DSLM : Dynamic Synchronous Language with Memory

Pejman Attar

▶ To cite this version:

Pejman Attar. DSLM : Dynamic Synchronous Language with Memory. 2012. hal-00779192v1

HAL Id: hal-00779192
https://hal.science/hal-00779192v1

Submitted on 21 Jan 2013 (v1), last revised 23 Feb 2013 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

DSLM : Dynamic Synchronous Language with Memory*

Pejman Attar
INRIA - INDES
Pejman.Attar@inria.fr

January 21, 2013

Abstract

We propose a new scripting language called `DSLM` based on the synchronous/reactive model. In `DSLM`, systems are composed of several sites executed asynchronously, and each site is running agents in a synchronous way. Each agent executes its script in synchronous parallel way. Scripts may call functions that are considered in an abstract way: their effect on the memory is not considered,

but only their "orchestration" i.e. the organisation of their calls in time and in place (the site where they are called). The mapping of sites onto cores allows one to benefit from multicore architectures. Two properties are assumed by DSL: reactivity of sites and absence of interferences between scripts run by distinct sites. We consider several variants of DSL. In the first variant, functions are defined in FunLoft. In the second variant of DSL, functions are defined in ReactiveML and the JoCaml system is used for asynchronous inter-sites communications. The third variant is based on SugarCubes which is a Java based framework for reactive programming. Finally, in the fourth variant, functions are defined in Scheme/Bigloo.

*with support from ANR-08-EMER-010, project PARTOUT

1 Introduction

Concurrency and parallelism are among the main problems of systems and programming languages. Up to now, issues concerning parallelism and concurrency were concentrated at the operating system level and left to experts of this domain.

Nowadays, multi-core machines are everywhere: in servers, PCs and even in mobile phones. These machines are widely used by the public. Concurrency problems are no more expert problems but they now also concern software programmers.

There exist several approaches to concurrent programming. In this paper, we are mainly considering the shared memory concurrency model [14] : the first and most used variant is called *preemptive multi-threading*. In this model, concurrent programs are system threads scheduled and preempted by the system in an arbitrary way. This model major problem is the freedom schedulers have in choosing the threads to be executed; this leads to so-called *time-dependent errors* which are generally considered as extremely difficult to tame [17].

Another model is *cooperative multi-threading* [9] [25]. In this model, the system loses the possibility to arbitrarily preempt threads. In order to be given to a new thread, the control must be explicitly released by the currently executing thread. Thanks to this, time-dependent errors do not occur anymore. However, the cooperative approach suffers from major drawback as a single thread can freeze the whole system if it never releases the control, preventing thus the system from giving it to the other threads.

The intrinsic difficulty of problems raised by concurrency and parallelism calls for *formal* techniques, and more specifically formal semantics. Among formal techniques, are *operational semantics* [18] which are usually classified in two categories: structural operational or *small-step* semantics, and natural or *big-step* semantics. Small-step semantics is close to program execution and describes each step of evaluation by an abstract interpretor. On the other hand, big-step semantics describes how the overall execution result is obtained, possibly using abstract means such as least fix-points of functionals. Small-step semantics are closer to implementation than big-step semantics, but more difficult to reason with.

Other models of concurrency and parallelism have also been proposed like the Actor model [3], Petri nets [22], Transactional memory [19], etc. Among these proposals, is the *synchronous approach*[2] that we now describe

by means of an example written in the Esterel language [4]. Let us consider a program made of two parallel statements, one awaiting for an event e, then producing event f, and the other producing event e. Such a statement is written in Esterel as:

```
P1 = await immediate e; emit f|| emit e
```

Due to the synchronous parallelism used in Esterel, the program $P1$ immediately emits both e and f, and this is the only possible outcome. Actually, there is a general demand of Esterel that all programs are deterministic: Esterel thus defines a synchronous and deterministic parallel operator. The small-step semantics of Esterel over program $P1$ considers the various possible interleavings of the two branches of the parallel statement, and let the control progress until both emissions of e and f are performed. On the contrary, the big-step semantics guesses that e and f are present, and then verifies that this is a coherent outcome. Actually, the small-step semantics of $P1$ takes several steps, while the big-step semantics takes only one step.

Deterministic synchronous parallelism has difficulties to cope with memory. Indeed, uncontrolled concurrent accesses to memory, as in:

```
P2 = x:=1 || x:=2
```

produces non-deterministic results: the outcome can be either $x = 1$ (if the second branche is executed first) or $x = 2$ otherwise. Moreover, non-determinism can result from non-atomic access to the memory. Thus deterministic parallel programming demands for means to get atomicity in memory access. Atomicity is usually provided by locks, in the context of preemptive multi-threading. Locks, however, are problematic as they can produce deadlock situations, when used, and time-dependent errors when programmers forget to use them [24].

In the synchronous model, memory is difficult to deal with [16], [6]. In Esterel, the solution is rather drastic: a variable cannot be read by one branch of a parallel statement and written by the other [5]. Thus, the previous program $P2$ would be rejected by the compiler. However, Esterel does not control concurrent accesses made at lower level by procedures and functions. Consider the following statement where two functions are called in parallel:

```
P3 = f1 () || f2 ()
```

The Esterel compiler is unable to verify that no concurrent access occurs through the calls of $f1$ and $f2$. Actually, one may think that the Esterel solution to avoid concurrent accesses to the memory is over-restrictive, specially in the context of multi-core programming in which access to the memory are the basic communication and synchronization means.

In this text, we propose a new model in which data races and time-dependent errors are eliminated by construction in the context of multi-cores architectures.

Structure of the text

The model is introduced in Section 2. A language based on the model is informally presented in Section 3. The domains in which the semantics is build are presented in Section 4. The semantics of scripts is given in Section 5. Sites and systems semantics are presented in the Section 6. A type system to verify safety of the language is given in Section 7. A refined semantics of the language is presented the section 8. Finally, related work and conclusion are given in Section 10 and 11.

2 Proposal

We propose a synchronous-based model which uses a deterministic parallel operator, and is able to deal with the memory in a safe way, without possibilities of time-dependent errors or data-races. We give our formalism a small-step semantics and introduce means to take benefit from multi-core architectures. Our model is called `DSLM` stand for *Dynamic Synchronous Language with Memory*.

Our model simplifies the programming of parallelism, compared to standard approaches. Simplification basically results from a simple and clean semantics due to the use of a synchronous and deterministic parallel operator. As in standard synchronous models, a notion of instant is present. Instants define a logical time, different from the physical time; an instant is terminated when all the parallel components have reached a synchronization barrier. Our proposal contains a solution to the issue of non-terminating instants which would prevent the system to reach the synchronization barrier (this problem is closely related to the freezing problem of the cooperative model).

In the synchronous model, events are a mean for communication. At each instant, an event is either absent, or present if it is produced. However, in *Esterel*, *causality cycle* can appear when no coherent solution can be found for the absence/presence status of an event. For example, consider:

```
P4 = present e else emit e end
```

There is a causality cycle in $P4$, as the status of e cannot be determined: if e is absent, then it is emitted, which is contradictory. However, if e is present, then it is not emitted, which is also a contradiction. Thus, $P4$ has no coherent solution in determining the status of e.

In addition to the possibilities of non-terminating instants, and of causality cycles, the standard synchronous model has difficulties in facing real parallelism and use of multi-core machines. Indeed, the use of multi-cores in the synchronous context implies the necessity to statically reject all kinds of time-dependent errors, and the possibility of dynamic creation (of events and parallel components) which is a extremely difficult task for standard synchronous languages.

Our model is based on the reactive variant [2] of the synchronous approach, which allows for dynamic creation and avoids causality cycles by construction (by prohibiting instantaneous reaction to absence of events). Moreover, there exists solutions in the reactive variant to insure by construction the termination of instants, even at the lower level of function and procedure calls (in the FunLoft [10] language).

Our proposal contains four main notions : scripts, events, agents and sites:

- Scripts : scripts are the basic parallel components. The syntax of scripts and their semantics is given in Section 3.

- Events: instantaneous broadcast of events is possible in our model. Values can be associated to events and event and their associated values are seen by all receivers in the same way. Events can be created dynamically during the execution.

- Agents: an agent encapsulates a script which can be a parallel script composed of several components (the components belonging to the agent) sharing the same agent memory. The only parallel components

ht

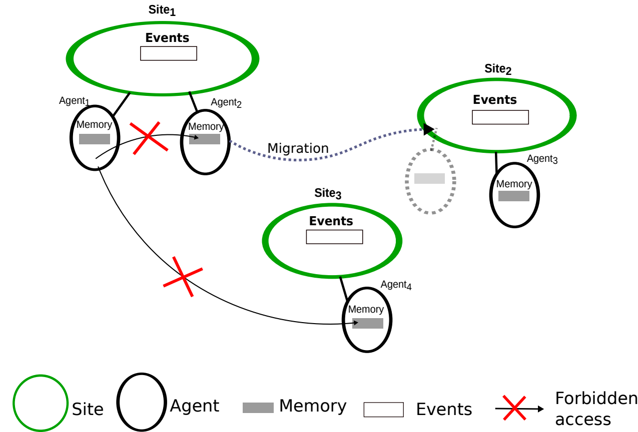

Figure 1: DSLM model

that can access the memory of an agent are the one belonging to the agent.

