in Coordination Languages and Models, D. Garlan and D. LeMétayer (eds),
LNCS 1282, pp. 115-130, 1997.

Coordinating Durative Actions(*)

I. Nunes
J.L.Fiadeiro
Department of Informatics
Faculty of Sciences, University of Lisbon
Campo Grande
1700 Lisboa, Portugal
{in,llf}@di.fc.ul.pt

W.M.Turski
Institute of Informatics
Warsaw University
ul. Banacha 2
02-097 Warsaw, Poland
wmt@mimuw.edu.pl

Abstract. A computing paradigm is presented for coordinating the execution of durative actions, i.e. actions which, although executed atomically on a private local state, have a duration in the sense that the system state in which they finish executing is not necessarily the same in which they started. Just as in traditional coordination languages, the coordination model that is responsible for controlling the interference between the actions is independent from the computation model in which actions execute. This coordination model is formalised through an operational and a denotational semantics, both parameterised by those of the underlying computational model.

1. Introduction

The behavioural paradigm developed in [11,12] for supporting multiprocessor computing is presented as a coordination model for controlling the execution of durative actions, i.e. actions which, although executed atomically on a private local state, have a duration in the sense that the system state in which they finish executing is not necessarily the same in which they started. Systems of durative actions arise naturally from the need to coordinate the behaviour of individual programs that can access a shared data space, namely to achieve interoperability of, possibly heterogeneous, components in the context of reuse-based and incremental system development, and for which the atomicity imposed through the use of a single processor executing a single program at the time is simply unrealistic and undesirable.

The proposed coordination model relies on a small set of primitives that support, on the one hand, local computations without interference of the environment and, on the other hand, the decoupling between the global states in which actions are launched and the global states in which they find the system when they finish executing.

(*) This work was partially supported by JNICT through contracts PRAXIS XXI 2/2.1/MAT/46/94 (ESCOLA), PCSH/OGE/1038/95 (MAGO) and PRAXIS XXI 2/2.1/TIT/1662/95 (SARA).

Like in (more) traditional coordination languages, the coordination model that is responsible for controlling the interference between the actions is independent from the computation model in which actions execute. That is to say, it does not really matter if the computations performed locally by the actions are programmed in a, say, imperative or functional style.

In section 2, we present the behavioural paradigm in detail, from an operational, but informal point of view, and compare it to (more) traditional coordination models. In section 3, behavioural programs are formalised as sets of coordination primitives used to relate programs written in some programming language whose details are considered irrelevant. In section 4, we formalise the rules that provide an operational semantics for behavioural programs. In section 5, we formalise the relationship between this operational semantics and a denotational semantics defined over Durative Transition Systems – an extension of labelled transition systems introduced in [7,8] to cope with action duration.

2. Coordination in the behavioural paradigm

It seems hard to improve on the characterisation of the distinction between *computation* and *coordination* that was proposed in [5]: "[A] computation model allows programmers to build a single computational activity: a single-threaded, step-at-a-time computation; [a] coordination model is the glue that binds separate activities into an ensemble".

The aim of this paper is to present a computing paradigm proposed in [11,12] – the *behavioural paradigm* – which explores this very distinction between coordination and computation, and proposes primitives for coordinating the interference between actions which perform computations over a common state space.

In order to understand the essence of the behavioural paradigm, consider the chemical reaction metaphor which is at the base of coordination formalisms, namely Gamma [1]:

Reaction condition $\rightarrow$ Action

Execution consists in removing from the "chemical solution" (the multiset that provides the global state space) the elements that satisfy the reaction condition, replacing them by the results of the action. Evaluation of the reaction condition and computation of the effects of the action are done atomically ("test and set") in the sense that the state of the system (the multiset) does not change between the evaluation of the condition and the computation of the action. The computation of the action is also done atomically in this sense: its effects are going to update the same state that was used for evaluating the condition and determining the course of the computation.

One of the characteristics of the behavioural paradigm is the relaxation of the assumption on the atomicity of the execution of the action. More precisely, the action computes in a local copy of the relevant part of the state of the system and, hence, atomically in the sense that the computation proceeds without interference from the environment. However, while the computation proceeds on the local copy, the state of the system may change because other actions may have terminated. Hence, the state in which the computation induced by the action ends is not necessarily the same in which the execution started.

Because the state in which the reaction condition is evaluated is still the same in which the execution starts, this relaxation implies that the conditions that lead to the execution of the action are not necessarily met when the execution ends. Hence, it may well be true that the effects computed locally are no longer of interest. To determine whether the global state should or should not be updated with the effects computed by the action, the behavioural paradigm relies on a second condition – the *post-guard* – which is evaluated when the local execution of the action ends (the so-called *accepting state*). The global state is updated iff the post-guard evaluates to *true*. The evaluation of the post-guard and the update of the global state are done atomically ("test and set").

