Correctness of compiling Occam to Transputer code*

Egon Börger & Igor Đurđanović

Dipartimento di Informatica
Università di Pisa
Cso Italia 40
I-56125 Pisa
boerger@di.unipi.it
fax: xx39–50–887226

University Paderborn
Fachbereich 17 – Informatik
Warburgerstr. 100
33098 Paderborn, Germany
igor@uni-paderborn.de
fax: xx49–5251–603338

This paper contributes to the development of a rigorous mathematical framework for the study of provably correct compilation techniques. The proposed method is developed through an implementation of a real–life non–toy imperative programming language with nondeterminism and parallelism – namely Occam – to a commercial machine, namely the Transputer. We provide a mathematical definition of the Transputer Instruction Set architecture for executing Occam together with a correctness proof for a general compilation schema of Occam programs into Transputer code.

We start from the ground model, an abstract processor, running a high and a low priority queue of Occam processes, which formalizes the semantics of Occam at the abstraction level of atomic Occam instructions. We develop here increasingly more refined levels of Transputer semantics, proving correctness (and when possible also completeness) for each refinement step. Along the way we collect our proof assumptions, a set of natural conditions for a compiler to be correct, thus making our proof applicable to a large class of compilers. As a by–product our construction provides a challenging realistic case study for proof verification by theorem provers.

1. INTRODUCTION

It is well known that a reliable compilation method which includes compiler verification requires that the semantics of both source and target language have been rigorously defined. In [BĐR:94] a high–level mathematical model for the truly concurrent semantics of Occam has been developed which captures the intuitive programmer's view of the dynamics of Occam in terms of atomic Occam instructions. In this paper we provide a mathematical model for the Transputer Instruction Set architecture. We use these two models to prove the correctness of the compilation scheme proposed in [Inmos:88] for the compilation of Occam programs into the Transputer instruction set.

<u>**Main Theorem**</u> Every compiler which satisfies the conditions listed in this paper compiles arbitrary Occam programs correctly into Transputer instructions.

The main problem in proving the theorem consists in bridging the gap between the abstraction levels of Occam and the Transputer. We relate the Occam ground model to the Transputer model by a series of stepwise refined intermediate models. At each refinement step we show the correctness and when possible also the completeness of the implementation. As a side product of our work for the correctness proof we obtain a detailed explanation of the rationale of the compilation scheme in [Inmos:88].

Several remarks have to be made to avoid a possible misunderstanding of the theorem.

- Correctness is to be understood as relative to the formal Occam and Transputer ground models; along the specification of these models it is made explicit and precise which parameters of the high–level model are correctly preserved through the refinement steps. Unfortunately we could not make reasonable use of any of the many refinement notions in the literature. There is also no general refinement notion for *evolving algebras*; but for each specific refinement step we explicitly define what this refinement means. Therefore it is crucial that these models are simple and transparent and can independently be justified, on pragmatic grounds, as adequate formalization of the programming language Occam and of the Transputer

*In Computer Journal 1996. Preliminary version appeared in Evolving Algebras Mini-Course, BRICS Notes Series NS-95-4, ISSN 0909-3206, pp. 153-194, University of Aarhus, 1995.

processor respectively.

- The correctness claim and its proof are not absolute but relative; indeed the mathematical models contain a certain number of interfaces to the environment which are supposed to work in accordance with those properties which are used as assumptions in the proof. Such relative correctness proofs are the best one can reach by rigorous methods, given the huge complexity of the problems under study.
- The proof is a mathematical proof in the classical sense of the term, based upon human reasoning and insight and providing understanding, not a machine level verification of details. It is split into numerous steps which have been introduced in order to break down the complexity of the whole construction into manageable and well understood small pieces. In order to be faithful to the Transputer implementation of Occam (see [Inmos]) we have to meet the Compiler Writer's Guide [Inmos:88]. The definitions of the semantics of Occam and of the Transputer which are authoritative for us come from these INMOS books, not from the well known denotational descriptions of Occam. This condition of having to reflect faithfully the given INMOS definitions also implies that we do not aim at providing any new ideas about what Occam and the Transputer were constructed for. However, in order to achieve the desired correctness proof we have to structure the Compiler Writer's Guide by decomposing it through many refinement steps whose correctness is amenable to the precise *formulation* and to the *proof* of the relevant properties.
  We have also made a particular effort in order to achieve that most of our proofs become "local", i.e. have to do with well defined specific features and leave the whole rest abstract or unchanged; this locality makes the approach modular in a strong (not only syntactical) sense: we isolate orthogonal Transputer instruction set components and often can proceed with our proofs even instructionwise. The guiding principle for breaking complex statements into simpler ones has been to stop only where the proofs become routine exercises which can be carried out by an automatic theorem proving system. Clearly we concentrate our attention on a precise outline and full proof of the global proof strategy. The routine exercises which consist in carrying out simple inductions, case distinctions, etc. are left to the reader. Our teaching experience is that in class, average students solve these exercises without problems, satisfactorily.
- We prove the correctness of a compiling specification. This means that we do not describe any specific compiler but formulate explicit conditions on the compilation function, namely those properties of the compilation process which we use for the correctness proof. These assumptions represent useful directives for correct compiler design and can also be used for variations of the design scheme for the underlying architecture, preserving correctness without need for testing of the unchanged components.

The overall structure of our refinement hierarchy is divided into two parts (sections 3. – 4. and 5. – 6.). It highlights essential points of provably correct compiler development, namely the implementation of the control structure and of environments (including auxiliary functions and dealing with relative addressing). The Occam ground model comprises a refinement of the concurrent Occam semantics to an abstract sequential processor which runs two queues of processes (one of low and one of high priority); in order to concentrate on Occam's distributed features — i.e. communication, parallelism and alternation — the Occam ground model has been based upon the usual layout of (imperative) programs as flowchart along which the process(es) (daemon(s)) are supposed to walk, each carrying along his own environment in which he executes at each node an atomic Occam instruction. In section 4. the generation of this flowchart is replaced by compilation into still abstract code. In section 6. the abstract code will be refined to Transputer code whose execution is based upon the Transputer model developed in section 5..

The refinement of the flowchart generation to code compilation consists in replacing the walking of daemons through the flowchart by moving instruction pointers through abstract code produced (together with the environment) by compilation. Thus it was natural to split this compilation again into two steps: compilation of the control structure and compilation of the environment. For the control structure compilation we first linearize the flowchart (by introducing goto–instructions) and then describe its machine internal representation by loading the result of a compilation function. For the compilation of the environment we first refine environments to be determined by the program structure and not any more by the daemons; we then implement environments by blocks of memory (obtaining eventually relocatable environment access by relative addresses of identifiers with respect to daemons as base addresses).

Section 6. refines the compilation from abstract to Transputer instructions. Here again we have four refinement steps: first we introduce the Transputer ground model consisting of various registers used for the execution of Transputer instructions.

Then we eliminate the environment from the run–time by building the relative addresses for identifiers into the compilation (with run–time calculation of the absolute address). As third step we implement abstract functions refining daemons to workspace addresses. At the end we make the Transputer code relocatable by introducing the technique of relative branching. At compile–time the instruction address offset (or distance) is calculated whereas at run–time this offset is

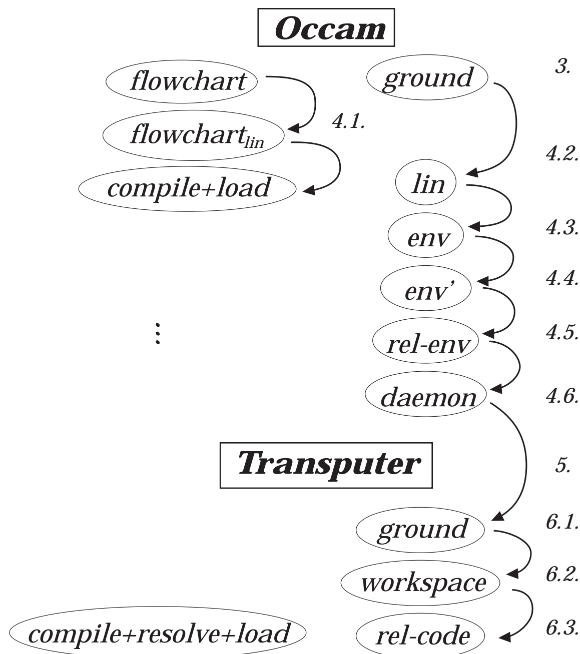


FIGURE 1. Refinement structure

used to obtain the absolute addresses.

Diagram 1 summarizes the overall structure of our refinement hierarchy. The Appendixes 7.2. and 7.3. summarize the complete rule system for the result of sections 4. and 6.

In contrast to other work on compiler correctness in the literature, we do not verify a series of compilers which compile between various intermediate languages to give a multi–stage compiler for the whole Occam language. Rather each refinement of our function *compile* compiles directly from the Occam source language and thus constitutes in itself a new compiler.

The refinement hierarchy and the correctness proofs work for the full Occam language. Since our main goal is to let the characteristic Occam features stand out in a transparent way along the whole implementation process we have decided to leave the evaluation and compilation of expressions and the implementation of values still abstract; one can add to our refinement chain further levels which deal with this, including the particular Transputer scheme for encoding instructions and providing large operands (i.e. postfixing and prefixing). (See the careful treatment of expression compilation in [MMO:95], pages 38–49.) For notational convenience and without loss of generality we also work with only one processor (Transputer) but without loosing its multi–task capabilities (queue).

The method which allows us the appropriate refinements of the abstraction level, providing full mathematical rigor but avoiding heavy formal overhead, makes use of Gurevich's notion of evolving algebras [Gur:95]. One essential feature of the potential of the evolving algebra approach to specification and verification of large systems is the fact that evolving algebra models can be read and understood without any specific previous formal training. We invite the reader who does not know the notion of evolving algebra to read our models as "pseudo–code over abstract data"; that suffices for an understanding of the specification. To carry out the proofs, some more technical understanding of what constitutes computations by evolving algebras is needed however. To avoid a possible misunderstanding we want to stress the point that our rules, which the practitioner may read as abstract pseudo-code, do have however a precise mathematical meaning, derived from Gurevich's rigorous definition of the semantics of *evolving algebras* in [Gur:95].

Section 2. summarizes the basic definitions and notation. Section 3. recalls the basic constituents of the Occam ground model which are the starting point of the refinements in section 4.. Along the way we use the chance to adapt the Occam oriented ground model to the needs of the Transputer, the target model of this paper. We pay attention that these modifications preserve the correctness theorems of [BĐR:94]. As a by–product this permits the reader to follow the present paper without knowing [BĐR:94].

We suppose the reader to be familiar with (the problems of) provably correct compiler development and to have some idea about Occam or at least about the notion of parallel computation.

2. EVOLVING ALGEBRA: PREREQUISITES AND NOTATION

In our specifications we use the notion of *evolving algebras*, see [Gur:95]. Evolving algebras represent a mathematically rigorous form of fundamental operational intuitions of computing. This permits to read and understand our description as 'pseudocode over abstract data', without any particular theoretical prerequisites. For the sake of completeness we review here our notation and refer the interested reader for the foundational justification to [Gur:95].

We treat abstract data as elements of (possibly not furthermore specified) sets (domains, *universes*). The operations allowed on universes will be represented by partial *functions*. Dynamic changes are obtained by executing *function updates* of form

$$f(t_1, \ldots, t_r) := t$$

whose execution is to be understood as *setting* (modifying) the value of the function f at given arguments. Note that the 0-ary functions play the role of *variables* in programming.

An *evolving algebra* is defined by a finite set of *transition rules* of form "**if** *Cond* **then** *Updates*" where *Cond* (condition or guard) is an expression, the truth of which triggers *simultaneous* execution of all updates in the finite set of *Updates*. Simultaneous execution helps us

avoid fussing and coding to, say, interchange two values. Since functions may be partial, equality in the guards is to be understood as implying that both arguments are defined.

Unless explicitly declared to be static, functions are dynamic. For a given evolving algebra $\mathcal{A}$ a function must be dynamic if it can be updated in rules of $\mathcal{A}$, i.e. if it is an f of a functional update; otherwise this update would be syntactically incorrect. If we want to stress that a dynamic function f may change its value without being updated by a rule of $\mathcal{A}$ we declare this function as external.

In applications an evolving algebra usually comes together with a set of *integrity constraints*, i.e. extralogical axioms and/or rules of inference which specify the intended domains.

Our rules will always be constructed so that the guards imply *consistency* of updates.

Evolving algebras transform structures (abstract machine states) into structures, the term being taken in the standard sense of (first–order) logic. Thus they can be understood as *transition systems* whose states are first order structures. This intuitively clear semantics of evolving algebras has a rigorous definition given in [Gur:95].

In applications of evolving algebras one usually encounters *heterogenous* signatures with several universes, which may in general grow and shrink in time. Therefore we use the following update form to extend a universe:

extend A **by** $t_1, \ldots, t_r$ **with** *Updates* **endextend**

where *Updates* may (and should) depend on t_i's, setting the values of some functions on *newly created* elements t_i of A. [Gur:95] has shown how to reduce these domain extensions and heterogeneous structures to the basic model of a homogenous signature (with one universe).

The forms obviously reducible to the above basic syntax, which we shall freely use as abbreviations, are **where**, **let** and **if_then_else**.

As stated above all the updates appearing in a rule are executed simultaneously. In the rare cases where we need sequentiality we will use the update form:

seq *Updates* **endseq**

where the *Updates* are executed sequentially.

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:

if $p_1 \mid p_2 \mid p_3$
$\quad \wedge \ \ldots$
then $u_1 \mid u_2 \mid u_3$
$\quad \ldots$

We shall assume that we have the standard mathematical universes of booleans, integers, lists of whatever etc (as well as the standard operations on them) at our disposal without further mention. In general we will write *NAME* for the universe of *objects* of type *name*. We use notations $x_1.x_2.\ldots.x_r$, $x_1, x_2, \ldots, x_r$, $(x_1, x_2, \ldots, x_r)$ etc. for lists.

An evolving algebra, as given above, determines the dynamics of a transition system. Evolving algebra descriptions of systems are deliberately what is often called "operational". they support directly the users' point of view of a system which evolves due to actions which take place in time. It has been explained in [Boerger:95] why this rule–based but abstract modeling of process does not lead to consideration of irrelevant or "dirty" implementation details and why "operational" is by no means contradictory to "abstract". We are usually only interested in states reachable from some designated *initial states*, which may be specified in various ways.

3. REVIEW OF THE $OCCAM_{ground}$ MODEL

We summarize here the final version of the Occam ground model developed in [BĐR:94] which is the starting point of the refinements in this paper. We make an effort to explain that ground model from scratch in order to enable the reader to understand this paper without knowing the details of [BĐR:94]. The reader who is familiar with [BĐR:94] can skip this section and come back to it should he want to check the justification for some slight technical (mostly notational) changes which we incorporate here in order to smoothen the transition from flowchart generation to compilation.

The starting point $OCCAM_{ground}$ of this paper is itself the result of various refinement steps defined in [BĐR:94]. They lead from a high–level truly concurrent model of Occam to abstract sequential processors running two queues of processes (for low and for high priority) with time–slicing and interrupt mechanism; these sequential processors (Transputers) run concurrently to other processors and external channels.

For notational convenience we restrict our attention here to only one processor and skip external channels; technically speaking this means only to suppress in certain functions and rules of our model the parameter ranging over the universe of *PROCESSOR*s. In order to avoid repetitions we will also deal with just one queue, the low priority queue which is subject to time–slicing in order to preserve completeness of the sequential implementation of concurrent runs; it would be easy to incorporate the interrupt handling by adding a new rule within the Fetch-Execute mechanism.

The flowchart

The usual layout of (imperative) programs as flowchart with nodes marked by atomic instructions is formalized by a set of *NODE*s with functions:

$$next : NODE \times \mathbf{N} \rightarrow NODE$$
$$cmd : NODE \rightarrow CODE$$

representing the edges and marks respectively. For notational convenience we use the following macros:

$$next(n) \stackrel{\text{def}}{=} next(n,0)$$
$$yes(n) \stackrel{\text{def}}{=} next(n,0)$$
$$no(n) \stackrel{\text{def}}{=} next(n,1)$$

Processes are represented as daemons (elements of an abstract set *DAEMON*) which are placed into the flowchart by a dynamic function a (for "agent"):

$$loc \quad : \quad DAEMON \rightarrow NODE.$$

The processor

The processor is formalized by a queue of running (i.e. non sleeping) daemons, a currently active daemon and clock functions (for time–slicing). Formally — the set of of non sleeping processes, waiting to become active — $QUEUE \subseteq DAEMON^*$ comes with external functions:

$$first, last \quad : \quad QUEUE \rightarrow DAEMON$$
$$rest \quad : \quad QUEUE \rightarrow QUEUE$$

and a 0-ary dynamic function:

$$q \quad : \quad QUEUE$$

which represents the current incarnation of the queue. The unique currently active process (the "agent") is formalized by a 0-ary dynamic function:

$$a \quad : \quad DAEMON \cup \{ nil \}.$$

The processor clock is formalized by an external function:

$$timer \quad : \quad \mathbf{N}$$

which is used to set the dynamic 0-ary function:

$$start \quad : \quad \mathbf{N}$$

when a process is taken out of the queue to become active. The dequeuing (activating the next process in the queue) is done by the macro[1]:

$$\begin{array}{lll} dequeue & \stackrel{\text{def}}{=} & \textbf{if } \text{not } empty\ q \\ & & \textbf{then } a := first(q) \\ & & \qquad q := rest(q) \\ & & \qquad start := timer \\ & & \textbf{else } a := nil \end{array}$$

[1] In [BDR:94] we had introduced the dequeue update as independent rule "**if** $a = nil \wedge$ not *empty q* **then** *dequeue*" which was the only executable rule each time a was set to nil (by time–slice, stop or end rule or by sending a process to sleep). To smoothen the transition to the Transputer we incorporate here the effect of dequeuing directly into the relevant rules; it is obvious that this modification does not change the semantics. The present definition for dequeuing is in accordance with the INMOS manual [Inmos:88] which says that the special daemon nil indicates that no valid process is present. Our present definition reflects that when the Transputer runs out of processes to be executed nothing happens: the processor runs in *fetch* mode (see below) until some process shows up.

At places which are safe for time–slicing the currently active process $a \neq nil$ will be put back into the queue when its time is elapsed, i.e. exceeds the time–slicing *period*, a 0-ary external (implementation defined) function[2]:

$$period \quad : \quad \mathbf{N}$$

This is done by the rule[3]:

$$\begin{array}{ll} \boxed{\text{time–slice}} & \textbf{if } a \neq nil \wedge elapsed \\ & \textbf{then seq } q := q.a \\ & \qquad\qquad dequeue \\ & \quad \textbf{endseq} \end{array}$$

where

$$elapsed \stackrel{\text{def}}{=} timer - start > period$$

Beside of the queue q of running processes, there is another queue for administrating time–waiting processes[4]:

$$time_q \quad : \quad QUEUE$$

If a daemon is in the time queue, then the dynamic function:

$$t_{min} \quad : \quad DAEMON \rightarrow \mathbf{N}$$

holds the minimal time he is waiting for. We will use the following macros[5] for (non–deterministic) insertion/deletion of daemons into/from the time-queue *time_q*:

$$\begin{array}{ll} time\text{-}insert & \stackrel{\text{def}}{=} \\ & \textbf{let } time_q = x_1.\ldots.x_i.x_{i+1}.\ldots.x_r \\ & \textbf{let } t_{min}(x_i) \leq t_{min}(a) \leq t_{min}(x_{i+1}) \\ & time_q := x_1.\ldots.x_i.a.x_{i+1}.\ldots.x_r \\ time\text{-}delete\ x & \stackrel{\text{def}}{=} \\ & \textbf{let } time_q = x_1.\ldots.x_i.x.x_{i+1}.\ldots.x_r \\ & time_q := x_1.\ldots.x_i.x_{i+1}.\ldots.x_r \end{array}$$

Dynamics of Processes

Daemons x are sitting at their current flowchart node (see 7.1.) $loc(x)$ waiting to be activated by the processor

[2] The restriction to one processor comes up to have suppressed the *PROCESSOR* parameter for the functions q, a, *timer*, *period*, *start*. The restriction to one priority queue comes up to suppress the parameter q of *dequeue*.

[3] Note the sequential execution of enqueuing and dequeuing of a daemon. This is done to avoid problems with simultaneous execution when only one daemon exists.

[4] Note that in the Transputer there are two timer-queues, one for each priority. In [BDR:94] we could formalize time queues simply as sequences *min_time* of pairs of daemons and their waiting time. For a smooth transition to the Transputer model it turns out to be advantageous to split *min_time* into a list *time-q* of daemons and a function t_{min} which records the waiting times. We leave it as an exercise to show that this implementation is correct.

[5] At the level of abstraction of the Occam interpreter in [BDR:94] we could afford to abstract from the order of the processes in the timer queue. Since in reality the hardware is responsible for taking waiting processes out of the time queue, we have to make this sequentiality in the Transputer model explicit; see the time_wakeup rule below.

for execution of the instruction $cmd(loc(x))$ in this node. We abbreviate this condition for a daemon to get his instruction executed by the following:

$$cmd\ is\ c \stackrel{\text{def}}{=} cmd(loc(a)) = c \wedge \text{not } elapsed$$

Note that this condition is false if the time–slice rule can fire[6]. After having executed the instruction of a daemon the processor will proceed to the next instruction; we abbreviate this action by:

$$proceed \stackrel{\text{def}}{=} loc(a) := next(loc(a))$$

A daemon can become inactive either by time–slicing or because for some reason he has to wait (for a communication partner to be ready or for a time condition to become satisfied) and therefore "goes to sleep". Going to sleep means to become inactive without going back into the process queue:

$$sleep\ at\ n \stackrel{\text{def}}{=} loc(a) := n, dequeue$$

Being woken up then corresponds to return into the queue:

$$wakeup\ x \stackrel{\text{def}}{=} enqueue\ x$$
$$enqueue\ x \stackrel{\text{def}}{=} q := q.x$$

The behavior of Occam programs can now be rigorously defined by giving for each atomic instruction rules which define the semantics of that instruction.