- Sites: a site is a location where execution of agents takes place. Each site runs one or more agents and manages a set of events shared by the site agents. There is no dynamic creation of sites. The model requires the uniqueness of site names. All the agents belonging to the same site are sharing the same instants and events. Agent can *migrate* from a site to another; the syntax is `migrate to` *site* for migrating the executing agent to site *site*. A *system* is composed of a set of sites which are run asynchronously and agent migration is the only communication means between sites.

`DSLM` does not provide any means to define functions. However, scripts may call functions defined in a "host" language (the language in which we define our model is called the host language). Function are required to teminate instantaneously (i.e. in the same instant it is started).And Modules execution can last several instants or even never terminate.

Execution of an agent consists in the execution of the agent's script in the context of the agent's memory. Execution of a site consists in the synchronous execution of all the agents belonging to the site, up to a state where they are all suspended or terminated. Then, end of the current instant is decided and

all the events which have not been generated during the instant are considered as absent. During site executions, the agents of the site can communicate and synchronize using the events they dynamically create.

Note that our model can be considered as a member of the GALS family (Globally asynchronous, Locally Synchronous) [23], as sites are executed asynchronously and agents in the same site are executed in a synchronous way.

In our model, scripts, agents, sites and systems receive a small-step semantics describe in Sections 5, 6.

3 Informal Language Description

This section contains the informal description of the language. First, we describe scripts and expressions. Then, a description of the model execution semantics is given.

3.1 Scripts and Expressions

Scripts are considered in a synchronous context and execution of a script at one instant has two possible outcomes:

- the script is terminated (nothing remains to be executed);
- the script is suspended: either it is waiting for an event to be generated, or it is waiting for the end of instant. In the first case execution will be continued during the current instant upto, either the generation of the event (which thus become present), or, the end of the current instant (and then the awaited event is considered as absent).

The informal description of scripts is as follows:

- `nothing` does nothing and terminates. In the semantics nothing is the only statement which is terminated and there is no rewriting rule that applies to it.
- $s_1; s_2$ runs scripts s_1 and s_2 in sequence; execution of s_2 immediately starts when s_1 is terminated;
- $x := e$ puts the values of e in the memory location of x, and terminates;

- $s_1 \nmid s_2$ runs the script s_1 and s_2 in parallel. The execution is terminated when both parallel branches are terminated. During each instant, both parallel branches are executed in a deterministic way: execution always starts with s_1 and switches to s_2 when s_1 is suspended. If necessary, execution returns to s_1 when s_2 becomes suspended, and this process continues until s_1 and s_2 are both terminated or suspended.

- `let` $x = e$ `in` s `end` defines a new variable x whose scope is s. A new location is associated to x, and the values of e is stored in this location. Then, the script behaves like s;

- `cooperate` suspend the execution for the current instant, waiting for the next instant. At the next instant the cooperate instruction is replaced by `nothing`;

- `generate` ev `with` e generates event ev with the value of the expression e and terminates;

- `await` ev has no effect and terminates if event ev is present. Otherwise, it suspends execution waiting for the event ev;

- `get_all` ev `in` l stores all the values associated to the generations of event ev during the current instant into the location l; execution is suspended upto the end of instant;

- `do` s `watching` ev executes the script s while event ev is not present. Execution of s is aborted as soon as ev is generated; In case of abortion, execution of the watching statement is suspended upto the end of instant;

- `repeat` e `do` s `end` runs the script s, n times in sequence, where n is the result of the evaluation of e;

- `loop` s `end` cyclically runs the script s: execution of s is restarted as soon as it terminates. However, if s terminates instantaneously (i.e. in the same instant it is started), the loop waits for the next instant to restart s. There is thus no possibility to get an *instantaneous loop* which would cycle for ever during the current instant, freezing the whole system;

- `launch` $m(e)$ launches the module m with the parameter e. Execution cannot terminate instantaneously. Execution may never terminates;
- `if` e `then` s_1 `else` s_2 `end` runs the script corresponding to the result of evaluation of the expression e (s_1 corresponds to true, and s_2 to false);
- `migrate to` *site* makes a request for moving the executing agent from the current site to the site *site*. Execution of the instruction is suspended for the current instant and resumes after the migration is effective on *site*. Due to parallelism, there can be several migration requests during the same instant (we call this *schizophrenia*). In this case, only the first request is considered and the other ones are ignored;
- `createAgent` s `in` *site* creates a new agent which encapsulate s with an empty memory. The agent is added to the list of agents requesting to be incorporated to *site* and its execution starts when it is effectively incorporated.

Expressions are the following:

- a basic value v;
- a vector of expressions $\vec{e}$;
- a variable x whose value is the location associated to x;
- $!x$ whose value is the content of the location associated to x;
- `ref` e which returns a new location where the value of e is stored;
- $f(e)$ which calls the function f with the value of e (it can be a vector) as parameter. Execution of the call starts immediately and is required to be instantaneous, i.e. terminates instantly.

3.2 Agents, Sites and Systems

Execution of an agent consists in the execution of the agent's script in the context of the agent's memory. Execution of an agent terminates when the agent's script is terminated.

Execution of a site consists in the synchronous execution of all the agents belonging to the site, up to a state where they are all suspended or terminated. During each instant, the agents of the site can communicate and synchronize using events.

The end of the current instant is decided at the site level when all the agents belonging to the site are suspended or terminated. Then, the followings actions are performed:

- the requests of migration to other sites are processed;
- the agents requesting to be incorporated in the site are actually added to the site;
- the events which have not been generated during the instant are considered as absent and the suspended scripts are reconstructed for the next instant (i.e. `cooperate` is change in `nothing`);
- finally, the site event set is reset to the empty set.

when these actions are performed, the execution of the site for the next instant can start.

A system is a statically defined set of sites, each of them with a distinct name. Execution of sites is completely asynchronous: sites are chosen and executed in a totally arbitrary way (even in real parallelism). Nothing is shared between different sites and the only means of communication is agent migration.

The semantics of the language (given in the next section) is separated in several levels which describes systems, sites, agents, and scripts. The semantics at a given level uses lower levels; for example, the semantics of an agent is based on script semantics.

Here are the main characteristics of the semantics:

(1) The semantics of agents and scripts is completely deterministic, even at the level of memory manipulations. We choose a fully deterministic parallel operator (*merge*) and a small-step semantics for that purpose. Moreover, to deal with the possiblity for a script to get suspended, we chose the smallest possible grain of parallelism; for example, a full step is devoted to the evaluation of the boolean expression of a test, while the execution of the chosen branch is left for a future step. In this way, the execution of the test can progress *even if the chosen branch is suspended.*

(2) The semantics of sites is confluent, at the event level. This results from two items: first, the synchronous execution of agents, and second, the memory encapsulation in agents. With these two elements, it becomes possible to get a confluent semantics for the sites: during one instant of a site, not two different event sets can be produced.

(3) The semantics of systems is completely non-deterministic which makes possible to model distribution as well as multi-core aspects.

4 Domains

In this section the domains used in this paper are given.

The following disjoint countable sets are defined: **LocName** (locations), **VarName** (variable names), **FunName** (function names), **ModuleName** (module names), **SiteName** (site names) and **EventName** (event names). Each set has an associated function which returns an unused element of the set (for example, each call of the function *new_loc* returns a new unused location in **LocName**).

We use the following notation to define domains:

- $A \times B$ denotes the cartesian product of the domain A and B;
- $A \oplus B$ denotes the disjoint union of the domain A and B;
- $\mathbb{N}^Z$ denotes the multi-set containing the elements of Z;
- $\uplus$ is the union of multi-sets;
- **None** is the domain that contains the unique distinguished element $None$. In the sequal, we does not distinguish between **None** and $None$;
- $\overrightarrow{A}$ denotes the domain of heterogeneous vectors of domain A.
- $A \rightarrow B$ is the domain of (partial) functions from A to B.

The set **Basic** is the set of basic values which, for simplicity, is defined as follows:

$$b \in \textbf{Basic} = \textbf{Bool} \oplus \textbf{Integer} \oplus \textbf{Double} \oplus \textbf{String}$$

The set **Value** which contains basic values, locations, and vectors of values is defined as:

$$v \in \mathbf{Value} = \mathbf{Basic} \oplus \mathbf{LocName} \oplus \overrightarrow{\mathbf{Value}}$$

4.1 Memory

A memory M belonging to **Mem** is a partial function that associates a value with its location or variable.

$$M \in \mathbf{Mem} : \mathbf{VarName} \oplus \mathbf{LocName} \rightarrow \mathbf{Value}$$

One notes $M[l \leftarrow v]$ the memory M' defined by: $M'(l) = v$ and for $x \neq l$, $M'(x) = M(x)$. If $M(x)$ is a location, $M[x \leftarrow v]$ is an abbreviation for $M[M(x) \leftarrow v]$.

