Proposal for a Dynamic Synchronous Language

Pejman Attar, Frédéric Boussinot, Louis Mandel, Jean-Ferdy Susini

Proposal for a Dynamic Synchronous Language*

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

Frédéric Boussinot
INRIA - INDES
Frederic.Boussinot@inria.fr

Louis Mandel
Université Paris 11 - LRI
INRIA - PARKAS
Louis.Mandel@lri.fr

Jean-Ferdy Susini
CNAM - Cédric
Jean-Ferdinand.Susini@cnam.fr

May 3, 2011

Abstract

We propose a new scripting language called DSL based on the synchronous/reactive model. In DSL, systems are composed of several sites executed asynchronously, and each site is running scripts in a 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.

1 Introduction

Synchronous programming [6] simplifies concurrency, compared to standard approaches based on the exclusive use of the classic model of threads (pthreads or Java threads). The simplification basically results from a cleaner and simpler semantics, which reduces the number of possible interleavings in parallel computations. However, standard synchronous languages introduce specific issues (namely, non-termination of instants) and have major difficulties to cope with dynamic creation (of threads, components, or signals); moreover they are generally not able to fully benefit from real parallelism, as that provided by multicore machines. We propose a new synchronous language, called DSL (for *Dynamic Synchronous Language*), which addresses the above issues in the following way:

- We use the reactive variant [4] of the synchronous approach, which is able to express dynamic creation; basically, in this variant "causality cycles" are eliminated "by construction", by forbidding immediate reaction to absence. Several related proposals are based on this variant (e.g. Reactive-C [9], SugarCubes [14], ReactiveML [17]).
- We introduce the notion of *site*, for combining synchronous and asynchronous aspects of system decompositions into a unique framework. This combination is inspired by the FairThreads model [10] which mixes cooperative and preemptive threads. Sites can be run by dedicated cores, thus giving a way to fully exploit multicore architectures.

DSL is designed with a simple formal semantics, describing without ambiguity how the system evolves.

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

Our approach is an alternative to the use of locks for memory protection in a classic threading context. We claim that our proposal makes *both* semantics and programming easier.

1.1 Orchestration Model

The computing model we consider is the following: there are N sites, each of them being composed of two levels, the orchestration level and the host level. The sites are completely autonomous and are run asynchronously (possibly on different processing resources).

At the orchestration level, each site runs a script which is fundamentally parallel. At that level, inputs are:

- new scripts dropped by the other sites, by the external world, or by the host level of the site; these new scripts are put in parallel with the one currently executed by the site; events are input as simple scripts generating them.
- boolean values coming from the host level and used by `if` instructions;
- integer values coming from the host level and used by `repeat` statement.

The outputs of the orchestration level are:

- new scripts sent to other sites;
- calls of functions belonging to the host level of the site. Functions are treated in an abstract way: we do not consider their effect on the memory but only their "orchestration" i.e. the organisation of their calls in time and in place (the site where they are called).

Scripts run by the same site synchronise by means of local events which are broadcast in the whole site. Two properties are assumed in DSL: reactivity of sites, and absence of interferences between sites (i.e. sites do not share instants, nor memory, nor events). These two properties of sites allow them to be mapped on distinct cores while preserving the semantics, and to therefore benefit from multicore architectures.

Sites bear an analogy with a musical orchestra: the orchestration level corresponds to the orchestra conductor who leads the host level corresponding to the music players. The conductor follows a music partition (script) and communicates with the musicians by sending them orders and signals (modelled by events) and by listening to the music they play (also modelled by events; imagine events corresponding to sound waves produced by the instruments or by the voices). The conductor must be able to do several things in parallel: she must direct and listen to all instruments at the same time. Of course, as needed by any orchestra, a common clock is defined by the conductor; in our model, this clock which should be shared by the whole orchestra is given by instants. One can see the presence of several orchestras (sites) playing asynchronously as what happens sometimes in music festivals, when several stages are used independently, possibly simultaneously (of course in this case, the absence of interferences between distinct stages is mandatory: nothing should be shared by distinct sites).

1.2 Variants of DSL

We consider several variants of DSL, depending on the language (which we call the *host language*) in which functions are expressed.

