup>). The Shared Variables Assumption ensures that the values of boolean guards do not change during the execution of `alt`. The refinement of the flowchart for `alt` is as follows: `alt( $\vec{G}$ )` is ‘compiled’ to



With new abbreviations:  $enable(b, x, C) \stackrel{\text{def}}{=} \mathbf{if} \ eval(b, env(x)) \ \mathbf{then} \ reader(C) := x$ ,  $disable(b, x, C) \stackrel{\text{def}}{=} \mathbf{if} \ eval(b, env(x)) \ \mathbf{then} \ reader(C) := nil$ , the rules take the following

<sup>6</sup>Note that there is no need to reset  $min\_time(x)$  to `undef` because before its next use it will be updated in rule `alt_a` to the appropriate new value.

form where for notational convenience guards of the same type are grouped together:

$\boxed{\text{alt\_a}(\vec{G}, x)}$   
**if**  $x$  does  $\text{alt\_a}(\vec{G})$   
**then**  $\text{enable}(b_1, x, \bar{c}_1) \dots \text{enable}(b_k, x, \bar{c}_k)$ ,  $\text{min\_time}(x) := t_{\min}$   
     **if**  $\forall i (eval(b_i, env(x)) \Rightarrow \text{idle } \bar{c}_i) \wedge \text{timer}(x) \leq t_{\min} \wedge \text{not SKIP}$   
     **then** put  $x$  asleep at  $\text{next}(loc(x))$ ,  $c\_mode(x) := \text{alt\_sleep}$   
     **else** proceed  $x$ ,  $c\_mode(x) := \text{alt\_running}$   
**where**  $G_i = b_i : c_i?v_i$  ( $1 \leq i \leq k$ )  
      $t_{\min} := \min_j \{eval(t_j, env(x)) \mid eval(b_j, env(x)) = \text{true}\}$   
     ( $\infty$  if the set is empty)  
      $G_j = b_j : \text{TIME?AFTER } t_j$  ( $k+1 \leq j \leq k+m$ )  
     SKIP  $\stackrel{\text{def}}{=} \text{there is at least one SKIP in the guards,}$   
     i.e.  $m < l$  for  $G_{j'} = b_{j'} : \text{SKIP}$  ( $k+m+1 \leq j' \leq k+l$ )

$\boxed{\text{alt\_s\_com}(\vec{G}, i, x)}$  |  $\boxed{\text{alt\_s\_time}(\vec{G}, i, x)}$  |  $\boxed{\text{alt\_s\_skip}(\vec{G}, i, x)}$   
**if**  $x$  does  $\text{alt\_s}(\vec{G}) \wedge eval(b_i, env(x))$   
      $\wedge \text{writer}(\bar{c}_i) \neq \text{nil} \mid \text{timer}(x) > eval(t_i, env(x)) \mid$   
**then**  $\text{disable}(b_1, x, \bar{c}_1) \dots \text{disable}(b_k, x, \bar{c}_k)$   
      $loc(x) := \text{next}(loc(x), i)$   
**where**  $G_i = b_i : c_i?v_i \mid G_i = b_i : \text{TIME?AFTER } t_i \mid G_i = b_i : \text{SKIP}$

$\boxed{\text{chan\_wakeup}(C)}$	$\boxed{\text{time\_wakeup}(x)}$
<b>if</b> ready $C$	<b>if</b> $\text{timer}(x) > \text{min\_time}(x)$
$\wedge c\_mode(\text{reader}(C)) = \text{alt\_sleep}$	$\wedge c\_mode(x) = \text{alt\_sleep}$
<b>then</b> $c\_mode(\text{reader}(C)) := \text{alt\_running}$	<b>then</b> $c\_mode(x) := \text{alt\_running}$
wakeup $\text{reader}(C)$	wakeup $x$

where ‘put  $x$  asleep at  $n$ ’  $\stackrel{\text{def}}{=} mode(x) := \text{sleeping}$ ,  $loc(x) := n$  and ‘wakeup  $x$ ’  $\stackrel{\text{def}}{=} mode(x) := \text{running}$ .

In 4.1 we will prove how these new rules implement the previous  $\text{alt}$ -rules.  $\text{alt\_skip}(\vec{G}, i, x)$  gets implemented by  $\text{ALT\_SKIP} \stackrel{\text{def}}{=} \text{alt\_a}(\vec{G}, x) \text{ alt\_s\_skip}(\vec{G}, i, x)$ . For  $\text{alt\_time}(\vec{G}, i, x)$  there are two cases, depending on whether or not, when  $\text{alt\_a}(\vec{G}, x)$  is executed, process  $x$  is ready for selection (by satisfaction of a time requirement or by readiness of an input guard channel). The former case is implemented by  $\text{ALT\_TIME} \stackrel{\text{def}}{=} \text{alt\_a}(\vec{G}, x) \text{ alt\_s\_time}(\vec{G}, i, x)$  or by  $\text{out}(z, e, u) \text{ ALT\_TIME}$  for some  $G_j = b : e?u$  in  $\vec{G}$  and some  $z$ , the latter by  $\text{alt\_a}(\vec{G}, x) \text{ WAKE\_UP } \text{alt\_s\_time}(\vec{G}, i, x)$  where  $\text{WAKE\_UP}$  is  $\text{time\_wakeup}(x)$  or  $\text{out}(z, e, u) \text{ chan\_wakeup}(\text{bind}(e, env(z)))$  for some  $z, e, u$ .