4.2 Events

Elements of **EventEnv** are multi-sets of pairs composed of an event name and an associated basic value:

$$E \in \mathbf{EventEnv} : \mathbb{N}^{(\mathbf{EventName} \times \mathbf{Basic})}$$

To simplify notation one notes $ev \in E$, if there exist v such that $(ev, v) \in E$.

The function *get_values* is used to collect all the basic values associated with an event ev in a set of events E:

$$get_values : \mathbf{EventName} \times \mathbf{EventEnv} \rightarrow \overrightarrow{\mathbf{Basic}}$$

4.3 Expressions

The set **Expr** denotes the expressions and is defined by the following grammar:

$$\begin{array}{llll} e \in \mathbf{Expr} ::= & v & \mid x & \mid\, !\,x \\ & \mid \vec{e} & \mid \texttt{ref}\ e & \mid f(\vec{e}) \end{array}$$

4.4 Scripts

The set **Script** denotes the scripts and is defined by:

$$
\begin{array}{rl}
s \in \textbf{Script} ::= & \texttt{nothing} \\
 & \mid s; s \\
 & \mid x := e \\
 & \mid s \nmid s \\
 & \mid \texttt{let}\, x = e\, \texttt{in}\, s\, \texttt{end} \\
 & \mid \texttt{cooperate} \\
 & \mid \texttt{generate}\, ev\, \texttt{with}\, e \\
 & \mid \texttt{await}\, ev \\
 & \mid \texttt{get_all}\, ev\, \texttt{in}\, l \\
 & \mid \texttt{do}\, s\, \texttt{watching}\, ev \\
 & \mid \texttt{repeat}\, e\, \texttt{do}\, s\, \texttt{end} \\
 & \mid \texttt{loop}\, s\, \texttt{end} \\
 & \mid \texttt{launch}\, m(e) \\
 & \mid \texttt{if}\, e\, \texttt{then}\, s\, \texttt{else}\, s\, \texttt{end} \\
 & \mid \texttt{migrate to}\, site \\
 & \mid \texttt{createAgent}\, s\, \texttt{in}\, site
\end{array}
$$

4.5 Agents

The set **Agent** denotes the agents which are triples of the form (s, M, η), where $s \in \textbf{Script}$, $M \in \textbf{Mem}$, and η is a migration request:

$$Ag \in \textbf{Agent} = \textbf{Script} \times \textbf{Mem} \times \textbf{Migr}$$

$$\eta \in \textbf{Migr} = \textbf{None} \oplus \textbf{SiteName}$$

Drop orders code agent migration requests. A drop order can be either the **None** value to indicate the absence of migration request, or a demand for migration of the current agent, or a demand for migration of a newly created agent. The set **D** of drop orders is defined as:

$$d \in \textbf{D} = \textbf{None} \oplus \textbf{SiteName} \oplus (\textbf{Agent} \times \textbf{SiteName})$$

A drop order $site \in \textbf{SiteName}$ is a demand for the migration of the current agent to *site*. A drop order $(Ag, site) \in \textbf{Agent} \times \textbf{SiteName}$ is a demand for the migration to *site* of the newly created agent Ag.

We define the combination of two migration requests:

$$\blacktriangleright: \mathbf{Migr} \times \mathbf{SiteName} \rightarrow \mathbf{SiteName}$$

$$\blacktriangleright (\eta, site) = \begin{cases} site & if\, \eta = None \\ site_\eta & if\, \eta = site_\eta \end{cases}$$

4.6 Sites

The set **Site** denotes the sites which are quadruples of the form:

$$S \in \mathbf{Site} = \mathbf{SiteName} \times \mathbb{N}^{\mathbf{Agent}} \times \mathbb{N}^{\mathbf{Agent}} \times \mathbf{EventEnv}$$

The site $(site, \mathcal{A}, \mathcal{I}, E)$ is interpreted as follows:

- *site* is the name of the site;
- $\mathcal{A}$ is the multi-set of the agents running in the site;
- $\mathcal{I}$ is the multi-set of agents which will be incorporated in the site at the next instant.
- E is the multi-set of the events generated in the site.

Let $S = (site, \mathcal{A}, \mathcal{I}, E)$ be a site; one notes $sn(S) \subseteq \mathbf{SiteName}$ the set of site names occurring in $\mathcal{A}$.

4.7 Systems

The set $\Sigma \in \mathbf{Sys}$ denotes the systems which are sets of sites:

$$\Sigma = \{S_1, ..., S_n\}$$

One says that a system $\Sigma = \{S_1, ..., S_n\}$ where $S_i = (site_i, \mathcal{A}_i, \mathcal{I}_i, E_i)$ is *well-formed* if the following two requirements are fulfilled:

1. No two sites have the same name:
   $\forall i, j \in \{1, \ldots, n\} : \quad i \neq j \;\Rightarrow\; site_i \neq site_j$;
2. The target of a migration is always defined:
   $\forall i \in \{1, \ldots, n\} : \quad site \in sn(S_i) \;\Rightarrow\; \exists j \,.\, site = site_j$.

4.8 Reconstruction Function

A reconstruction function Ω (Section 6.3) is used to prepare an agent for the next instant. Its domain is:

$$\Omega : \mathbf{Agent} \times \mathbf{EventEnv} \rightarrow \mathbf{Agent}$$

5 Semantics of Scripts

This section presents the semantics of scripts. First, in the Section 5.1 the semantics of expressions is defined. Then, the suspensions predicate for scripts is presented in Section 5.2. Finally, the transition relation which defines the small-step semantics of scripts is defined in Section 5.3.

5.1 Expressions

The evaluation of an expression is noted:

$$e, M \rightsquigarrow v, M' \tag{1}$$

where:

- e is the initial expression;
- M is the memory in which the expression e is evaluated;
- v is the result of the evaluation of e;
- M' is the new memory after the evaluation of e;

Evaluation of expressions is defined by the following rules:

- A value evaluates in itself:

$$v, M \rightsquigarrow v, M \tag{2}$$

- To access a variable, the variable must denote a location; the evaluation returns the value stored in it:

$$!\,x, M \rightsquigarrow M(x), M \tag{3}$$

- The elements of a vector of expressions are evaluated in increasing order:

$$\frac{e_i, M_i \rightsquigarrow v_i, M_{i+1}}{(e_1, ..., e_n), M_1 \rightsquigarrow (v_1, ..., v_n), M_{n+1}} \tag{4}$$

- Evaluation of `ref` e returns a new location in which the value of e is stored:

$$\frac{e, M \rightsquigarrow v, M' \quad l = new_loc()}{\texttt{ref}\ e, M \rightsquigarrow l, M'[l \leftarrow v]} \tag{5}$$

- Evaluation of a function call should be instantaneous. The only changes in the memory are the ones resulting from the evaluation of the arguments:

$$\frac{\overrightarrow{e}, M \rightsquigarrow \overrightarrow{v}, M' \quad f(\overrightarrow{v}) = v'}{f(\overrightarrow{e}), M \rightsquigarrow v', M'} \tag{6}$$

5.2 Suspension Predicate

Reactive programs suspend execution either waiting for signals which are not already produced, or waiting for the end of current instant. The *suspension predicate* of scripts is noted ‡ and one writes $\langle s, E\rangle\ddagger$ to indicate that script s is suspended in the environment E. ‡ is defined inductively by the following rules:

- A cooperate instruction is suspended in all environments:

$$\langle cooperate, E\rangle\ddagger \tag{7}$$

- A get all instruction is suspended in all environments:

$$\langle \texttt{get_all}\; ev \;\texttt{in}\; l, E\rangle\ddagger \tag{8}$$

- An await instruction is suspended when the awaited event is not present:

$$\frac{ev \notin E}{\langle \texttt{await}\; ev, E\rangle\ddagger} \tag{9}$$

- A watching statement is suspended if its body is suspended:

$$\frac{\langle s, E\rangle\ddagger}{\langle \texttt{do}\; s \;\texttt{watching}\; ev, E\rangle\ddagger} \tag{10}$$

- A sequence is suspended if its first branch is suspended (the second branch is not considered):

$$\frac{\langle s_1, E\rangle\ddagger}{\langle s_1; s_2, E\rangle\ddagger} \tag{11}$$

- A parallel statement is suspended if the first branch is suspended and the second one is either suspended or terminated:

$$\frac{\langle s_1, E\rangle \ddagger \qquad \langle s_2, E\rangle \ddagger \;\; or \;\; s_2 = \texttt{nothing}}{\langle s_1 \nmid s_2, E\rangle \ddagger}$$

The suspension predicate of scripts is naturally extended to agents:

$$\frac{\langle s, E\rangle \ddagger}{\langle (s, M, \eta), E\rangle \ddagger} \tag{12}$$

A site $S = (site, \mathcal{A}, \mathcal{I}, E)$ is suspended if all its agents are suspended or terminated (an agent is terminated if its script is):

$$\frac{\forall Ag \in \mathcal{A} \quad \langle Ag, E\rangle \ddagger \quad \vee \quad Ag = (\texttt{nothing}, M, \eta)}{(site, \mathcal{A}, \mathcal{I}, E) \ddagger}$$

The predicate $\natural$ indicates the absence of migration request in a site:

$$\frac{\forall (s, M, \eta) \in \mathcal{A} \quad \eta = None}{(site, \mathcal{A}, \mathcal{I}, E)\natural}$$

Rule 35 describes the detection of the end of current instant of a site, which is only possible when the two previous predicates are valid.