Declarations

In the flowchart model of Occam daemons carry their *ENV*ironment, associated to them by a dynamic function:

$$env : DAEMON \rightarrow ENV$$

This function is updated through execution of declaration instructions[7]. Without loss of generality we can replace the abstract notion of *ENV*ironment used in [BĐR:94] by defining ENV as $(ID \times (VAR \cup CHANNEL))^*$; clearly VAR and $CHANNEL$ are supposed to be disjoint.

The semantics of declaration instructions is defined by the following simple rules:

decl_var$(id_1, \ldots, id_r)$

if *cmd is* decl_var$(id_1, \ldots, id_r)$
then extend VAR **by** $v_1, \ldots, v_r$ **with**
  $env(a) := append(e, env(a))$
 endextend
 proceed
where $e = (id_r, v_r), \ldots, (id_1, v_1)$

[6] The condition $a \neq nil$ in *cmd is* from [BĐR:94] could be skipped with the understanding that $cmd(undef) = undef$, $loc(nil) = undef$.

[7] As already in [BĐR:94] we skip the data-types of Occam and procedures as they are not in any way characteristic for the language and can be incorporated into our models in a standard way.

For channel declarations one has to add the initialization of the channel agent (reader or writer) to *nil*; the agents needed for the asynchronous arrival of communications partners (see below) are formalized by a dynamic function:

$$agent : CHANNEL \rightarrow DAEMON \cup \{nil\}.$$

decl_chan$(id_1, \ldots, id_r)$

if *cmd is* decl_chan$(id_1, \ldots, id_r)$
then extend $CHANNEL$ **by** $c_1, \ldots, c_r$ **with**
  $env(a) := append(e, env(a))$
  $agent(c_1) := nil$
  ...
  $agent(c_r) := nil$
 endextend
 proceed
where $e = (id_r, c_r), \ldots, (id_1, c_1)$

Note that the Occam scoping scheme is respected: the later declaration overrides the previous one (see the definition of *bind* function below). The environment is restored once the daemon has left the scope of declaration:

decl_end(r)

if *cmd is* decl_end(r)
then let $env(a) = (id_r, v_r), \ldots, (id_1, v_1), e'$
  $env(a) := e'$
  proceed

Expressions

Expression evaluation is kept abstract using a function:

$$bind : ID \times ENV \rightarrow VAR \cup CHANNEL$$

for binding identifiers in a given environment and two functions for evaluating variables and expressions (in a given environment) to *VAL*ues which we keep abstract[8]:

$$eval : VAR \rightarrow VAL$$
$$eval : EXP \times ENV \rightarrow VAL.$$

As expected *bind* is recursively defined by:

$$bind(id, [H \mid T]) = \begin{cases} x & \text{if } H = (\text{id,x}) \\ bind(id, T) & \text{otherwise} \end{cases}$$

Notationally we use $\bar{v}, \bar{c}$ instead of $bind(v, env(a))$, $bind(c, env(a))$ respectively. Thus we have the following

[8] We clearly assume these two evaluation functions to be related in the standard way.

rules for assignment, timing, test of boolean condition:

$\boxed{\mathsf{ass}(v,t)}$

if *cmd is* $v := t$
then $eval(\bar{v})$ **:=** $eval(t, env(a))$
proceed

$\boxed{\mathsf{time}(v)}$

if *cmd is* $\mathsf{TIME}\,?\,v$
then $eval(\bar{v})$ **:=** *timer*
proceed

$\boxed{\mathsf{if}(b)}$

if *cmd is* $\mathsf{if}(b)$
then **if** $eval(b, env(a))$
then $loc(a)$ **:=** $yes(loc(a))$
else $loc(a)$ **:=** $no(loc(a))$

The rules for skip and stop are as follows:

$\boxed{\mathsf{skip}}$

if *cmd is* SKIP **then** *proceed*

$\boxed{\mathsf{stop}}$

if *cmd is* STOP **then** *dequeue*

Communication

The communicating daemons might arrive independently to the synchronizing channel. If a reader or writer arrives at a channel c before his communication partner, he will go to sleep after having notified to the channel his identity (updating the function $agent(c)$) and the place where he wants the input or the message he wants to output respectively. For this purpose in addition to the function *agent* two dynamic functions:

$$mssg : DAEMON \rightarrow VAL$$
$$place : DAEMON \rightarrow VAR$$

are introduced which are updated in the following two rules, where $idle\ c \stackrel{\text{def}}{=} agent(c) = nil$:

$\boxed{\mathsf{inp_idle}(c,v)}$
if *cmd is* $c\,?\,v$
$\wedge$ *idle* $\bar{c}$
then *sleep at* $next(loc(a))$
$agent(\bar{c})$ **:=** a
$place(a)$ **:=** $\bar{v}$
$com_mode(a)$ **:=** *input*

$\boxed{\mathsf{out_idle}(c,t)}$
if *cmd is* $c\,!\,t$
$\wedge$ *idle* $\bar{c}$
then *sleep at* $next(loc(a))$
$agent(\bar{c})$ **:=** a
$mssg(a)$ **:=** $eval(t, env(a))$

Note that the auxiliary function:

$$com_mode : DAEMON \rightarrow \{input, alt_sleep, alt_run\}$$

is set here to the value *input* in order to distinguish *ordinary* input requests from input requests which appear in the guard of an alternative (see the ALT-statement below). Since we consider here only internal channels, a daemon arriving as second communication partner to a non idle channel – formalized by $ready\ c \stackrel{\text{def}}{=} agent(c) \neq nil$ – completes the communication by placing the message to the place in question, by waking up the communication partner and by cleaning the channel[9]. This is formalized by the following two rules where *clear c* denotes the update $agent(c)$ **:=** *nil* which clears the channel once the communication is done:

$\boxed{\mathsf{inp_ready}(c,v)}$
if *cmd is* $c\,?\,v$
$\wedge$ *ready* $\bar{c}$
then $eval(\bar{v})$ **:=** $mssg(agent(\bar{c}))$
wakeup $agent(\bar{c})$
clear $\bar{c}$
proceed

$\boxed{\mathsf{out_ready}(c,t)}$
if *cmd is* $c\,!\,t$
$\wedge$ *ready* $\bar{c}$
$\wedge$ $com_mode(agent(\bar{c})) = input$
then $eval(place(agent(\bar{c})))$ **:=** $eval(t, env(a))$
wakeup $agent(\bar{c})$
clear $\bar{c}$
proceed

There is the special case of a daemon trying to output to a channel which received an input request through the guard of an alternative, formally which has been enabled by the first rule for the ALT-statement (alt_a, see below). In this case the output is allowed to be done only after the alternative in question has been really selected, but meantime, the outputting daemon has to announce his readiness (doing the updates of the out_idle rule) and to wake up the potentially input expecting communication partner; this includes to put him from the *alt_sleep* mode into the *alt_run* mode. This is for-

[9] The restriction to internal channels means that we do not have to consider the chan rule of [BDR:94], page 496.

malized by the rule:

$\boxed{\mathsf{out_alt}(c,t)}$
if *cmd is* $c\,!\,t$
  $\wedge$ *ready* $\bar{c}$
  $\wedge$ $com_mode(agent(\bar{c})) \neq input$
then *sleep at* $next(loc(a))$
  $mssg(a) := eval(t, env(a))$
  $agent(\bar{c}) := a$
  if $com_mode(agent(\bar{c})) = alt_sleep$
  then *time-delete* $agent(\bar{c})$
    wakeup $agent(\bar{c})$
    $com_mode(agent(\bar{c})) := alt_run$

Alternation

In the sequential implementation of Occam the execution of ALT-statements is split into two phases:

- announcing — for each of the input requests $c_i\,?\,v_i$ or time requirements TIME ? AFTER t_j where the corresponding boolean condition in the guard is *true* — the readiness to select the corresponding alternative once the corresponding input is ready or the time requirement is satisfied,
- selecting among the alternatives whose guards are satisfied.

The first phase is defined by the alt_a rule: the channels c_i appearing in guards $G_i = b_i : c_i?v_i$ with true boolean condition b_i are "enabled" by setting their agents to the executing daemon a:

$$enable(b,c) \stackrel{\text{def}}{=} \textbf{if } eval(b, env(a)) \wedge agent(c) = nil \textbf{ then } agent(c) := a$$

The smallest among the enabled time requirements to be checked against *timer* is recorded into the function t_{min}. If no input is ready and none of the time requirements is satisfied yet, the daemon goes to sleep (setting his communication mode to *alt_sleep*) and inserts himself into the time-queue *time_q* (if there was at least one enabled time guard); it can be woken up by an outputting communication partner (see above the out_alt rule) or because the minimal waiting time t_{min} has been reached by *timer* (see time_wakeup rule below).

If, when enabling, a daemon finds at least one input to be ready or one time requirement to be satisfied – the latter is immediately true in case of the empty requirement, denoted by SKIP – he goes into communication mode *alt_run* and proceeds to the execution of the alt_s instruction after having "disabled" the channels which had been "enabled":

$$disable(b,c) \stackrel{\text{def}}{=} \textbf{if } eval(b, env(a)) \wedge agent(c) = a \textbf{ then } agent(c) := nil$$

Note that the communication takes place only after the corresponding alternative has been chosen.

$\boxed{\mathsf{alt_a}(\vec{G})}$
if *cmd is* $\mathsf{alt_a}(\vec{G})$
then
  $enable(b_1, \bar{c}_1)$
  ...
  $enable(b_p, \bar{c}_p)$
  if $\exists\, i\ (eval(b_i, env(a)) \wedge ready\ \bar{c}_i)$
    $\vee\ t_wait < timer$
    $\vee\ \exists\, k\ (eval(b_k, env(a)))$
  then $com_mode(a) := alt_run$
    proceed
  else $com_mode(a) := alt_sleep$
    sleep at $next(loc(a))$
    if there is an enabled time guard
    then seq $t_{min}(a) := t_wait$
        time-insert
      endseq
where
  $G_i = b_i : c_i\,?\,v_i$
  $G_j = b_j : \mathsf{TIME\,?\,AFTER}\,t_j$
  $G_k = b_k : \mathsf{SKIP}$
  $$t_wait = \begin{cases} \infty \text{ if } \forall\, j\ (\neg eval(b_j, env(a))), \text{ otherwise} \\ \min_j\{eval(t_j, env(a)) \mid eval(b_j, env(a))\} \end{cases}$$
  $1 \le i \le p < j \le q < k \le r$

Obviously, the sentence "there is an enabled time guard" stands for the expression $\exists\, j\ (eval(b_j, env(a)))$, where $p < j \le q$. Note that here and in the sequel we use indices and "..." only to have a concise notation without contradicting the formal character of our model.

$\boxed{\mathsf{alt_s_com}} \mid \boxed{\mathsf{alt_s_time}} \mid \boxed{\mathsf{alt_s_skip}}\ (i, \vec{G})$
if *cmd is* $\mathsf{alt_s}(\vec{G})$
  $\wedge\ eval(b_i, env(a))$
  $\wedge\ agent(\bar{c}_i) \neq\ a \mid timer > eval(t_i, env(a)) \mid$
then
  $disable(b_1, \bar{c}_1)$
  ...
  $disable(b_p, \bar{c}_p)$
  $loc(a) := next(loc(a), i)$
where
  $G_i = b_i : (\, c_i\,?\,v_i \mid \mathsf{TIME\,?\,AFTER}\,t_i \mid \mathsf{SKIP}\,)$

The value p (as in previous rule) is the count of input guards appearing in ALT construct. (Note our "|" notation explained in the prerequisites section 2..)

$\boxed{\mathsf{time_wakeup}}$
if *time_q* not *empty*
  $\wedge\ timer > t_{min}(first(time_q))$
then $com_mode(first(time_q)) := alt_run$
  wakeup $first(time_q)$
  time-delete $first(time_q)$

Note that by our definition of insertion into the time queue processes in that queue are ordered by the time they are waiting for.

Parallelism

In executing a PAR-statement the newly created daemons have to be linked to the currently executing daemon using a function:

$$father \ : \ DAEMON \rightarrow DAEMON$$

and the currently executing daemon – who will go to sleep – has to record how many processes he has created, updating a function:

$$count \ : \ DAEMON \rightarrow \mathbf{N}$$

The two rules to execute (begin and end of) PAR-statements are as follows[10]:

par(r) **if** *cmd is* par(r)
then extend *DAEMON* **by** $x_1 \ldots x_r$ **with**
$q := q.x_1. \ldots . x_r$
...
$father(x_i) := a$
$loc(x_i) := next(loc(a), i)$
$env(x_i) := env(a)$
...
$count(a) := r$
sleep at next($loc(a)$)
endextend
where $1 \leq i \leq r$

end **if** *cmd is* end
then *dequeue*
if $r = 1$
then *wakeup father*(a)
$r := r - 1$
where $r = count(father(a))$

4. FLOWCHART AND ENVIRONMENT COMPILATION

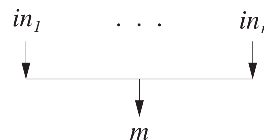
In this section we compile into still abstract code the Occam control structure (which is embodied in the flowchart of the ground model) and the environment (which in the ground model is associated to the daemons). In 4.1. the flowchart is linearized (by introducing goto-instructions instead of multiple in/outgoing edges). In 4.2. the resulting sequence of nodes, marked by atomic Occam instructions, is described by a compilation function whose output is loaded into memory. In 4.3. environments are refined as determined recursively by the program structure, resulting in their still abstract definition by the compilation function. In 4.4. this definition is refined by placing variables and channels into memory using an auxiliary function which computes the needed environment size recursively along the program structure. This allows us in 4.5. to make the memory access relocatable by introducing identifier addresses which are relative to daemons as base addresses.

4.1. Flowchart linearization

The goal of our paper is to prove the correctness of compiling specifications and not the correctness of compiler optimization techniques. Therefore the attention is focused on what happens to *atomic* Occam instructions during the compilation. As a consequence the use of the tree structure for the optimization of the code compiled for the given Occam program is not relevant for our analysis. This is the reason why we deviate from the usual practice in compilers and linearize the given flowchart at the very beginning of our analysis (instead of keeping the linearization of the generated instruction blocks for the end of the compilation).

In the sequel let S be an arbitrary Occam program and *flowchart*(S) the flowchart by which the ground model $OCCAM_{ground}$ is initialized (see 7.1.). We transform *flowchart*(S) into a sequence $lin(S)$ of nodes which are marked with atomic Occam instructions or with new goto-instructions whenever those are needed to avoid multiple in/outgoing edges. We refine correspondingly the $OCCAM_{ground}$ – rules for branching instructions – i.e. if, par, alt_s; adding also a new goto-rule we obtain a new evolving algebra $OCCAM_{lin}$ which if initialized by $lin(S)$ is equivalent to $OCCAM_{ground}$ initialized by *flowchart*(S).

The transformation follows standard techniques: through the linearization nodes which (at run time) are the next ones might physically not be any more the next ones; for such cases one has to introduce goto instructions. More formally: multiple incoming edges to a node m from nodes $in_1, \ldots, in_r$



are replaced by new nodes $n_1, \ldots, n_r$ marked by the new instruction goto(m); multiple outgoing edges, say from a node n with $cmd(n) = instr$ to nodes $m_1, \ldots, m_q$ are replaced by assigning a new instruction $instr(m_1, \ldots, m_q)$ to n with a modified rule which will assume the intended flow control. The precise form depends on the four possible cases for *instr* where multiple outgoing edges can occur in *flowchart*(S), namely IF, WHILE, PAR, ALT.

For a smooth transition (in 4.2.) from the linearized flowchart to the use of a compilation function whose result is loaded into *NODE* we treat the parameters for the modified instructions as “labels” to which an auxiliary function:

$$labeled_loc \ : \ LABEL \rightarrow NODE$$

will associate the intended target – location. In this

[10] Since we concentrate the attention to one processor we do not consider the PLACED PAR statement which is semantically different from the PAR-statement, it is allowed to be used only once – at the beginning of the Occam program – to “place” daemons on different processors; the daemons are not supposed to terminate and are not linked to the father.

way the new $OCCAM_{lin}$-function:

$$next \quad : \quad NODE \rightarrow NODE$$

becomes monadic and is related to the old binary $OCCAM_{ground}$ function *next* by:

$$next(n, i) \quad = \quad \begin{cases} next(n) & \text{if } i = 0 \\ labeled_loc(l_i) & \text{otherwise} \end{cases}$$

In the case of nodes with an if instruction, one out-

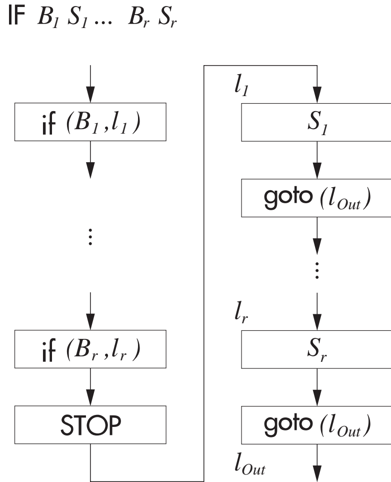


FIGURE 2. Flowchart Linearization for IF

going edge is chosen as *next*-exit, the other exit m is implemented by using a new node marked with the new instruction goto(m). For conventional reasons the choice of the *next*-exit depends on whether the if instruction comes from an IF (see Fig. 2) or from a WHILE (see Fig. 3) statement in the original Occam program; the *next*-exit is the no-exit in the first and the yes-exit in the second case. For flowcharts correspond-

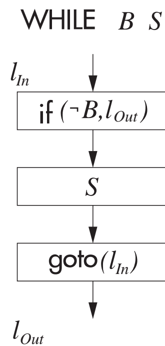


FIGURE 3. Flowchart Linearization for WHILE

ing to a statement of form PAR $S_1 \ldots S_r$ (see Fig. 4) or ALT $G_1\, S_1 \ldots G_r\, S_r$ (see Fig. 5) the linear sequences constructed by induction for the subprograms are connected in the order in which the latter appear in the Occam program. The correct transfer of control to the entry node of $lin(S_i)$ or $lin(G_i, S_i)$ labeled by l_i is implemented by the additional assignment of $labeled_loc(l_i)$ as location in the modified par and alt rule. Formally

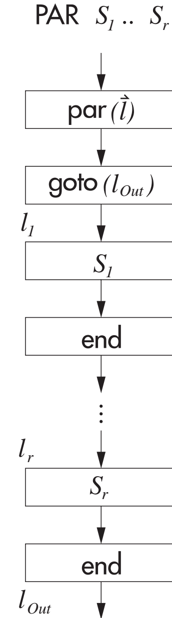


FIGURE 4. Flowchart Linearization for PAR

this linearized flowchart can be generated by a simple system of EA rules, similar to the flowchart generating rules given in appendix 7.1.. We leave this as an exercise to the reader. The new $OCCAM_{lin}$ algebra has new rules for three refined instructions (if, alt_s and par of the preceding section) and for one new instruction (goto). In the branching rules the additional parameter l is introduced together with a location update by $labeled_loc(l)$.

[if(b,l)] **if** *cmd is* if(b, l)
then if $eval(b, env(a))$
  then $loc(a)$:= $labeled_loc(l)$
  else *proceed*

[goto(l)] **if** *cmd is* goto(l)
then $loc(a)$:= $labeled_loc(l)$

[par($\vec{l}$)] **if** *cmd is* par($l_1, \ldots, l_r$)
then extend *DAEMON* **by** $x_1 \ldots x_r$ **with**
  q := $q.x_1.\ldots.x_r$
  ...
  $father(x_i)$:= a
  $loc(x_i)$:= $labeled_loc(l_i)$
  $env(x_i)$:= $env(a)$
  ...
  $count(a)$:= r
  sleep at $next(loc(a))$
  endextend
where $1 \leq i \leq r$

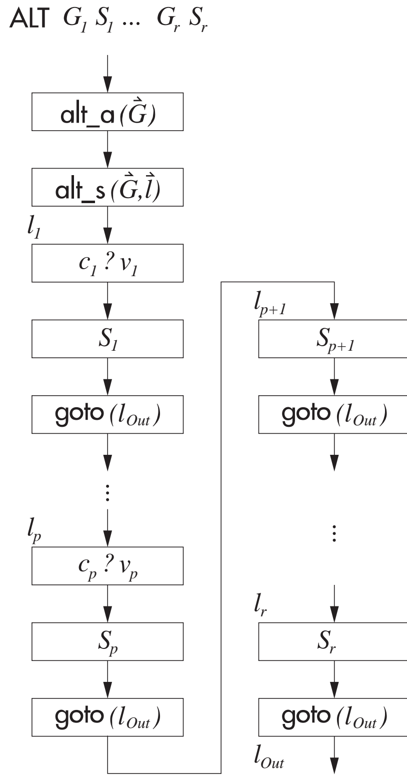


FIGURE 5. Flowchart Linearization for ALT

alt_s_com | alt_s_time | alt_s_skip $(i, \vec{G}, \vec{l})$

if *cmd is* alt_s$(\vec{G}, \vec{l})$
  $\wedge\, eval(b_i, env(a))$
  $\wedge\, agent(\bar{c}_i) \notin \{\, nil, a \,\} \mid timer > eval(t_i, env(a)) \mid$
then
  $disable(b_1, \bar{c}_1)$
  ...
  $disable(b_p, \bar{c}_p)$
  $loc(a)$:= $labeled_loc(l_i)$
where
  $\vec{l} = l_1, \ldots, l_r$
  $\vec{G} = G_1, \ldots, G_r$
  $G_i = b_i \;:\; (\, c_i\,?\,v_i \mid$ TIME ? AFTER $t_i \mid$ SKIP)

The value p is the count of input guards appearing in the ALT construct.

It is easy to define the notion of corresponding runs and of the equivalence between $OCCAM_{ground}$ on *flowchart*(S) and $OCCAM_{lin}$ on *lin*(S) where homonymous rules — corresponding to each other via $next(n, i) = labeled_loc(l_i)$ — have the same effect. This allows us to prove the following simple proposition:

PROPOSITION 4.1. *Corresponding runs of $OCCAM_{ground}$ on flowchart(S) and of $OCCAM_{lin}$ on lin(S) are equivalent.*

COROLLARY 4.2. *Corresponding runs of $OCCAM_{ground}$ on flowchart(S) and of $OCCAM_{lin}$ on lin(S) preserve daemons together with their environment and the values of variables appearing there, communication traces, termination, deadlock and divergence.*

4.2. Compilation and Loading of programs

In this section we describe the generation of the linearized flowchart *lin*(S) through an abstract compilation function:

$$compile \;:\; STATEMENT \rightarrow (CODE \cup LABEL)^*$$

In order to achieve a simple definition of *compile* we will use the natural and quite standard recursion on the program structure. Therefore we separate the definition of *compile*(S) from its "loading", i.e. from the generation of nodes which will be marked by the atomic instructions in *compile*(S). As a consequence we separate now the syntactical status of labels l — to be names for parameters of instructions goto, if, par, alt_s — from their semantical interpretation as nodes *labeled_loc*(l) associated to them during the loading process. The loading itself is described by two new rules which constitute the *LOAD* algebra: they assign instructions to the next free node – "memory location" *loading_loc* – and assign to labels the current loading location *loading_loc* as value (to which value the next compiled instruction will be associated).