Similarly, the implementation of  $\text{alt\_com}(\vec{G}, i, x; y, d, t)$  breaks into two cases:

- (i)  $\text{READY\_COM SEL\_COM}$  with  $\text{READY\_COM}$  being the composition of  $\text{out}(y, d, t)$  and  $\text{alt\_a}(\vec{G}, x)$  in any order and  $\text{SEL\_COM} \stackrel{\text{def}}{=} \text{alt\_s\_com}(\vec{G}, i, x) \text{ in}(x, c_i, v_i) \text{ chan}(\bar{c}_i)$ ,
- (ii)  $\text{alt\_a}(\vec{G}, x) \text{ WAKE\_UP SEL\_COM}$  where  $\text{WAKE\_UP}$  is  $\text{time\_wakeup}(x)$  or  $\text{out}(y, d, t) \text{ chan\_wakeup}(\bar{c}_i)$  or  $\text{out}(z, e, u) \text{ chan\_wakeup}(\bar{c}_j) \text{ out}(y, d, t)$  for some  $z, e, u, j$ . Note that in the last case  $j$  is one of the alternatives enabled by  $\text{alt\_a}$  (given that  $\text{bind}(e, env(z)) = \bar{c}_j$ )

with  $j \neq i$ —if several alternatives become available, nothing compels the `alt_s` rules to select exactly that one which has woken the daemon up.

In 4.1., following the above intuitive reasoning, we provide explicit meaning and a proof of

**Theorem 1.** The implementation, in algebra  $\text{Occam}_1$ , of communication and alternation in algebra  $\text{Occam}_0$ , is correct and complete.

### 3.2. Processors and Internal Communication

The Transputer supports concurrent execution of several processes on one processor. Here a ‘processor’ will be an abstract object, element of a domain  $PROCESSOR$ , and the association of processors to daemons (‘placement’ of processes) will be represented by a function  $p : DAEMON \rightarrow PROCESSOR$ . The `placed par` construct of Occam can now be realized by a rule identical to `par`, setting in addition  $p(x_i)$  to explicitly listed processors.

We concentrate here on refining the action of a *single* processor. We can thus suppress the processor from the notation.

Processes residing on the same processor will communicate by *internal* channels (implemented in the Transputer as memory locations). For internal channels the execution of communication can be optimized, since reader and writer share the store, and their environments may be assumed to be both ‘immediately available’. The daemon that wants to communicate with a nonidle internal channel may thus complete the communication in one blow, making the `chan` rule spurious. Therefore *reader* and *writer* may be merged to an *agent*. Also, *mssg* and *place* may be reasonably attached to the channel’s *agent* rather than to the channel—we thus have (homonymous) functions *mssg*, *place* with domain  $DAEMON$  (it is a memory-saving device in the Transputer implementation, allowed by the Channel Assumption).

Communication over internal channels can then be optimized in the sense that new versions of `in` and `out` rules may perform immediately also the work of `chan` and `chan_wakeup`. Rules for input and output will refine to special rules for a) input from an idle or external channel, b) input from a ready internal channel, c) output to an idle or external channel, d) output to a ready internal channel, e) output to an internal channel enabled by `alt`.

We shall occasionally, in order to avoid repetition in the rules, rely on notation and abbreviations from the ‘external channels’ model, under the convention that, for any internal channel  $C$ , *reader*( $C$ ), *writer*( $C$ ) should both be read as *agent*( $C$ ). Also, *mssg*( $C$ ), *place*( $C$ ) should, for internal  $C$ , be read as *mssg*(*agent*( $C$ )), *place*(*agent*( $C$ )) respectively. Correspondingly we have to refine the definitions of enable/disable as follows:

$$\begin{aligned} \text{enable}(b, x, c) &\stackrel{\text{def}}{=} \mathbf{if} \text{eval}(b, \text{env}(x)) \wedge \text{agent}(c) = \text{nil} \mathbf{then} \text{agent}(c) := x \\ \text{disable}(b, x, c) &\stackrel{\text{def}}{=} \mathbf{if} \text{eval}(b, \text{env}(x)) \wedge \text{agent}(c) = x \mathbf{then} \text{agent}(c) := \text{nil} \end{aligned}$$

Minding the notation from [7], of  $R?$  and  $R!$  for, respectively, guards and updates of a rule  $R$ , we have the following new in/out rules:

$$\begin{array}{ll} \boxed{\text{in\_idle}(x, c, v)} & \boxed{\text{out\_idle}(x, c, t)} \\ \mathbf{if} \ x \text{ does } c?v & \mathbf{if} \ x \text{ does } c!t \\ \wedge (\text{internal } \bar{c} \wedge \text{idle } \bar{c}) \text{ or external } \bar{c} & \wedge (\text{internal } \bar{c} \wedge \text{idle } \bar{c}) \text{ or external } \bar{c} \\ \mathbf{then} \ \text{in}(x, c, v)! & \mathbf{then} \ \text{out}(x, c, t)! \end{array}$$