5.3 Transition Relation

The small-step semantics of scripts is presented as a set of rewriting rules. Rules represent transitions (execution) of scripts. The general format of a script transition is:

$$\langle s,\, E,\, M\rangle \xrightarrow{d} \langle s',\, E',\, M'\rangle$$

where:

- s is the script which is rewritten;

- E is a multi-set of pairs (ev, v) where ev is a generated event and v is an associated value.
- s' is the *residual* script (what remains to be done at the next step);
- E' is the multi-set of events generated during the rewriting of script s, coupled with their values;
- M is the memory in which s is rewritten;
- M' is the new memory obtained after the rewriting of s;
- d is a drop order indicating if a migration request has been issued from the rewriting of s, and if it is the case, the nature of the request.

The semantics of instructions is given by the following rules:

- The semantics of a sequence considers the case where the first branch is terminated, and the case where it is not. There is no ambiguity in the choice of the rule to apply because no rewriting rule is applicable to nothing. The first rule considers the case where the first branch is different from nothing: it is executed and the instruction rewrites in a new sequence:

$$\frac{\langle s_1,\, E,\, M\rangle \xrightarrow{d} \langle s_1',\, E',\, M'\rangle}{\langle s_1; s_2,\, E,\, M\rangle \xrightarrow{d} \langle s_1'; s_2,\, E',\, M'\rangle} \tag{13}$$

  The control passes to the second branch when the first is terminated; this corresponds to the following rule:

$$\langle \texttt{nothing}; s_2,\, E,\, M\rangle \xrightarrow{\textbf{None}} \langle s_2,\, E,\, M\rangle \tag{14}$$

- Execution of a parallel instruction always starts by execution of the left branch:

$$\frac{\langle s_1,\ E,\ M\rangle \xrightarrow{d} \langle s_1',\ E',\ M'\rangle}{\langle s_1 \nmid s_2,\ E,\ M\rangle \xrightarrow{d} \langle s_1' \nmid s_2,\ E',\ M'\rangle} \tag{15}$$

A parallel instruction rewrites its right branch when the left branch is suspended:

$$\frac{\langle s_1, E\rangle \ddagger \quad \langle s_2,\ E,\ M\rangle \xrightarrow{d} \langle s_2',\ E',\ M'\rangle}{\langle s_1 \nmid s_2,\ E,\ M\rangle \xrightarrow{d} \langle s_1 \nmid s_2',\ E',\ M'\rangle} \tag{16}$$

Execution of a parallel switches to the right branch when the left branch is terminated:

$$\langle \texttt{nothing} \nmid s,\ E,\ M\rangle \xrightarrow{\textbf{None}} \langle s,\ E,\ M\rangle \tag{17}$$

- A let instruction declares a variable x in the scope of a script s; the initial value of x is obtained by evaluating an expression e:

$$\frac{e, M \rightsquigarrow v, M'}{\langle \texttt{let}\ x = e\ \texttt{in}\ s\ \texttt{end},\ E,\ M\rangle \xrightarrow{\textbf{None}} \langle s,\ E,\ M'[x \leftarrow v]\rangle} \tag{18}$$

- An assignment puts a new value in the memory location associated to a variable. The type system of Section 7 insures that the value of the variable is always a location.

$$\frac{e, M \rightsquigarrow v, M'}{\langle x := e,\ E,\ M\rangle \xrightarrow{\textbf{None}} \langle \texttt{nothing},\ E,\ M'[x \leftarrow v]\rangle} \tag{19}$$

- A loop statement executes its body cyclically. The body is restarted as soon as it terminates. A loop instruction never terminates. A cooperate statement is systematically added in parallel to the body to avoid instantaneous loops:

$$\frac{\langle s \nmid \texttt{cooperate},\, E,\, M\rangle \xrightarrow{d} \langle s',\, E',\, M'\rangle}{\langle \texttt{loop}\, s\, \texttt{end},\, E,\, M\rangle \xrightarrow{d} \langle s'; \texttt{loop}\, s\, \texttt{end},\, E',\, M'\rangle} \tag{20}$$

- A generate instruction produces an event in the environment and associates a value obtained from the evaluation of an expression to this production:

$$\frac{e, M \rightsquigarrow v, M' \quad E' = E \uplus \{(ev, v)\}}{\langle \texttt{generate}\, ev\, \texttt{with}\, e,\, E,\, M\rangle \xrightarrow{\textbf{None}} \langle \texttt{nothing},\, E',\, M'\rangle} \tag{21}$$

  The pair made of the event and its value is added in the event environment which is a multi-set. Thus, several productions of the same event with the same value are possible during a same instant.

- An await instruction terminates if the awaited event is present in the environment:

$$\frac{ev \in E}{\langle \texttt{await}\, ev,\, E,\, M\rangle \xrightarrow{\textbf{None}} \langle \texttt{nothing},\, E,\, M\rangle} \tag{22}$$

  There is no rule corresponding to an event which is not present. In this case the instruction is suspended: $\langle \texttt{await}\, ev, E\rangle\ddagger$.

- A watching statement executes its body, if not terminated, and rewrites in a new watching statement:

$$\frac{\langle s,\, E,\, M\rangle \xrightarrow{d} \langle s',\, E',\, M'\rangle}{\langle \texttt{do}\, s\, \texttt{watching}\, ev,\, E,\, M\rangle \xrightarrow{d} \langle \texttt{do}\, s'\, \texttt{watching}\, ev,\, E',\, M'\rangle} \tag{23}$$

If the body is terminated (i.e. it is nothing), then the watching statement rewrites in a cooperate instruction (thus, it suspends until the end of the current instant):

$$\langle \texttt{do nothing watching}\, ev,\ E,\ M\rangle \xrightarrow{\textbf{None}} \langle \texttt{cooperate},\ E,\ M\rangle \tag{24}$$

- A module call launches a new separate thread run by the operating system to execute a specific module. Execution of a module up to termination must take several instants (at least one). The thread executing a module is supposed to generate a termination event when the module terminates:

$$\frac{\overrightarrow{e}, M \rightsquigarrow \overrightarrow{v}, M' \quad ev = new_event() \quad m(\overrightarrow{v}, ev) \Uparrow}{\langle \texttt{launch}\ m\,(\overrightarrow{e}),\ E,\ M\rangle \xrightarrow{\textbf{None}} \langle \texttt{await}\ ev,\ E,\ M'\rangle} \tag{25}$$

The notation $m(\overrightarrow{v}, ev) \Uparrow$ means that the module m is launched in a separate thread. Event ev is generated to signal the end of the thread. The difference between module call and function call is that execution of a module can take several instants or even never terminate, while a function call should always be instantaneous. A second difference is that a function call returns a value, while a module call does not.

- A repeat statement executes its body s in sequence n times, where n is the integer obtained by evaluating e:

$$\frac{e, M \rightsquigarrow n, M'}{\langle \texttt{repeat}\ e\ \texttt{do}\ s\ \texttt{end},\ E,\ M\rangle \xrightarrow{\textbf{None}} \langle \overbrace{s; \ldots; s}^{n\ times},\ E,\ M'\rangle} \tag{26}$$

- A conditional evaluates its boolean test and rewrites in the appropriate branch:

$$\frac{e, M \rightsquigarrow tt, M'}{\langle \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end},\ E,\ M\rangle \xrightarrow{\textbf{None}} \langle s_1,\ E,\ M'\rangle} \tag{27}$$

$$\frac{e, M \rightsquigarrow f\!f, M'}{\langle \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end},\ E,\ M\rangle \xrightarrow{\textbf{None}} \langle s_2,\ E,\ M'\rangle} \tag{28}$$

The fact that the rewriting of the chosen branch is delayed to a future execution step is an essential feature of the small-step semantics of scripts, as previously discussed in Section 3.2.

- An agent creation produces a drop order $Ag \downarrow site$ to demand the migration in *site* of a new agent Ag containing a script s and a new empty memory $\emptyset$:

$$\begin{array}{l}\langle \texttt{createAgent}\ s\ \texttt{in}\ site,\ E,\ M\rangle \\ \qquad \xrightarrow{(s,\emptyset,\textbf{None})\downarrow site} \langle \texttt{nothing}.\ E,\ M\rangle\end{array} \tag{29}$$

The absorbtion of the newly created agent by the system is described in Rule (32).

- A migration instruction produces a drop order *site* to demand the migration in *site* of the executing agent, and suspends up to the end of the current instant:

$$\langle \texttt{migrate to}\ site,\ E,\ M\rangle \xrightarrow{site} \langle \texttt{cooperate},\ E,\ M\rangle \tag{30}$$

The processing of the migration request is described in Rule (34) .

Theorem: For any script s, event environment E and memory M, there is only one possible execution.

Proof: To proof the determinism of execution of all the scripts we should verify all the possible script execution. Between all the scripts only the parallel ($\nmid$), watching and conditional have several possible execution.