Formally the *LOAD* algebra has the two new dynamic functions:

$$loading_loc \;:\; NODE$$
$$load_prg \;:\; (CODE \cup LABEL)^*$$

for the current loading location and the remaining program to be loaded which are thought to be initialized by *begin* – the initial node where the program execution starts – and *compile*(S) respectively. The two rules are:

load_label(l,t)

if $load_prg = [\, l \mid t \,]$
  $\wedge\, l \in LABEL$
then $labeled_loc(l)$:= $loading_loc$
  $load_prg$:= t

load_cmd(c,t)

if $load_prg = [\, c \mid t \,]$
  $\wedge\, c \in CODE$
then $cmd(loading_loc)$:= c
  $load_prg$:= t
  extend *NODE* **by** n **with**
    $next(loading_loc)$:= n
    $loading_loc$:= n
  endextend

In defining now the compilation function by induction on S we tacitly assume that each time where labels appear they are distinct from each other and fresh for the compilation process; this assumption reflects the fact

that semantically the labels have to represent different nodes in $lin(S)$, generated during the construction of the (linearized) flowchart. We abstain from the routine formalization of this assumption.

$compile(\mathsf{SKIP}) = \mathsf{SKIP}$

$compile(\mathsf{STOP}) = \mathsf{STOP}$

$compile(v := t) = v := t$

$compile(\mathsf{TIME}\,?\,v) = \mathsf{TIME}\,?\,v$

$compile(c\,!\,t) = c\,!\,t$

$compile(c\,?\,v) = c\,?\,v$

$compile(\mathsf{VAR}\ id_1, \ldots, id_r : S) =$
$\mathsf{decl_var}(id_1, \ldots, id_r)$,
$compile(S)$,
$\mathsf{decl_end}(r)$

$compile(\mathsf{CHAN}\ id_1, \ldots, id_r : S) =$
$\mathsf{decl_chan}(id_1, \ldots, id_r)$,
$compile(S)$,
$\mathsf{decl_end}(r)$

$compile(\mathsf{IF}\ B_1\,S_1\ \ldots\ B_r\,S_r) =$
...
$\mathsf{if}(B_i, l_i)$,
...
$compile(\mathsf{STOP})$,
...
l_i,
$compile(S_i)$,
$\mathsf{goto}(l_{Out})$,
...
l_{Out}
where $l_1, \ldots, l_r, l_{Out}$ are new labels
$1 \leq i \leq r$.

$compile(\mathsf{WHILE}\ B\,S) =$
l_{In},
$\mathsf{if}(\neg B, l_{Out})$,
$compile(S)$,
$\mathsf{goto}(l_{In})$,
l_{Out}
where l_{In}, l_{Out} are new labels.

$compile(\mathsf{PAR}\ S_1 \ldots S_r) =$
$\mathsf{par}(l_1, \ldots, l_r)$,
$\mathsf{goto}(l_{Out})$,
...
l_i,
$compile(S_i)$,
end,
...
l_{Out}
where $l_1, \ldots, l_r, l_{Out}$ are new labels
$1 \leq i \leq r$.

$compile(\mathsf{SEQ}\ S_1 \ldots S_r) =$
$compile(S_1)$,
...,
$compile(S_r)$

$compile(\mathsf{ALT}\ G_1\,S_1 \ldots G_r\,S_r) =$
$\mathsf{alt_a}(\vec{G})$,
$\mathsf{alt_s}(\vec{G}, \vec{l})$,
...
l_i,
$compile(c_i\,?\,v_i)$,
$compile(S_i)$,
$\mathsf{goto}(l_{Out})$,
...
l_j,
$compile(S_j)$,
$\mathsf{goto}(l_{Out})$,
...
l_{Out}
where $\vec{l} = l_1, \ldots, l_r$ are new labels
$\vec{G} = G_1, \ldots, G_r$
$G_i = b_i\ :\ c_i\,?\,v_i$
$G_j = b_j\ :\ (\ \mathsf{TIME}\,?\,\mathsf{AFTER}\,t_j$ or $\mathsf{SKIP}\ )$
$1 \leq i \leq p < j \leq r$.

Using the convention on fresh labels during the computation of $compile(S)$ it is easy to see that the following is true:

PROPOSITION 4.3. *The LOAD algebra, started with load_prg = compile(S), computes a sequence of nodes marked with* Occam *instructions which is isomorphic to lin(S).*

Proof. Induction on S, observing that labels l_i occurring during the recursive computation of $compile(S)$ correspond to the nodes $labeled_loc(l_i)$ associated to them by the loading rules.

COROLLARY 4.4. *Let $OCCAM_{compile}$ be the union of $OCCAM_{lin}$ and the LOAD algebra. The runs of $OCCAM_{lin}$ on lin(S) and of $OCCAM_{compile}$ started with load_prg = compile(S) are equivalent. Therefore they preserve daemons together with their environments and the values of the variables appearing there, communication traces, termination, deadlock and divergence.*

You would probably have expected that we define $OCCAM_{compile}$ by using the modification of $OCCAM_{lin}$ where the all rules get the additional guard $load_prg =$ []. As a matter of fact it is irrelevant whether applications of $OCCAM_{lin}$ rules and of *LOAD* rules are interleaved or not. ■

REMARK 4.1. Once the flowcharts have been linearized and the programs are compiled and loaded, it is of no help any more to speak of nodes as placeholders for instructions. In the following we switch therefore to the following new notation and naming which brings us

closer to the intuition of the Transputer memory:

$$NODE = LOC \quad next = +1 \quad begin = 0.$$

Since at this level of abstraction we do not care about efficient use of locations, we assume $+1 : LOC \rightarrow LOC$ to be a total (if you wish even injective) function on locations (we will also use the binary + function defined in the usual way from +1). As a consequence the extend update in the load_cmd rule can be replaced equivalently by:

$$loading_loc \quad := \quad loading_loc + 1.$$

As further step towards the Transputer we introduce the Transputer store abstractly by a function:

$$content \quad : \quad LOC \rightarrow VAL$$

which yields the value stored in a location. From now on we consider programs to be "stored in memory" by requiring *cmd* to be subfunction of *content*. Clearly these cosmetic changes (pure data refinements) of the $OCCAM_{compile}$ algebra do not effect the truth of proposition 4.3.

4.3. Environment Compilation

This section is devoted to the compilation of the environment which however still remains abstract and will be refined in the next section.

For the interpreter model $OCCAM_{ground}$ it did pay out to let daemons carry their environment; daemons only have to extend or shrink their current environment upon execution of a declaration begin or end instruction. In order to obtain efficient code the management of environments is assigned as much as possible to the compiler. Indeed for the Occam language an environment is really determined by the structure of the piece of program to which it belongs, namely it is the result of the still active declarations which lie on the path of locations (instructions) traversed by the executing daemon.

Therefore the environments can be computed in advance by the compilation function and be used at run time by the daemons as context for the execution of atomic Occam instructions. This means that $env(a)$ is replaced by an environment parameter e of the atomic Occam instructions, computed by *compile*.

This is the idea for the following refinement of $OCCAM_{compile}$ into the evolving algebra $OCCAM_{env}$. The function *compile* receives as additional parameter the environment within which the (sub)program has to be executed:

$$\begin{aligned} compile \quad &: \quad STATEMENT \times ENV \\ &\rightarrow \quad ((CODE \times ENV) \cup LABEL)^*. \end{aligned}$$

For better readability we will write *code(e)* instead of *(code, e)* and suppress the environment parameter for code where it doesn't matter (i.e. for SKIP, STOP, goto, end).

The rules decl_var, decl_chan, decl_end of $OCCAM_{compile}$ in section 3. are deleted because their effect is now computed by the following clauses which refine the clauses for declarations of the function *compile* of the preceding section. As we did already for labels, we abstain from the routine formalization of the creation of "new" variables and channels.

$$\begin{aligned} & compile(\mathsf{VAR}\ id_1, \ldots, id_r : S, e) \ = \ compile(S, e') \\ & \textbf{where}\ e' = append(e_{new}, e) \\ & \qquad e_{new} = (id_r, v_r), \ldots, (id_1, v_1) \\ & \qquad v_1, \ldots, v_r \text{ are "new" variables} \end{aligned}$$

$$\begin{aligned} & compile(\mathsf{CHAN}\ id_1, \ldots, id_r : S, e) \ = \\ & \quad \mathsf{init_chan}(id_1, \ldots, id_r, e'), \\ & \quad compile(S, e') \\ & \textbf{where}\ e' = append(e_{new}, e) \\ & \qquad e_{new} = (id_r, c_r), \ldots, (id_1, c_1) \\ & \qquad c_1, \ldots, c_r \text{ are "new" channels} \end{aligned}$$

Note that our compilation of declarations "allocates" statically variables and channels. During the run–time they are *reused*, therefore the initialization of channels has to be re–done every time a channel is "created". Obviously this cannot be done at compile time, therefore we have to generate an init_chan instruction with a rule which executes the updates (initializations) of the decl_chan instruction of $OCCAM_{compile}$:

$$\begin{aligned} & \textbf{if}\ cmd\ is\ \mathsf{init_chan}(c_1, \ldots, c_r, e) \\ & \textbf{then}\ agent(\bar{c_1}) := nil \\ & \qquad \ldots \\ & \qquad agent(\bar{c_r}) := nil \\ & \textbf{where}\ \bar{c_i} = bind(c_i, e) \end{aligned}$$

Note also that the generation of the instruction decl_end in $OCCAM_{compile}$ in section 3. can be eliminated because e is the "restored" environment of e'.

The compilation of atomic Occam instructions whose semantics depends upon the environment generates corresponding instructions which refine the instructions generated in the preceding subsection by passing to them also the environment parameter:

$$\begin{aligned} compile(v := t, e) &= \mathsf{ass}(v, t, e) \\ compile(\mathsf{TIME}\,?\,v, e) &= \mathsf{time}(v, e) \\ compile(c\,!\,t, e) &= \mathsf{out}(c, t, e) \\ compile(c\,?\,v, e) &= \mathsf{inp}(c, v, e) \end{aligned}$$

The rules for those refined instructions are obtained from corresponding $OCCAM_{compile}$ rules of section 3. by using the generated environment parameter e instead of $env(a)$ (except for the instructions SKIP, STOP, goto, end where nothing does change because their semantics

does not depend upon the environment):

ass(v,t,e)

if *cmd is* ass(v, t, e)
then $eval(\bar{v})$ **:=** $eval(t, e)$
proceed

time(v,e)

if *cmd is* time(v, e)
then $eval(\bar{v})$ **:=** *timer*
proceed

if(b,l,e)

if *cmd is* if(b, l, e)
then if $eval(b, e)$
then $loc(a)$ **:=** $labeled_loc(l)$
else $loc(a)$ **:=** $next(loc(a))$

Analogously for the refined rules for input and output and for alt_a and alt_s. For WHILE, IF, ALT, SEQ, PAR we have the same compilation as in $OCCAM_{compile}$ (see section 3.), adding to *compile* and to the generated instructions the parameter e. We assume as before that each generated label is "new".

compile(WHILE $B\ S, e$) =
l_{In},
if$(\neg B, l_{Out}, e)$,
compile(S, e),
goto(l_{In}),
l_{Out}

compile(IF $B_1\ S_1\ \ldots\ B_r\ S_r, e$) =
...
if(B_i, l_i, e),
...
compile(STOP$, e)$,
...
l_j,
compile(S_j, e),
goto(l_{Out}),
...
l_{Out}
where $1 \le i, j \le r$

Analogously *compile*(SEQ $S_1, \ldots, S_r$) is refined introducing the parameter e.

compile(ALT $G_1\ S_1 \ldots G_r\ S_r, e$) =
alt_a$(\vec{G}, e)$,
alt_s$(\vec{G}, \vec{l}, e)$,
...
l_i,
compile$(c_i\ ?\ v_i, e)$,
compile(S_i, e),
goto(l_{Out}),
...
l_j,
compile(S_j, e),
goto(l_{Out}),
...
l_{Out}
where $\vec{l} = l_1, \ldots, l_r$
$\vec{G} = G_1, \ldots, G_r$
$G_i = b_i\ :\ c_i\ ?\ v_i$
$G_j = b_j$: (TIME ? AFTER t_j or SKIP)
$1 \le i \le p < j \le r$

Analogously *compile*(PAR $S_1, \ldots, S_r$) is refined by introducing the parameter e. In the refined rule for par the update of the environment function *env* is deleted.

It is easy (see the proof below) to define the correspondence of runs of $OCCAM_{compile}$ and the refined evolving algebra $OCCAM_{env}$ started with $compile(S)$ and $compile(S, [\,])$ respectively.

PROPOSITION 4.5. *Corresponding runs of $OCCAM_{compile}$ and of $OCCAM_{env}$, started with $compile(S)$ and $compile(S, [\,])$ respectively, are equivalent. Therefore they preserve daemons, environments, the values of variables appearing there, communication traces, termination, deadlock and convergence.*

Proof. By induction on the structure of S. The linear structure of the compiled code is the same in both algebras, except for the declaration code of $OCCAM_{compile}$. No such code is generated in $OCCAM_{env}$ where instead *compile* – assisted by the channel initializing instruction – provides the environment computed in $OCCAM_{compile}$ at run time by the declaration code. Homonymous rules in both algebras have the same effect. The environment $env(a)$ within which an instruction is executed in $OCCAM_{compile}$ is the same as the environment parameter e of the corresponding instruction in the corresponding run of $OCCAM_{env}$. ■

In the sequel our compilation function will undergo further refinements. Besides instructions and environments several other entities will be generated like labels, daemons, etc. We use them as parameters in very the same way as we did with e. Therefore they are to be understood without further mentioning to go together with the introduction or refinement of corresponding dynamic functions which will be updated correctly in the *LOAD* algebra.

REMARK 4.2. Note that for the compilation of environments in $OCCAM_{env}$ we assume fresh variables to have value *undef*, as is the case when new variables are created in $OCCAM_{compile}$. In any case it is hardly the compiler which should be blamed if a variable used by a program is wrongly initialized.

4.4. Refining variables and channels

In this section we implement variables and channels as locations. Thus the abstract environments become mappings to blocks of memory. Formally we require:

$$VAR, CHANNEL \subseteq LOC$$

and refine the variable evaluation function *eval* and the function *agent* to be subfunctions of *content*. Due to the static memory allocation for Occam programs we can store environments internally as blocks of memory. As preparation for the refinement to relocatable memory access in the next subsection we provide the compilation function with an additional parameter m for the next free memory position starting from which the environment will be stored in a contiguous memory area:

$$\begin{aligned} compile &: STATEMENT \times ENV \times LOC \\ &\rightarrow ((CODE \times ENV) \cup LABEL)^* \end{aligned}$$

The recursive definition of the placement of environments into memory by the compilation function is assisted by an auxiliary function:

$$env_size : STATEMENT \rightarrow \mathbf{N}$$

which yields the number of memory locations needed by a subprogram for its variables and channels. This function is also recursively defined; the definition takes advantage from the decision to place environments into contiguous memory positions. We first give this recursive definition and then show how the refinement of *compile* affects the defining clauses for declarations and PAR-statements. Through declarations, the environments are extended:

$$\begin{aligned} env_size(\mathsf{VAR}\ id_1, \ldots, id_r : S) &= r + env_size(S) \\ env_size(\mathsf{CHAN}\ id_1, \ldots, id_r : S) &= r + env_size(S). \end{aligned}$$

Some statements don't introduce new variables, therefore in those cases we have:

$$\begin{aligned} env_size(\mathsf{SKIP}) &= 0 \\ env_size(\mathsf{STOP}) &= 0 \\ env_size(\mathsf{TIME}\,?\,v) &= 0 \\ env_size(c\,?\,v) &= 0 \end{aligned}$$

For output and assignment statements additional (temporary) variables might be needed through the still abstract evaluation. Thus we require:

$$\begin{aligned} env_size(c\,!\,t) &= eval_size(t) \\ env_size(v := t) &= eval_size(t) \end{aligned}$$

where *eval_size* is an auxiliary function which will be specified together with expression evaluation.

Sequential execution of SEQ components requires that the reserved place must cover the maximal space requirement among the components. Therefore:

$$env_size(\mathsf{SEQ}\ S_1 \ldots S_r) = max_{i=1}^{r}\ env_size(S_i)$$

Also statements operating on boolean expression(s) may need some additional temporary variables for evaluation; this explains the definition of *eval_size* for ALT, IF and WHILE:

$$\begin{aligned} &env_size(\mathsf{ALT}\ G_1\,S_1 \ldots G_r\,S_r) = \\ &\quad max_{i=1}^{r}\ (eval_size(b_i), env_size(S_i)) \end{aligned}$$

$$\begin{aligned} &env_size(\mathsf{IF}\ B_1\,S_1, \ldots B_r\,S_r) = \\ &\quad max_{i=1}^{r}\ (eval_size(B_i), env_size(S_i)) \end{aligned}$$

$$\begin{aligned} &env_size(\mathsf{WHILE}\ B\,S) = \\ &\quad max(\ eval_size(B), env_size(S)\ ) \end{aligned}$$

In the case of a PAR program statement, its component statements run *concurrently*, so their variables/channels must be present concurrently. Therefore the *env_size* sums the size of the components:

$$env_size(\mathsf{PAR}\ S_1 \ldots S_r) = \sum_{i=1}^{r} env_size(S_i).$$

During the compilation of declarations the first free memory location has to be updated (advanced to the next location) to express the effect of the "location in use". Thus the compilation of variable declarations of the preceding section is refined as follows.

$$\begin{aligned} &compile(\mathsf{VAR}\ id_1, \ldots, id_r : S, e, m) = \\ &\quad compile(S, e', m + r) \\ &\mathbf{where} \\ &\quad e' = append(e_{new}, e) \\ &\quad e_{new} = (id_r, m + r - 1), \ldots, (id_1, m). \end{aligned}$$

The refinement of the compilation function is similar for all other cases except for PAR. During the compilation of PAR-statements we have to pass now to each son a different free memory location for his private declarations. The requirement that all variables/channels have to be placed in a contiguous block comes handy now. We simply give each son enough space (determined using the value of *env_size* for the subprograms of the sons). Hence the compilation of PAR statements

of the preceding subsection is refined as follows:

$$\begin{aligned}
&\mathit{compile}(\mathsf{PAR}\ S_1 \dots S_r, e, m) \ = \\
&\quad \mathsf{par}(\vec{l}), \\
&\quad \mathsf{goto}(l_{Out}), \\
&\quad \dots \\
&\quad l_i, \\
&\quad \mathit{compile}(S_i, e, m_i), \\
&\quad \mathsf{end}, \\
&\quad \dots \\
&\quad l_{Out} \\
&\textbf{where } \vec{l} = l_1, \dots, l_r \\
&\qquad m_1 = m \\
&\qquad m_j = m_{j-1} + \mathit{env_size}(S_{j-1}) \\
&\qquad 1 \le i \le r \\
&\qquad 2 \le j \le r
\end{aligned}$$

The environment space splitting is shown in figure[11] 6. The space is reserved by the $LOAD$ algebra by setting *loading_loc* initially to $\mathit{env_size}(S)$, and passing 0 as first free location for the compilation function. In this way the instructions are loaded immediately after the space dedicated to the environment, i.e. starting at $\mathit{env_size}(S)$. The refinement $OCCAM_{env'}$

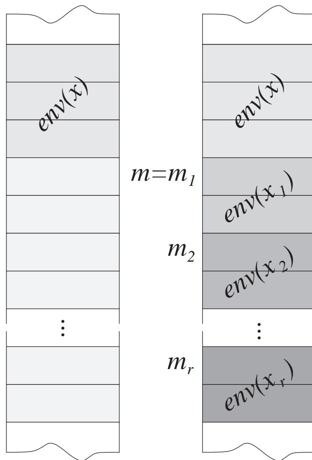


FIGURE 6. Splitting of space among new daemons x_i

of $OCCAM_{env}$ obtained by placing environments into memory is a pure data refinement for *compile* and for the initialization by $\mathit{compile}(s, [\,], 0)$. It does not involve new rules.

PROPOSITION 4.6. *Proposition 4.5 is true for* $OCCAM_{env'}$ *modulo the refinement of variables and channels.*

[11] Here and in the sequel we will put the content of the memory into boxes, the boxes themselves represent arbitrary chunks of memory. Where needed the size of each chunk is indicated next to the boxes. Sometimes, where known, the "address" where the boxes start is also annotated. Different shadowing of the boxes is intended for visual distinguishing semantically different data placed into the memory.

4.5. Relative addressing for environments

In this section we formalize a standard technique to introduce relative addressing for bindings of identifiers. The goal is to make the environment relocatable.

In the Occam ground model the role of daemons is that of agents who execute Occam programs. In the Transputer this role is reduced to that of holding the base address of the corresponding environment. Indeed the compilation function does not need to know (and in a real system cannot know) where the memory block for the environment will begin at run time.

For the compilation it is sufficient "to assume" a parameter x as base address with respect to which addresses are calculated as distances $n - x$ from x. At run time the actual address can then be calculated by adding this distance to the current base address, namely the active daemon a in our model. Making the addressing of locations mapped to identifiers relative to a daemon x as base address means that the execution of a piece of code "by x" is done by the Transputer through a computation in which the locations mapped to identifiers are accessed through x.