<div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 5px;"><b>in_ready</b>(<math>x, c, v</math>)</div> <p><b>if</b> <math>x</math> does <math>c?v</math></p> <p style="padding-left: 20px;"><math>\wedge</math> internal <math>\bar{c} \wedge</math> not idle <math>\bar{c}</math></p> <p><b>then</b> write <math>mssg(agent(\bar{c}))</math> to <math>x</math> at <math>v</math></p> <p style="padding-left: 20px;">wakeup <math>agent(\bar{c})</math></p> <p style="padding-left: 20px;">proceed <math>x</math></p> <p style="padding-left: 20px;">clear <math>\bar{c}</math></p>	<div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 5px;"><b>out_ready</b>(<math>x, c, t</math>)</div> <p><b>if</b> <math>x</math> does <math>c!t</math></p> <p style="padding-left: 20px;"><math>\wedge</math> internal <math>\bar{c} \wedge</math> not idle <math>\bar{c}</math></p> <p style="padding-left: 20px;"><math>\wedge</math> <math>c\_mode(agent(\bar{c})) = input</math></p> <p><b>then</b> write <math>eval(t, env(x))</math> to <math>agent(\bar{c})</math></p> <p style="padding-left: 20px;">at <math>place(agent(\bar{c}))</math></p> <p style="padding-left: 20px;">wakeup <math>agent(\bar{c})</math></p> <p style="padding-left: 20px;">proceed <math>x</math></p> <p style="padding-left: 20px;">clear <math>\bar{c}</math></p>
---	---

The implementation of communication by  $(in(x, c, v) \mid out(y, d, t)) \text{ chan}(\bar{c})$  gets, in case of internal  $\bar{c}$ , optimized to  $in\_idle(x, c, v) \text{ out\_ready}(y, d, t)$  or  $out\_idle(y, d, t) \text{ in\_ready}(x, c, v)$ .

In case of output to an internal channel enabled by `alt_a`, which is then by definition also ready, the corresponding output rule is not allowed yet to do the output, but must announce its readiness as if the channel were idle, and in addition do the work of `chan_wakeup`. The corresponding rule is

**out\_alt**( $x, c, t$ )

**if**  $x$  does  $c!t \wedge$  internal  $\bar{c} \wedge$  not idle  $\bar{c} \wedge$   $c\_mode(agent(\bar{c})) \neq input$

**then** put  $x$  asleep at  $next(loc(x))$

$mssg(x) := eval(t, env(x)), agent(\bar{c}) := x$

**if**  $c\_mode(agent(\bar{c})) = alt\_sleep$  **then** wakeup  $agent(\bar{c})$

$c\_mode(agent(\bar{c})) := alt\_running$

The implementation of `alt_time`( $\vec{G}, i, x$ ) involving internal channels is the same as in 3.1, replacing in the first case `out(z, e, u)` by `out_idle(z, e, u)`, and replacing in `WAKE_UP` `out(z, e, u) chan_wakeup(bind(e, env(z)))` by `out_alt(z, e, u)`.

Also the implementation of `alt_com`( $\vec{G}, i, x$ ), for an internal selected channel, is as in 3.1 but with the following definition of `READY_COM` and `WAKE_UP`: `READY_COM` is `out_idle(y, d, t) alt_a(\vec{G}, x)` or `alt_a(\vec{G}, x) out_alt(y, d, t)`, `WAKE_UP` is `time_wakeup(x)` or `out_alt(y, d, t)` or `out_alt(z, e, u) out_alt(y, d, t)` for some  $z, e, u$ .

An explicit meaning and a proof of the following theorem can be found in 4.2.

**Theorem 2.** The refinement of `Occam1` to the algebra `Occam2` with internal channels is correct and complete.

### 3.3. Sequential Processors

Here we describe the sequential implementation of concurrent processes using a simple queue of daemons. Such a queue structure will be given by functions  $fst, last : QUEUE \rightarrow DAEMON$ ,  $rest : QUEUE \rightarrow QUEUE$ ; we denote the operation of adding a daemon to the (back of the) queue as  $q.x$  and assume the usual queue axioms. We use the abbreviations enqueue  $x, q \stackrel{\text{def}}{=} q := q.x$  and dequeue  $q, a \stackrel{\text{def}}{=} (a := fst(q), q := rest(q))$ .

A *sequential processor* may then be, at this level of abstraction, viewed as carrying a *queue* of active (non-sleeping) processes, all of them sharing the (external)  $timer : PROCESSOR \rightarrow \mathbf{N}^7$ , and a single process being really executed—its  $agent$ .<sup>8</sup> These data

<sup>7</sup>Formally this means to replace everywhere  $timer(x)$  by  $timer(p(x))$ .

<sup>8</sup>Note that at this level of abstraction, we still consider the processor and the agents for the `chan`-rule and the `wakeup`-rules to be concurrent.

will be represented by functions:  $Q : PROCESSOR \rightarrow QUEUE$ ,  $A : PROCESSOR \rightarrow DAEMON$ . Concentrating on a single processor  $P$ , we write just  $Q, A$  for  $Q(P), A(P)$ .

The *mode* of processes can now be dropped, since we view a process as running iff it is in the queue, i.e. reachable as  $fst(rest^n(Q))$  for some  $n \geq 0$ . This interpretation of ‘running’ and ‘sleeping’ will be realized if we refine the actions of ‘putting to sleep’ to ‘put  $A$  asleep at  $n$ ’ (which by definition means  $loc(A) := n$ ,  $A := nil$ ) and ‘waking up’ to ‘wakeup  $x' \stackrel{\text{def}}{=} \text{enqueue } x, Q^9$ .

Sequentiality is realized by allowing *only*  $A$  to execute (that is why only  $A$  may be put to sleep), i.e. by refining the notion of ‘doing’ an instruction to making it applicable only to  $A$ :  $x \text{ does } C \stackrel{\text{def}}{=} x = A \wedge cmd(loc(x)) = C$ .