For parallel ($\nmid$), there are three possible execution. But none of them can be executed in the same time and create a non-deterministic execution. between Rule (15) and Rule (16) the non-determinism is not possible since if script s_1 is suspended it cannot be executed. Neither, between (15) and Rule (17) since there is no execution rule for `nothing` instruction. Finally, between Rule (16) and Rule (17), there is no possible non-deterministic execution: `nothing` is not a suspended instruction.

For the watching statement the execution is deterministic since there is no possible execution for `nothing`, so there cannot be any interaction between Rule (23) and Rule (24).

At last, for conditional, an expression cannot be in the same time true and false, since there is only one possible execution for expressions.

6 Semantics of Sites and Systems

The small-step semantics of sites and systems is given in this section. The micro-step execution of a site during one instant is described by the first three rules (31)-(33). The next two rules deals with end of instants. Migration requests are processed in Rule (34). The passing to the next instant is described by Rule (35). Finally, the transformation of the suspended terms for the next instant is described in Section 6.3

The format for system rewritting is:

$$\Sigma \rightarrow \Sigma'$$

where Σ and Σ' are systems.

6.1 Sites

There are three rules for defining system rewriting. These rules describe the choice of a site S, the choice of an agent Ag in S, and the execution of Ag in the event environment of S.

- The first rule considers the case where no drop order is issued from the agent execution:

$$
\frac{\begin{array}{c} S = (site, \mathcal{A} \uplus (s, M, \eta), \mathcal{I}, E) \\ \langle s,\, E,\, M\rangle \xrightarrow{\textbf{None}} \langle s',\, E',\, M'\rangle \end{array}}{\Sigma[S] \quad \rightarrow \quad \Sigma[(site, \mathcal{A} \uplus (s', M', \eta), \mathcal{I}, E')]} \tag{31}
$$

After execution, the agent is reintegrated in the site and the site event environment is replaced by the produced event set.

- The second rule corresponds to the production of the drop order of a new agent Ag in the target site. Agent Ag is put in the set of agents requesting to be incorporated into $site_0$:

$$
\frac{\begin{array}{c} S = (site, \mathcal{A} \cup (s, M, \eta), \mathcal{I}, E) \\ \langle s,\, E,\, M\rangle \xrightarrow{Ag \downarrow site_0} \langle s',\, E',\, M'\rangle \\ S' = (site_0, \mathcal{A}_0, \mathcal{I}_0 \cup Ag, E_0) \\ S'' = (site, \mathcal{A} \cup (s', M', \eta), \mathcal{I}, E') \end{array}}{\begin{array}{l} \Sigma[(site_0, \mathcal{A}_0, \mathcal{I}_0, E_0)][S] \\ \qquad\qquad\qquad \rightarrow \Sigma[S'][S''] \end{array}} \tag{32}
$$

- The third rule corresponds to the production of a migration request $site_0$ for the current agent. There are two cases: either a migration request is already present in the agent, and then $site_0$ is simply ignored (a way to prevent schizophrenia); or, there is no previous migration request in the agent, and then $site_0$ becomes the agent migration request:

$$
\frac{\begin{array}{c} S = (site, \mathcal{A} \cup (s, M, \eta), \mathcal{I}, E) \\ \langle s,\, E,\, M\rangle \xrightarrow{site_0} \langle s',\, E',\, M'\rangle \end{array}}{\Sigma[S] \quad \rightarrow \quad \Sigma[(site, \mathcal{A} \cup (s', M', \eta \blacktriangleright site_0), \mathcal{I}, E')]} \tag{33}
$$

6.2 End of Instants

The end of the current instant of a site is decided when the site is suspended, that is when all its agents are suspended. In this case, the site decides the end of the current instant, and can start the next instant.

Two rules are needed to process suspended sites. The first one considers migration requests and the second prepares the site for the next instant. In both cases, suspended agents are transformed to take in acount absent events and the passing to the next instant. These two transformations are defined using the function Ω described in Section 6.3.

- The first rule considers the case where an agent Ag_1 of site $site_1$ requests to migrate to site $site_2$. First, suspended instructions of Ag_1 are processed in order to take in account the end of current instant (function Ω); then, the transformed agent is added to the set of agents requesting to be incorporated in $site_2$:

$$\frac{\begin{array}{c} S_1 \ddagger \\ S_1 = (site_1, \mathcal{A}_1 \uplus (s, M, site_2), \mathcal{I}_1, E_1) \\ S_2 = (site_2, \mathcal{A}_2, \mathcal{I}_2, E_2) \\ S_1' = (site_1, \mathcal{A}_1, \mathcal{I}_1, E_1) \\ S_2' = (site_2, \mathcal{A}_2 \uplus \Omega\,(s, M), \mathcal{I}_2, E_2) \end{array}}{\Sigma[S_1][S_2] \quad \rightarrow \quad \Sigma[S_1'][S_2']} \tag{34}$$

- The second rule considers the case where there is no migration request. In this case, suspended instructions are processed in order to take in account the end of current instant, and the agents requesting to be incorporated in the site are added to the agent set. Moreover, the site event environment is reset ($E = \emptyset$):

$$\frac{S \ddagger, S \natural \quad S = (site, \mathcal{A}, \mathcal{I}, E)}{\Sigma[S] \quad \hookrightarrow \quad \Sigma[(site, \Omega\,(\mathcal{A}, E) \cup \mathcal{I}, \emptyset, \emptyset)]} \tag{35}$$

In the rule, $\Omega\,(\mathcal{A}, E)$ means:

$$\Omega\,(\mathcal{A}, E) = \{\forall Ag \in \mathcal{A} \quad | \quad \Omega\,(Ag, E)\}$$

6.3 Reconstruction for Next Instant

The resconstruction function Ω is used at each end of instant in order to reconstruct suspended agents, with regard to an event environment E, and to prepare them for execution at the next instant. To reconstruct an agent means to transform its script and this reconstruction can possibly modify the agent's memory. The Ω function is inductively defined in Figure 2. There are 4 basic cases for script reconstruction:

$$\Omega\left((\texttt{cooperate}, M, \eta), E\right) = (\texttt{nothing}, M, \eta)$$

$$\Omega\left((\texttt{get_all}\, ev\, \texttt{in}\, l, M, \eta), E\right) = \\ (\texttt{nothing}, M[l \leftarrow get_values(ev, E)], \eta)$$

$$\frac{ev \in E}{\Omega\left((\texttt{do}\, s\, \texttt{watching}\, ev, M, E), E\right) = (\texttt{nothing}, M, \eta)}$$

$$\frac{ev \notin E \quad \Omega\left((s, M, \eta), E\right) = (s', M', \eta)}{\Omega\left((\texttt{do}\, s\, \texttt{watching}\, ev, M, \eta), E\right) = (\texttt{do}\, s'\, \texttt{watching}\, ev, M', \eta)}$$

$$\Omega\left((\texttt{await}\, ev, M, \eta), E\right) = (\texttt{await}\, ev, M, \eta)$$

$$\frac{\Omega\left((s_1, M, \eta), E\right) = (s_1', M_1, \eta)}{\Omega\left((s_1; s_2, M, \eta), E\right) = (s_1'; s_2', M_2, \eta)}$$

$$\frac{\Omega\left((s_1, M, \eta), E\right) = (s_1', M_1, \eta) \quad \Omega\left((s_2, M_1, \eta), E\right) = (s_2', M_2, \eta)}{\Omega\left((s_1 \dagger s_2, M, \eta), E\right) = (s_1' \dagger s_2', M_2, \eta)}$$

Figure 2: Ω function

- `cooperate` is reconstructed in `nothing`;
- `do` s `watching` ev is reconstructed in `nothing` if ev is present in E; otherwise, ($ev \notin E$), the instruction is reconstructed in `do` s' `watching` ev where s' is the reconstruction of s in E.
- `await` ev is reconstructed in itself;
- `get_all` ev `in` l is reconstructed in `nothing`; moreover, the values associated with ev in E are collected in a list which is assigned to l.

Note that this is the only reconstruction step that possibly modifies the agent memory.

7 Typing system

The purpose of the proposed type system is twofold: first, to insure that values are correctly used; this is traditional type checking, to verify for instance that in `if` e `then` s_1 `else` s_2 `end`, e is a boolean expression. Second, that no data-race occurs. For example, consider the following fragment:

```
let x = ref e1 in
    createAgent
       !x
    in remote;
    x := e2;
end
```

There is a data-race as `x` is read by an agent belonging to site `remote`, while it is written in the current site. To prevent this kind of errors, the type system checks that a reference not belonging to an agent memory cannot be accessed by the agent.

A *type* is either the name of a basic type (*int, bool,* etc), or a references on a *type*:

$$Basic ::= bool \mid unit \mid int \mid String$$

$$\tau ::= Basic \mid ref\ \tau \mid \overrightarrow{\tau}$$

A typing environment Γ is a possibly empty set[1] of elements of the form $x : \tau$, where x is a variable and τ is a type:

$$\Gamma ::= x_1 : \tau_1, \cdots, x_n : \tau_n$$

The general form of typing rules is:

[1] In the sequel, the brackets of the standard set notation are omitted.

$$\Gamma \vdash s : \tau \tag{36}$$

where :

- Γ is the typing environment;
- s is the script to be typed;
- τ is the type of s.