The other important characteristic of the behavioural paradigm is that it addresses *multiprocessor computing*. Systems evolve in an environment populated by one or more agents which are identical in their ability to perform (local) programs, except perhaps, for efficiency. Agents are equipped with memories in which they execute the actions that they are assigned. For that purpose, they are able to *load* into their memories the relevant part of the state of the system when the execution of an action is assigned, and to *discharge* their local states into the global state of the system when the execution of an action ends and the corresponding post-guard is true. Loading and discharging are instantaneous.

All actions that *can* be initiated in a given state *are* initiated, regardless of any possible future conflict; conflicts are resolved in the accepting states. Moreover, no agent is allowed to idle in a system state in which a reaction condition (*pre-guard* in the terminology of the behaviour paradigm) is satisfied. In particular, several copies of the same action can be launched if there is more than one agent free or if an agent becomes free while the pre-guard of the action remains true. This is justified by the fact that some agents may be slower than other agents; the first to terminate gets the chance to update the global state of the system if the post-guard is true; the effects of the action may falsify the post-guard thus preventing the agents performing the other copies of the action to terminate productively. But, as in nature, success may be reserved for some slower agent that gets its chance from changes performed in its environment. The "waste" of computations is only apparent because, on the one hand, all that is done is to keep busy agents that would otherwise be idle (pre-guards that are false can only become true due to the termination of some action, which automatically frees an agent) and, on the other hand, the more the copies in execution the better the chances of success become. (Again, nature is very good at launching several processes in concurrency even if it is known that at most one will be successful!) Besides, it is assumed that actions are assigned to agents in a fair way: when more than one can be assigned, the actual choice is unbiased. Moreover, as processors become cheaper and cheaper, multiprocessor environments can be easily populated with a fair number of agents.

The model that we have just described also justifies that, unlike the tuple space model with generative communication used by coordination languages [2,4], where agents communicate by putting and consuming tuples in the tuple space, the only moment when, in the behavioural paradigm, the state of the system is modified is at discharging, which can be seen as a substitution of some values for other ones. That is, evaluation of a pre-guard (reaction condition) does not change (consume from) the global state. This happens precisely to allow for existing free agents to get a chance of picking another instance of the launched action.

A behavioural program is an initial condition together with a set of rules where each rule is called a double guarded action – *dga* – and takes the form

$$\text{(pre-guard,postguard)} \rightarrow [\text{loading} \mid \text{program} \mid \text{discharging}]$$

By *loading* and *discharging* we are denoting lists of attribute symbols. These are the names of the attributes of the system (the "program variables") which, in the behavioural paradigm, can be typed as usual in programming languages. This is a departure from the multiset / (tuple space) representations typical of coordination languages, but not an essential one.

The loading and discharging lists account for the attributes whose values an agent has to load into its local state when assigned to perform the action, and the attributes that it has to update when it discharges its local state. Loading lists are optional in the sense that the full state of the system could be loaded each time an action is launched. The same does not hold for the discharging list. This list performs an important role in determining the global state of the system that results from the acceptance of an action (see below).

In order to illustrate the coordination model just described, consider the "dining philosophers". The following specification is that of a given philosopher (this example is taken from [11]).

$$(LOT \wedge ROT \wedge \neg H)$$

$$\begin{array}{rcl}
(\neg H, \neg H) & \rightarrow & [\ \mid \text{think} \mid H] \\
(H \wedge PR \wedge PL,\ H \wedge PR \wedge PL) & \rightarrow & [\ \mid \text{eat} \mid H, PR, ROT, PL, LOT] \\
(H \wedge PL \wedge \neg PR \wedge \neg ROT,\ \neg ROT) & \rightarrow & [\ \mid \text{rel_L} \mid PL, LOT] \\
(H \wedge PR \wedge \neg PL \wedge \neg LOT,\ \neg LOT) & \rightarrow & [\ \mid \text{rel_R} \mid PR, ROT]
\end{array}$$

$$(H \wedge LOT, H \wedge LOT) \quad \rightarrow \quad [\ \ |\ take\_L\ |\ PL, LOT]$$

$$(H \wedge ROT, H \wedge ROT) \quad \rightarrow \quad [\ \ |\ take\_R\ |\ PR, ROT]$$

where the propositional symbols are used with the following meanings:

- H – the philosopher is hungry
- LOT – the left fork is on the table
- ROT – the right fork is on the table
- PL – the philosopher possesses the left fork
- PR – the philosopher possesses the right fork

The condition ($\neg H \wedge LOT \wedge ROT$) defines initial states: when program execution begins, the philosopher must not be hungry and both left and right forks must be on the table.

In this example all loading lists are empty. This means that the computations performed locally by each action do not depend on the state in which the action is launched.

The intended semantics of the rel_L action specification, for example, is the following. When a philosopher is hungry (H), possesses the left fork (PL) and does not possess the right fork ($\neg$PR), and the right fork is not on the table ($\neg$ROT), he will release the left fork (the action is launched because the pre-guard is true). This is so specified in order to prevent deadlock. If, while our philosopher is in the process of releasing his left fork, the philosopher at his right releases his left fork (our philosopher's right fork), then our philosopher should not bring about the action. This is expressed through the post-guard: if the right fork is on the table (ROT) when the philosopher finishes executing the release, the action is rejected. The pre-guard of the action *take_R* is now true and so the process of picking-up the right fork is launched.