To realize this idea we augment the compilation function of the preceding and the binding function of section 3. with an additional parameter for the environment base address ("daemon") and place daemons into memory by requiring $DAEMON \subseteq LOC$:

$$\begin{aligned}
\mathit{compile} &: STATEMENT \times ENV \times LOC \times DAEMON \\
&\rightarrow ((CODE \times ENV \times DAEMON) \cup LABEL)^* \\
\mathit{bind} &: ID \times ENV \times DAEMON \rightarrow LOC
\end{aligned}$$

For the refined code produced by the new compilation function we put again the new daemon parameter as argument of the original code — writing *code(x)* instead of *(code, x)* — and suppress the parameter x in instructions where it plays no role.

The new function *compile* is obtained from the function of $OCCAM_{env'}$ by passing the new parameter x uniformly from arguments to rules whenever the base address (daemon) does not change. The scheme is illustrated for assignment. The compilation of assignments in subsection 4.3. is refined as follows:

$$\mathit{compile}(v := t, e, m, x) \ = \ \mathsf{ass}(v, t, e, x)$$

The new rule for assignment instructions will use the compile-time parameter x to calculate the distance and from that the correct run-time address of the correct binding of v and the variables in t (see below). New daemons appear only as consequence of the execution of PAR instructions where the new compilation function will compute these new base addresses. It is natural to define the ith son as the memory position m_i which had been assigned by the compilation function of $OCCAM_{env'}$ for his private declarations. Hence the compilation of PAR statements of the preceding subsec-

tion is refined as follows:

$$\begin{aligned}
&compile(\mathsf{PAR}\ S_1 \ldots S_r, e, m, x) \ = \\
&\quad \mathsf{par}(x, \vec{m}, \vec{l}), \\
&\quad \mathsf{goto}(l_{Out}), \\
&\quad \ldots \\
&\quad l_i, \\
&\quad compile(S_i, e, m_i, m_i), \\
&\quad \mathsf{end}, \\
&\quad \ldots \\
&\quad l_{Out} \\
&\mathbf{where} \\
&\ \vec{l} = l_1, \ldots, l_r \\
&\ \vec{m} = m_1, \ldots, m_r \\
&\ m_1 = m \\
&\ m_j = m_{j-1} + env_size(S_{j-1}) \\
&\ 1 \leq i \leq r \\
&\ 2 \leq j \leq r
\end{aligned}$$

The new rule for $\mathsf{par}(x, \vec{m}, \vec{l})$ will use the compile time distance $x_i - x$ between father x and son x_i to calculate the correct run-time base address $a + (x_i - x)$ for each i.

Similarly the new binding function computes the distance:

$$bind(id, e, x) \quad = \quad bind(id, e) - x$$

between the absolute addresses $bind(id, e)$ – computed by the binding function of $OCCAM_{env'}$ – and the base address x. In this way the relation between compile time and run time addresses is characterized by the simple equation (see Fig. 7):

run-time address = compile-time relative address + a

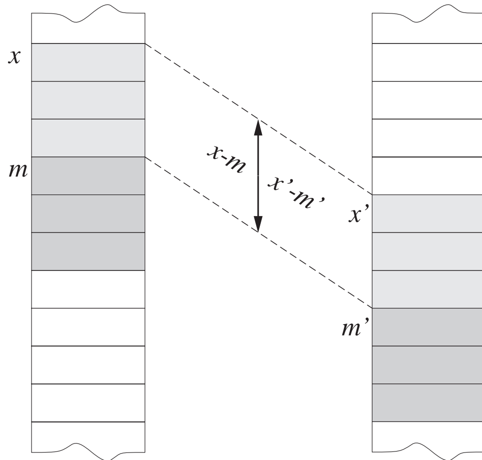


FIGURE 7. Assumed and Real addresses

Therefore the rules of the new evolving algebra $OCCAM_{rel-env}$ with relocatable environments are obtained from the rules of $OCCAM_{env'}$ as follows. Whenever $bind(id, e)$ is used it is replaced by $bind(id, e, x)$ with the appropriate x, given by the rule context. Whenever $content(\bar{v})$ is used, it is replaced by $content(\bar{v} + a)$. A typical example is the assignment rule of subsection 4.3. which is refined as follows:

$\boxed{\mathsf{ass}(v,t,e,x)}$

if *cmd is* $\mathsf{ass}(v, t, e, x)$
then $content(\bar{v} + a)$ **:=** $eval(t, e, x)$
proceed

The relative addressing of a location associated to an identifier v in an environment e with respect to the compile-time base address x is reflected by the fact that at run time the distance $\bar{v} = bind(v, e, x)$ of the identifier's address from the compile time base address x is added to the run time base address, here the value of the active daemon a. Note that the expression evaluation function *eval* which has indirectly to call *bind* to get the values of the variables which occur in the expression has to be extended by an argument for the value of the daemon which has been assumed at compile time:

$$eval \quad : \quad EXP \times ENV \times DAEMON \rightarrow VAL.$$

The same refinement is done for time and input. The new rule for $\mathsf{par}(x, \vec{m}, \vec{l})$ refines the corresponding rule in $OCCAM_{env'}$ (compare the formulation in subsection 4.1.) by computing the new daemons from the compile time parameters $x, \vec{m}$, i.e. the abstract creation of new daemons is replaced by a definition of the new base addresses they represent. The definition of the run time value x'_i for the ith son adds the run-time value of the creating father – namely a – to the distance $x_i - x$ of the ith son from its father which has been computed by the compilation function.

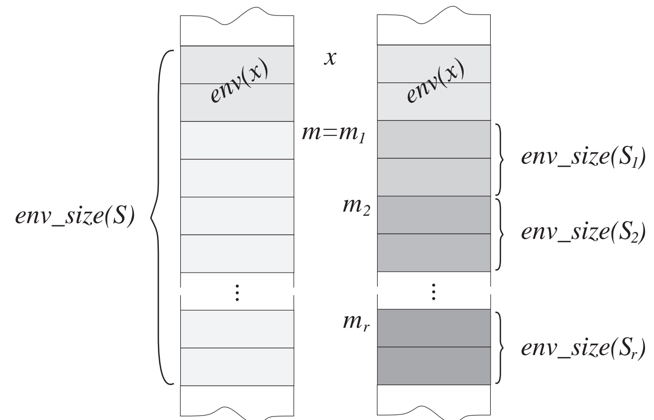


FIGURE 8. "Creation" of daemons in PAR

$\boxed{\mathsf{par}(x, \vec{x}, \vec{l})}$

if *cmd is* $\mathsf{par}(x, \vec{x}, \vec{l})$
then $q := q.x'_1. \ldots .x'_r$
...
$father(x'_i) := a$
$loc(x'_i) := labeled_loc(l_i)$
...
$count(a) := r$
sleep at $next(loc(a))$
where $\vec{l} = l_1, \ldots, l_r$
$\vec{x} = x_1, \ldots, x_r$
$x'_i = a + x_i - x$
$1 \leq i \leq r$

Figure 8-a shows the interpretation of daemons as base addresses, figure 8-b illustrates the optimization of the next section where the father x takes over the role of his first son. The effect of the relative addressing is expressed by the following lemma.

LEMMA 4.7. *Let m, x, m', x' be such that $m - x = m' - x'$. Then the computations of $OCCAM_{rel-env}$ started with $compile(S, [], m, x)$ and $compile(S, [], m', x')$ respectively are equivalent.*

Proof. The claim follows by induction an S and induction on the computations using the definition of the new binding function and the definition of new daemons for the PAR construct, which have been given in terms of distances to x. ∎

COROLLARY 4.8. *Corresponding runs of $OCCAM_{env'}$ and $OCCAM_{rel-env}$, started with $compile(S, [], 0)$ and $compile(S, [], 0, 0)$ respectively, are equivalent, preserving daemons, environments, the values of variables appearing there, communication traces, termination, deadlocks and convergence.*

4.6. Optimization for PAR statements

Figure 8 shows that when a daemon x creates sons $x_1, \ldots, x_k$, his next free memory position m coincides with the first free memory position m_1 of his first son x_1. Due to the fact that the father is sleeping while his sons are running we can reuse the father to do the work of his first son; indeed due to our compilation schema the environment of the first son is adjacent to the father's environment. In this way we can save the creation of one daemon. This idea can be realized by the following refinement of $OCCAM_{rel-env}$ to an optimized evolving algebra $OCCAM_{daemon}$. The assignment of the environment space is refined by not creating x_1 any more. The compilation function is refined on the PAR statement by eliminating the first element from $\vec{m}$ and $\vec{l}$ — i.e. creating only $k-1$ instead of k sons — and by eliminating the instruction $\mathsf{goto}(l_{Out})$, i.e. by letting the father proceed with the execution of the subprogram of the old first son. As a consequence the last (finishing) son must take up the role of his father instead of waking him up.

This implies refinements of the rules for the instructions par and end of the preceding subsection (see below).

$compile(\mathsf{PAR}\ S_1 \ldots S_r, e, m, x) =$
$\mathsf{par}(x, \vec{m}, \vec{l}, l_{Out})$,
l_1,
$compile(S_1, e, m, x)$,
end,
l_2,
$compile(S_2, e, m_2, m_2)$,
end,
...
l_r,
$compile(S_r, e, m_r, m_r)$,
end,
l_{Out}
where
$\vec{l} = l_2, \ldots, l_r$
$\vec{m} = m_2, \ldots, m_r$
$m_1 = m$
$m_i = m_{i-1} + env_size(S_{i-1})$
$2 \leq i \leq r$

When the last son finishes his work — i.e. when he executes the end instruction — he has to reset his base address to the base address of his father and to proceed with the program of his father. For this purpose the goto-parameter l_{Out} has to be made available. We store it through a function:

$$end_par : DAEMON \rightarrow LABEL$$

The par-instruction is refined to include l_{Out} as parameter which upon execution of the par-rule is assigned to $end_par(a)$; upon execution of the refined end-rule it will be used to let the last son take up the work of his father.

$\boxed{\mathsf{par}(x, \vec{x}, \vec{l}, l_{Out})}$

if *cmd is* $\mathsf{par}(x, \vec{x}, \vec{l}, l_{Out})$
then $q := q.x'_1. \ldots .x'_r$
...
$father(x'_i) := a$
$loc(x'_i) := labeled_loc(l_i)$
...
$count(a) := r$
$end_par(a) := l_{Out}$
proceed
where $\vec{l} = l_1, \ldots, l_r$
$\vec{x} = x_1, \ldots, x_r$
$x'_i = a - x + x_i$
$1 \leq i \leq r$

```
end
if cmd is end
then if r > 1
       then dequeue
              r := r − 1
       else a := x_f
              loc(x_f) := a_Out
where r = count(x_f)
          x_f = father(a)
          l_Out = end_par(x_f)
          a_Out = labeled_loc(l_Out)
```

PROPOSITION 4.9. *Let $OCCAM_{daemon}$ be the optimization of $OCCAM_{rel-env}$, obtained by saving one son in the compilation of* PAR *statements. Runs of $OCCAM_{daemon}$ are equivalent to the corresponding runs of $OCCAM_{rel-env}$. Therefore environments, the values of variables appearing there, communication traces, deadlocks and divergence are preserved.*

Proof. The runs in the two algebras are in one-to-one correspondence: the same rules are applied although by different agents. Indeed fathers work for their first sons and they do this correctly, as assured by the refined clause for the *compile* function: x goes to execute the code of his first son (because the goto instruction has been eliminated); the last son instead of killing himself takes up the role of his father by reseting himself (the base address) to that of his father and his location to where the father had stopped to do his own job. The compilation of the variables of x_1 is done correctly because the environment produced for x_1 is the same as before, it is only accessed from the father's base address x instead of x_1. ■

We can summarize the results of this section by the following theorem:

THEOREM 4.10. *$OCCAM_{daemon}$ is a correct and complete implementation of $OCCAM_{ground}$, i.e. for each* Occam *program, its $OCCAM_{ground}$ semantics is the same as its $OCCAM_{daemon}$ semantics.*

5. Transputer (GROUND) MODEL

In this section we introduce the signature of the Transputer ground model which is relevant for the execution of the Transputer code to which the abstract code of the preceding section will be refined in the next section.

We introduce here the registers *Wreg* (for the current agent) and *Ireg* (for the pointer to the current instruction), the three (stack like) registers *Areg, Breg, Creg* for the management of instruction operands and the registers for the implementation of the queues q and *time_q*. We also define which Transputer instructions are safe for time–slicing and introduce the instruction fetch mechanism which also takes care of the Transputer time–slicing mechanism.

5.1. Queue Implementation

Two special–purpose Transputer registers[12]:

$$FPtrReg, BPtrReg \;:\; DAEMON \cup \{nil\}$$

known as *front/back* (queue) pointer registers, together with the function[13]:

$$next \;:\; DAEMON \rightarrow DAEMON$$

implement the abstract queues of $OCCAM_{daemon}$.

As a matter of fact, the latter can be recovered by the following definition of the list $queue(l)$:

$$q \;=\; queue(FPtrReg)$$

$$queue(l) \;=\; \begin{cases} l & \text{if } l = BPtrReg \\ l.queue(next(l)) & \text{otherwise} \end{cases}$$

(Remember that we use the dot for list notation.) Clearly $first(q) = FPtrReg$, $last(q) = BPtrReg$, $rest(q) = queue(next(FPtrReg))$. The queue is said to be empty if $FPtrReg = nil$; we assume that the queue registers are initialized with *nil*. The abstract updates for enqueuing and dequeuing of the Occam algebra in section 3. are now implemented by the following macro:

```
enqueue x ≝ BPtrReg := x
            if FPtrReg = nil
            then FPtrReg := x
            else next(BPtrReg) := x
```

The macro *dequeue* formalizes the dequeuing mechanism. Note that if the process queue is empty, then the cpu is idle, waiting for some daemon to appear (either by completing an external communication or by an application of the time_wakeup rule).

```
dequeue ≝ if FPtrReg = nil
          then a := nil
          else a := FPtrReg
               loc(a) := loc(FPtrReg)
               start := timer
               if FPtrReg = BPtrReg
               then FPtrReg := nil
               else FPtrReg := next(FPtrReg)
```

Note that strictly speaking the update $loc(a) := loc(FPtrReg)$ is unnecessary; we include it already here because in subsection 5.3. we refine $loc(a)$ to be the register *Ireg* which is independent of a.

It is a routine exercise to formalize and to prove the correctness and completeness statement for this implementation of queues of $OCCAM_{daemon}$. Note that this refinement is essentially a data refinement.

5.2. Time queue

The administration of time is implemented in the Transputer by Timer–hardware. It takes care of incre-

[12] Really two for each priority

[13] Not to be confused with the function *next* in the flowcharts of $OCCAM_{ground}$.

menting the *timer* function(s) and administrating the time queue.

The time queue is implemented by a 0-ary function[14] (a reserved location (register) Time Pointer Location):

$$TPtrLoc \quad : \quad DAEMON \cup \{nil\}$$

together with the above indicated function *next*. Note that we are allowed to reuse the same *next* function for scheduler and timer queue because these two queues are always disjoint. Assume that an ALT process is waiting for a communication partner and for the time; at the moment he is woken up by his communication partner he is thrown out from the timer queue and put into the scheduler queue (see out_alt and OUT_alt rules). Clearly *time_q* can be recovered in the standard way from $TPtrLoc = first(time_q)$ using *next*. In the Transputer we find for efficiency reasons also another reserved location (register):

$$TNextReg \quad : \quad LOC$$

to hold the time the process in *TPtrLoc* is waiting for (i.e. $TNextReg = t_{min}(TPtrLoc)$).

It is routine to refine the notions of *time-insert* and *time-delete* accordingly.

When the time recorded in *TNextReg* is reached the refined time_wakeup rule fires and wakes up the corresponding waiting daemons. The refined version of the time_wakeup rule of section 3. is the following:

time_wakeup
if $TPtrLoc \neq nil \wedge timer > TNextReg$
then $com_mode(TPtrLoc) := alt_run$
enqueue TPtrLoc
time-delete TPtrLoc

We leave again as an exercise to the reader to formalize and prove the correctness and the completeness statement for this implementation of time queues — which is again a pure data refinement. Note that the time_wakeup rule runs concurrently to the other rules, thus reflecting that an independent piece of hardware is responsible for timing.

Note that on the basis of the notions of process queue and time queue and with appropriate register initialization we can define deadlock as process and timer queue being empty without any process waiting for an external communication.

5.3. Registers

The Transputer has three general purpose registers:

$$Areg, Breg, Creg \quad : \quad VAL$$

which behave like a stack, allowing us to access directly only the top of the stack, *Areg*[15]. Therefore we will use the following stack–like macros for accessing registers:

$push(v) \stackrel{\text{def}}{=} Areg := v,\ Breg := Areg,\ Creg := Breg$
$pop \stackrel{\text{def}}{=} Areg := Breg, Breg := Creg,\ Creg := random$
$pop(v) \stackrel{\text{def}}{=} v := Areg,\ pop$

In *pop* we use a 0-ary external function *random* which takes values in *VAL*. The update of *Creg* by *random* in *pop* formalizes the non–deterministic behavior of the hardware which may assign any value to *Creg*.

The Transputer has two special–purpose registers:

$$Wreg, Ireg \quad : \quad VAL$$

to hold the current agent and a pointer to the current instruction. From the *workspace (pointer) register Wreg* and the *instruction (pointer) register Ireg* we can recover *a* and *loc* of $OCCAM_{daemon}$ as follows:

$$a \stackrel{\text{def}}{=} Wreg$$
$$loc(a) \stackrel{\text{def}}{=} Ireg$$

Correspondingly the macro *proceed* of previous sections becomes:

$$proceed \stackrel{\text{def}}{=} Ireg := Ireg + 1$$

Similarly the definitions and rules where *a* and *loc*(*a*) appear have to be interpreted with *Wreg* and *Ireg* (see in particular the dequeue rules). For the time being it suffices to assume that the general purpose Transputer registers range over a subdomain of *LOC* so that the function *content* can be applied to registers.

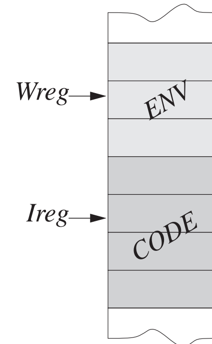


FIGURE 9. Example of *Wreg* and *Ireg*

Since this introduction of Transputer registers is a pure data refinement, with respect to an appropriate register initialization it preserves the correctness and the completeness statement.

5.4. Time slicing

As a rule the scheduling of processes is a time consuming operation. In fact before actually starting a process one has to restore his context, represented by certain processor registers which have to be preserved on

[14] Really for each priority

[15] By requiring that the registers take the values in *VAL* we abstract from the hardware implementation of registers; e.g. the implementation of the workspace register introduced below, in **T800** uses only 30 bits for values and one bit for priority.

rescheduling. For the sake of efficiency this context — in a narrow sense — has been reduced in the Transputer to *Wreg* and *Ireg* only. The introduction of the notion of safe places defines "safe" points in a program starting from where the code doesn't depend on the context — in a broad sense — including values of *Areg*, *Breg* and *Creg* registers which have been generated before reaching such a point. It is the duty of the compilation function to meet the proposed requirement for being a safe place when the code is generated. At such safe places time-slicing can be applied correctly.

For time-slicing it is natural that its execution is coupled to the *fetch* phase in which, anyway, the CPU has to recognize the type of instruction to be executed. We formalize this by introducing a 0-ary function:

$$mode \quad : \quad \{fetch, execute\}$$

together with a function[16]:

$$cmd \quad : \quad CODE$$

which holds the instruction fetched by the CPU for execution so that $cmd = content(Ireg - 1)$.

Since all the rules defined so far have to do with the execution of instructions they are put under the additional guard $mode = execute$ by refining the macro *cmd is* of section 3. to:

$$cmd\ is\ c \quad \stackrel{\text{def}}{=} \quad cmd = c \wedge mode = execute \wedge Wreg \neq nil$$

and by adding to each rule the update $mode := fetch$ and deleting the update *proceed*. The time–slice rule of section 3. is refined correspondingly as follows:

```
time–slice
if mode = fetch ∧ Wreg ≠ nil
then if elapsed
        ∧ safe(content(Ireg))
     then seq enqueue Wreg
              dequeue
          endseq
     else cmd := content(Ireg)
          proceed
          mode := execute
```

Under the assumption that in an infinite run there are infinitely many *safe* instructions which are executed this new rule is a correct implementation of the previous time–slice rule. We reassume this section in the following proposition.

PROPOSITION 5.1. *The refinement of $OCCAM_{daemon}$ by implementing process and time queues and by introducing time–slicing as part of the fetch mechanism is correct and complete.*

Proof. It is easy to check comparing homonymous rules that the $OCCAM_{daemon}$ update *proceed* is handled correctly by the time–slicing rule. By definition a "safe" place in a program is represented by code which doesn't depend on the "context" in the broad sense generated before reaching that place but only on the "context" in the narrow sense. The "context" in the broad sense is represented in the Transputer by *Wreg*, *Ireg* and the three stack registers *Areg*, *Breg*, *Creg* for holding the parameters. The claim therefore follows by induction on the number of times when the time has elapsed for time–slicing, using the fact that Occam processes which run in parallel are independent and in particular do not compete for resources.

Note that the Transputer designers have defined the jump instruction (as well as the loop end instruction LEND) to be safe. (Note that our compilation function prevents pieces of code which follow a jump instruction to depend on the values of *Areg*, *Breg*, *Creg* unless these registers are set by that piece of code.) The compilation of WHILE constructs contains a jump so that no execution of a compiled WHILE program on the Transputer can diverge without containing infinitely many executions of jumps and therefore time–slicing. ■

[16] Not to be confused with the function *cmd* of the Occam algebra.

6. COMPILATION INTO Transputer CODE

In this section we refine the *compile* function of section 4. to output instructions which are executed in the Transputer model defined in section 5.. We do this in three steps: first we compile into code which still uses the abstract auxiliary functions of $OCCAM_{daemon}$ but implements the relative addressing of variables and channels by the run–time calculation of the absolute addresses from the distance from the base address which has been calculated at compile–time. Then we implement the abstract auxiliary Occam functions by generalizing *Wreg* from the daemon's base address to the process workspace it represents (points to). We refine the *compile* function according to the workspace implementation and finally make the code relocatable by applying the technique of relative branching (relative addressing for target positions of branching instructions).

Our definition of the *compile* function really describes a specific compiler. However this definition is by no means unique. Any definition will do which satisfies the constraints which we use for our correctness proof.