- In the first variant, the host language is FunLoft [11, 12]. The basic properties of reactivity and of absence of data-races are checked by the FunLoft compiler. In order to insure these properties, the use of functions is restricted (for example functions cannot be stored in memory). As FunLoft does not currently provide any means for code distribution, this feature is not presently available.
- In the second variant of DSL, functions are defined in ReactiveML [17] which is an extension of ML to reactive programming. Higher-order functions can be defined without restriction in this variant of DSL. The JoCaml system [3] is used for asynchronous inter-site communications. Moreover, the ReactiveML variant provides users with the scripting top-level of ReactiveML. The ReactiveML variant does not check the reactivity and interference freeness properties which are thus left to the programmer's responsability.
- The third variant is based on SugarCubes [14] which is a Java based framework for reactive programming. Like with the ReactiveML variant, the basic reactivity and data-race freeness properties are not checked. The SugarCubes variant allows one to use the object approach of Java to program functions. Communication

between two sites is implemented with the RMI protocol of Java.

- The fourth variant is based on Scheme/Bigloo [1]. It directly implements the semantic rules. In the current implementation, distribution aspects are not covered but we plan to address them using the Hop [2] system implemented in Scheme/Bigloo.

1.3 Implementations

At the implementation level, each variant of DSL has its own particularities. The most interesting aspect of comparison is how the notion of instant in DSL is reflected in the various host languages.

With ReactiveML and SugarCubes, instants of DSL and instants of the host language are in a direct one-to-one correspondence. Actually, there is no need to introduce any supplementary mechanism to run DSL scripts, which are just directly translated in the host language to be executed. This is possible because the expressivity of these host languages is comparable to that of DSL: actually, each DSL primitive (except `drop`) has a direct counterpart in them.

Structure of the text

The language is first described informally in Section 2. Then, its formal semantics is given in Section 3. The FunLoft variant of DSL is described in Section 4; the SugarCubes variant is described in Section 5; the ReactiveML variant is described in Section 6; finally, the Scheme/Bigloo variant is described in Section 7. Section 8 contains some experiments with a multicore machine. Related work is considered in Section 9 and a conclusion is given in Section 10. Annex A contains the grammar of DSL. Annex B describes the evaluation of DSL scripts in FunLoft. Annex C describes the evaluation of DSL scripts in ReactiveML.

2 Language Description

A program is composed of several independent sites, each of them executing a script made of parallel components. To add a new script into a site, one puts the script in parallel with the already existing parallel components.

In the current version, DSL does not provide any means to define functions. However, scripts may call functions defined in a "host" language (different variants of DSL correspond to different host languages). These functions have parameters of basic types only (integer, boolean, string).

Tasks are special functions whose execution is not immediate; actually, execution of a task does not start immediately, but at the next instant; moreover, the execution of a a task can last several instants or even never terminate. Tasks are called using a specific keyword (`launch`). Scripts "orchestrate" the execution of functions and tasks on the various sites that compose a program.

We first present scripts in 2.1, then introduce sites in 2.2 and events in 2.3. The basic properties of DSL are presented in 2.4. Finally, an example which will be used to benchmark implementations is described in 2.5.

2.1 Scripts

Scripts are made of basic instructions whose informal semantics is as follows:

- `nothing` does nothing
- `cooperate` terminates the execution for the current instant. Execution resumes at the next instant.
- $f\,(v_1, \ldots, v_n)$ calls the function f with the parameters $v_1, ..., v_n$. Execution starts immediately and is instantaneous. To call a non-existing function is considered as an empty statement.
- `launch` $t\,(v_1, \ldots, v_n)$ launches the task t with the parameters $v_1, ..., v_n$. Execution takes several (at least, one) instants to terminate, or may even never terminate. To call a non-existing task is considered as an empty statement.
- $s_1; s_2$ runs the two scripts s_1 and s_2 in sequence.
- $s_1 \parallel s_2$ runs the two scripts s_1 and s_2 in parallel. The parallel script terminates as soon as both s_1 and s_2 are terminated.
- `if` e `then` s_1 `else` s_2 `end` runs the script corresponding to the result of the evaluation of the boolean expression e.
- `loop` s `end` cyclically runs the script s. Execution of s is restarted as soon as it terminates, except if it terminates instantly (i.e. in the same instant it is started); in this last case, 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 same instant.

- `repeat` e `do` s `end` runs n times the script s, where n is the result of the evaluation of the integer expression e.
- `generate` e generates the event e.
- `await` e blocks execution while the event e is not generated. Execution resumes as soon as e is generated.
- `do` s `watching` e executes the script s while the event e is not generated. The execution of s is aborted when e is generated. The `watching` instruction terminates normally when s terminates.
- `drop` s `in` *site* adds the script s in the remote site *site*. Execution continues immediately without waiting for the completion of s.

The BNF syntax of scripts is given in Annex A.

2.2 Sites

Sites are asynchronous (i.e., each site is possibly run by a distinct native thread). On the contrary, scripts are executed synchronously on a site: they share the same instants and thus proceed at the same pace.