The only situation for which no action is specified is the one where the philosopher is hungry, holds no fork and no fork is on the table. This is not, fortunately, a stable state: the missing forks are held by neighbours, and they will eventually release them. For all other situations well-defined actions are prescribed; if the results of these actions are not accepted at any time it is because a neighbour successfully completes an action of his own. In a system composed of several philosophers this means that some action is accepted in each state. Thus this specification is deadlock free.

It is perhaps worthwhile to observe that this specification avoids deadlocks without referring to any information on the actual state of the philosopher's neighbours. As soon as the information on neighbours is explicitly available and used, a conspiracy to starve a common neighbour becomes possible. In our design, where this information is not available (nor needed), starvation is possible either because of a gross heterogeneity of philosophers, or as a chance phenomenon. Even if the philosophers are roughly similar, a pattern of fork movements preventing one or more of their number from eating would not be inconsistent with the design, just as an arbitrarily long run of heads is not inconsistent with repeated tossing of a fair coin. This seems unavoidable, but practically irrelevant.

We have deliberately omitted the definition of the program executed locally by each action to remind that the behavioural paradigm is all about coordinating durative actions and not about programming their effects. This is achieved through the pre and post-guards and by the loading and discharging lists. Naturally, these local programs have to be supplied for the behavioural program to be complete. For instance, in the case of rel_L, this could be

PL:=false; LOT:=true

assuming that the programming language in which rel_L is given uses local attributes with the same names as the global attributes of the system. But it could well be that rel_L uses different attributes, in which case the discharging of the local state has to be a bit more than a simple copy. However, it is essential that discharging (as well as loading) are atomic (and efficient!).

3. Coordination Primitives + Programs = Behavioural Programs

Summarising the informal discussion of the previous section, the behavioural paradigm is characterized by a collection of primitives whose aim is to control the interference between durative actions, and which clearly separates computation concerns from coordination ones:

- The programming language in which computations are programmed is not constrained to any particular one: we rely on an interface that is provided by the loading and discharging operations;
- Pre- and post-guards stand for activation/permission criteria to coordinate the execution and productivity of actions.

A behavioural program is a set of double-guarded actions (*dga*s), each of which includes two guards – the pre and post-guards – two lists of attributes – the loading and the discharging lists – and the program executed locally by the action.

In other words, a behavioural program is a collection of individual programs implicitly related through some coordinating and comunication primitives. These are the guards and the interface lists of attributes. The guards control *why* and *what* is to be made active in the system, as reactions to the system state. The loading list defines what an agent has to know about the system state in order to execute an action. The discharging list defines to what extent does an accepted action change the system state.

In order to formalize the notion of behavioural program we begin by defining the concept of system signature which consists of a declaration of the specific vocabulary symbols that are relevant for that system description.

For simplicity, we shall assume that a collection of data types has been fixed which provides a data signature U=(S,F), where S is a set of sorts and F is an $S^* \times S$-indexed family of function symbols.

Definition 3.1: A signature Θ is a pair (At,Ac) where:

- At is an S-indexed family of sets of attribute symbols,
- Ac is a set of action symbols equipped with two mappings L and D providing, for each action $g \in Ac$, its loading (L_g) and discharging (D_g) lists, each of which is an S-indexed family of sets of attribute symbols. ♦

Definition 3.2: The language $\mathcal{L}(At)$ for Θ of state formulas, is defined by:

$$e_s ::= p | f(e_{1_{s_1}}, ..., e_{n_{s_n}})$$

where $p \in At_s$ and $f:s_1,\ldots,s_n \rightarrow s \in F$

$$A ::= e_{1_s} = e_{2_s} | \neg A | A_1 \supset A_2$$ ♦

We will use the traditional propositional constants and connectives tt, ff, $\wedge$, $\vee$, $\equiv$, defined through the usual abbreviations.

As already mentioned, the behavioural paradigm is not characterised by a specific syntax for its programs but through a collection of primitives whose aim is to coordinate durative actions. In particular, the durative actions may be programmed in any programming language, subject to some restrictions. These restrictions, or assumptions, on the nature of the language that can be used for programming actions, constitute what we call a *context*. In so far as this section is concerned, we only need to know about syntactical aspects, which we call a *programming context* for a given signature.

Definition 3.3: A programming context $\mathcal{T}_p(\Theta)$ for a signature Θ is a language PL. ♦

No assumption is made on the actual syntax of the programming language PL besides the fact that it has to work over the data types defined in the signature. The idea is that, to coordinate durative actions, we do not need to know explicitly how their internal computations are programmed. Instead, we rely on an interface that is provided through the global attributes At and the loading and discharging lists for each action.

Definition 3.4: Given a programming context $\mathcal{T}_p(\Theta)$ for a signature Θ=(Ac,At), a behavioural program is a pair BP = (I,BDY) where I is an initial condition and BDY assigns, to every $g \in Ac$, a triple (A_g,P_g,C_g) where:

- A_g and P_g are formulas of $\mathcal{L}(At)$, standing for the pre- and the post-guard of action *g*, respectively;

- C_g is a program in the language PL.