In this section we make extensive use of the locality principle built into evolving algebras which allows us to proceed instructionwise. Instruction per instruction we define the new function *compile* together with the corresponding rule(s) for the execution of the produced target code and prove the correctness with respect to $OCCAM_{daemon}$. These local proofs are mostly trivial and altogether will prove the following theorem:

THEOREM 6.1. *($TRANSPUTER_{ground}$ correctness) Arbitrary runs of $OCCAM_{daemon}$ on $compile_{daemon}$ $(S, [], 0, 0)$ are implemented correctly by corresponding runs of $TRANSPUTER_{ground}$ on $compile_{ground}(S, [], 0, 0)$ where $TRANSPUTER_{ground}$*

is the evolving algebra defined in section 5. and 6.1. and where $compile_{daemon}$ *and* $compile_{ground}$ *are the respective compile functions.*

From the following definition of the new rules for $TRANSPUTER_{ground}$ which execute the code produced by the new function *compile* it will be clear how to define the "correspondence" of runs on the basis of "local" correspondences of single $OCCAM_{daemon}$ instructions to short sequences of $TRANSPUTER_{ground}$ instructions.

Note that the theorem cannot be extended to completeness of corresponding runs due to the particular Transputer strategy for the computation of ALT.

6.1. Abstract Transputer compilation

In this section we define the compilation to Transputer instructions which still uses abstract auxiliary $OCCAM_{daemon}$ functions.

We proceed stepwise, defining for each Occam statement S the value of *compile* together with the $TRANSPUTER_{ground}$ rules for the execution of the code. Each time we show that this implements correctly the semantics of S as compiled to and executed in $OCCAM_{daemon}$.

Declarations

The compilation of variable declarations remains the same as in $OCCAM_{daemon}$. For the channel declarations (see subsection 4.3.) we have to compile the pseudo instruction init_chan for the initialization of channels to *nil*. This is realized by first loading *nil* into the register *Areg* (using the `MINT` instruction) and then storing it from there to the channel (using the local storing instruction `STL`) with appropriate address:

$$
\begin{aligned}
&compile(\mathsf{CHAN}\ id_1,\ldots,id_r : S, e, m, x) = \\
&\quad compile(\mathsf{init_chan}(\vec{id}), e', m, x), \\
&\quad compile(S, e', m+r, x) \\
&\mathbf{where}\ \vec{id} = id_1,\ldots,id_r \\
&\qquad e' = (id_r, m+r-1),\ldots,(id_1,m), e
\end{aligned}
$$

$$
\begin{aligned}
&compile(\mathsf{init_chan}(\vec{c}), e, m, x) = \\
&\quad \ldots \\
&\quad \texttt{MINT}, \\
&\quad \texttt{STL}\ bind(c_i, e, x), \\
&\quad \ldots \\
&\mathbf{where}\ \vec{c} = c_1,\ldots,c_r \\
&\qquad 1 \le i \le r
\end{aligned}
$$

if *cmd is* `MINT` **then** $push(nil)$

if *cmd is* `STL` n **then** $pop(content(Wreg + n))$

Note that the binding function (see section 4.5.) yields the offset $bind(id, e, x)$ calculated at compile time with respect to the compile–time base address x; this relative address is used at run time by the instruction `STL` to calculate the absolute address by adding the current base address *Wreg*. In this way the environment becomes spurious at run-time.

Looking at the rule for init_chan in $OCCAM_{daemon}$ (see section 4.3.) it should become clear that the following correctness lemma holds for declaration statements:

LEMMA 6.2. (CORRECTNESS LEMMA) *For declaration statements S of* Occam, *the execution of the rules of* $OCCAM_{daemon}$ *on* $compile_1(S, e, m, x)$ *is correctly implemented by the execution of the rules of* $TRANSPUTER_{ground}$ *on* $compile_2(S, e, m, x)$ *where* $compile_1$, $compile_2$ *denote the function compile of* $OCCAM_{daemon}$ *and* $TRANSPUTER_{ground}$ *respectively.*

Assignment

We still keep the expression evaluation abstract by using an abstract instruction eval which stores the evaluation result in *Areg*. The compilation of $v := t$ has then to make sure that from *Areg* this value gets stored into the right variable. Remembering that for evaluation of variables we have $eval(v, e, x) = content(\bar{v} + Wreg)$, where $\bar{v} = bind(v, e, x)$, makes clear that the following compilation of assignment statements satisfies the correctness lemma 6.2 (see the compilation of assignments in section 4.5.); similarly for time statements where instead of an expression the current value of *timer*[17] is loaded into *Areg*.

$$
\begin{aligned}
&compile(v := t, e, m, x) = \\
&\quad \mathsf{eval}(t, e, x) \\
&\quad \texttt{STL}\ bind(v, e, x)
\end{aligned}
$$

$$
\begin{aligned}
&compile(\mathsf{TIME}\,?\,v, e, m, x) = \\
&\quad \texttt{LDTIMER} \\
&\quad \texttt{STL}\ bind(v, e, x)
\end{aligned}
$$

if *cmd is* eval(t, e, x) **then** $push(eval(t, e, x))$

if *cmd is* `LDTIMER` **then** $push(timer)$

There is no Transputer equivalent for the *no-operation* instruction SKIP, therefore it disappears:

$$compile(\mathsf{SKIP}, e, m, x) =$$

The STOP instruction is directly implemented by the Transputer instruction `STOPP` (see the stop rule in section 3.):

$$compile(\mathsf{STOP}, e, m, x) = \texttt{STOPP}$$

if *cmd is* `STOPP` **then** *Areg, Breg, Creg* := *random* *dequeue*

Note that in our compilation schema the process executing `STOPP` will never be woken up, thus rendering the necessity of storing *Ireg* in the workspace — as it is really done in the Transputer — superfluous.

Obviously the correctness lemma 6.2 holds for the new compilation of SKIP and STOP.

The compilation of SEQ remains unchanged.

[17] The *timer* function for the priority of the currently active process is used.

Input

The compilation of an input statement starts with instructions LDLP which loads the absolute variable/channel addresses into the first two stack registers. Then the instruction IN uses the information stored there and actually takes the message from the channel and stores it into the variable.

$compile(c\,?\,v, e, m, x)$ =
  LDLP $bind(c, e, x)$,
  LDLP $bind(v, e, x)$,
  IN

if *cmd is* LDLP n **then** $push(Wreg + n)$

The corresponding rules are the following:

IN_idle
if *cmd is* IN $\wedge$ $content(Breg) = nil$
then $content(Breg)$:= $Wreg$
  $place(Wreg)$:= $Areg$
  $com_mode(Wreg)$:= $input$
  dequeue

IN_ready
if *cmd is* IN $\wedge$ $content(Breg) \neq nil$
then $content(Areg))$:= $mssg(content(Breg))$
  $content(Breg)$:= nil
  enqueue $content(Breg)$

Clearly the rules IN_idle and IN_ready directly implement the $OCCAM_{daemon}$ rules inp_idle(x, c, v) and inp_ready(x, c, v) for internal channels, with a being $Wreg$, $\bar{c}$ being $Breg$ and $\bar{v}$ being $Areg$ (loaded by LDLP) (see the input rules in section 3. and 4.3.). This proves the correctness lemma 6.2 for the compilation of input commands. [18]

Output

Also in the output case[19] we abstract from the memory organization (see footnote 18). Note however that we could have use a specialized OUTW instruction which explicitly uses a unit size.

$compile(c\,!\,t, e, m, x)$ =
  LDLP $bind(c, e, x)$,
  eval(t, e, x)
  OUT

As in the input case, the following rules OUT_idle, OUT_ready and OUT_alt directly implement the $OCCAM_{daemon}$ rules out_idle(x, c, t), out_ready(x, c, t) and out_alt(x, c, t); this proves the correctness lemma 6.2 for the compilation of output commands.

OUT_idle
if *cmd is* OUT
  $\wedge$ $content(Breg) = nil$
then $content(Breg)$:= $Wreg$
  $mssg(Wreg)$:= $Areg$
  dequeue

OUT_ready
if *cmd is* OUT
  $\wedge$ $content(Breg) \neq nil$
  $\wedge$ $com_mode(content(Breg)) = input$
then $content(place(content(Breg)))$:= $Areg$
  $content(Breg)$:= nil
  enqueue $content(Breg)$

OUT_alt
if *cmd is* OUT
  $\wedge$ agent $\neq nil$
  $\wedge$ com_mode(agent) $\neq input$
then *dequeue*
  agent := $Wreg$
  $mssg(Wreg)$:= $Areg$
  if com_mode(agent) = alt_sleep
  then *time-delete* agent
    com_mode(agent) := alt_run
    enqueue agent
where agent = $content(Breg)$

Conditionals

For IF and WHILE we compile the pseudo instruction if by using an unconditional jump instruction:

$compile$(IF $B_1\,S_1\ \ldots\ B_r\,S_r, e, m, x)$ =
  ...
  $compile$(if$(B_i, l_i), e, m, x)$,
  ...
  $compile$(STOP$, e, m, x)$,
  ...
  l_j,
  $compile(S_j, e, m, x)$,
  J l_{Out},
  ...
  l_{Out}
where $1 \le i, j \le r$

$compile$(if$(B, L), e, m, x)$ =
  eval$(\neg B, e, x)$,
  CJ L

The negated boolean expression $(\neg B)$ is evaluated because CJ performs the conditional branch if the value in

[18] Since at this level of abstraction we don't deal with the organization of memory into bytes, words, etc., we omit the compilation of the message size which is taken here as unit size.

[19] Note that when the compiler does also the compilation of expressions then the expression evaluation would first generate code for the expression evaluation and then will load the address of the channel (and then reorder the values in the registers).

Areg is *false*:

> **if** *cmd is* CJ L
> **then if** *Areg* = *false*
> **then** *Ireg* := *labeled_loc*(L)
> **else** *Areg* := *Breg*
> *Breg* := *Creg*
> *Creg* := *random*

compile(WHILE $B\ S, e, m, x$) =
l_{In},
compile(if($\neg B, l_{Out}$), e, m, x),
compile(S, e, m, x),
J l_{In},
l_{Out}

Clearly, this compilation of IF and WHILE satisfies the correctness lemma 6.2 (see their compilation in section 4.2.).

Time–Slicing

The pseudo instruction goto is implemented by the Transputer instruction J and serves as *safe* place to perform time–slicing.

Statistical analysis of program behavior has taught us the well known locality rule for optimizations, saying that *the locations which will be used in the next execution step won't be far away from the data used in the last step*. The rule can be split into two rules, one for data and one for code. From this viewpoint branching instructions are expensive. They destroy the locality of code because the next instruction can be located very far away from the previous instruction location; implicitly they also destroy the locality of data because it is highly improbable that the context for code which is far away from the last instruction will have anything in common with the current context. From the point of view of locality branching is similar to rescheduling. Therefore branching places are a *natural* place for rescheduling of processes[20], i.e. to define function *safe* to be *true* only for the jump instruction.

if *cmd is* J L **then** *Ireg* := *labeled_loc*(L)

To prove the correctness of this implementation of time slicing it is sufficient to consider the following. The only instruction in Occam which can produce a loop is WHILE. The compilation of WHILE however contains a jump which will be executed at the end of each iteration. Therefore no compiled Occam program, when executed on the Transputer, can diverge without containing an infinite number of executions of jumps and thereby of time slicing.[21]

For ALT-statements we have to compile the two pseudo-instructions alt_a and alt_s. The compilation of alt_a results in "enabling" code, the compilation of alt_s in code for "disabling" and the selection of one alternative.

ALT announce

The parallelism which is built into evolving algebras through the simultaneous execution of possibly many updates allowed us in $OCCAM_{daemon}$ to formulate alt_a as *one* computation step of the following simultaneous actions for the guards (see section 3.):

1. check whether among the communication guards there are some with true boolean condition; if yes, enable the corresponding channel;
2. compute the new minimal time requirement $t_{min}(a)$ appearing in time guards with true boolean condition;
3. check whether there is at least one SKIP command with true boolean condition;
4. depending on the result of 1 – 3 proceed to compute the communication mode (*com_mode*) for a or otherwise insert a into the time queue and send him to sleep.

The Transputer has to do these computations in some order. *compile* will produce for each communication guard the channel enabling instruction ENBC (preceded by instructions LDLP to load the channel $bind(c, e, x)$ and the evaluated boolean condition eval(b, e, x)). Upon execution ENBC will also record into an auxiliary function:

$$alt_ready \quad : \quad DAEMON \rightarrow \{true, false\}$$

whether there is already an outputting partner waiting for the communication. (An initialization instruction, namely ALT or TALT, will set this auxiliary function at the beginning to *false*.) For each time guard *compile* produces the time enabling instruction ENBT (preceded by instructions to evaluate the time argument eval(t, e, x) and the boolean condition eval(b, e, x)). Upon execution ENBT records into an auxiliary function:

$$time_enabled \quad : \quad DAEMON \rightarrow \{ true, false \}$$

whether there is at least one enabled time guard and updates $t_{min}(a)$ if the encountered enabled time value is smaller then the previously encountered ones. (The appropriate initialization of *time_enabled*(a) to *false* is again done by the initialization instruction TALT.) For each SKIP guard *compile* produces an ENBS instruction (preceded by an instruction eval(b, e, x)) which will set *alt_ready*(a) to *true* if *eval*(b, e, x) is *true*.

The compilation of alt_a terminates with an alt–waiting instruction ALTWT or TALTWT whose execution computes (through evaluation of *alt_ready*, t_{min} and

[20]The conditional jump differs from the unconditional jump because its execution destroys the locality only in one of the two possible cases. That is the reason why usually the locality preserving non jumping case is favored.

[21]Note that there are however simple Transputer programs which produce infinite runs without any unconditional jump. Here is an example, implementing an unconditional jump by a conditional jump with guard which is constantly false: L, LDC 0, CJ L. Such programs do not appear as result of our compilation function.

time_enabled) whether the current daemon is to be sent to sleep (and into timer queue) or whether he can proceed to execute the code for alt_s.

This explains the following definitions for the compilation of alt_a and alt_s instructions[22] and proves the correctness lemma 6.2 for them. (A technical detail: in the initializing instructions ALT and TALT *com_mode*(*a*) is set to *alt_run* which will be changed to *alt_sleep* if ALTWT or TALTWT finds out that indeed the process has to go to sleep.) For an explanation of the update of the auxiliary function *alt_choice* in ALTWT and TALTWT see below.

compile(alt_a($\vec{G}$), e, m, x) =
  ALT,
  ...
  LDLP *bind*(c_i, e, x),
  eval(b_i, e, x),
  ENBC,
  ...
  eval(t_j, e, x),
  eval(b_j, e, x),
  ENBT,
  ...
  eval(b_k, e, x),
  ENBS,
  ...
  ALTWT
where
  $\vec{G} = G_1, \ldots, G_r$
  $G_i = b_i \ : \ c_i \,?\, v_i$
  $G_j = b_j \ :$ TIME ? AFTER t_j
  $G_k = b_k \ :$ SKIP
  $1 \leq i \leq p < j \leq q < k \leq r$

In case $l \neq p$, the same sequence of instructions but with TALT and TALTWT instead of ALT and ALTWT respectively is generated.

if *cmd is* TALT | ALT
then *time_enabled*(*Wreg*) := *false* |
  com_mode(*Wreg*) := *alt_run*
  alt_ready(*Wreg*) := *false*

if *cmd is* ENBC
then if *Areg* = *true*
  then if *content*(*Breg*) = *nil*
    then *content*(*Breg*) := *Wreg*
    elsif *content*(*Breg*) ≠ *Wreg*
    then *alt_ready*(*Wreg*) := *true*
  Breg := *Creg*
  Creg := *random*

Note that the Transputer scheme keeps *Areg* untouched by "enabling" instructions ENBC, ENBT and ENBS in order to offer the possibility to collect the boolean guard appearing there.

if *cmd is* ENBT
then if *Areg* = *true*
  then if *time_enabled*(*Wreg*) = *false*
    then *time_enabled*(*Wreg*) := *true*
      t_{min}(*Wreg*) := *Breg*
    elsif *Breg* < t_{min}(*Wreg*)
    then t_{min}(*Wreg*) := *Breg*
  Breg := *Creg*
  Creg := *random*

if *cmd is* ENBS
then if *Areg* = *true*
  then *alt_ready*(*Wreg*) := *true*

if *cmd is* TALTWT|ALTWT
then if t_{min}(*Wreg*) ≥ *timer* ∧ |
    alt_ready(*Wreg*) = *false*
  then if *time_enabled*(*Wreg*)
    then *time-insert*
    com_mode(*Wreg*) := *alt_sleep*
    dequeue
  Areg, Breg, Creg := *random*
  alt_choice(*Wreg*) := *nil*

The selection of an alternative through alt_s is done in the Transputer by searching sequentially through all communication, time and skip guards for the first one which is ready. This deterministic Transputer strategy is responsible for the loss of completeness of $TRANSPUTER_{ground}$ with respect to $OCCAM_{daemon}$; only correctness can be preserved.

A realization of this deterministic Transputer strategy needs also a mechanism to prevent the processor from getting involved in a further ready alternative once the first one has been encountered. We formalize this by an auxiliary function:

alt_choice : *DAEMON* → *LABEL* ∪ { *nil*}

which will hold (the label for) the first ready alternative and has to be initialized to *nil* **before** starting the execution of the alt_s code. This is why *alt_choice* is initialized in the rule for the last instruction issued by *compile* for alt_a (see above).

compile will produce for each guard an instruction which covers the disjunctive choice among readiness through communication, time or skip guards; these instructions are DISC, DIST, DISS and each of them will be preceded by instructions which load the corresponding channel or time value, the boolean guard and the label marking the code for the alternative. The last instruction generated for the compilation of alt_s, namely ALTEND, branches to the selected first (ready) alternative by setting *Ireg* to the location recorded in *alt_choice*(*Wreg*).

ALT select

[22] For the sake of definiteness we choose as order communication guards followed by time guards followed by SKIP guards. Since completeness cannot be preserved when a particular order is chosen, any order will do for the correctness claim. A suggestion of INMOS [Inmos:88] for making ALT appear non deterministic is to let the function *compile* reorder the alternatives randomly.

This explains the following compilation [23] of alt_s and proves the correctness lemma 6.2 for ALT–statements.

```
compile(alt_s(G⃗, l⃗), e, m, x) =
    ...
    LDLP bind(c_i, e, x),
    eval(b_i, e, x),
    LDC l_i,
    DISC,
    ...
    eval(t_j, e, x),
    eval(b_j, e, x),
    LDC l_j,
    DIST,
    ...
    eval(b_k, e, x),
    LDC l_k,
    DISS,
    ...
    ALTEND
  where
    l⃗ = l_1, ..., l_r
    G⃗ = G_1, ..., G_r
    G_i = b_i : c_i ? v_i
    G_j = b_j : TIME ? AFTER t_j
    G_k = b_k : b_k : SKIP
    1 ≤ i ≤ p < j ≤ q < k ≤ r
```

```
if cmd is DISC
then if Breg
     then if content(Creg) = Wreg
          then content(Creg) := nil
               Areg := false
          elsif content(Creg) ≠ nil
                ∧ alt_choice(Wreg) = nil
          then alt_choice(Wreg) := Areg
               Areg := true
          else Areg := false
     else Areg := false
     Breg, Creg := random
```

```
if cmd is DIST
then if Breg
     then if Creg < timer
             ∧ alt_choice(Wreg) = nil
          then alt_choice(Wreg) := Areg
               Areg := true
          else Areg := false
     else Areg := false
     Breg, Creg := random
```

[23]To avoid repetition we skip the compilation of PRI ALT; it would come up to respect the order in which the guards appear in the source program.

```
if cmd is DISS
then if Breg
     then if alt_choice(Wreg) = nil
          then alt_choice(Wreg) := Areg
               Areg := true
          else Areg := false
     else Areg := false
     Breg, Creg := random
```

```
if cmd is ALTEND
then Ireg := labeled_loc(alt_choice(Wreg))

if cmd is LDC n then push(n)
```

Parallelism

For PAR statements we have to compile the two pseudo instructions par and end. The Transputer has to start the processes sequentially, for definiteness we stick here to the order in which they appear in the Occam PAR statement. The start instruction STARTP is preceded by instructions which load the location l_i and the (relative) base address $x_i - x$ (of x_i with respect to his father); upon execution of STARTP the run–time base address of the ith son is calculated by adding the run–time base address *Wreg* of his father to the relative base address $x_i - x$ which at this moment is loaded into *Areg*. This explains the following definition:

```
compile(par(x, x⃗, l⃗, l_Out), e, m, x) =
    count(Wreg) := r,
    end_par(Wreg) := l_Out,
    ...
    LDC l_i,
    LDLP x_i − x
    STARTP,
    ...
  where l⃗ = l_1, ..., l_r
        x⃗ = x_1, ..., x_r
        1 ≤ i ≤ r
```

```
if cmd is STARTP
then loc(Wreg + Areg) := labeled_loc(Breg)
     enqueue Wreg + Areg
```

The ENDP instruction terminates the process, causing rescheduling except when the process to be terminated is the last son. Note that ENDP is preceded by an LDLP instruction which loads into *Areg* the negative value of the distance between son and the father which has to be added to the current value of *Wreg*, namely $-(x_i - x)$ $= x - x_i$. This explains the following compilation of the end instruction.

$$compile(\mathsf{end}(x_i), e, m, x) = \mathtt{LDLP}\; x - x_i,\ \mathtt{ENDP}$$

if *cmd is* ENDP
then if $count(x_f) > 1$
  then *dequeue*
    $count(x_f) := count(x_f) - 1$
  else $Ireg := labeled_loc(end_par(x_f))$
    $Wreg := x_f$
where $x_f = Wreg + Areg$

The arguments above show that this compilation of the instructions par and end correctly implements their abstract versions in $OCCAM_{daemon}$ (see section 4.5.) and thus prove the correctness lemma 6.2 for PAR statements. The proof of the theorem 6.1 follows by induction on S putting together the preceding local correctness lemmata.

6.2. Compilation of Workspace

In this section we implement our abstract functions. The definition of a daemon as process base address will be extended to daemons as workspace address. This term describes a memory block which is associated with a process. It is used to implement the environment together with all the functions which are needed by the process at run time. We will refine the function *compile* accordingly and show that the resulting evolving algebra $TRANSPUTER_{workspace}$ implements $TRANSPUTER_{ground}$ correctly.