7.1 Typing Rules

To type a variable x, its type should be specified in the environment:

$$\Gamma \cup \{x : \tau\} \vdash x : \tau \tag{37}$$

A value has a unique type τ:

$$\Gamma \vdash v : \tau \tag{38}$$

To type a let statement defining a variable x as e in s, we should first type the expression e by τ_1; then, the script s should be typed in the new environment in which x has type τ_1:

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \cup x\!: \tau_1 \vdash s : \tau_2}{\Gamma \vdash \texttt{let}\, x = e\, \texttt{in}\, s\, \texttt{end} : \tau_2} \tag{39}$$

We suppose that the type of any function is known by the type system. The type of a function contains the type of the function's arguments and the type of the result. To type a function call, the arguments are type checked and the call receives the type of the function result:

$$\frac{\Gamma \vdash \vec{e} : \vec{\tau} \quad f :\ \vec{\tau} \rightarrow \tau'}{\Gamma \vdash f(\vec{e}) : \tau'} \tag{40}$$

The type of an accessed variable must be a reference type present in the typing environment:

$$\frac{\Gamma \vdash x : \texttt{ref}\,\tau}{\Gamma \vdash !x : \tau} \tag{41}$$

To type a sequence, both branch must be typed:

$$\frac{\Gamma \vdash s_1 : \tau_1 \quad \Gamma \vdash s_2 : \tau_2}{\Gamma \vdash s_1 \;;\; s_2 : \texttt{unit}} \tag{42}$$

To type a conditional, the condition should be typed to a boolean, then the two branches s_1 and s_2 should be typed:

$$\frac{\Gamma \vdash e : \mathit{bool} \quad \Gamma \vdash s_1 : \tau \quad \Gamma \vdash s_2 : \tau}{\Gamma \vdash \texttt{if}\; e\; \texttt{then}\; s_1\; \texttt{else}\; s_2\; \texttt{end} : \tau} \tag{43}$$

To type a `repeat` e `do` s `end` statement, the expression e should be typed as an integer, before typing s:

$$\frac{\Gamma \vdash e : \texttt{int} \quad \Gamma \vdash s : \tau}{\Gamma \vdash \texttt{repeat}\; e\; \texttt{do}\; s : \texttt{unit}} \tag{44}$$

To type a loop, one types its body:

$$\frac{\Gamma \vdash s : \tau}{\Gamma \vdash \texttt{loop}\; s : \texttt{unit}} \tag{45}$$

To type a parallel statement, one types both branches in the same environment:

$$\frac{\Gamma \vdash s_1 : \tau_1 \quad \Gamma \vdash s_2 : \tau_2}{\Gamma \vdash s_1 \nmid s_2 : \texttt{unit}} \tag{46}$$

We suppose that the type of any module is known by the type system. The type of a module contains the type of the module's arguments and the type of the result is always `unit`. To type a module call, the arguments are type checked and the call is typed as `unit`:

$$\frac{\Gamma \vdash \vec{e} : \vec{\tau} \quad \vec{e} = (e_1, ..., e_n) \quad m :\ \vec{\tau} \rightarrow \texttt{unit}}{\Gamma \vdash \texttt{launch}\ m\,(\overrightarrow{e}) : \texttt{unit}} \tag{47}$$

The cooperate statement is simply typed in `unit`:

$$\Gamma \vdash \texttt{cooperate} : \texttt{unit} \tag{48}$$

To type a generate statement, the associated value should be of a basic type, and the type of the statement is `unit`:

$$\frac{\Gamma \vdash e : Basic}{\Gamma \vdash \texttt{generate}\ ev\ \texttt{with}\ e : \texttt{unit}} \tag{49}$$

An await statement is simply typed in `unit`:

$$\Gamma \vdash \texttt{await}\ ev : \texttt{unit} \tag{50}$$

To type a watching statement means to type its body:

$$\frac{\Gamma \vdash s : \tau}{\Gamma \vdash \texttt{do}\ s\ \texttt{watching}\ ev : \texttt{unit}} \tag{51}$$

A *get_all* statement is simply typed in `unit`:

$$\Gamma \vdash \texttt{get_all}\ ev\ \texttt{in}\ l : \texttt{unit} \tag{52}$$

To type an agent creation one should type its body in an empty typing environment:

$$\frac{\emptyset \vdash s : \tau}{\Gamma \vdash \texttt{createAgent}\ s\ \texttt{in}\ site : \texttt{unit}} \tag{53}$$

This is the central rule to prevents the possibility of data-races in the language.

The migration of the current agent to a site is typed in `unit`:

$$\Gamma \vdash \texttt{migrate}\ \texttt{to}\ site : \texttt{unit} \tag{54}$$

The following theorem can be proved from the previous definition of the type system:

Theorem: If a program P is well-typed, then no data-race can occur during its execution.

Proof: To have a data-race in a program we need to access to another agent memory during the execution. To access to another agent memory, it should be able to access to one of its memory location. If an agent has access to another location than its memory and want to access to this location, it should use $!\,x$. A well-typed program cannot have $!\,x$ alone in its core, since each time we are creating an agent we are reseting the typing enivronement. This program cannot be typed.

8 Refined Semantics

In this section, we consider the context of multi-processor/multi-core architectures. Our main goal is to give the system the possibility to maximize the usage of computing resources (processors or cores).

We propose to define in the model a new level of parallelism in which the agents are mapped to parallel components called *schedulers*. The sites are composed of several schedulers sharing the same instants and the same event and are called *synchronized schedulers*.

ht

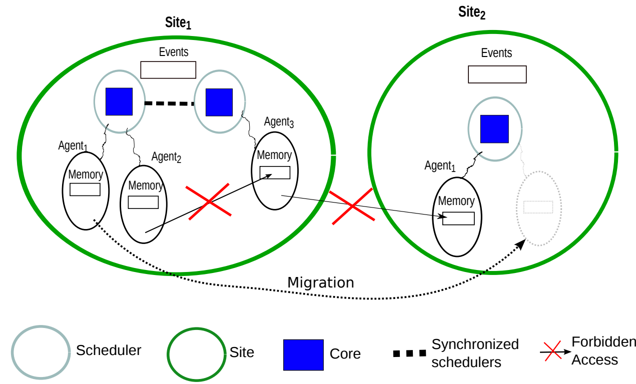

Figure 3: DSLM: Synchronized schedulers

At implementation level, the guess is that each scheduler is executed by a distinct thread (for example, in a Linux-SMP architecture), or by a distinct processor (for example, in a cluster). Schedulers among a site are supposed to be run in real parallelism and to synchronize at the end of each instant (via a synchronization barrier).

The number of synchronized schedulers belonging to a given site dynamically changes during the site execution, according to the load of agents that are present on the site and according to the availability of the computing resources. Moreover, agents can be *transparently moved* from one scheduler to another scheduler of the same site, to adapt the charge load over the site. The transparency basically results from the fact that agents do not share memory. The only way to share information in a site is to use events. The model is shown in Fig. 3.

According to this approach, the structuration of a program at the scheduler level is not statically fixed in order to allow the system to use resources in an efficient way. For example, in a multi-core context, the system is free to optimize the mapping of the schedulers (and consequently, the mapping of agents) in a way that maximises the use of the real cores.

The number of sites present in a program is statically fixed, as the model does not allow the dynamic creation of sites. The number of computing resources available for execution is fixed and depends on the executing machine,

and thus can be considered as a constant. We fix the number of schedulers to be the maximum of the number of sites and of the number of computing resources.

Initially, one scheduler, arbitrarilly chosen, is associated to each site. The remaining schedulers, if any, are the *unused schedulers*. At run time, two actions are possible for the sites: first, the taking of an unused scheduler (which thus becomes used); second; the releasing of a scheduler (which becomes unused). The first action is called *site expansion* and the second *site contraction*. The conditions and the moments expansions and contractions are performed is not specified and left to the implementation, in order to maximize the possibilities of optimisations.

The new way sites are structured in sets of synchronized schedulers is formalized through a new semantics for sites and systems described in the next section. New rules are introduced for site expansion, site contraction, and transparent agent migration in the same site, and previous rules for sites and systems are adapted to the new context.

The rest of the section is structured as follows: first, the semantics domains are defined in Section 8.1; then, the suspension predicate for sites and schedulers is defined in Section 8.2; finally, the semantics of sites and systems is given in Section 8.3.

8.1 Domains

The new definition of **Site** is:

$$\mathit{site} \in \mathbf{Site} = \mathbf{SiteName} \times \mathbf{SyncSched} \times \mathbb{N}^{\mathbf{Agent}} \times \mathbf{EventEnv}$$

A scheduler is a multi-set of agents and a synchronized scheduler is a multi-set of schedulers:

$$\mathit{sched} \in \mathbf{Sched} = \mathbb{N}^{\mathbf{Agent}}$$

$$\mathtt{scheds} \in \mathbf{SyncSched} = \mathbb{N}^{\mathbf{Sched}}$$

8.2 Suspension Predicate

The suspension predicate defined in Section 5.2 is extended to schedulers and redefined for sites.