We add a rule for dequeuing whenever  $A$  gets *nil*:

**dequeue** if  $A = nil \wedge$  not empty  $Q$  then dequeue  $Q, A$ .

Due to the fact that all processes which are run by the processor share the *timer*, *min\_time* now associates to the processor the list of pairs of daemons and the time they are waiting for, sorted according to the latter:  $min\_time : PROCESSOR \rightarrow (DAEMON \times \mathbf{N})^*$ . Correspondingly the *min\_time* update in the **alt\_a** rule becomes:  $min\_time := insert(min\_time, \langle x, t_{min} \rangle)$  (with an *insert* function which respects the ordering). The **time\_wakeup** rule is changed to:

**if**  $min\_time \neq \langle \rangle \wedge timer > t_{wait} \wedge c\_mode(d_{wait}) = alt\_sleep$   
**then**  $c\_mode(d_{wait}) := alt\_running$ , wakeup  $d_{wait}$ ,  $min\_time := delete(min\_time, d_{wait})$

where  $t_{wait}$  is the minimal waiting time in *min\_time* and  $d_{wait}$  is any daemon waiting for it.<sup>10</sup> In all **alt\_s** rules we have to add the update:  $min\_time := delete(min\_time, x)$  meaning that if there is a pair with first component  $x$  in *min\_time*, then it is deleted.

The rules for parallelism and stop now take the form:

<p><b>par</b>(<math>x, k</math>)  <b>if</b> <math>x</math> does <b>par</b> <math>k</math>  <b>then</b> create <math>x_1 \dots x_k</math>  <math>Q := Q.x_1 \dots x_k</math>  <math>\dots</math>  <math>father(x_i) := x</math>  <math>loc(x_i) := next(loc(x), i)</math>  <math>env(x_i) := env(x)</math>  <math>\dots</math>  <math>count(x) := k</math>  put <math>x</math> asleep at <math>next(loc(x))</math></p>	<p><b>end</b>(<math>x</math>)  <b>if</b> <math>x</math> does <b>end</b>  <b>then</b> <math>x := nil</math>  <math>k := k - 1</math>  <b>if</b> <math>k = 1</math>  <b>then</b> wakeup <math>father(x)</math>  <b>where</b> <math>k = count(father(x))</math></p> <p><b>stop</b>(<math>x</math>)  <b>if</b> <math>x</math> does <b>stop</b> <b>then</b> <math>x := nil</math></p>
---	--

Note that there is no more need for distributed counting.

The evolving algebra corresponding to one processor implements daemons sequentially — by allowing at most one daemon to execute his rule at any time, as can be easily verified by inspecting the guards. In 4.3. we prove the following theorem:

<sup>9</sup>Any implementation certainly dequeues the next daemon immediately. For the sake of clarity and modularity of the description, at this level of abstraction we separate ‘putting to sleep’ and dequeuing.

<sup>10</sup>Many processes might be waiting for the same time moment. At this level of abstraction we still disregard the order in which they are put into *min\_time* and from there into the queue of processes.

**Theorem 3.** The sequential implementation  $Occam_s$  of  $Occam_2$  is correct. Given non-divergence, it is also complete.

### 3.3.1. Time-slicing

The active daemon (one at  $A$ ) will remain active as long as it is not descheduled or interrupted. Since descheduling is always caused by communication, a *divergent* daemon (one that can execute infinitely many steps without ever communicating with the outside world) might run forever, preventing thereby other daemons from becoming active. This might cause incompleteness of our implementation of parallelism—some runs may not be implemented in presence of a divergent daemon (cf. 4.1).

In the Transputer such incompleteness is prevented by *time-slicing*—permitting any daemon to be active just for a finite period of time. When his time has elapsed, he is forcibly enqueued, and another daemon gets his chance.

To represent timeslicing we introduce a function *start* holding the starting time of the low priority active process (the value of *timer* when the daemon was dequeued and became active). A function *period* defines the amount of time each daemon is allowed to remain active:  $period, start \in \mathbf{N}$ .

The dequeue rules have to include updating the *start* function to (the current value of) *timer* when the new low priority daemon becomes active.

The following abbreviation is used for checking whether the active daemon has spent its time:  $\text{elapsed} \stackrel{\text{def}}{=} timer - start > period$ . We can prevent the active daemon from doing anything after its time is elapsed by refining the abbreviation ‘ $x$  does  $C$ ’ by the conjunct ‘not elapsed’. Enqueueing of the active daemon will have to be done explicitly, by a dedicated time-slicing rule:

$$\boxed{\text{time-slice}} \text{ if elapsed } \wedge A \neq nil \text{ then } Q := Q.A, A := nil$$

Since time-slicing prevents divergent behavior, theorems 1,2,3 prove the main theorem.

### 3.3.2. Priority

The notion of *priority*, embodied in the **pri par** construct of Occam, can be easily realized by associating to each process its priority, introducing, for each processor, *two* queues,  $Q_{high}$  and  $Q_{low}$  and a 0-ary function  $A_{low}$  for holding the *interrupted low priority agent*. Correspondingly *timer* and *min.time* split into low and high versions with appropriate insertion and deletion. The **par** rule should now let the children inherit also the priority of their *father*, and the enqueue update is refined to enqueue each process according to its priority:

$$\text{enqueue } x \stackrel{\text{def}}{=} \text{ if } priority(x) = \text{high} \text{ then } Q_{high} := Q_{high}.x \text{ else } Q_{low} := Q_{low}.x.$$