6.2.1. Defining the abstract functions

The abstract functions which are used in our rules are here mapped into the memory by defining them relative to daemons. It would be a routine exercise to prove the correctness for a scheme which simply reserves for each of the eleven abstract functions and each daemon x some space, say $content(workspace(x, i))$, which is kept separate from the space for the environment of x. More sophistication is needed for the Transputer layout with optimized use of memory characterized by re-use of space for different non conflicting purposes. In accordance with the organization of environments as falling stack — where local variables of process x are addressed as positive offsets from $x = Wreg$ — locations with negative offsets from x are used for recording the values $f_i(x)$ of the abstract functions. The values which are likely to be used most often are put closest to x, namely $loc(x)$ and $next(x)$ which are needed when x is in the queue:

$$next(x) = content(x-2) \qquad loc(x) = content(x-1)$$

Since every daemon x can execute at any time only one Occam statement, functions which belong to different statements can be mapped to the same location without creating a conflict among them. This is the case for $mssg(x)$, $end_par(x)$ and $alt_choice(x)$ whose values are stored at $Wreg = x$:

$$mssg(x), end_par(x), alt_choice(x) = content(x)$$

Note that as consequence of this decision in [Inmos:88] to use the location *Wreg* as an extra 'register" for certain instructions one has to take care to avoid any conflicting use of this location for storing the first local variable of the daemon $x = Wreg$. This will be done by allocating in these cases an extra workspace slot for the use of *Wreg* to record one of $mssg(x)$, *end_par*, $alt_choice(x)$; see below the introduction of *Wreg* adjustment instructions AJW into the compilation of output, ALT and PAR statements.

The three functions *place*, *alt_ready*, *com_mode* can be encoded in a consistent way by one function whose value is mapped to location $content(x-3)$. To avoid conflicts we introduce one new value *alt_ready* by use of which we can restore the three functions as follows:

$$alt_ready(x) = \begin{cases} false & \text{if } content(\text{x-3}) \in \{\ alt_sleep,\ alt_run\} \\ true & \text{if } content(\text{x-3}) = alt_ready \\ undef & \text{otherwise} \end{cases}$$

$$com_mode(x) = \begin{cases} alt_sleep & \text{if } content(\text{x-3}) = alt_sleep \\ alt_run & \text{if } content(\text{x-3}) \in \{\ alt_run,\ alt_ready\ \} \\ input & \text{otherwise} \end{cases}$$

$$place(x) = \begin{cases} undef & \text{if } content(\text{x-3}) \in \\ & \quad \{\ alt_sleep,\ alt_run, alt_ready\ \} \\ content(x-3) & \text{otherwise} \end{cases}$$

For processes x executing the code for an ALT statement with time guards the locations $x-4$, $x-5$ are used to encode the relevant values of the functions *time_enabled* and t_{min}:

$$time_enabled(x) = content(x-4)$$
$$t_{min}(x) = content(x-5)$$

Note that the compilation schema defined below optimizes the use of memory by distinguishing ALT statements with and without time guards. For a PAR executing process x location $x+1$ is used to encode the son counter:

$$count(x) = content(x+1)$$

This use of location $x+1$ for workspace makes a corresponding workspace adjustment instruction necessary (see below definition of *env_size* and the refinement of *compile*). Note that we still keep the function *labeled_loc* here; it will disappear in section 6.3.1..

The layout of these functions into consecutive positions $x+1, \ldots, x-5$ realizes an optimized use of memory: each time a daemon is in the queue, locations $x-1$ and $x-2$ are used. When a daemon executes input/output for variables, also $x-3$ is needed, for values also x is used. When a daemon executes an ALT statement, locations $x, \ldots, x-3$ are used. In addition also $x-4, x-5$ are used if ALT contains a time guard. For a daemon executing a PAR statement locations $x+1, x+2$ are used. It is easy to formalize these observations for a proof of the following interesting

Lemma 6.3. (Workspace Optimization lemma) *Assuming a proper initialization it never happens in runs of $TRANSPUTER_{workspace}$ that there are any holes between used locations.*

In order to incorporate this implementation of functions by *content* correctly into the rules of $TRANSPUTER_{ground}$ we have to reserve for each daemon x enough space to encode his functions without touching the space of other daemons.

For the convenience of the reader we collected the final rules in appendix (see 7.2.).

6.2.2. Workspace Size

In this section we refine the function *env_size* (which calculates the space needed by a daemon for his environment) to the real Transputer workspace by coupling it with a function *spec_size* which takes into account the special space needed for the implementation of the abstract functions as defined above. We introduce two new functions:

$$spec_size : STATEMENT \rightarrow \mathbf{N}$$
$$work_size : STATEMENT \rightarrow \mathbf{N}$$
$$work_size(S) = env_size(S) + spec_size(S)$$

spec_size calculates the space to be reserved for the abstract functions. *spec_size* for declarations is already covered by *env_size*, therefore:

$$spec_size(\mathsf{VAR}\ id_1, \ldots, id_r : S) = spec_size(S)$$
$$spec_size(\mathsf{CHAN}\ id_1, \ldots, id_r : S) = spec_size(S)$$

For each statement it must be foreseen that the process is in the queue; this requires two locations:

$$spec_size(\mathsf{SKIP}) = 2$$
$$spec_size(\mathsf{STOP}) = 2$$
$$spec_size(v := t) = 2$$
$$spec_size(\mathsf{TIME}\,?\,v) = 2$$

The input statements use in addition the functions *place* and *com_mode*, both implemented by $content(x-3)$:

$$spec_size(c\,?\,v) = 3$$

Output statements require the function *mssg* and *com_mode*. As we want to output the value of a variable without actually having the variable, a temporary variable is allocated using the function *mssg* and its address is stored in *place*. Since *com_mode* and *place* are mapped to the same location we define:

$$spec_size(c\,!\,t) = 4$$

Due to the sequential execution of component statements of SEQ, IF and WHILE, *spec_size* is defined there as maximum of the *spec_size* of components.

$$spec_size(\mathsf{SEQ}\ S_1 \ldots S_r) = max_{i=1}^{r}\ spec_size(S_i)$$

$$spec_size(\mathsf{IF}\ B_1\, S_1, \ldots B_r\, S_r) = max_{i=1}^{r}\ (spec_size(S_i)$$

$$spec_size(\mathsf{WHILE}\ B\, S) = spec_size(S)$$

In ALT statements we distinguish two cases: case (a) without timer guard(s): the functions *alt_ready*, *alt_choice* and *com_mode* are needed; case (b) with timer guard(s), where also *time_enabled* and t_{min} are needed. Therefore:

$$spec_size(\mathsf{ALT}\ G_1\, S_1 \ldots G_r\, S_r) = \begin{cases} max_{i=1}^{r}\ (4, spec_size(S_i)) & \text{case (a)} \\ max_{i=1}^{r}\ (6, spec_size(S_i)) & \text{case (b)} \end{cases}$$

For PAR statements we need the additional functions *count* and *end_par*. Because of the father doing his first son's job we must check the first component statement:

$$spec_size(\mathsf{PAR}\ S_1 \ldots S_r) = max(4, spec_size(S_1))$$

We must extend also *env_size* because each son (except the first who is covered by his father) has to receive enough space not only for his environment but also for his functions:

$$env_size(\mathsf{PAR}\, S_1 \ldots S_r) = env_size(S_1) + \sum_{i=2}^{r} work_size(S_i)$$

6.2.3. Workspace Adjustment

In this section we refine the function *compile* with respect to the implementation of the abstract Occam functions by the special workspace assigned to daemons. This refinement has to prevent conflicting requests for already used locations x and $x+1$ in the case of daemon x executing statements which make use of Occam functions encoded there. (We will see that in general what is and has to be prevented are conflicting requests for the use of already used locations.) The three cases are output, ALT, and PAR statements.

In the case of a daemon x executing an output statement we must prevent that $content(x) = mssg(x)$ overwrites a local variable stored in $content(x)$. Function *compile* takes care of this by generating an instruction which provides an extra location for the encoding of $mssg(x)$: the instruction `AJW` adjusts the workspace address by n, in this case by minus one, making temporarily location $x-1$ location x and restoring the original workspace address once the output is done:

```
compile(c!t, e, m, x) =
    LDLP bind(c, e, x),
    eval(t, e, x)
    AJW −1
    OUT
    AJW +1
```

The rule which executes the new instruction is:

if *cmd is* `AJW` n **then** $Wreg := Wreg + n$

Note that the definition of *spec_size* for output statements provides the three locations which are needed to make this workspace sliding conflict free. This proves the correctness lemma for the implementation of output statements in $TRANSPUTER_{workspace}$.

This proves the following lemma:

LEMMA 6.4. (CORRECTNESS LEMMA) *For output statements S of* Occam*, the execution of the rules of* $TRANSPUTER_{ground}$ *on* $compile_1(S, e, m, x)$ *is correctly implemented by the execution of the rules of* $TRANSPUTER_{workspace}$ *on* $compile_2(S, e, m, x)$ *where* $compile_1$, $compile_2$ *denote the function compile of* $TRANSPUTER_{ground}$ *and* $TRANSPUTER_{workspace}$ *respectively.*

A daemon x who executes an ALT statement will make use of $alt_choice(x)$, stored as $content(x)$. We can proceed as in the case of output statements to prevent the use of location x as first position for the environment part which contains the local variables. Since any alternative might be chosen by executing ALT, *compile* emits a workspace adjusting instruction at the entry of each branch. Note that in accordance with the initial workspace adjustment by -1 the compilation of alt_a and alt_s receives the workspace pointer parameter $x-1$ to ensure the correct access to the environment part.

Note that the definition of *env_size* for ALT statements makes sure that the space which is necessary for a conflict free workspace adjustment has been provided.

Since we don't know in advance which branch will be chosen by ALT, we have to generate the re-sliding instruction in each branch:

$compile(\mathsf{ALT}\ G_1\, S_1 \ldots G_r\, S_r, e, m, x) =$
`AJW` -1,
$compile(\mathsf{alt_a}(\vec{G}), e, m, x-1)$,
$compile(\mathsf{alt_s}(\vec{G}, \vec{l}), e, m, x-1)$,
...
l_i,
`AJW` $+1$,
$compile(c_i\ ?\ v_i, e, m, x)$,
$compile(S_i, e, m, x)$,
`J` l_{Out},
...
l_j,
`AJW` $+1$,
$compile(S_j, e, m, x)$,
`J` l_{Out},
...
l_{Out}
where $\vec{l} = l_1, \ldots, l_r$
$\vec{G} = G_1, \ldots, G_r$
$G_i = b_i\ :\ c_i\ ?\ v_i$
$G_j = b_j\ :\ (\ \mathsf{TIME}\ ?\ \mathsf{AFTER}\ t_j$ or $\mathsf{SKIP}\ )$
$1 \leq i \leq p < j \leq r$

We can now prove the correctness lemma 6.4 for ALT statements. Note that at the moment when `ALTEND` is executed the slided *Wreg* points to the correct location where the reference to the chosen alternative is stored.

For a daemon x executing a PAR statement we have to consider two possible sources for conflicting reuse of memory. One is similar to the situation we know already from output and ALT statements: location $Wreg = x$ is used to store $end_par(x)$ and location $x+1$ for $count(x)$. Conflicting use of these two locations for local variables is avoided by a workspace adjustment through the instruction `AJW` -2. The second conflict to prevent is a conflict between x and his sons. First of all each son has to receive enough space for his own values of the relevant abstract Occam functions; this is provided by the refinement of m_i in the refinement of *compile* which includes the special workspace (see figure 11). But note that also the workspace conflict between x acting as father and x acting as his first son has to be prevented in case that his son is again responsible for a PAR statement. Here is an example:

```
PAR
   PAR
      p
      q
      r
L:  s
L':
```

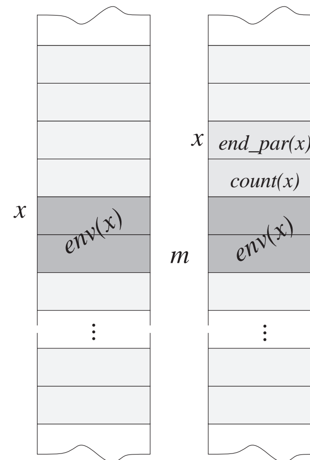


FIGURE 10. Sliding of the father

The overwriting of the values $content(x) = \mathtt{L}$[24] and $content(x+1) = 3$ belonging to the father by the values $end_par(x_1) = \mathtt{L}'$ and $count(x_1) = 2$ belonging to the first son $x_1 = x$ is avoided by the initial workspace pointer adjustment `AJW` -2 (see figure 10). Note that

[24] Note that the labels `L` and `L'` we have inserted here are not part of the Occam program but will be generated by *compile* to mark the exit of a PAR statement.

here as for ALT statements the adjusted value $x-2$ becomes the workspace pointer passed as parameter for the compilation of par and end.

$$\begin{aligned}
&compile((\text{PAR } S_1 \dots S_r), e, m, x) = \\
&\quad \texttt{AJW } -2 \\
&\quad compile(\text{par}(x, \vec{m}, \vec{l}, l_{Out}), e, m, x-2), \\
&\quad compile(S_1, e, m_1, x-2), \\
&\quad compile(\text{end}(x-2), e, m, x-2), \\
&\quad l_2, \\
&\quad compile(S_2, e, m_2, m_2), \\
&\quad compile(\text{end}(m_2), e, m_2, x-2), \\
&\quad \dots \\
&\quad l_r, \\
&\quad compile(S_r, e, m_r, m_r), \\
&\quad compile(\text{end}(m_r), e, m_r, x-2), \\
&\quad l_{Out}, \texttt{ AJW } +2 \\
&\textbf{where } \vec{l} = l_2, \dots, l_r \\
&\qquad \vec{m} = m_2, \dots, m_r \\
&\qquad m_1 = m \\
&\qquad m_i = m_{i-1} + env_size(S_{i-1}) + spec_size(S_i) \\
&\qquad 2 \le i \le r
\end{aligned}$$

Together with the implementation of the abstract functions *count* and *end_par* the first two abstract assignment instructions in the definition of $compile(\text{par}(x, \vec{x}, \vec{l}, l_{Out}), e, m, x)$ in $TRANSPUTER_{ground}$ are replaced by:

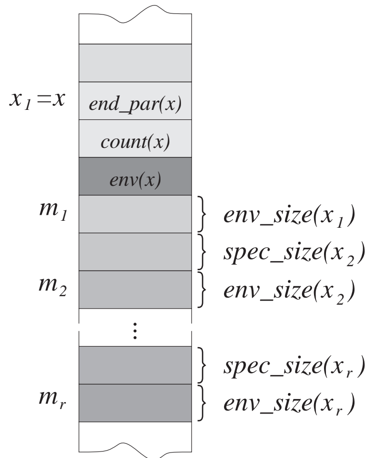


FIGURE 11. Sliding of sons

```
LDC r,
STL 1,
LDC l_Out
STL 0
```

As we have seen each process upon its creation must be (initially) slided into the plus direction pointing to the base address of the environment as shown in figure 12. For the very first process — the daemon Demiurge — we must do this manually, therefore the initial state of the loading algebra is refined by:

$$\begin{aligned}
load_prg \;=\; &\texttt{AJW } + spec_size(S) \\
&compile(S, [\,], 0, 0) \\
&\texttt{AJW } - spec_size(S)
\end{aligned}$$

The preceding reasoning proves the correctness lemma

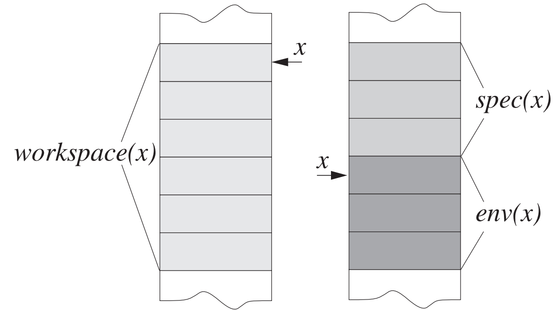


FIGURE 12. Initial sliding

6.4 for PAR statements.

THEOREM 6.5. *The implementation of $TRANSPUTER_{ground}$ in $TRANSPUTER_{workspace}$ is correct. Corresponding runs of the two evolving algebras on arbitrary* Occam *programs S are equivalent.*

Proof. By induction on S. The cases where there is no conflict among the abstract functions are covered by *compile* providing enough workspace $work_size(S)$. The non trivial case where a conflict might have occurred are covered by the above correctness lemmata for output, ALT and PAR statements. ■

6.3. Relocatable code

The real Transputer code uses relative branching instructions and doesn't have labels. In this section we eliminate labels from the compiled code (and thereby the function *labeled_loc* from our rules) in two steps: first we refine the function *compile* to incorporate relative addressing of instructions, then we add a label resolution phase to the loading procedure which is based upon location distances.

It is easy to apply the technique of relative addressing to instructions. What matters for the execution of a branching instruction is to know its distance from the place where the target instruction resides. Here is the refinement for the compilation of WHILE commands and the related jump rule:

$$\begin{aligned}
&compile(\text{WHILE } B\, S, e, m, x) = \\
&\quad l_{In}, \\
&\quad compile(\text{if}(\neg B, l_{Out}), e, m, x), \\
&\quad compile(S, e, m, x), \\
&\quad \texttt{J } l_{In} - l_{Out}, \\
&\quad l_{Out}
\end{aligned}$$

if *cmd is* J $l_1 - l_2$
then $Ireg := Ireg + (labeled_loc(l_1) - labeled_loc(l_2))$

Clearly the correctness lemma holds for this refinement of WHILE statements.

For alt_s we have a similar refinement of *compile* and of the related ALTEND rule (where again the auxiliary function *labeled_loc* disappears). Note that the jumping action takes place in ALTEND, therefore from l_{here} (forward) to the chosen alternative with label l_i.

```
compile(alt_s(G⃗, l⃗), e, m, x) =
    ...
    LDLP bind(c_i, e, x),
    eval(b_i, e, x),
    LDC l_i − l_here,
    DISC,
    ...
    eval(t_j, e, x),
    eval(b_j, e, x),
    LDC l_j − l_here,
    DIST,
    ...
    eval(b_k, e, x),
    LDC l_k − l_here,
    DISS,
    ...
    ALTEND,
    l_here
where l⃗ = l_1, ..., l_r
      G⃗ = G_1, ..., G_r
      G_i = b_i : c_i ? v_i
      G_j = b_j : TIME?AFTER t_j
      G_k = b_k : SKIP
      1 ≤ i ≤ p < j ≤ q < k ≤ r
```

if *cmd is* ALTEND
then *Ireg* := *Ireg* + *alt_choice*(*Wreg*)

The same technique applies to the refinement of *compile* on IF statements:

```
compile(IF B_1 S_1 ... B_r S_r, e, m, x) =
    ...
    compile(if(B_i, l_i), e, m, x),
    ...
    compile(STOP, e, m, x),
    ...
    l_i,
    compile(S_i, e, m, x),
    J l_Out − l_{i+1},
    ...
    l_Out
where l_{r+1} = l_Out
      1 ≤ i ≤ r
```

For the corresponding environment of *compile* on if(B, L) we have no name l_{Out} and therefore create a new label l_{here} to mark the place where the instruction pointer *Ireg* will point to when the Transputer is executing the branching instruction (see *fetch* phase in 5.4.). Thus we have the following refinement of *compile* on if(B, L) and of the rule for conditional jump (note that the auxiliary function *labeled_loc* disappears from the rule):

```
compile(if(B, L), e, m, x) =
    eval(¬B, e, x),
    CJ L − l_here,
    l_here
```

if *cmd is* CJ $l_1 - l_2$
then if *Areg* = *false*
  then *Ireg* := *Ireg*+ (*labeled_loc*(l_1) − *labeled_loc*(l_2))
  else *Areg* := *Breg*
    Breg := *Creg*
    Creg := *random*

Again it is obvious that this refinement satisfies the correctness lemma for IF statements. For the compilation of PAR we can proceed in a similar way replacing the starting address l_i of the ith son by the distance $l_i - l'_i$ to the location l'_i of the instruction STARTP which actually starts the process. The rule for STARTP is refined correspondingly by adding the relevant distance to the current value of *Ireg*; here too *labeled_loc* disappears.

There is a slight additional complication. For the execution of the jump in the instruction ENDP the absolute (return) address is needed. It can be computed upon entering the PAR code when the distance $l_{Out} - l_{here}$ is loaded into *Areg* to get stored into *content*(x): it suffices to add to this compile–time value the current value of *Ireg*. This is done by the new instruction LDPI which is inserted after LDC $l_{Out} - l_{here}$.

```
compile(par(x, x⃗, l⃗, l_Out), e, m, x) =
    LDC r,
    STL 1,
    LDC l_Out − l_here
    LDPI,
    l_here,
    STL 0,
    ...
    LDC l_i − l'_i,
    LDLP x_i − x
    STARTP,
    l'_i,
    ...
where l⃗ = l_1, ..., l_r
      x⃗ = x_1, ..., x_r
      1 ≤ i ≤ r
```

if *cmd is* LDPI
then *Areg* := *Ireg* + *Areg*

if *cmd is* STARTP
then *loc*(*Wreg* + *Areg*) := *Breg* + *Ireg*
  enqueue Wreg + *Areg*

Let $TRANSPUTER_{rel-code}$ be the refinement of $TRANSPUTER_{workspace}$ by introducing relative addressing of instructions for relocatable code as defined

in this section. The preceding arguments prove the following:

Lemma 6.6. *$TRANSPUTER_{rel-code}$ is a correct implementation of $TRANSPUTER_{workspace}$.*

6.3.1. Resolving Labels

The rules of $TRANSPUTER_{rel-code}$ show that at run-time only the distances $labeled_loc(l_1) - labeled_loc(l_2)$ between labeled locations are used but not any more the labels l_1, l_2 themselves. We therefore eliminate in this section the appearance of labels from the code which is loaded into the Transputer memory.