Each such triple is called a *dga* (double-guarded action specification). ♦

4. An operational semantics for behavioural programs

Here we present an operational semantics as a set of rules which describe, formally, execution in the behavioural paradigm. These rules specify how expressions are evaluated and how *dga*s are executed.

Before proceeding we must define the notion of state for a signature. Again, we assume a fixed U-algebra $\mathcal{U} = (S_{\mathcal{U}}, F_{\mathcal{U}})$ that interprets the data types corresponding to U, where $S_{\mathcal{U}} = \{s_{\mathcal{U}} | s \in S\}$ and $F_{\mathcal{U}} = \{f_{\mathcal{U}}: s_{1\mathcal{U}} \times ..... \times s_{n\mathcal{U}} \to s_{\mathcal{U}} | f: s_1, \ldots, s_n \to s\}$.

Definition 4.1: A state σ for a signature Θ=(At,Ac) is an S-indexed family $(\sigma_s: At_s \to s_{\mathcal{U}})\ s \in S$ where, for every $s \in S$, σ_s assigns a value over its domain to each attribute in At_s. The notation σ_Ω stands for the projection of state σ in set Ω, where $\Omega_s \subseteq \mathcal{A}t_s$ for all $s \in S$. The notation $\sigma[p]$ stands for the value of attribute p in state $\sigma \in \Sigma$. The notation $\sigma'[\sigma_\Omega/\Omega]$ stands for the state that is equal to σ' except for the values of attributes in Ω whose values are given by σ_Ω. ♦

State formulas as defined in 3.2 can be evaluated over such states:

Definition 4.2: The value of an expression $f(e_1, \ldots, e_n)$ at σ – written $\sigma[f(e_1,\ldots,e_n)]$– is $f_{\mathcal{U}}(\sigma[e_1],\ldots,\sigma[e_n])$. ♦

Definition 4.3: We define the satisfaction relation $\models_\Theta: \Sigma \times \mathcal{L}(At)$ inductively as follows:

$\sigma \models_\Theta e_1 = e_2$	iff	$\sigma[e_1] = \sigma[e_2]$
$\sigma \models_\Theta \neg A$	iff	$\sigma \not\models_\Theta A$
$\sigma \models_\Theta A_1 \supset A_2$	iff	$\sigma \models_\Theta A_1$ implies $\sigma \models_\Theta A_2$ ♦

Now that we already have a precise notion of state we may extend the programming context with the denotational semantics of all programs of actions of Ac.

Definition 4.4: An extension $\mathcal{T}_d$ of a programming context $\mathcal{T}_p$=PL for a signature Θ consists of a pair (PL,$\{[\![C_g]\!] | g \in Ac\}$) where,

- $[\![C_g]\!]:(\Sigma_{Lg} \to \Sigma_{Dg})$ is a denotational semantics for program C_g; we denote by Σ_{Lg} the set $\{\sigma_{Lg} | \sigma \in \Sigma\}$ and by Σ_{Dg} the set $\{\sigma_{Dg} | \sigma \in \Sigma\}$;

We also call such an extension $\mathcal{T}_d$, a *denotational context* for Θ. ♦

At the local level, we see that, because they are deterministic, actions are functions and, because they are atomic, they are functions from a local state to another local state.

Local computations are executed by agents which have an identity. That is, agents may be referred to, even though a given agent does not need to know the identity of the agents with which it cooperates. Each agent works over a (local) private state.

When an agent is assigned an action, a *loading* of the values it needs to accomplish its task is performed into its private state and the agent starts execution. If the system is receptive at the time the agent finishes executing the action, that is, the post-guard of the action is satisfied by the system state, then a *discharging* of its output values must be done (from its private state).

The need for the loading and discharging operations results from the fact that we allowed each agent to have its own states. Hence, the loading operation defines how the attributes in L_g are *loaded*: whether they are just copied or translated in some way to define a local state is a detail that is not relevant. Whatever the loading operation is, it should be a simple one because it needs to be performed atomically. The discharging operation gives to the system attributes in D_g a value computed by action g. As with the loading operation, the details of the discharging operation are not relevant here.

In order to define the operational semantics of behavioural programs, we need to extend programming contexts as defined in 3.4 to account for the executing agents and the loading and discharging operations as well as the operational semantics of the language used to program actions.

Definition 4.5: An operational extension of a programming context $\mathcal{T}_p$=PL for a signature Θ consists of a tuple $(PL,Pr,\{\Sigma_a|a\in Pr\},\{(l^a_g,d^a_g,\xrightarrow{a})|(a,g)\in Pr\times Ac\})$ where,

- Pr is a set of agent symbols that stand for the existing executing agents;
- Σ_a is the local state space (memory) of agent *a*;
- $l^a_g: \Sigma_{Lg} \rightarrow \Sigma_a$ gives semantics to the *loading* operations;
- $d^a_g: \Sigma_a \rightarrow \Sigma_{Dg}$ gives semantics to the *discharging* operations;
- $\xrightarrow{a}:(PL\times\Sigma_a \rightarrow \Sigma_a)$ is an operational semantics for comands in PL, for agent *a* .