Priority is usually explained by ‘low priority process being allowed to execute only if no high priority process is running’. For this purpose the notion of ‘doing’ should be refined yet another time, as

$$x \text{ does } C \stackrel{\text{def}}{=} x = A \wedge cmd(loc(x)) = C \\ \wedge (priority(A) = \text{high} \vee (\text{empty } Q_{high}) \wedge \text{not elapsed})$$

refining also the **dequeue** rule so as to reflect the priority, and adding a rule to interrupt a low priority  $A$  when  $Q_{\text{high}}$  is not empty:

```

dequeue
if  $A = nil$  thenif not empty  $Q_{\text{high}}$ 
    then dequeue  $Q_{\text{high}}, A$ 
    elseif  $A_{\text{low}} \neq nil$ 
        then  $A := A_{\text{low}}, A_{\text{low}} := nil$ 
        elseif not empty  $Q_{\text{low}}$  then dequeue  $Q_{\text{low}}, A$ 

interrupt
if  $priority(A) = low \wedge$  not empty  $Q_{\text{high}}$  then  $A := nil, A_{\text{low}} := A$ 

```

Since time-slicing is done only for low-priority processes, the time-slicing rule is refined by adding the conjunct  $priority(A) = low$  to the guard and by replacing  $Q$  by  $Q_{\text{low}}$ . By the above interpretation of ‘does’, a low-priority daemon is frozen as soon as a high-priority daemon is enqueued—the only action possible in such a situation is **interrupt** followed by **dequeue**, which ‘store the state’ and ‘service the interrupt’.

#### 4. RELATING THE MODELS

In this section we prove the theorems stated above, using the framework of [7].

We shall say that (possible) runs  $\rho, \sigma$  are *equivalent*,  $\rho \sim \sigma$ , if  $\langle \rho \rangle \phi \Leftrightarrow \langle \sigma \rangle \phi$  for any formula  $\phi$ .

*Sequentialization* of runs is defined by: the sequentialization of a simple rule  $R$  is  $R$ ; if  $\rho', \sigma'$  are any sequentializations of, respectively,  $\rho, \sigma$ , then  $\rho' \sigma'$  is a sequentialization of  $\rho \sigma$ , and any interleaving of  $\rho', \sigma'$  is a sequentialization of  $\rho \mid \sigma$ .

*Strong equivalence*,  $\simeq$ , is the smallest equivalence relation which puts any run together with all its sequentializations.

Strongly equivalent runs are equivalent (see [7]). Under strong equivalence we forget rearrangements of actions allowed by independence. These notions are, of course, always to be understood as relative to a static algebra in which the runs considered are possible.

We shall relate runs of different Occam models by ‘implementation maps’ mapping runs of the ‘more concrete’ algebra to those of the ‘more abstract’ one. We always consider runs possible in the initial states, determined by Occam programs. Implementation maps will be constructed incrementally, proceeding in a uniform way, by defining an ‘increment map’:  $\Delta f : B \times B^* \rightarrow A^*$  where  $A^*, B^*$  are, respectively, sets of runs of the abstract algebra  $A$  and concrete algebra  $B$ , so that  $\Delta f(R, \rho)$  will be defined whenever  $\langle \rho R \rangle$ . Note that, for any nonempty  $\rho$  ( $\sigma$ ), we have  $\rho \simeq R\rho'$  and  $\rho \mid \sigma \simeq (R \mid S)(\rho' \mid \sigma')$  for some  $R, \rho', (S, \sigma')$ , as soon as the left hand sides are possible. In order to define an implementation map  $f : B^*/\simeq \rightarrow A^*/\simeq$  it will then suffice to stipulate

$$f(\varepsilon) = \varepsilon \quad f(\rho R) = f(\rho) \Delta f(R, \rho) \quad f(\rho(R \mid S)) = f(\rho) (\Delta f(R, \rho) \mid \Delta f(S, \rho))$$

as soon as we establish, inductively, the **Increment Properties**

$$\langle \rho R \rangle \Rightarrow \langle f(\rho) \Delta f(R, \rho) \rangle \quad \langle \rho \rangle I(R, S) \Rightarrow \langle f(\rho) \rangle I(\Delta f(R, \rho), \Delta f(S, \rho)).$$

By simple induction we can then see that  $f$  preserves possibility and independence,

$$\langle \rho \rangle \Rightarrow \langle f(\rho) \rangle \quad I(\rho, \sigma) \Rightarrow I(f(\rho), f(\sigma)).$$

The  $f$ 's relating our Occam algebras will also preserve the sets of daemons together with their environments and locations, communication traces, termination, deadlock and divergence.

In this context we say that a run  $\rho$  of  $Occam_0$  is *convergent* if, for some  $k$  and any continuation  $\tau$  of  $\rho$  of length  $\geq k$  one of the following holds:

$$\rho\tau \simeq \rho\tau_1 \text{com}(x, c, v; y, d, t)\tau_2 \quad \rho\tau \simeq \rho\tau_1 \text{alt}(\vec{G}, i; x; y)\tau_2.$$

A non-convergent run is *divergent*—it is possible, after a divergent run, to run indefinitely (by König's lemma) without ever again communicating with the outside world. A program is divergent if it has a divergent run.