The idea is to transfer the computation of the distances between labeled locations from the execution time of Transputer instructions to the very moment where the values $labeled_loc(l_i)$ become known. Before loading we therefore pre-process the compiled program $compile(S, [], 0, 0)$ by a variant *LOAD-LABEL* of the *LOAD* algebra, whose function is to compute the values of the function *labeled_loc* — followed by the evaluation of the distances:

$$labeled_loc(l_1) - labeled_loc(l_2).$$

Let *LOAD-LABEL* be obtained from the algebra *LOAD* (see section 4.2.) by adding the guard $mode = load\text{-}label$ to the rules and by replacing the update:

$$cmd(labeled_loc) := c$$

by the update:

$$tmp_prg := append(tmp_prg, c)$$

where $tmp_prg : (CODE \cup LABEL)^*$ is a new 0-ary function. *LOAD-LABEL* algebra is initialized by:

$$\begin{aligned} load_prg &= \texttt{AJW}\ spec_size(S) \\ &\ compile(S, [], 0, 0) \\ &\ \texttt{AJW}\ -spec_size(S) \\ tmp_prg &= [] \\ loading_loc &= 0 \end{aligned}$$

LOAD-LABEL computes $labeled_loc(l)$ for each label l appearing in *load_prg* and stores a copy of *load_prg* in *tmp_prg*. Upon termination of *LOAD-LABEL* the following rule switches to the computation of distances between labeled locations:

if $mode = load\text{-}label \wedge load_prg = []$
then $loading_loc := 0$
  $mode := compute\text{-}distances$

The following rule replaces $l_1 - l_2$ by $labeled_loc(l_1) - labeled_loc(l_2)$ in the code produced by *compile*:

if $mode = compute\text{-}distances \wedge tmp_prg \neq []$
then $loading_loc := loading_loc+$
  $tmp_prg := tail(tmp_prg)$
  if $first(tmp_prg) = c\ l_1 - l_2$
  then $load_prg := append(load_prg,$
    $c\ labeled_loc(l_1) - labeled_loc(l_2))$
  else $load_prg := append(load_prg, first(tmp_prg))$

When this rule cannot be applied any more, the following rules switches to the algebra obtained from *LOAD* by adding the condition $mode = load$ to the guard and by deleting the rule load_label:

if $mode = compute\text{-}distances \wedge tmp_prg = []$
then $mode := load$

Where the loading rule cannot be applied any more, the following rule switches to the rules of $TRANSPUTER_{rel-code}$:

if $mode = load \wedge load_prg = []$
then $mode := fetch$

Let *TRANSPUTER* be the evolving algebra consisting of all the rules described above added to the rules of $TRANSPUTER_{rel-code}$. It is an exercise to prove the following lemma:

Lemma 6.7. *TRANSPUTER implements $TRANSPUTER_{rel-code}$ correctly.*

The proof of the theorem 6.1 follows by induction on S putting together the single local correctness lemmata stated and proved above. The 4.10 theorem, the 6.1 theorem and the 6.5 theorem prove the Main theorem.

CONCLUSION AND OUTLOOK

We have built a formal model for the Transputer instruction set architecture and have used it to formulate and prove a correctness theorem for a general compilation schema of Occam programs into Transputer code. Occam and Transputer here served as non trivial paradigm taken from the real world and independent from the compiler verification project. The method developed in this paper to support correct compiler design is general and can be applied *mutatis mutandis* to other architectures and other programming languages which exhibit the characteristic features of distributed computing, namely nondeterminism and concurrency.

There are several directions for future work which we consider challenging and worthwhile to be investigated. One is to refine the Transputer actions to sequences of hardware actions which are executed by a hardware interpreter; other processors might also be interesting for this line of research. Another direction is to figure out whether our proofs can be carried out in interesting deductive frameworks. In this context it would be especially interesting to compare our approach with the work done on the subject within the PROCOS II Esprit Basic Research Action. A directly related investigation is Müller–Olm's forthcoming doctoral dissertation (see [MMO:95] where a code generator correctness proof for a sequential sublanguage of Occam is given by delivering increasingly more abstract levels starting from the Transputer). We guess that some of the evolving algebras developed here are models of the algebraic laws employed for Müller–Olm's proofs. In general it could be interesting to investigate the relation between

evolving algebras and Hoare's refinement algebra approach to prove correctness of compiling specifications (see [Hoare:91], [Hoare et al.:87], [Hoare et al.:93]). A challenging project is to verify our proofs by theorem provers. Note that the corresponding computer verification of the WAM correctness proof of [BöRo:95b] in *KIV* and *ISABELLE* at present looks promising.

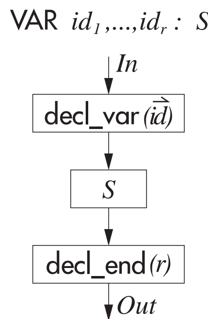
Acknowledgments

We thank the following colleagues for criticism and useful comments on previous versions of this paper: Peter Baumann, Markus Mueller-Olm, Kirsten Winter, Wolf Zimmermann and two anonymous referees. We also thank Schloß Dagstuhl for many weekends we have been offered there to work on this paper.

7. APPENDIX

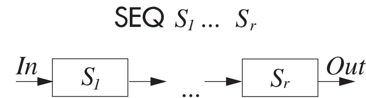
7.1. Generating Flowcharts (section 4.1)

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).

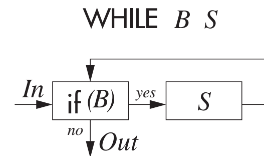


if $cmd(n)$ = VAR $id_1, \ldots, id_r$: S
then extend *NODE* **by** n_1, n_2 **with**
$cmd(n)$:= decl_var($\vec{id}$)
$next(n)$:= n_1
$cmd(n_1)$:= S
$next(n_1)$:= n_2
$cmd(n_2)$:= decl_end(r)
$next(n_2)$:= $next(n)$
endextend
where $\vec{id} = id_1, \ldots, id_r$

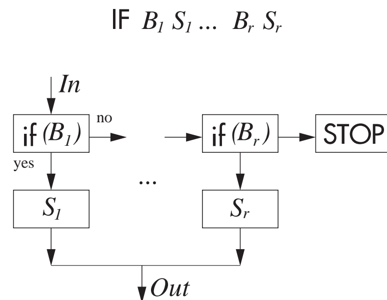
if $cmd(n)$ = CHAN $id_1, \ldots, id_r$: S
then extend *NODE* **by** n_1, n_2 **with**
$cmd(n)$:= decl_chan($\vec{id}$)
$next(n)$:= n_1
$cmd(n_1)$:= S
$next(n_1)$:= n_2
$cmd(n_2)$:= decl_end(r)
$next(n_2)$:= $next(n)$
endextend
where $\vec{id} = id_1, \ldots, id_r$



if $cmd(n)$ = SEQ $S_1 \ldots S_r$
then extend *NODE* **by** $n_2, \ldots, n_r$ **with**
...
$cmd(n_i)$:= S_i
$next(n_i)$:= n_{i+1}
...
endextend
where $n_1 = n$
$n_{r+1} = next(n)$
$1 \leq i \leq r$



if $cmd(n)$ = WHILE B S
then extend *NODE* **by** n_1 **with**
$cmd(n)$:= if(B)
$no(n)$:= $next(n)$
$yes(n)$:= n_1
$cmd(n_1)$:= S
$next(n_1)$:= n
endextend

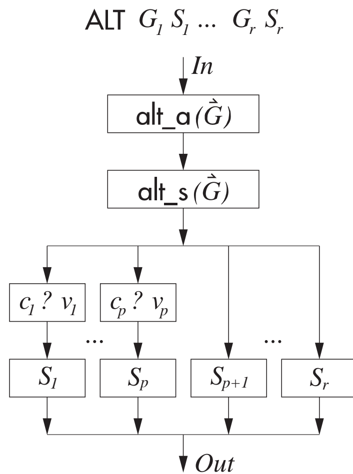


if $cmd(n) = \mathsf{IF}\ B_1\ S_1 \ldots B_r\ S_r$
then extend $NODE$ **by** $n_1, \ldots, n_r,$ **with**
$m_2, \ldots, m_{r+1}$

$\ldots$
$cmd(m_i) := \mathsf{if}(B_i)$
$yes(m_i) := n_i$
$no(m_i) := m_{i+1}$
$cmd(n_i) := S_i$
$next(n_i) := next(n)$
$\ldots$
$cmd(m_{r+1}) := \mathsf{STOP}$
endextend

where $m_1 = n$
$1 \leq i \leq r$

ALT $G_1\ S_1 \ldots G_r\ S_r$

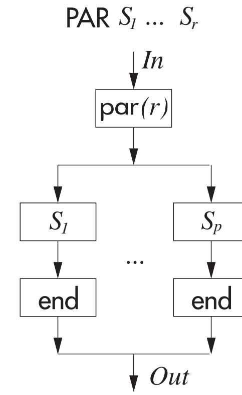


if $cmd(n) = \mathsf{ALT}\ G_1\ S_1 \ldots G_r\ S_r$
then extend $NODE$ **by** $n_0, \ldots, n_r,$ **with**
$m_1, \ldots, m_p$

$cmd(n) := \mathsf{alt_a}(\vec{G})$
$next(n) := n_0$
$cmd(n_0) := \mathsf{alt_s}(\vec{G})$
$\ldots$
$next(n_0, i) := n_i$
$cmd(n_i) := c_i\ ?\ v_i$
$next(n_i) := m_i$
$cmd(m_i) := S_i$
$next(m_i) := next(n)$
$\ldots$
$next(n_0, j) := n_j$
$cmd(n_j) := S_j$
$next(n_j) := next(n)$
$\ldots$
endextend

where $\vec{G} = G_1, \ldots, G_r$
$G_i = b_i\ :\ c_i\ ?\ v_i$
$G_j = b_j\ :\ (\ \mathsf{TIME}\ ?\ \mathsf{AFTER}\ t_j \text{ or } \mathsf{SKIP}\ )$
$1 \leq i \leq p < j \leq r$

PAR $S_1 \ldots S_r$



if $cmd(n) = \mathsf{PAR}\ S_1 \ldots S_r$
then extend $NODE$ **by** $n_1, m_1 \ldots, n_r, m_r$ **with**

$cmd(n) := \mathsf{par}(r)$
$\ldots$
$next(n, i) := n_i$
$cmd(n_i) := S_i$
$next(n_i) := m_i$
$cmd(m_i) := \mathsf{end}$
$\ldots$
endextend

where $1 \leq i \leq r$

7.2. Summary of $OCCAM_{daemon}$ (section 4.6)

The Processor

$content\ :\ LOC \rightarrow VAL$
$+1\ :\ LOC \rightarrow LOC$

$a\ :\ DAEMON \cup \{nil\}$
$QUEUE \subseteq DAEMON^*$
$first, last\ :\ QUEUE \rightarrow DAEMON$
$rest\ :\ QUEUE \rightarrow QUEUE$
$q\ :\ QUEUE$

$enqueue\ x \stackrel{\text{def}}{=} q := q.x$
$dequeue \stackrel{\text{def}}{=}$ **if** not $empty\ q$
then $a := first(q)$
$q := rest(q)$
$start := timer$
else $a := nil$

$timer\ :\ \mathbf{N}$
$start\ :\ \mathbf{N}$
$period\ :\ \mathbf{N}$
$elapsed \stackrel{\text{def}}{=} timer - start > period$

[time–slice] **if** $a \neq nil \wedge elapsed$
then seq $q := q.a$
dequeue
endseq

$$time_q \; : \; QUEUE$$
$$t_{min} \; : \; DAEMON \rightarrow \mathbf{N}$$

$time\text{-}insert \overset{\text{def}}{=}$
let $time_q = x_1.\ldots.x_i.x_{i+1}.\ldots.x_r$
let $t_{min}(x_i) \leq t_{min}(a) \leq t_{min}(x_{i+1})$
$time_q := x_1.\ldots.x_i.a.x_{i+1}.\ldots.x_r$

$time\text{-}delete\ x \overset{\text{def}}{=}$
let $time_q = x_1.\ldots.x_i.x.x_{i+1}.\ldots.x_r$
$time_q := x_1.\ldots.x_i.x_{i+1}.\ldots.x_r$

$$proceed \overset{\text{def}}{=} loc(a) := loc(a) + 1$$
$$sleep\ at\ n \overset{\text{def}}{=} loc(a) := n, dequeue$$
$$wakeup\ x \overset{\text{def}}{=} enqueue\ x$$
$$cmd\ is\ c \overset{\text{def}}{=} content(loc(a)) = c \wedge \text{not } elapsed$$

Compilation & Loading

$$compile \; : \; STATEMENT \times ENV \times LOC \times DAEMON \rightarrow ((CODE \times ENV \times DAEMON) \cup LABEL)^*$$

$$loading_loc \; : \; NODE$$
$$load_prg \; : \; ((CODE \times ENV \times DAEMON) \cup LABEL)^*$$

[load_label(l,t)]
if $load_prg = [\,l \,|\, t\,] \wedge l \in LABEL$
then $labeled_loc(l) := loading_loc$
$load_prg := t$

[load_cmd(c,t)]
if $load_prg = [\,c \,|\, t\,] \wedge c \in CODE$
then $content(loading_loc) := c$
$load_prg := t$
$loading_loc := loading_loc+$

Declarations

$compile(\mathsf{VAR}\ id_1, \ldots, id_r : S, e, m, x) =$
$compile(S, e', m + r, x)$
where $e' = append(e_{new}, e)$
$e_{new} = (id_r, m + r - 1), \ldots, (id_1, m).$

$compile(\mathsf{CHAN}\ id_1, \ldots, id_r : S, e, m, x) =$
$\mathsf{init_chan}(\vec{id}, e', x),$
$compile(S, e', m + r, x)$
where $e' = append(e_{new}, e)$
$e_{new} = (id_r, m + r - 1), \ldots, (id_1, m).$

$$ENV = (ID \times (VAR \cup CHANNEL))^*$$
$$bind \; : \; ID \times ENV \times DAEMON \rightarrow VAR \cup CHANNEL$$
$$bind(id, [\,H \,|\, T\,], x) = \begin{cases} o - x & \text{if } H = (id,o) \\ bind(id, T, x) & \text{otherwise} \end{cases}$$
$$\bar{v} = bind(v, e)$$
$$\bar{c} = bind(c, e)$$

[init_chan($\vec{c}, e, x$)]
if *cmd is* $\mathsf{init_chan}(c_1, \ldots, c_r, e, x)$
then $agent(\bar{c_1}) := nil$
...
$agent(\bar{c_r}) := nil$

Expressions

$$eval \; : \; VAR \rightarrow VAL$$
$$eval(v) \overset{\text{def}}{=} content(v)$$
$$eval \; : \; EXP \times ENV \times DAEMON \rightarrow VAL$$

$$compile(v := t, e, m, x) = \mathsf{ass}(v, t, e, x)$$

[ass(v,t,e,x)]
if *cmd is* $\mathsf{ass}(v, t, e, x)$
then $content(\bar{v} + a) := eval(t, e, x)$
proceed

$$compile(\mathsf{TIME}\ ?\ v, e, m, x) = \mathsf{time}(v, e, x)$$

[time(v,e,x)]
if *cmd is* $\mathsf{time}(v, e, x)$
then $content(\bar{v} + a) := timer$
proceed

Conditionals

$compile(\mathsf{WHILE}\ B\ S, e, m, x) =$
$l_{In},$
$\mathsf{if}(\neg B, l_{Out}, e, x),$
$compile(S, e, m, x),$
$\mathsf{goto}(l_{In}),$
l_{Out}

$compile(\mathsf{IF}\ B_1\, S_1\ \ldots\ B_r\, S_r, e, m, x) =$
...
$\mathsf{if}(B_i, l_i, e, x)$,
...
STOP,
...
l_j,
$compile(S_j, e, m, x)$,
$\mathsf{goto}(l_{Out})$,
...
l_{Out}
where $1 \leq i, j \leq r$

if(b,l,e,x)
if *cmd is* $\mathsf{if}(b, l, e, x)$
then if $eval(b, e, x)$
then $loc(a)$ **:=** $labeled_loc(l)$
else *proceed*

goto(l)
if *cmd is* $\mathsf{goto}(l)$
then $loc(a)$ **:=** $labeled_loc(l)$

$compile(\mathsf{SKIP}, e, m, x) = \mathsf{SKIP}$

skip **if** *cmd is* SKIP **then** *proceed*

$compile(\mathsf{STOP}, e, m, x) = \mathsf{STOP}$

stop **if** *cmd is* STOP **then** *dequeue*

Input

$$idle\ c \stackrel{\text{def}}{=} agent(c) = nil$$
$$clear\ c \stackrel{\text{def}}{=} agent(c) := nil$$
$$ready\ c \stackrel{\text{def}}{=} agent(c) \neq nil$$

$compile(c\,?\,v, e, m, x) = \mathsf{inp}(c, v, e, x)$

$\mathsf{inp_idle}(c, v, e, x)$
if *cmd is* $\mathsf{inp}(c, v, e, x) \wedge idle\ \bar{c}$
then *sleep at* $next(loc(a))$
$agent(\bar{c})$ **:=** a
$place(a)$ **:=** $\bar{v} + a$
$com_mode(a)$ **:=** *input*

$\mathsf{inp_ready}(c, v, e, x)$
if *cmd is* $\mathsf{inp}(c, v, e, x) \wedge ready\ \bar{c}$
then $content(\bar{v} + a)$ **:=** $mssg(agent(\bar{c}))$
wakeup $agent(\bar{c})$
proceed
clear $\bar{c}$

Output

$compile(c\,!\,t, e, m, x) = \mathsf{out}(c, t, e, x)$

$\mathsf{out_idle}(c, t, e, x)$
if *cmd is* $\mathsf{out}(c, t, e, x) \wedge idle\ \bar{c}$
then *sleep at* $next(loc(a))$
$agent(\bar{c})$ **:=** a
$mssg(a)$ **:=** $eval(t, e, x)$

$\mathsf{out_ready}(c, t, e, x)$
if *cmd is* $\mathsf{out}(c, t, e, x)$
$\wedge\ ready\ \bar{c}$
$\wedge\ com_mode(agent(\bar{c})) = input$
then $content(place(agent(\bar{c})))$ **:=** $eval(t, e, x)$
wakeup $agent(\bar{c})$
proceed
clear $\bar{c}$

$\mathsf{out_alt}(c, t, e, x)$
if *cmd is* $\mathsf{out}(c, t, e, x)$
$\wedge\ ready\ \bar{c}$
$\wedge\ com_mode(agent(\bar{c})) \neq input$
then *sleep at* $next(loc(a))$
$mssg(a)$ **:=** $eval(t, e, x)$, $agent(\bar{c})$ **:=** a
if $com_mode(agent(\bar{c})) = alt_sleep$
then *time-delete* $agent(\bar{c})$
wakeup $agent(\bar{c})$
$com_mode(agent(\bar{c}))$ **:=** alt_run

Alternation

$compile(\mathsf{ALT}\ G_1\, S_1 \ldots G_r\, S_r, e, m, x) =$
$\mathsf{alt_a}(\vec{G}, e, x)$,
$\mathsf{alt_s}(\vec{G}, \vec{l}, e, x)$,
...
l_i,
$\mathsf{inp}(c_i, v_i, e, x)$,
$compile(S_i, e, m, x)$,
$\mathsf{goto}(l_{Out})$,
...
l_j,
$compile(S_j, e, m, x)$,
$\mathsf{goto}(l_{Out})$,
...
l_{Out}
where $\vec{l} = l_1, \ldots, l_r$
$\vec{G} = G_1, \ldots, G_r$
$G_i = b_i\ :\ c_i\,?\,v_i$
$G_j = b_j\ :\ (\ \mathsf{TIME}\,?\,\mathsf{AFTER}\, t_j$ or $\mathsf{SKIP}\ )$
$1 \leq i \leq p < j \leq r$

```
alt_a(G⃗, e, x)
if cmd is alt_a(G⃗, e, x)
then
  enable(b_1, c̄_1)
  ...
  enable(b_p, c̄_p)
  if ∃ i (eval(b_i, e, x) ∧ ready c̄_i)
     ∨ t_wait < timer
     ∨ ∃ k (eval(b_k, e, x))
  then com_mode(a) := alt_run
       loc(a) := next(loc(a))
  else com_mode(a) := alt_sleep
       sleep at next(loc(a))
       if ∃ j (eval(b_j, e, x))
       then seq t_min(a) := t_wait
                time-insert
            endseq
where
  G_i = b_i : c_i ? v_i
  G_j = b_j : TIME ? AFTER t_j
  G_k = b_k : SKIP
```

$$t_wait = \begin{cases} \infty \text{ if } \forall\, j\ (\neg eval(b_j, e, x)), \text{ otherwise} \\ \min_j \{eval(t_j, e, x) \mid eval(b_j, e, x)\} \end{cases}$$

$1 \le i \le p < j \le q < k \le r$

```
alt_s_com | alt_s_time | alt_s_skip (i, G⃗, l⃗, e, x)
if cmd is alt_s(G⃗, l⃗, e, x)
  ∧ eval(b_i, e, x)
  ∧ agent(c̄_i) ∉ { nil, a } | timer > eval(t_i, e, x) |
then
  disable(b_1, c̄_1)
  ...
  disable(b_p, c̄_p)
  loc(a) := labeled_loc(l_i)
where
  l⃗ = l_1, ..., l_r
  G⃗ = G_1, ..., G_r
  G_i = b_i : ( c_i ? v_i | TIME ? AFTER t_i | SKIP )
```

```
time_wakeup
if time_q not empty
   ∧ timer > t_min(first(time_q))
then com_mode(first(time_q)) := alt_run
     wakeup first(time_q)
     time-delete first(time_q)
```

Parallelism

$env_size(\mathsf{SKIP}) = 0$

$env_size(\mathsf{STOP}) = 0$

$env_size(\mathsf{TIME}\,?\,v) = 0$

$env_size(c\,?\,v) = 0$

$env_size(c\,!\,t) = eval_size(t)$

$env_size(v := t) = eval_size(t)$

$env_size(\mathsf{VAR}\ id_1, \ldots, id_r : S) = r + env_size(S)$

$env_size(\mathsf{CHAN}\ id_1, \ldots, id_r : S) = r + env_size(S).$

$env_size(\mathsf{WHILE}\ B\, S) = max(\, eval_size(B), env_size(S)\,)$

$env_size(\mathsf{IF}\ B_1\, S_1, \ldots B_r\, S_r) = max_{i=1}^{r}\ (eval_size(B_i), env_size(S_i))$

$env_size(\mathsf{SEQ}\ S_1 \ldots S_r) = max_{i=1}^{r}\ env_size(S_i)$

$env_size(\mathsf{ALT}\ G_1\, S_1 \ldots G_r\, S_r) = max_{i=1}^{r}\ (eval_size(b_i), env_size(S_i))$