A scheduler is suspended if all the agents belonging to it are suspended or terminated:

$$\frac{\langle Ag_i, E\rangle \ddagger \quad \vee \quad Ag_i = (\texttt{nothing}, M_i, \eta_i)}{\langle \{Ag_1, \ldots, Ag_m\}, E\rangle \ddagger} \tag{55}$$

A set of synchronized schedulers is suspended if all the schedulers belonging to it are suspended:

$$\frac{\langle sched_i, E\rangle \ddagger}{\langle \{sched_1, ..., sched_n\}, E\rangle \ddagger} \tag{56}$$

A site is suspended if the synchronized schedulers in it are suspended:

$$\frac{\langle \texttt{scheds}, E\rangle \ddagger}{(site, \texttt{scheds}, I, E)\ \ddagger} \tag{57}$$

8.3 Sites and Systems

8.3.1 Sites

The three rules 31-33 are changed in the same way: they now describe the execution of the script of an agent chosen in a synchronized scheduler of a site.

The first rule corresponds to the absence of migration oder (similar to rule 31):

$$\frac{sched = sched_0 \uplus (s, M, \eta) \quad \langle s,\ E,\ M\rangle \xrightarrow{\textbf{None}} \langle s',\ E',\ M'\rangle}{\begin{array}{l}\Sigma[(site, \texttt{scheds}[sched], I, E)] \\ \qquad \rightarrow \Sigma[(site, \texttt{scheds}[sched_0 \uplus (s', M', \eta)], I, E')]\end{array}} \tag{58}$$

The creation of a new agent is described by (similar to rule 32):

$$\frac{\begin{array}{c} sched = sched_0 \uplus (s, M, \eta) \\ \langle s,\, E,\, M\rangle \xrightarrow{Ag\downarrow site_2} \langle s',\, E',\, M'\rangle \\ sched' = sched_0 \cup (s', M', \eta) \\ S' = (site_2, \texttt{scheds}_2, I_2 \cup Ag, E_2) \end{array}}{\begin{array}{c} \Sigma[(site_1, \texttt{scheds}_1[sched], I_1, E_1)][(site_2, \texttt{scheds}_2, I_2, E_2)] \\ \Sigma[(site_1, \texttt{scheds}'_1[sched'], I_1, E'_1)][S'] \end{array}} \tag{59}$$

Finally, the migration to another site is described by (similar to rule 33):

$$\frac{\begin{array}{c} sched = sched_0 \uplus (s, M, \eta) \quad \langle s,\, E,\, M\rangle \xrightarrow{site'} \langle s',\, E',\, M'\rangle \\ sched' = sched_0 \uplus (s', M', \blacktriangleright (\eta, site')) \end{array}}{\Sigma[(site, \texttt{scheds}[sched], I, E)] \rightarrow \Sigma[(site, \texttt{scheds}[sched'], I, E')]} \tag{60}$$

8.3.2 End of Instant

When there exists migration orders, they are executed when the site is suspended (similar to rule 34):

$$\frac{\begin{array}{c} sched = sched_0 \uplus (s, M, site_2) \quad S\,\ddagger \\ S_1 = (site_!, \texttt{scheds}[sched_0], I_1, E_1) \\ S_2 = (site_2, \texttt{scheds}_2, I_2 \cup \Omega(s, M), E_2) \end{array}}{\begin{array}{c} \Sigma[(site_1, \texttt{scheds}_1[sched], I_1, E_1)][(site_2, \texttt{scheds}_2, I_2, E_2)] \\ \rightarrow \Sigma[S_1][S_2] \end{array}} \tag{61}$$

The end of current instant is reached when the site is suspended and there is no migration order to be processed (similar to rule 35):

$$\frac{\begin{array}{c} S = (site, Sched, I, E) \quad S\,\ddagger \quad S\natural \\ S' = (site, \Omega'(\texttt{scheds}) \uplus I, E) \end{array}}{\Sigma[S] \rightarrow \Sigma[S']} \tag{62}$$

In this rule, Ω' extends the reconstruction function Ω of Section 6.3 and is defined by:

$$\Omega'(sched_1 \uplus ... \uplus sched_n) = \Omega'(sched_1) \uplus ... \uplus \Omega'(sched_n)$$

and:

$$\Omega'(Ag_1 \uplus ... \uplus Ag_m) = \Omega(Ag_1) \uplus ... \uplus \Omega(Ag_m)$$

8.3.3 Expansion and Contraction

The two site expansion and site contraction actions depend on the number of unused schedulers, which is an integer global to the system. This integer is name `unused_schedulers`. The `free_scheduler`() function returns a arbitrary scheduler chosen among the unused schedulers, turns its state to used, and decrements the counter `unused_schedulers`. Conversely, the `kill_sched` function takes a used scheduler in parameter, turns its state to unused, and increments the counter `unused_schedulers`.

The two rules for site expansion and site contraction used freely the counter `unused_schedulers` and the previous functions.

Expansion of a site adds a new synchronized scheduler to a site:

$$\frac{\begin{array}{c} S = (site, \texttt{scheds}, \mathcal{I}, E) \\ \texttt{unused_schedulers} > 0 \\ sched = \texttt{free_scheduler}() \end{array}}{\Sigma[S] \rightarrow \Sigma[(site, \texttt{scheds} \uplus sched, \mathcal{I}, E)]} \tag{63}$$

The removal of a scheduler *sched* in a site cannot occur unless the number of agents of the scheduler, noted $\#sched$, is equal to zero. In this case, contraction means that the scheduler is removed from the site:

$$\frac{\begin{array}{c} \texttt{scheds} = \texttt{scheds}_0 \uplus sched_i \\ \#sched_i = 0 \\ \texttt{kill_scheduler}(sched_i) \end{array}}{\Sigma[(site, \texttt{scheds}, \mathcal{I}, E)] \rightarrow \Sigma[(site, \texttt{scheds}_0, \mathcal{I}, E)]} \tag{64}$$

8.3.4 Transparent Migration

During execution of a site, the implementation can choose to arbitrarilly transfer agents between the schedulers, in particular for optimization purposes. These transfers are called transparent migrations as they do not introduce any change in the execution of agents.

Transparent migration simply means to transfer an agent from a scheduler of a site to another scheduler of the same site:

$$\begin{aligned}\Sigma[(site, \texttt{scheds}[sched_1 \uplus Ag][sched_2], \mathcal{I}, E)] \rightarrow \\ \Sigma[(site, \texttt{scheds}[sched_1][sched_2 \uplus Ag], \mathcal{I}, E)]\end{aligned} \tag{65}$$

9 Example

This section describes the coding of a 2D simulation of colliding particles. The simulation is divided in two containers in which particles are moving and are bouncing against the borders. Collisions are elastic ones. There is a "Migration point" in each container: when a particle falls into a migration point, it migrates into the other container (in the same state). The simulation is shown on Fig 4.

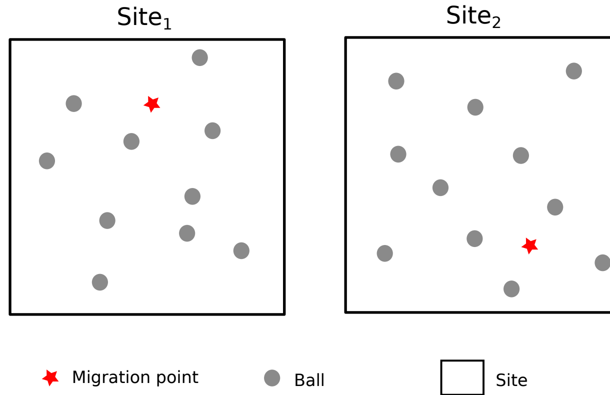

Figure 4: Example: Particle collision

The simulation is made of two sites $site_1$ and $site_2$, one for each container. Initially, each site contains N particles. Each particle is implemented as one agent. As they belong to the same site, all the particles present in a container share the same instants, and communicate their position by generating the shared event `position`. At each instant, a particle with coordinates (x, y) generates event `position` with the couple (x, y) as value. Then, the particle gets all the values of `position` and calculates if there is a potential collision with another particle (using the function `collision`, not described here). Finally, the particle computes its next position (function `next_position`) according to its state.

The script of a particle is a a loop whose body is a parallel instruction with 3 branches[2]. The first branch implements the signaling of position and the passing to the next position previously described. The second branch draw the particle on screen (function `draw`). Finally, the third branch decides if a migration must occur (function `should_migrate`).

The code for the first site is:

```
createAgent =
   Repeat N do
      createAgent =
       let l = ref () in
       let d = ref random_direction() in
       let (x,y) = ref random_position() in
         loop
             (generate "position" with (!x,!y);
              get_all "position" in l;
              d := collision(!l,!x,!y,!d);
              (x,y) := next_position(!x,!y,!d))
           ||
             draw(!x,!y)
           ||
             (if (should_migrate_site2(!x,!y)) then
                 migrate to site_2
              else
                 if (should_migrate_site1(!x,!y)) then
                   migrate to site_1
                 else
                   nothing)
        end
      in site_1
```

[2]The script definition of Section 5.3 is slightly extended to allow parallel instructions with more than 2 branches.

```
   end
in site_1
```

The code for the second site is similar except that `site_1` and `site_2` are exchanged.