#### 4.1. Descheduling Processes

In view of the Independence Property and the Channel Assumption we have:

**Lemma 1.1.** If, in  $Occam_1$ ,  $\langle \rho \text{alt\_s\_time}(\vec{G}, i, x) \rangle$ , then for some  $\sigma$  and with WAKE\_UP as defined in 3.1., one of the following holds:

- (1)  $\rho \simeq \sigma \text{alt\_a}(\vec{G}, x)$  or  $\rho \simeq \sigma \text{out}(z, e, u) \text{alt\_a}(\vec{G}, x)$  for some  $G_j = b : e?u$  in  $\vec{G}$  and  $z$
- (2)  $\rho \simeq \sigma \text{alt\_a}(\vec{G}, x)$  WAKE\_UP

**Lemma 1.2.** If in  $Occam_1$ ,  $\langle \rho \text{chan}(C) \rangle$ , then for some  $\sigma, x, c, v, y, d, t$  such that  $\text{bind}(c, \text{env}(x)) = \text{bind}(d, \text{env}(y)) = C$ , either (1)  $\rho \simeq \sigma(\text{in}(x, c, v) \mid \text{out}(y, d, t))$  or for some  $\vec{G}_i, i$  such that  $c = c_i$ , one of the following holds: (2)  $\rho \text{chan}(C) \simeq \sigma \text{READY\_COM SEL\_COM}$  or (3)  $\rho \text{chan}(C) \simeq \sigma \text{alt\_a}(\vec{G}, x)$  WAKE\_UP SEL\\_COM with READY\\_COM, SEL\\_COM and WAKE\_UP as defined in 3.1.

Since by the Channel Assumption  $x, y$  are unique,  $\Delta f : Occam_1 \times Occam_1^* \rightarrow Occam_0^*$  can be defined by:

$$\begin{aligned} \Delta f(\text{alt\_s\_skip}(\vec{G}, i, x), \rho) &= \text{alt\_skip}(\vec{G}, i, x) \\ \Delta f(\text{alt\_s\_time}(\vec{G}, i, x), \rho) &= \text{alt\_time}(\vec{G}, i, x) \\ \Delta f(\text{chan}(C), \rho) &= \begin{cases} \text{com}(x, c, v; y, d, t) & \text{in case (1) of lemma 1.2} \\ \text{alt\_com}(\vec{G}, i; x; y, d, t) & \text{in cases (2),(3) of lemma 1.2} \end{cases} \\ \Delta f(R, \rho) &= \begin{cases} \varepsilon & \text{if } R \in \{\text{in, out, alt\_a, alt\_s\_com,} \\ & \text{chan\_wakeup, time\_wakeup}\} \\ R & \text{otherwise} \end{cases} \end{aligned}$$

The Increment Properties hold and the implementation map  $f$  is well defined by Lemma 1.

**Lemma 2.** For any  $\rho$  and ALT\\_TIME, WAKE\_UP, READY\\_COM, SEL\\_COM as in 3.1:

$$\begin{aligned} \langle \rho \text{alt\_a}(\vec{G}, x) \text{alt\_s\_skip}(\vec{G}, i, x) \rangle &\Leftrightarrow \langle f(\rho) \text{alt\_skip}(\vec{G}, i, x) \rangle & (1) \\ \langle \rho \text{ALT\_TIME} \rangle &\text{or} & (2) \end{aligned}$$



$$\langle \rho \text{ alt\_a}(\vec{G}, x) \text{ WAKE\_UP alt\_s\_time}(\vec{G}, i, x) \rangle \Leftrightarrow \langle f(\rho) \text{ alt\_time}(\vec{G}, i, x) \rangle$$

$$\langle \rho (\text{in}(x, c, v) \mid \text{out}(y, d, t)) \text{ chan}(\vec{c}) \rangle \Leftrightarrow \langle f(\rho) \text{ com}(x, c, v; y, d, t) \rangle \quad (3)$$

$$\langle \rho \text{ READY\_COM SEL\_COM} \rangle \quad \text{or} \quad (4)$$

$$\langle \rho \text{ alt\_a}(\vec{G}, x) \text{ WAKE\_UP SEL\_COM} \rangle \Leftrightarrow \langle f(\rho) \text{ alt\_com}(\vec{G}, i; x; y, d, t) \rangle$$

The new ‘intermediate states’, in which some channel is not idle, do not correspond very well to any states of  $Occam_0$ . Let us then distinguish those states in which all channels are idle as *significant*. Runs which preserve significance of states, i.e. which have already completed every communication they had started, will also be called significant. It is easy to see that all maximal nondeadlocking runs, finite and infinite, are significant, and that significant runs are dense up to strong equivalence. We then have

**Proposition 3.** On significant runs  $f$  preserves communication traces, the set of daemons, their locations, environments and modes.

**Proposition 4.**  $f$  preserves termination, deadlock and divergence.

**Proposition 5.**  $f$  is surjective, i.e. every run of  $Occam_0$  is strongly equivalent to  $f(\sigma)$  for some (significant)  $\sigma \in Occam_1^*$ .

Propositions 3 and 4 establish *correctness* of ‘implementing’  $Occam_0$  by  $Occam_1$  — Proposition 5 establishes its *completeness*, in a rather strong sense. This proves Theorem 1.

## 4.2. Internal Channels

Mapping runs of  $Occam_2$  to those of  $Occam_1$  is simple: if  $R$  is one of  $\text{in}(x, c, v)$  or  $\text{out}(x, c, t)$ , we set

$$\begin{aligned} \Delta f(R\_idle, \rho) &= R & \Delta f(R\_ready, \rho) &= R \text{ chan}() \\ \Delta f(\text{out\_alt}(x, c, t), \rho) &= \text{out}(x, c, t) \end{aligned}$$

Otherwise  $\Delta f(R, \rho) = R$ . Propositions 3-5 then extend to  $Occam_2$ .