$env_size(\mathsf{PAR}\ S_1 \ldots S_r) = \sum_{i=1}^{r} env_size(S_i)$

```
compile(PAR S_1 ... S_r, e, m, x) =
    par(x, m⃗, l⃗, l_Out),
    l_1,
    compile(S_1, e, m, x),
    end,
    l_2,
    compile(S_2, e, m_2, m_2),
    end,
    ...
    l_r,
    compile(S_r, e, m_r, m_r),
    end,
    l_Out
where l⃗ = l_2, ..., l_r
      m⃗ = m_2, ..., m_r
      m_1 = m
      m_i = m_{i-1} + env_size(S_{i-1})
      2 ≤ i ≤ r
```

```
par(x, x⃗, l⃗, l_Out)
if cmd is par(x, x⃗, l⃗, l_Out)
then q := q.x'_1. ... .x'_r
     ...
     father(x'_i) := a
     loc(x'_i) := labeled_loc(l_i)
     ...
     count(a) := r
     end_par(a) := l_Out
     proceed
where l⃗ = l_1, ..., l_r
      x⃗ = x_1, ..., x_r
      x'_i = a − x + x_i
      1 ≤ i ≤ r
```

```
[end]
if cmd is end
then if r > 1
       then dequeue
            r := r - 1
       else a := x_f
            loc(x_f) := a_Out
where r = count(x_f)
      x_f = father(a)
      l_Out = end_par(x_f)
      a_Out = labeled_loc(l_Out)
```

7.3. Summary of $TRANSPUTER_{rel-code}$

Processor

$$content : LOC \to VAL$$
$$+1 : LOC \to LOC$$

$$FPtrReg, BPtrReg : DAEMON \cup \{nil\}$$
$$next : DAEMON \to DAEMON$$

```
enqueue x =def BPtrReg := x
               if FPtrReg = nil
               then FPtrReg := x
               else next(BPtrReg) := x

dequeue =def if FPtrReg = nil
             then a := nil
             else a := FPtrReg
                  loc(a) := loc(FPtrReg)
                  start := timer
                  if FPtrReg = BPtrReg
                  then FPtrReg := nil
                  else FPtrReg := next(FPtrReg)
```

$$TPtrLoc : DAEMON \cup \{nil\}$$
$$TNextReg : LOC$$

$$Areg, Breg, Creg : VAL$$
$$random : VAL$$

```
push(v) =def
   Areg := v, Breg := Areg, Creg := Breg
pop =def
   Areg := Breg, Breg := Creg, Creg := random
pop(v) =def
   v := Areg, pop
```

$$Wreg, Ireg : VAL$$
$$proceed \stackrel{\text{def}}{=} Ireg := Ireg + 1$$

$$cmd : CODE$$
$$cmd \text{ is } c \stackrel{\text{def}}{=} cmd = c \wedge mode = execute \wedge Wreg \neq nil$$
$$mode = \{load\text{-}label, compute\text{-}distance, load, fetch, execute\}$$

```
[time-slice]
if mode = fetch ∧ Wreg ≠ nil
then if elapsed
          ∧ safe(content(Ireg))
       then seq enqueue Wreg
                dequeue
            endseq
       else cmd := content(Ireg)
            proceed
            mode := execute
```

Compilation, Resolving, Loading, ...

$$compile : STATEMENT \times ENV \times LOC \times DAEMON \to (CODE \cup LABEL)^*$$

$$loading_loc : LOC$$
$$tmp_prg, load_prg : (CODE \cup LABEL)^*$$

```
if mode = load-label
then if load_prg = []
       then mode := compute-distances
       else let load_prg = [H | T]
            load_prg := T
            if H ∈ LABEL
            then labeled_loc(l) := loading_loc
            if H ∈ CODE
            then tmp_prg := append(tmp_prg, c)
                 loading_loc := loading_loc+

if mode = compute-distances
then if load_prg = []
       then mode := load
       else let tmp_prg = [H | T]
            loading_loc := loading_loc+
            tmp_prg := T
            if H = c l_1 - l_2
            then load_prg := append(load_prg, H')
            else load_prg := append(load_prg, H)
where H' = c labeled_loc(l_1) - labeled_loc(l_2)

[load_cmd(c,t)]
if mode = load
then if load_prg = []
       then mode := fetch
       else let load_prg = [H | T]
            load_prg := T
            content(loading_loc) := H
            loading_loc := loading_loc+
```

Declarations

$compile(\mathsf{VAR}\ id_1, \ldots, id_r : S, e, m, x) =$
$compile(S, e', m + r, x)$
where $\vec{id} = id_1, \ldots, id_r$
$e' = (id_r, m + r - 1), \ldots, (id_1, m) \mid e$

$compile(\mathsf{CHAN}\ id_1, \ldots, id_r : S, e, m, x) =$
$compile(\mathsf{init_chan}(\vec{id}), e', m, x),$
$compile(S, e', m + r, x)$
where $\vec{id} = id_1, \ldots, id_r$
$e' = (id_r, m + r - 1), \ldots, (id_1, m) \mid e$

$compile(\mathsf{init_chan}(\vec{c}), e, m, x) =$
...
`MINT`,
`STL` $bind(c_i, e, x)$,
...
where $\vec{c} = c_1, \ldots, c_r$
$1 \leq i \leq r$

$$ENV = (ID \times LOC)^*$$
$$bind : ID \times ENV \times DAEMON \rightarrow LOC$$
$$bind(id, [H \mid T], x) = \begin{cases} o - x & \text{if H = (id,o)} \\ bind(id, T, x) & \text{otherwise} \end{cases}$$

if *cmd is* `MINT` **then** $push(nil)$

if *cmd is* `STL` n **then** $pop(content(Wreg + n)$

Expressions

$$eval : EXP \times ENV \times DAEMON \rightarrow VAL$$

$compile(v := t, e, m, x) =$
$\mathsf{eval}(t, e, x)$
`STL` $bind(v, e, x)$

if *cmd is* $\mathsf{eval}(t, e, x)$ **then** $push(eval(t, e, x))$

$compile(\mathsf{TIME}\ ?\ v, e, m, x) =$
`LDTIMER`
`STL` $bind(v, e, x)$

if *cmd is* `LDTIMER`
then $push(timer)$

$compile(\mathsf{SKIP}) =$

$compile(\mathsf{STOP}, e, m, x) =$ `STOPP`

if *cmd is* `STOPP`
then $Areg, Breg, Creg := random$
dequeue

Conditionals

$compile(\mathsf{WHILE}\ B\ S, e, m, x) =$
l_{In},
$compile(\mathsf{if}(\neg B, l_{Out}), e, m, x),$
$compile(S, e, m, x),$
`J` $l_{In} - l_{Out}$,
l_{Out}

$compile(\mathsf{if}(B, L), e, m, x) =$
$\mathsf{eval}(\neg B, e, x),$
`CJ` $L - l_{here}$,
l_{here}

if *cmd is* `J` n **then** $Ireg := Ireg + n$

if *cmd is* `CJ` n
then if $Areg = false$
then $Ireg := Ireg + n$
else $Areg := Breg$
$Breg := Creg$
$Creg := random$

$compile(\mathsf{IF}\ B_1\ S_1\ \ldots\ B_r\ S_r, e, m, x) =$
...
$compile(\mathsf{if}(B_i, l_i), e, m, x),$
...
$compile(\mathsf{STOP}, e, m, x),$
...
l_i,
$compile(S_i, e, m, x),$
`J` $l_{Out} - l_{i+1}$,
...
l_{Out}
where $l_{r+1} = l_{Out}$
$1 \leq i \leq r$

Input

$compile(c\ ?\ v, e, m, x) =$
`LDLP` $bind(c, e, x)$,
`LDLP` $bind(v, e, x)$,
`IN`

if *cmd is* `LDLP` n
then $push(Wreg + n)$

IN_idle
if *cmd is* `IN` $\wedge\ content(Breg) = nil$
then $content(Breg) := Wreg$
$content(Wreg - 3) := Areg$
dequeue

IN_ready
if *cmd is* `IN` $\wedge\ content(Breg) \neq nil$
then $content(Areg) := content(agent)$
$agent := nil$
enqueue agent
where $agent = content(Breg)$

Output

$compile(c\,!\,t, e, m, x) =$
 LDLP $bind(c, e, x)$,
 eval(t, e, x)
 AJW -1
 OUT
 AJW $+1$

if *cmd is* AJW n **then** $Wreg := Wreg + n$

OUT_idle

if *cmd is* OUT $\wedge$ $content(Breg) = nil$
then $content(Breg) := Wreg$
 $content(Wreg) := Areg$
 dequeue

OUT_ready

if *cmd is* OUT
 $\wedge$ $agent \neq nil$
 $\wedge$ $place \notin \{ alt_sleep, alt_run, alt_ready \}$
then $content(place) := Areg$
 $agent := nil$
 enqueue agent
where $agent = content(Breg)$
 $place = content(agent - 3)$

OUT_alt

if *cmd is* OUT
 $\wedge$ $agent \neq nil$
 $\wedge$ $place \in \{ alt_sleep, alt_run, alt_ready \}$
then *dequeue*
 $agent := Wreg$
 $content(Wreg) := Areg$
 if $place = alt_sleep$
 then *time-delete agent*
  $place := alt_run$
  enqueue agent
where $agent = content(Breg)$
 $place = content(agent - 3)$

Alternation

$compile(\mathsf{ALT}\ G_1\, S_1 \ldots G_r\, S_r, e, m, x) =$
 AJW -1,
 $compile(\mathsf{alt_a}(\vec{G}), e, m, x-1)$,
 $compile(\mathsf{alt_s}(\vec{G}, \vec{l}), e, m, x-1)$,
 ...
 l_i,
 AJW $+1$,
 $compile(c_i\,?\,v_i, e, m, x)$,
 $compile(S_i, e, m, x)$,
 J l_{Out},
 ...
 l_j,
 AJW $+1$,
 $compile(S_j, e, m, x)$,
 J l_{Out},
 ...
 l_{Out}
where $\vec{l} = l_1, \ldots, l_r$
 $\vec{G} = G_1, \ldots, G_r$
 $G_i = b_i\ :\ c_i\,?\,v_i$
 $G_j = b_j\ :\ (\ \mathsf{TIME}\,?\,\mathsf{AFTER}\,t_j$ or $\mathsf{SKIP}\ )$
 $1 \leq i \leq p < j \leq r$

$compile(\mathsf{alt_a}(\vec{G}), e, m, x) =$
 ALT,
 ...
 LDLP $bind(c_i, e, x)$,
 eval(b_i, e, x),
 ENBC,
 ...
 eval(t_j, e, x),
 eval(b_j, e, x),
 ENBT,
 ...
 eval(b_r, e, x),
 ENBS,
 ...
 ALTWT
where
 $\vec{G} = G_1, \ldots, G_r$
 $G_i = b_i\ :\ c_i\,?\,v_i$
 $G_j = b_j\ :\ \mathsf{TIME}\,?\,\mathsf{AFTER}\,t_j$
 $G_k = b_k\ :\ b_k\ :\ \mathsf{SKIP}$
 $1 \leq i \leq l < j \leq p < k \leq r$

if *cmd is* TALT | ALT
then $content(Wreg - 4) := false$ |
 $content(Wreg - 3) := alt_run$

```
if cmd is ENBC
then if Areg = true
      then if content(Breg) = nil
            then content(Breg) := Wreg
            elsif content(Breg) ≠ Wreg
            then content(Wreg − 3) := alt_ready
      Breg := Creg
      Creg := random

if cmd is ENBT
then if Areg = true
      then if content(Wreg − 4) = false
            then content(Wreg − 4) := true
                 content(Wreg − 5) := Breg
            elsif Breg < content(Wreg − 5)
            then content(Wreg − 5) := Breg
      Breg := Creg
      Creg := random

if cmd is ENBS
then if Areg = true
      then content(Wreg − 3) := alt_ready

if cmd is TALTWT|ALTWT
then if content(Wreg − 5) ≥ timer ∧ |
        content(Wreg − 3) = alt_run
      then if content(Wreg − 4)
            then time-insert
            content(Wreg − 3) := alt_sleep
            dequeue
      Areg, Breg, Creg := random
      content(Wreg) := nil
```

$compile(\mathsf{alt_s}(\vec{G}, \vec{l}), e, m, x) =$

```
    ...
    LDLP bind(c_i, e, x),
    eval(b_i, e, x),
    LDC l_i − l_here,
    DISC,
    ...
    eval(t_j, e, x),
    eval(b_j, e, x),
    LDC l_j − l_here,
    DIST,
    ...
    eval(b_k, e, x),
    LDC l_k − l_here,
    DISS,
    ...
    ALTEND,
    l_here
```

where $\vec{l} = l_1, \ldots, l_r$

$\vec{G} = G_1, \ldots, G_r$

$G_i = b_i \; : \; c_i \, ? \, v_i$

$G_j = b_j \; : \; \mathsf{TIME} \, ? \, \mathsf{AFTER} \, t_j$

$G_k = b_k \; : \; b_k \; : \; \mathsf{SKIP}$

$1 \leq i \leq p < j \leq q < k \leq r$

```
if cmd is DISC
then if Breg
      then if content(Creg) = Wreg
            then content(Creg) := nil
                 Areg := false
            elsif content(Creg) ≠ nil
                 ∧ content(Wreg) = nil
            then content(Wreg) := Areg
                 Areg := true
            else Areg := false
      else Areg := false
      Breg, Creg := random

if cmd is DIST
then if Breg
      then if Creg < timer ∧ content(Wreg) = nil
            then content(Wreg) := Areg
                 Areg := true
            else Areg := false
      else Areg := false
      Breg, Creg := random

if cmd is DISS
then if Breg
      then if content(Wreg) = nil
            then content(Wreg) := Areg
                 Areg := true
            else Areg := false
      else Areg := false
      Breg, Creg := random

if cmd is ALTEND
then Ireg := Ireg + content(Wreg)
```

time_wakeup

```
if TPtrLoc ≠ nil ∧ timer > TNextReg
then com_mode(TPtrLoc) := alt_run
     enqueue TPtrLoc
     time-delete TPtrLoc
```

Parallelism

$$env_size \; : \; STATEMENT \rightarrow \mathbf{N}$$
$$spec_size \; : \; STATEMENT \rightarrow \mathbf{N}$$
$$work_size \; : \; STATEMENT \rightarrow \mathbf{N}$$
$$work_size(S) = env_size(S) + spec_size(S)$$

$env_size(\mathsf{SKIP}) = 0$

$env_size(\mathsf{STOP}) = 0$

$env_size(\mathsf{TIME} \, ? \, v) = 0$

$env_size(c \, ? \, v) = 0$

$$env_size(c\,!\,t) = eval_size(t)$$
$$env_size(v := t) = eval_size(t)$$

$$env_size(\mathsf{VAR}\ id_1, \ldots, id_r : S) = r + env_size(S)$$
$$env_size(\mathsf{CHAN}\ id_1, \ldots, id_r : S) = r + env_size(S).$$

$$env_size(\mathsf{WHILE}\ B\,S) = max(\,eval_size(B), env_size(S)\,)$$

$$env_size(\mathsf{IF}\ B_1\,S_1, \ldots B_r\,S_r) = max_{i=1}^{r}\ (eval_size(B_i), env_size(S_i))$$

$$env_size(\mathsf{SEQ}\ S_1 \ldots S_r) = max_{i=1}^{r}\ env_size(S_i)$$

$$env_size(\mathsf{ALT}\ G_1\,S_1 \ldots G_r\,S_r) = max_{i=1}^{r}\ (eval_size(b_i), env_size(S_i))$$

$$env_size(\mathsf{PAR}\ S_1 \ldots S_r) = env_size(S_1) + \sum_{i=2}^{r} work_size(S_i)$$

$$spec_size(\mathsf{SKIP}) = 2$$
$$spec_size(\mathsf{STOP}) = 2$$
$$spec_size(v := t) = 2$$
$$spec_size(\mathsf{TIME}\,?\,v) = 2$$
$$spec_size(c\,?\,v) = 3$$
$$spec_size(c\,!\,t) = 4$$

$$spec_size(\mathsf{VAR}\ id_1, \ldots, id_r : S) = spec_size(S)$$
$$spec_size(\mathsf{CHAN}\ id_1, \ldots, id_r : S) = spec_size(S)$$

$$spec_size(\mathsf{WHILE}\ B\,S) = spec_size(S)$$

$$spec_size(\mathsf{IF}\ B_1\,S_1, \ldots B_r\,S_r) = max_{i=1}^{r}\ (spec_size(S_i)$$

$$spec_size(\mathsf{SEQ}\ S_1 \ldots S_r) = max_{i=1}^{r}\ spec_size(S_i)$$

$$spec_size(\mathsf{ALT}\ G_1\,S_1 \ldots G_r\,S_r) = \begin{cases} max_{i=1}^{r}\ (4, spec_size(S_i)) & \text{case (a)} \\ max_{i=1}^{r}\ (6, spec_size(S_i)) & \text{case (b)} \end{cases}$$

$$spec_size(\mathsf{PAR}\ S_1 \ldots S_r) = max(4, spec_size(S_1))$$

$compile((\mathsf{PAR}\ S_1 \ldots S_r), e, m, x) =$
  `AJW` -2
  $compile(\mathsf{par}(x, \vec{m}, \vec{l}, l_{Out}), e, m, x-2)$,
  $compile(S_1, e, m_1, x-2)$,
  $compile(\mathsf{end}(x-2), e, m, x-2)$,
  l_2,
  $compile(S_2, e, m_2, m_2)$,
  $compile(\mathsf{end}(m_2), e, m_2, x-2)$,
  ...
  l_r,
  $compile(S_r, e, m_r, m_r)$,
  $compile(\mathsf{end}(m_r), e, m_r, x-2)$,
  l_{Out}, `AJW` $+2$
where $\vec{l} = l_2, \ldots, l_r$
  $\vec{m} = m_2, \ldots, m_r$
  $m_1 = m$
  $m_i = m_{i-1} + env_size(S_{i-1}) + spec_size(S_i)$
  $2 \le i \le r$

$compile(\mathsf{par}(x, \vec{x}, \vec{l}, l_{Out}), e, m, x) =$
  `LDC` r,
  `STL` 1,
  `LDC` $l_{Out} - l_{here}$
  `LDPI`,
  l_{here},
  `STL` 0,
  ...
  `LDC` $l_i - l'_i$,
  `LDLP` $x_i - x$
  `STARTP`,
  l'_i,
  ...
where $\vec{l} = l_1, \ldots, l_r$
  $\vec{x} = x_1, \ldots, x_r$
  $1 \le i \le r$

$compile(\mathsf{end}(x_i), e, m, x) =$ `LDLP` $x - x_i$, `ENDP`

if *cmd is* `STARTP`
then $loc(Wreg + Areg)$ **:=** $Breg$
  enqueue $Wreg + Areg$

if *cmd is* `ENDP`
then if $content(x_f) > 1$
  then *dequeue*
    $content(x_f + 1)$ **:=** $content(x_f + 1) - 1$
  else $Ireg$ **:=** $content(x_f)$
    $Wreg$ **:=** x_f
where $x_f = Wreg + Areg$

REFERENCES

[Boerger:95]: E.Börger: *On the use of evolving algebras in software engineering.* in: M.Bartosek, J.Staudek, J.Wiedermann (Eds), SOFSEM'95 22nd Seminar on Current Trends in Theory and Practice of Informatics, Springer Lecture Notes In Computer Science, vol. 1012, 1995, pp.30.

[BĐR:94]: E.Börger, I.Đurđanović & D.Rosenzweig, 1994, *Occam: Specification and Compiler Correctness. Part I: The Primary Model*, E.-R. Olderog (Ed.), Proc. PROCOMET'94 (IFIP Working Conference on Programming Concepts, Methods and Calculi), pages 489-508, North-Holland 1994

[BöRo:95a]: E.Börger & D.Rosenzweig, 1995, *A mathematical definition of Full Prolog*, in: *Science of Computer Programming* 24, pages 249-286.

[BöRo:95b]: E.Börger & D.Rosenzweig, 1995, *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, pages 20-90 (chapter 2).

[Bowen:93]: 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.

[BowHe:93]: J.P. Bowen, He Jifeng, 1993, *Specification, Verification and Prototyping of an Optimized Compiler*, in: *Formal Aspects of Computing.*

[BowHePand:90]: J.P. Bowen, He Jifeng, P.K. Pandaya, 1990, *An Approach to Verifiable Compiling Specification and Prototyping*, Springer Verlag, LNCS 456, pp. 45-59.

[Graham:90]: Ian Graham, 1990, *The Transputer Handbook*, Prentice Hall.

[Gur:95]: Y. Gurevich. Evolving Algebras 1993: Lipari Guide. *Specification and Validation Methods*, Ed. E. Börger, Oxford University Press, 1995.

[Hoare:91]: C.A.R. Hoare, *refinement algebra proves correctness of compiling specifications.* In C.C. Morgan and J.C.P. Woodcock, editors, 3rd *Refinement Workshop*, Workshops in Computer Science, pages 33–48. Springer Verlag, 1991.

[Hoare et al.:87]: C.A.R. Hoare, I.J. Hayes, He Jifeng, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sorenson, J.M. Spivey, and B.A. Sufrin. *Laws of programming.* Communications of the ACM, 30(8):672-687, August 1987.

[Hoare et al.:93]: C.A.R. Hoare, He Jifeng, and A. Sampaio. *Normal form approach to compiler design.* Acta Informatica, 30:701-739, 1993.

[Inmos]: INMOS, *Transputer Implementation of Occam.* In: *Communication Process Architecture*, Prentice Hall, note 21, 1989.

[Inmos:88]: INMOS, 1988, *Transputer Instruction Set – A compiler writer's guide*, INMOS document 72 TRN 119 05, Prentice Hall.

[Mitchell:90]: D.A.P. Mitchell et al, 1990, *Inside the Transputer*, Blackwell Scientific Publications.

[MMO:95]: Markus Müller–Olm, *Structuring Code Generator Correctness Proofs by Stepwise Abstracting the Machine Language's Semantics*, ProCos II Esprit basic Research 7071, [Kiel MMO 12/3] (doctoral dissertation).

[PageLuk:91]: 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.