The `drop` instruction is the means by which a script can influence remote sites. Note that, if nothing remains to be done after a `drop` instruction, one can see it as a *migration* to the remote site.

The creation of sites is not specified in the language; we suppose that for each script of the form `drop` s `in` *site*, *site* always exists and is accessible.

2.3 Events

Events are boolean values that are present or absent during instants. Events are not shared among sites. Once an event is generated during an instant, it remains present up to the end of the instant. Events are automatically reset to absent at the beginning of each instant. Events considered by the three instructions `generate`, `await`, and `watching` are created if they are not already existing on the site of execution.

2.4 Basic Properties

DSL demands that the two following fundamental properties are valid:

- No site can be prevented from passing to the next instant (*reactivity* property). This means that functions and tasks run by a site should not use all of the computing power of the site.
- No data-race can occur between scripts, functions and tasks (*interference freeness* property).

Basically, tasks can be executed in two ways: in the first way, the task is executed by the site in which it is launched (the task is said to be *linked* to the site); in the second way, a dedicated processing thread is devoted to the task execution (the task is *unlinked*).

In all the considered variants of DSL, except the Scheme/Bigloo variant, linked tasks are executed cooperatively by sites where instants exist. In these variants, data-races should only occur between tasks that are unlinked, or that belong to distinct sites. In the Scheme/Bigloo variant, tasks are always unlinked.

In the FunLoft variant, the fundamental properties are checked by the compiler which verifies that:

- Functions always terminate instantaneously.
- Linked tasks always cooperate.
- Memory can only be shared by functions or tasks linked to the same site.

In the other variants, the validity of the two fundamental properties is left to the programmer.

2.5 Example

We consider a system composed of 3 sites `site1`, `site2`, and `site3`, and a script supposed to be run by `site1`; the script is made of two sub-parts executed on `site2` and `site3`. Each sub-part calls the `consume` function (which heavily uses the CPU, according to the value of its parameter) and then drops back a script on `site1` to signal its termination. The two events generated upon termination are awaited in parallel. The code is:

```
repeat 1000 do
  (
    drop
        print ("0");
        consume (10000000);
```

```
      drop generate done0 in site1
   in site2
||
   drop
      print ("1");
      consume (10000000);
      drop generate done1 in site1
   in site3
||
   await done0
||
   await done1
 );
 cooperate
end
```

Note that there are similar parts in the code (for example, the two calls to `consume`). Actually, the DSL language does not give any means to share or parametrize scripts. In this respect, scripts are not very friendly and should thus be produced from some higher-level language; the definition of such a language is not in the scope of the present paper.

The two calls of `consume` can be executed in real parallelism (for example, on a dual-core machine). It is assumed that no interferences appear between them (for example, resulting from the sharing of a global counter). This assumption is statically verified in the FunLoft variant of DSL, while it is the responsibility of the programmer in the other variants. We shall return on this example later, when implementation is considered.

Remark: the body of a `repeat` statement is not demanded to be non-instantaneous, unlike the body of a `loop` statement. Indeed, a `repeat` script always terminates (provided its body terminates), and thus cannot prevent the other scripts to get the control. In the previous code, the justification for the `cooperate` is to prevent an instantaneous termination of the `repeat` if both `done0` and `done1` are received in the same instant; this is actually possible because of the asynchronous execution of sites.

3 Semantics

We give DSL a semantics expressed with rewriting rules. The semantics is "big step": one rewriting of a term represents the global execution of the term during one instant (as opposite, a "small step" semantics would describe the various execution steps occuring during the instant).

Evaluation of expressions is considered in 3.1. The (big-step) rewriting of scripts is first described in 3.2; then, fix-points are considered in 3.3; site execution is described in 3.4; three examples are considered in 3.5; finally, three variants of the semantics are analysed in 3.6.

3.1 Expressions

Expressions are either basic values (of type integer, boolean, or string), or calls of functions of the form $f(v_1, \ldots, v_n)$ where v_i are basic values. We adopt the following notation: we write $f(v_1, \ldots, v_n)\Uparrow$ if there is no function named f which is defined, or if the call is not well typed; in this case we say that we have a *wrong call*; we write $f(v_1, \ldots, v_n)\Downarrow$ otherwise.

The evaluation of a basic value returns itself. There are two cases for the evaluation of $f(v_1, \ldots, v_n)$:

- if $f(v_1, \ldots, v_n)\Downarrow$, the evaluation of the call returns the value of **f** applied to the list of values v_i, where **f** is the function associated to f;
- if $f(v_1, \ldots, v_n)\Uparrow$, then the value returned is the default value of the expected (basic) type (0 for integers, `false` for booleans, and the empty string `""` for strings).