We also call such an extended context $\mathcal{T}_o$, an *operational context* for Θ. ♦

Definition 4.6: An operational $\mathcal{T}_o = (PL,Pr,\{\Sigma_a|a\in Pr\},\{(l^a_g,d^a_g,\xrightarrow{a})|(a,g)\in Pr\times Ac\})$, and a denotational $\mathcal{T}_d = (PL,\{[\![C_g]\!]_g|g\in Ac\}$ contexts for a signature Θ are said to *agree* iff

for every $\sigma\in\Sigma,a\in Pr$, $[\![C_g]\!]\sigma_{Lg} = d^a_g(\xrightarrow{a}(C_g,l^a_g(\sigma_{Lg})))$ ♦

That is to say, the program C_g, whose denotational semantics is given by $[\![C_g]\!]$, is *implemented* by each agent *a* according to a "local" operational semantics $\xrightarrow{a}$ that, given program C_g and an agent *a*'s private state (the one resulting from applying *a*'s *loading* operation to $[\![C_g]\!]$ argument – σ_{Lg}), returns another private state of agent *a* (if we apply *a*'s *discharging* operation to this local state, we obtain $[\![C_g]\!]\sigma_{Lg}$ result – σ_{Dg}).

At the operational level, transitions between *configurations* are defined, that is, the several steps of the execution of a behavioural program BP are reflected in the global state of the system (updating it whenever an agent discharges the results of its task) and in the several configurations of the agents (updating them whenever an action is assigned to an agent, and whenever an agent becomes free, whether having discharged the results of its task or not).

Definition 4.7: Given an operational context $\mathcal{T}_o$ for a signature Θ, a *configuration* for Θ and Pr is a pair $<\sigma,\alpha>$ whose first component is a state for Θ and whose second component is a Pr-indexed set of agent configurations $\alpha = \{<st_a,ac_a,ls_a>|a\in Pr\}$ where:

- $st_a \in \{idle, busy, ready\}$ stands for the activity state of agent *a*, that is,
  – *a* is *idle* if *a* is unoccupied (with no action assigned to it);
  – *a* is *busy* if *a* is assigned an action and executing it;
  – *a* is *ready* if *a* has finished executing the action it was assigned to and it is waiting to become free (with or without discharging);
- $ac_a \in Ac$ stands for the action agent *a* is assigned to;
- $ls_a \in \Sigma$ stands for the state in which the system was in when the action in ac_a was assigned to agent *a*, that is, the action launching state. ♦

Configurations do not contain a program part because a behavioural program is a reactive program, that is, it does not "consume" itself when executing. Therefore, all *dga*s are always active in the sense that the program reacts to the environment at all times according to them.

Given a behavioural program BP and an operational context $\mathcal{T}_o$ for a signature Θ, we define a relation $\rightarrow \subseteq C\times C$, where C is the set of all configurations for Θ and Pr, by

$$\text{for all } c,c'\in C,\ \ c \rightarrow c' \text{ iff } c \rightarrow_{np} c' \text{ or } c \rightarrow_p c'$$

We represent by $<\sigma_1,\alpha_1> \rightarrow_{np} <\sigma_2,\alpha_2>$ the *non-productive* reaction of BP to configuration $<\sigma_1,\alpha_1>$, resulting in configuration $<\sigma_2,\alpha_2>$; *non-productive* means that no discharging of locally computed values happens in the transition from $<\sigma_1,\alpha_1>$ to $<\sigma_2,\alpha_2>$, and so, the system state σ_2 is the same as σ_1. A transition $<\sigma_1,\alpha_1> \rightarrow_p <\sigma_2,\alpha_2>$ represents a *productive* transition, that is, one where a discharging over the system state is done. This does not mean, however, that σ_1 is necessarily different from σ_2; it may happen that the discharge did not change the global state. These relations will be defined below through rules which provide an operational semantics for the behavioural paradigm.

The rules of the operational semantics have to control the launching and acceptance of actions according to the behaviour paradigm. With respect to launching, whenever the pre-guard of an action *g* is true of the system state, action *g* is assigned to an idle agent *a* (changing its configuration accordingly).

With respect to acceptance, whenever an agent *a* finishes executing an action *g*, which is signaled through the change of its state of activity to *ready*, and if the system state verifies the post-guard of *g*, it *discharges* the results computed locally by *g* into the system state according to *g*'s discharging list. After this, agent *a* returns to an *idle* state of activity.

If the system state does not verify *g*'s post-guard, agent *a* returns to an *idle* state of activity without discharging *g*'s results.

It can be the case that a configuration is reached such that the system state does not verify none of the action pre-guards and, moreover, it does not verify the post-guards of none of the pending actions: the executing actions (the ones that are assigned to *busy* agents) and the already executed ones (assigned to *ready* agents). In this case we say that the system reached a *terminal* configuration because no more productive transitions are possible. We will denote *terminal* configurations by $\langle\sigma,\alpha\rangle\surd$.