## 4.3. Sequential Processors

We relate  $Occam_s$  to  $Occam_2$  as follows.  $\Delta f(\text{dequeue}, \rho) \stackrel{\text{def}}{=} \varepsilon$ ,

$$\Delta f(\text{end}_s(x), \rho) \stackrel{\text{def}}{=} \begin{cases} \text{end}(x) & \text{if } \langle \rho \rangle(k > 1) \\ \text{end}(x) \text{ count}(y) & \text{if } \langle \rho \rangle(k = 1) \end{cases}$$

where  $y = \text{father}(x)$ .  $\Delta f(\text{time\_wakeup}, \rho)$  is defined as the sequence of  $\text{time\_wakeup}(x)$  for all  $x = d_{\text{wait}}$ .  $\Delta f(R, \rho) = R'$  for other rules, where  $R'$  is the  $Occam_2$  homonym of  $R$ .

The Increment Properties hold, and we have  $f$  preserving possibility and independence. In view of the Independence Property, (any) sequentialization shuffles the runs only up to  $\simeq$ . Since Propositions 3 and 4 are formulated in terms of  $\simeq$ , they hold also here, i.e. the sequential implementation of  $Occam_2$  is *correct*.

Without time-slicing however *completeness* would be lost. An action, possible in  $Occam_2$ , will namely have its analogon somewhere in the queue, which need not however be immediately possible, being way back in the queue. If the active daemon generates a divergent run, it might never get possible. However,

**Lemma 3.** If  $\rho$  is a run of  $Occam_s$  under time-slicing, the agent current after  $\rho$  will eventually get descheduled, i.e. if  $\langle \rho \rangle (A = x)$ , there is some  $k$  such that, for any continuation  $\sigma$  of  $\rho$  of length  $\geq k$ ,  $\rho\sigma \simeq \rho\sigma_1\sigma_2$  for some  $\sigma_1, \sigma_2$  so that  $\langle \rho\sigma_1 \rangle (A \neq x)$ .

Iterating the lemma along the queue, we obtain

**Lemma 4.** If  $\rho$  is a run in  $Occam_s$  under time-slicing, then, for some  $k$ , whenever in  $Occam_2$   $\langle f(\rho)R \rangle$ , there is in  $Occam_s$  a continuation  $\sigma$  of  $\rho$  of length  $\leq k$  such that for some  $\tau$  holds  $f(\rho\sigma) \simeq f(\rho)R\tau$ .

Iterating Lemma 4 along runs of  $Occam_2$ , we obtain

**Proposition 6.** Given time-slicing, every run of an  $Occam_2$  algebra is covered, up to strong equivalence, with runs in the image of  $f$ .

This establishes the Main Theorem.

## 5. CONCLUSION

The scope of the evolving algebra methodology is not exhausted by high-level descriptions, such as the one produced here for Occam. In [4] we have shown how such a description can be transformed (provably correctly) to a low level abstract machine like the WAM. The models of the present paper provide the starting point for further refinement all the way to a formal description of the Transputer Instruction Set architecture, accompanied by an incremental mathematical *correctness proof* for a general compilation scheme of Occam. We undertake to complete such a proof in a sequel to this paper.

Unlike formal studies of implementations of Occam sublanguages [1], [12] based on “The laws of Occam Programming” [17], we interpret the Occam programs as they are without reducing them to normal form. Note that we could have presented our rules in the form of Horn clauses and interpret them as “compiling” the described Occam constructs into Prolog; see [2] where the compiling specifications, obtained for the sequential sublanguage of Occam via “laws of Occam programming”, are transformed into logic programs.

Our proofs show the correctness of a compilation scheme. This is different from proving a concrete compiler to be correct, an approach investigated in [6].

The framework developed here could also be used to recast and compare other proposals for a “correct” implementation of Hoare’s CSP. One example is [5] where also output statements are allowed in the guards for alternative and iterative commands.

## Acknowledgements

We thank the following colleagues for criticism and useful comments on various previous versions of this paper: Jonathan Bowen, Bettina Buth, Werner Damm, Uwe Glässer, Tony Hoare, Burghard von Karger, Peter Mosses, Ernst-Rüdiger Olderog, two anonymous referees.

## 6. APPENDIX: GENERATING FLOWCHARTS

Given two nodes,  $Begin$  and  $End$ , such that  $next(Begin) = End$ , the following rules will generate the flowchart for an Occam program  $S = cmd(Begin)$  (assuming that there

are no further nodes or function values, and that  $S$  belongs to the fragment of Occam treated in the main text).