Note that the functions used by the system are defined in the host language (Section 2).

The example illustrates several aspects of the model and of the language:

1. Parallel components of agents are *naturally expressed* using the synchonous parallel operator. Each particle is considered as an autonomous object which moves accordingly to its defining script and interacts with the other particles through broadcast events.

2. Agents can be executed in real parallelism: two agents belonging to distinct schedulers can be run in real parallelism, by different computing resources. However, if two agents belong to the same site, the efficiency of the real parallelism of execution is moderated by the necessity for the schedulers to synchronize at each end of instant. But, two agents belonging to different sites can be executed in real parallelism without restriction, which can lead to efficient executions.

3. Each particle has its own memory containing its state (its coordinates) which is shared by all the components of its execution script. The language requires the absence of data-race in memory accesses, which is verified by the type system of Section 7. Thus, there is no risk of a data-race between any two agents, being or not on the same site.

4. There is no possibility of an instantaneous loop which would prevent execution form passing to the next instant, due to the semantics of the loop operator (rule 20).

10 Related Work

Ptolemy [15] is a whole complex framework which aim to model, design, and simulate concurrent, real-time embedded systems. Safety is not a central objective of Ptolemy. `DSLM` is just a language and is much smaller and oriented to safety and maximal usage of computing resources.

SugarCubes [11] is a framework for concurrent reactive programming in Java. `DSLM` is strongly inspired by SugarCubes. Both formalisms use the same totally deterministic parallel operator, called *merge* in SugarCubes. However, SugarCubes does not possess the notion of a synchronized schedulers and is not optimized for the multi computing resources systems. SugarCubes can be used over the network by using Java (RMI), which is not yet possible in our language.

FunLoft [10] (*"Functional Language over Fair Threads"*) is a language for safe reactive programming, with type inference. The FunLoft compiler checks that functions called by a program always terminate and only use a bounded amount of resources (memory and CPU). The basis of FunLoft is a theoretical Work described in [13]. `DSLM` is strongly linked to FunLoft in two aspects: first, the notion of synchronized schedulers of DSLM is coming from FunLoft. Second, there exists an experimental implementation of `DSLM` in FunLoft ([1]) in which functions are proved to terminate instantaneously. Compared to FunLoft which has a static approach, `DSLM` improves the usage of computing resources, by introducing the possibility of dynamical load balancing of agents among synchronized schedulers inside a same site.

`ReactiveML` [21] is a programming language for reactive programming in ML. As ML, `ReactiveML` is secure in the sense that there is no possibility of a crash during execution. However, there is no check in `ReactiveML` to insure that instants are always terminating. Moreover, `ReactiveML`, like ML, is not presently adapted for multi-core architectures. `ReactiveML` have the possibility to compile programs on the fly, which is not currently possible in `DSLM`.

ULM [8] is a programing model to address the unreliable character of accessing resources in a global computing context, focusing on giving a precise semantics for a small, yet expressive core language. Like `DSLM`, ULM has safety as a goal in accessing memory without using locks. However, in ULM, a script that wants to access a memory location is suspended if the location cannot be accessed, because it does not belong to the current site. ULM is not currently adapted for multi-core architectures.

CPC [20] is a programming language for concurrent systems in which native and cooperative threads are presented to the programmer as a single abstraction. CPC proposes a way to benefit from multi-core architecture, by spawning threads. The implementation technics of CPC is close to the one we use, as both are defining two levels of concurrency: a cooperative level (implementing the synchronous parallel in `DSLM`) and a preemptive level for

sites.

Work sharing/stealing [7] is a thread-based model in which threads are sharing their work with the other threads, and steal the work of the others when theirs are terminated. This approach is well adapted to multi-core architectures, but does not consider safety issues. However, one could envision to implement DSLM in the work sharing/stealing style.

11 Conclusion

We have presented a dynamic approach to parallel programming, based on the synchronous - reactive model. Synchronous programming is simpler than the traditional asynchronous approaches, based on the exclusive use of preemptive threads. However, three major issues are raised by synchronous programming: (1) how to be sure that the program is indeed reactive? (2) how to be sure that there is no harmful interferences between parallel computations (e.g. data-races)? (3) how to execute it efficiently on a multi-core/ multi-processor architecture? Our proposal gives answers to these questions. We assure the reactivity of a program by construction, and we require functions to be instantaneous. In the actual implementation which basically translates DSLM in FunLoft, this property is checked by the FunLoft compiler. DSLM defines agents which encapsulate their memory in a way which forbids harmful interferences due to memory sharing. The absence of memory sharing is checked by a type system. Finally, to benefit from multi-core/multi-processor architectures we define synchronized schedulers and transparent migration of agents to load balance the charge of agents over a site.

We envision the following tracks for Future Work:

- We plan to add security aspects to DSLM. The considered security aspects are control and tracking of information flows at all levels: memory, events, and agent migrations. This can be acheived by using security levels, an approach inspired by [12].

- In DSLM, functions and modules are defined in the host language, thus there is no insurance that the required properties (instantaneous termination of functions, and non-instantaneous execution of modules) are satisfied. We plan to allow function and module definitions directly in DSLM to be able to statically check their required properties.

- The possibility to give a big-step semantics at the level of agents and sites (and not at the script level) is currently under investigation. This semantics would give a view of systems and sites more abstract and more adapted to the distribution context than the small-step semantics presented in this paper.
- An implementation of `DSLM` is under development, based on a translation in FunLoft.

In this perspective, `DSLM` would be a safe and secure parallel programming language, adapted to multi-core/multi-processor architecture, which is, to our knowledge, something new and never proposed before.

References

[1] DSLM : Dynamic Script Language with Memory. https://gforge.inria.fr/projects/partout/.

[2] Reactive Programming. http://www-sop.inria.fr/indes/rp/.

[3] G.A. Agha. Actors: A model of concurrent computation in distributed systems. 1985.

[4] G. Berry. The Constructive Semantics of Pure Esterel, 1999.

[5] Gérard Berry. The Foundations of Esterel, 1998.

[6] Gérard Berry and Laurent Cosserat. The ESTEREL Synchronous Programming Language and Its Mathematical Semantics. In Stephen Brookes, Andrew Roscoe, and Glynn Winskel, editors, *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 389–448. Springer Berlin / Heidelberg, 1985.

[7] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work stealing. *J. ACM*, 46(5):720–748, September 1999.

[8] Gérard Boudol. ULM: A Core Programming Model for Global Computing. In David Schmidt, editor, *Programming Languages and Systems*, volume 2986 of *Lecture Notes in Computer Science*, pages 234–248. Springer Berlin Heidelberg, 2004.

[9] Gérard Boudol. Fair Cooperative Multithreading. In Lus Caires and Vasco Vasconcelos, editors, *CONCUR 2007 Concurrency Theory*, volume 4703 of *Lecture Notes in Computer Science*, pages 272–286. Springer Berlin / Heidelberg, 2007.

[10] F. Boussinot. *Safe Reactive Programming: The FunLoft Proposal*. Lambert Academic Publishing, 2010.

[11] Frédéric Boussinot and Jean-Ferdy Susini. The SugarCubes Tool Box: A Reactive Java Framework. *Software: Practice and Experience*, 28(14):1531–1550, 1998.

[12] Sara Capecchi, Ilaria Castellani, Mariangiola Dezani Ciancaglini, and Tamara Rezk. Session Types for Access and Information Flow Control. Technical report.

[13] Frédéric Dabrowski. Programmation Réactive Synchrone: langages et contrôle des ressources. 2007. PhD thesis.

[14] M. Dubois and C. Scheurich. Memory Access Dependencies in Shared-memory Multiprocessors. *Software Engineering, IEEE Transactions on*, 16(6):660 –673, jun 1990.

[15] J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127 – 144, jan 2003.

[16] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Pub., 1993.

[17] Per Brinch Hansen. Concurrent Programming Concepts. *ACM Comput. Surv.*, 5(4):223–245, December 1973.

[18] Matthew Hennessy. *The Semantics of Programming languages: An Elementary Introduction Using Structural Operational Semantics*. John Wiley & Sons, 1990.

[19] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

[20] Gabriel Kerneis and Juliusz Chroboczek. CPC: programming with a massive number of lightweight threads. *CoRR*, abs/1102.0951, 2011.

[21] Louis Mandel and Marc Pouzet. Reactiveml: A reactive extension to ml. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '05, pages 82–93, New York, NY, USA, 2005. ACM.

[22] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541 –580, apr 1989.

[23] J. Muttersbach, T. Villiger, and W. Fichtner. Practical Design of Globally-Asynchronous Locally-synchronous Systems. In *Advanced Research in Asynchronous Circuits and Systems, 2000. (ASYNC 2000) Proceedings. Sixth International Symposium on*, pages 52 –59, 2000.

[24] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[25] P. Schaumont, Bo-Cheng Charles Lai, Wei Qin, and I. Verbauwhede. Cooperative Multithreading on Embedded Multiprocessor Architectures Enables Energy-scalable Design. In *Design Automation Conference, 2005. Proceedings. 42nd*, pages 27 – 30, june 2005.