Definition 4.8: Given a Pr-indexed set α of agent configurations,

- $\alpha(a)$ stands for the configuration of agent *a*, that is, $\langle st_a, ac_a, ls_a\rangle$
- $\alpha(a).x$ stands for the value of x in *a*'s configuration, where x can be one of *st*,*ac* and *ls*
- $\alpha[\langle v_1,v_2,v_3\rangle/a]$ represents the set of configurations that results from α by assigning $\langle v_1,v_2,v_3\rangle$ to the configuration of agent *a* . When any of the v_i is not made explicit, it means that its value is not relevant in that context; $\alpha[\langle v_1, \, , \,\rangle/a]$ stands for the set of configurations that results from α by assigning v_1 to the *st* component of agent *a* . ♦

The operational semantics of a behavioural program is given over an operational context.

Definition 4.9: Given $\mathcal{T}_o = (PL, Pr, \{\Sigma_a | a \in Pr\}, \{(l_g^a, d_g^a, \xrightarrow{a}) | (a,g) \in Pr \times Ac\})$ – an operational context for a signature $\Theta = (At,Ac)$ – the following set of rules provides an operational semantics for a behavioural program $BP = (I, \{(A_g,P_g,C_g) | g \in Ac\})$:

1. $$\frac{\sigma \models_\Theta A_g \qquad \alpha(a).st = idle}{\langle\sigma,\alpha\rangle \rightarrow_{np} \langle\sigma,\alpha[\langle busy,g,\sigma\rangle/a]\rangle}$$

2. $$\frac{\sigma' \models_\Theta P_g \qquad \alpha(a) = \langle ready,g,\sigma\rangle \qquad \langle C_g, l_g^a(\sigma_{Lg})\rangle \xrightarrow{a} \sigma^a}{\langle\sigma',\alpha\rangle \rightarrow_p \langle\sigma'[d_g^a(\sigma^a)/D_g],\alpha[\langle idle, \, , \,\rangle/a]\rangle}$$

3. $$\frac{\sigma' \not\models_\Theta P_g \qquad \alpha(a) = \langle ready,g,\sigma\rangle}{\langle\sigma',\alpha\rangle \rightarrow_{np} \langle\sigma',\alpha[\langle idle, \, , \,\rangle/a]\rangle}$$

4. $$\frac{\bigwedge_{g\in Ac}(\bigwedge_{a\in Pr} ((\alpha(a) = \langle ready,g, \,\rangle \vee \alpha(a) = \langle busy,g, \,\rangle) \supset \sigma \not\models_\Theta P_g) \wedge \sigma \not\models_\Theta A_g)}{\langle\sigma,\alpha\rangle \rightarrow_{np} \langle\sigma,\alpha\rangle\surd}$$

♦

The rules for the evaluation of state formulas are as defined in 4.3.

The first rule says that whenever there is an action *g* whose pre-guard is true of the system state and there is an available agent *a*, action *g* is assigned to idle agent *a* and a copy of the system state is kept on *a*'s configuration. The system state remains the same. This is a non-productive transition.

The second rule says that whenever the system state satisfies the post-guard of some *locally executed* action (the agent which executed it is *ready*), the results of that action are *discharged* to the system attributes in D_g while the attributes in $\mathcal{A}t\backslash D_g$ keep their values. The set of agent configurations also changes insofar as the activity of the discharging agent is set to *idle*. This is a productive transition.

Notice how the duration of action execution is simulated: when an agent is assigned an action its state of activity is set to *busy*. The acceptance of its results is only considered when the agent is in a *ready* state of activity. This acts like some flag the agent waves when it ends its task. The actual transition from *busy* to *ready* is performed locally within the state of the agent. Because we are only concerned with the coordination model, this transition has to be determined according to the underlying computation model, and is not made explicit in the rule. The premiss $<C_g, l_g^a(\sigma_{Lg})> \xrightarrow{a} \sigma^a$ is the witness of the result of some local operational semantics and acts as interface between the computation and the coordination models.

Notice that the values of the attributes of the system – At – in the resulting state of a given action *g*, are completely determined. The values for attributes in D_g are given through the execution of C_g and depend on the values of the attributes in L_g at launching time (*g* is executed locally over the local state that is the result of *loading* the launching state. This launching state is the one that was stored in the agent's configuration at the time of the assignment); the other attributes – in $At\backslash D_g$ – depend on "what went on" in the system while *g* was executing, that is, they keep the value they had in the accepting state.

The third rule, a non-productive one, says that if the system state does not satisfy the post-guard of some locally executed action, the results of that action are ignored: the system state does not change. Only the set of agent configurations changes: the activity of the given agent is set to *idle*.

The fourth rule is the one that relates configurations for which no more productive transitions are possible, with the special terminal configuration.