<pre> <b>if</b> <math>cmd(n) = seq\ S_1 \dots S_k</math> <b>then</b> create <math>n_2, \dots, n_k</math>     ...     <math>cmd(n_i) := S_i</math>     <math>next(n_i) := n_{i+1}</math>     ...     (<math>1 \leq i \leq k</math>) <b>where</b> <math>n_1 = n, n_{k+1} = next(n)</math>  <b>if</b> <math>cmd(n) = while\ B\ S</math> <b>then</b> <math>cmd(n) := B</math>     <math>no(n) := next(n)</math>     create <math>n_1</math>     <math>yes(n) := n_1</math>     <math>cmd(n_1) := S</math>     <math>next(n_1) := n</math>  <b>if</b> <math>cmd(n) = alt\ G_1\ S_1 \dots G_k\ S_k</math> <b>then</b> <math>cmd(n) := alt(G_1, \dots, G_k)</math>     create <math>n_1, \dots, n_k</math>     ...     <math>next(n, i) := n_i</math>     <math>cmd(n_i) := S_i</math>     <math>next(m_i) := next(n)</math>     ... </pre>	<pre> <b>if</b> <math>cmd(n) = if\ B_1\ S_1 \dots B_k\ S_k</math> <b>then</b> create <math>n_1, n_2, m_2, \dots, n_k, m_k, m_{k+1}</math>     ...     <math>cmd(m_i) := B_i</math>     <math>yes(m_i) := n_i</math>     <math>no(m_i) := m_{i+1}</math>     <math>cmd(n_i) := S_i</math>     <math>next(n_i) := next(n)</math>     ...     (<math>1 \leq i \leq k</math>)     <math>cmd(m_{k+1}) := stop</math> <b>where</b> <math>m_1 = n</math>  <b>if</b> <math>cmd(n) = par\ S_1 \dots S_k</math> <b>then</b> <math>cmd(n) := par\ k</math>     create <math>n_1, m_1 \dots, n_k, m_k</math>     ...     <math>next(n, i) := n_i</math>     <math>cmd(n_i) := S_i</math>     <math>next(n_i) := m_i</math>     <math>cmd(m_i) := end</math>     ... </pre>
--	--

The rule for **alt** should be modified in the obvious way to fit its refinement. This could be viewed as a ‘compiler algebra’ for the Occam fragment. Note that there are no rules for ‘compiling’ atomic commands and boolean tests—when all composite commands are ‘compiled’, the algebra halts. In a sequel to this paper the ‘compiler algebra’ will be replaced by compilation function and refined to a ‘real’ compiler making further decomposition of atomic commands and boolean tests.

## REFERENCES

1. J.P. Bowen, He Jifeng, P.K. Pandaya, 1990, *An Approach to Verifiable Compiling Specification and Prototyping*, Springer Verlag, LNCS 456, pp 45-59.
2. J.P. Bowen, 1993, *From Programs to Object Code and back again using Logic Programming: Compilation and Decompilation*, in: *Journal of Software Maintenance: Research and Practice* 5(4):205-234.

3. E.Börger & D.Rosenzweig, 1994, *A mathematical definition of Full Prolog*, in: *Science of Computer Programming* (to appear).
4. E.Börger & D.Rosenzweig, 1994, *The WAM—Definition and Compiler Correctness*, in: *Logic Programming: Formal Methods and Practical Applications*, C.Beierle, L.Plümer, eds., North-Holland, Series in Computer Science and Artificial Intelligence.
5. G.N. Buckley, A. Silberschatz, 1982, *An Effective Implementation for the Generalized Input–Output Construct of CSP*, in *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 2, pp. 223-235.
6. B. Buth et. al., 1992, *Provably Correct Compiler Implementation*, in U. Karstens and P. Pfahler (eds.) *Compiler Construction*, Springer Verlag, LNCS 641, pp. 141-155.
7. P. Glavan & D. Rosenzweig, 1993, *Communicating Evolving Algebras*, in: *Computer Science Logic*, Selected Papers from CSL'92 (eds. E. Börger, S. Martini, G.Jäger, H.Kleine Büning, M. M. Richter), Springer LNCS 702, pp. 186–215.
8. Ian Graham, 1990, *The Transputer Handbook*, Prentice Hall.
9. Y.Gurevich, 1991, *Evolving Algebras. A Tutorial Introduction*, EATCS Bulletin 43, February 1991, pp. 264–284.
10. Y. Gurevich & J. K. Huggins, 1993, *The Semantics of the C Programming Language*, in: *Computer Science Logic*, Selected Papers from CSL'92 (eds. E. Börger, S. Martini, G.Jäger, H.Kleine Büning., M. M. Richter), Springer LNCS 702, pp. 274-308.
11. Y. Gurevich & L. Moss, 1990, *Algebraic Operational Semantics and Occam*, in: *CSL'89, 3d Workshop on Computer Science Logic* (E. Börger, H. Kleine Büning, M.M. Richter, eds.), Springer LNCS 440, 176–192.
12. He Jifeng, J.P. Bowen, 1993, *Specification, Verification and Prototyping of an Optimized Compiler*, in: *Formal Aspects of Computing*.
13. Inmos, *Transputer Implementation of Occam*. In: *Communication Process Architecture*, Prentice Hall, note 21.
14. Inmos, 1988, *Transputer Instruction Set – A compiler writer's guide*, INMOS document 72 TRN 119 05, Prentice Hall.
15. I. Page, W. Luk, 1991, *Compiling Occam into field-programmable gate arrays*, in *FPGAs, Oxford Workshop on Field Programmable Logic and Applications*, Abingdon EE&CS Books, pp 271-283.
16. G. Plotkin, 1981, *A structural approach to operational semantics*, International Report, CS Department, Aarhus University, DAIMI FN-19.
17. A.W. Roscoe, C.A.R. Hoare, 1988, *Laws of occam programming*, in *Theoretical Computer Science*, **60**, pp 177-229.
18. D.Scott, *Outline of a Mathematical Theory of Computation*, Technical Monograph PRG-2, November 1970, Oxford University Comp.Lab., Programming Res.Group, pp 1-24
19. D.A.P. Mitchell, et al, 1990, *Inside the Transputer*, Blackwell Scientific Publications.

PS. **The present paper appeared in:** E. - R.Olderog (Ed.), Proc. Procomet'94, IFIP TC 2 Working Conference on *Programming Concepts, Methods and Calculi*, North-Holland 1994, 489—508.