5. A denotational semantics for behavioural programs

The rules for operational semantics usually define labelled transitions systems (e.g. as in [6] and [10]). These are pairs $<\Sigma,\{\xrightarrow{g} | g\in Ac\}>$ where Σ is a non-empty set (of states), Ac is a set (of actions/labels) and, for each $g\in Ac$, $\xrightarrow{g}\subseteq\Sigma\times\Sigma$ is the transition relation (a partial function if actions are deterministic). Given $\sigma,\sigma'\in\Sigma$, $\sigma\xrightarrow{g}\sigma'$ means that action *g* takes the system from state σ to state σ'. In this context, each transition within the system is completely defined by an action and two states, the launching and the resulting ones.

In the behavioural paradigm, observable transitions relate not two but three states: the launching, the accepting and the resulting ones. We consider that the behaviour of a program obtains from productive transitions only, that is, we do not consider the assignment and deassignment of agents as part of the observable behaviour. Notice that, although we ignore the agents changes of activity, the information about which action they execute and the corresponding launching state is used in order to determine the resulting state of a given transition. The *ls* component of a *ready* agent represents the state the system was in when the action it executes was launched. This launching state is crucial, together with the accepting one, to determine the resulting state.

We are going to show how the operational semantics given in 4.9. defines an extended version (by inclusion of an initial state) of *durative transition systems* in the sense of [7,8] – $<\Sigma,\sigma_0,\{\xrightarrow{g} | g\in Ac\}>$ – where each *durative transition* – $\xrightarrow{g}:(\Sigma\times\Sigma)\rightarrow\Sigma$ – is the set of triples composed by the launching, accepting and resulting states of all instances of action g.

We shall need the following auxiliary definitions in order to define durative transitions:

Definition 5.1: Given an operational context $\mathcal{T}_o$ for a signature Θ we call α_{idle} the set of agents configurations where all agents are idle. ♦

Definition 5.2: A configuration $<\sigma,\alpha>$ is said to be *reachable from* a configuration $<\sigma_1,\alpha_1>$ in a given program BP iff it is possible to obtain $<\sigma,\alpha>$ from $<\sigma_1,\alpha_1>$ by successive application of the operational rules, that is,

$$<\sigma_1,\alpha_1> \rightarrow^* <\sigma,\alpha> \qquad ♦$$

Given a behavioural program BP = (I,BDY), let a productive rule be applied to a configuration $<\sigma',\alpha>$, where agent a is such that $\alpha(a)=<ready,g,\sigma>$, changing it to $<\sigma'',\alpha[<idle, , >/a]>$. Let configuration $<\sigma',\alpha>$ be reachable from $<\sigma_0,\alpha_{idle}>$ where σ_0 is a state where I holds; then we know that there was a transition which assigned action *g* to agent *a* – because *g*'s pre-guard was true of the system state – and stored that launching state in *a*'s configuration. Then, the triple $(\sigma,\sigma',\sigma'')$ represents the launching, accepting and resulting states of an instance of action *g*. The durative transition $\xrightarrow{g}$ is composed of all triples $(\sigma,\sigma',\sigma'')$ under these circumstances.

Definition 5.3: The durative transition system $\mathcal{D} = <\Sigma,\sigma_0,\{\xrightarrow{g}|g\in Ac\}>$ *defined by* a behavioural program BP = $(I,\{(A_g,P_g,C_g)|g\in Ac\})$ is such that:

- $\sigma_0 \models_\Theta I$
- $(\sigma,\sigma',\sigma'')\in\xrightarrow{g}$ iff for every $<\sigma',\alpha>$ s.t.

  $<\sigma',\alpha>$ *reachable from* $<\sigma_0,\alpha_{idle}>$ and for some a, $\alpha(a)=<ready,g,\sigma>$,

  $<\sigma',\alpha> \rightarrow_p <\sigma'',\alpha[<idle, , >/a]>$ ♦

The operational semantics just presented is shown to agree with an *extended* version of the denotational semantics presented in [8] by inclusion of the initial state.

Definition 5.4: Given a *denotational* context $\mathcal{T}_d = (PL, \{[\![C_g]\!] | g \in Ac\})$ for a signature Θ the denotational semantics of a program $BP = (I, \{(A_g, P_g, C_g) | g \in Ac\})$ for Θ is the durative transition system $\mathcal{D} = \langle \Sigma, \sigma_0, \{\mathcal{J}[\![(A_g, P_g, C_g)]\!] | g \in Ac\} \rangle$ where,

- $\sigma_0 \models_\Theta I$
- $\mathcal{J}[\![(A_g,P_g,C_g)]\!] = \{(\sigma,\sigma',\sigma'') | \sigma \models A_g,\ \sigma' \models P_g,\ \sigma'' = \sigma'[[\![C_g]\!](\sigma_{Lg})/D_g]\}$

$\mathcal{J}[\![(A_g,P_g,C_g)]\!]$ is the denotation of the *dga* (A_g,P_g,C_g). ♦

When the operational and denotational contexts for a given signature agree with each other as defined in 4.6, that is, the first *implements* the second and the second *abstracts* the first, the durative transitions defined operationally, that is, the triples of states $(\sigma,\sigma',\sigma'')$ where σ is the state the system is in when an agent is assigned an action, σ' is the state the system is in when the results of that action are accepted and σ'' is the resulting state, are the same as the ones obtained by the denotation of *dga*s.

Proposition 5.5: Given $\mathcal{T}_o = (PL, Pr, \{\Sigma_a | a \in Pr\}, \{(l_g^a, d_g^a, \xrightarrow{a}) | (a,g) \in Pr \times Ac\})$ and $\mathcal{T}_d = (PL, \{[\![C_g]\!] | g \in Ac\})$ – an operational and a denotational contexts for a signature $\Theta = (At, Ac)$ – which agree with each other, the *dts* defined by a behavioural program $BP = (I, \{(A_g, P_g, C_g) | g \in Ac\})$ is its denotation $\mathcal{D} = \langle \Sigma, \sigma_0, \{\mathcal{J}[\![(A_g,P_g,C_g)]\!] | g \in Ac\} \rangle$. ♦

6. Concluding remarks

The behavioural paradigm developed in [11,12] for supporting multiprocessor computing was presented as a coordination model for controlling the execution of durative actions, i.e. actions which, although executed atomically on a private local state, have a duration in the sense that the system state in which they finish executing is not necessarily the same in which they started.

In the proposed framework, each action has two associated guards. On the one hand, a pre-guard that characterizes the states of the system in which the action is desirable. As long as there are free processors, all actions that should be launched are launched regardless of any possible future conflict. Conflicts are resolved at acceptance time. This is achieved through a post-guard which characterises the system states in which an already "locally-executed" action is acceptable.

The interface between the global space of the system and the local space in which each action executes is made through two atomic operations of loading and discharging. When an action is launched, an atomic loading of the attributes that it needs to read is performed into the local state of the processor (agent) to which the action was assigned, and execution starts. When local execution ends, and if the post-guard of the action is satisfied by the system state, an atomic discharging of the effects computed locally is performed into the global state.

The proposed collection of primitives controls the interference between durative actions and clearly separates computation concerns from coordination ones as in [5]. The programming language in which computations are programmed is not constrained to any particular one: we rely on an interface that is provided by the loading and discharging operations. Pre- and post-guards stand for activation/permission criteria to coordinate the execution and productivity of actions.

Operational and denotational semantics were proposed for this paradigm which support, through context parameterisation, the separation between the coordination model and the computation model in which individual actions are programmed. The operational model formalises a multiprocessor environment through a collection of agents. The denotational semantics, which was proved to be equivalent to the operational one, is based on durative transition systems, an extension of labelled transitions systems proposed in [7,8] for capturing the notion of duration used in the behavioural paradigm. An axiomatic semantics is also available based on Durative Transition Logic, a modal logic introduced in [7] which is complete for durative transition systems. See [8] for further details.

Further work is under way for exploring the relationship between the behavioural paradigm and traditional coordination models in more detail, namely in what concerns multiset/ tuple space representations as well as the deductive approach to the operational semantics [9]. Applications to software architectures are also envisaged, for which we intend to integrate the behavioural formalisms in the categorical approach developed in [3].

References

1. J.P.Banâtre and D.Le Métayer, "Programming by Multiset Transformation", *Communications* ACM16, 1 pp. 55-77, 1993.
2. P.Ciancarini, C.Hankin, "Coordination Languages and Models", LNCS 1061, Springer-Verlag, 1996.
3. J.L.Fiadeiro and A.Lopes, "Semantics of Architectural Connectors", *Theory and Practice of Software Development*, M.Bidoit and M.Dauchet (eds), LNCS 1214, pp. 505-519, Springer-Verlag, 1997.
4. D.Gelernter, "Generative Communication in Linda", ACM *Trans. Prog. Lang. Syst.* 7, 1, pp. 80-112, 1985.
5. D.Gelernter, N.Carriero, "Coordination Languages and their Significance", *Communications* ACM 35, 2, pp. 97-107, 1992.
6. Z.Manna, A.Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag 1991.

7. I.Nunes, J.L.Fiadeiro and W.M.Turski, "A Modal Logic of Durative Actions", in *ICTL97*, Kluwer, in print, 1997.

8. I.Nunes, J.L.Fiadeiro and W.M.Turski, "Semantics of Behavioural Programs", Research Report, Department of Informatics, Faculty of Sciences, University of Lisbon, December 1996.

9. A.Porto, V.Vasconcelos, "Truth and Action Osmosis (the TAO Computational Model)", in J.M.Andreoli, C.Hankin and D.LeMetayer (eds), *Coordination Programming, mechanisms, models and semantics*, pp. 65-97, Imperial College Press, 1996.

10. C.Stirling, "Modal and Temporal Logics", in S.Abramsky, D.Gabbay and T.Maibaum (eds), *Handbook of Logic in Computer Science* 2, pp. 477-563, 1992.

11. W.M.Turski, "On Specification of Multiprocessor Computing", *Acta Informatica* 27, pp. 685-696, 1990.

12. W.M.Turski, "Extending the Computing Paradigm", *Structured Programming* 13, pp. 1-9, 1992.