The evaluation of the expression exp returning a value v is noted $exp \rightsquigarrow v$.

As with functions, we write $t(v_1, \ldots, v_n)\Uparrow$ if the task t does not exist or if the call is not well typed, and we write $t(v_1, \ldots, v_n)\Downarrow$ otherwise.

3.2 Scripts

The general format of the script semantics is:

$$P \vdash s \xrightarrow{b} s', G, D$$

- P is the set of present events; events not belonging to P are absent;
- s is the script which is rewriten;
- s' is the *residual* script ("what remains to do at the next instant");
- G is the set of events generated by the rewriting of s;
- D is the multiset of *dropped scripts* of the form $site\downarrow u$, where $site$ is a site name and u is a script; the union of multiset is noted $\uplus$;
- b is a boolean which is true (tt) if s' is terminated and false (ff) otherwise; the boolean conjunction is noted $\wedge$.

The semantics of scripts is given by the following rules:

Nothing

$$P \vdash \texttt{nothing} \xrightarrow{tt} \texttt{nothing}, \emptyset, \emptyset \tag{1}$$

Cooperate

$$P \vdash \texttt{cooperate} \xrightarrow{ff} \texttt{nothing}, \emptyset, \emptyset \tag{2}$$

Drop

$$P \vdash \texttt{drop}\ s\ \texttt{in}\ site \xrightarrow{tt} \texttt{nothing}, \emptyset, \{site \downarrow s\} \tag{3}$$

Sequence

$$\frac{P \vdash s_1 \xrightarrow{ff} s_1', G, D}{P \vdash s_1; s_2 \xrightarrow{ff} s_1'; s_2, G, D} \tag{4}$$

$$\frac{P \vdash s_1 \xrightarrow{tt} s_1', G_1, D_1 \quad P \vdash s_2 \xrightarrow{b} s_2', G_2, D_2}{P \vdash s_1; s_2 \xrightarrow{b} s_2', G_1 \cup G_2, D_1 \uplus D_2} \tag{5}$$

Parallel

$$\frac{P \vdash s_1 \xrightarrow{b_1} s_1', G_1, D_1 \quad P \vdash s_2 \xrightarrow{b_2} s_2', G_2, D_2}{P \vdash s_1 \parallel s_2 \xrightarrow{b_1 \wedge b_2} s_1' \parallel s_2', G_1 \cup G_2, D_1 \uplus D_2} \tag{6}$$

Loop

$$\frac{P \vdash s \parallel \texttt{cooperate} \xrightarrow{ff} s', G, D}{P \vdash \texttt{loop}\ s\ \texttt{end} \xrightarrow{ff} s'; \texttt{loop}\ s\ \texttt{end}, G, D} \tag{7}$$

Generate

$$P \vdash \texttt{generate}\ e \xrightarrow{tt} \texttt{nothing}, \{e\}, \emptyset \tag{8}$$

Await

$$\frac{e \in P}{P \vdash \texttt{await}\ e \xrightarrow{tt} \texttt{nothing}, \emptyset, \emptyset} \tag{9}$$

$$\frac{e \notin P}{P \vdash \texttt{await}\ e \xrightarrow{ff} \texttt{await}\ e, \emptyset, \emptyset} \tag{10}$$

Watching

$$\frac{P \vdash s \xrightarrow{tt} s', G, D}{P \vdash \texttt{do}\ s\ \texttt{watching}\ e \xrightarrow{tt} \texttt{nothing}, G, D} \tag{11}$$

$$\frac{e \in P \quad P \vdash s \xrightarrow{ff} s', G, D}{P \vdash \texttt{do}\ s\ \texttt{watching}\ e \xrightarrow{ff} \texttt{nothing}, G, D} \tag{12}$$

$$\frac{e \notin P \quad P \vdash s \xrightarrow{ff} s', G, D}{P \vdash \texttt{do}\ s\ \texttt{watching}\ e \xrightarrow{ff} \texttt{do}\ s'\ \texttt{watching}\ e, G, D} \tag{13}$$

Evaluation of expressions (function calls are expressions) appear in the following rules which are thus less formal than the previous ones; indeed, evaluation of expressions is not totally captured by the semantics of DSL.

Function

$$\frac{f\,(v_1, \ldots, v_n) \rightsquigarrow v}{P \vdash f\,(v_1, \ldots, v_n) \xrightarrow{tt} \texttt{nothing}, \emptyset, \emptyset} \tag{14}$$

Execution of a function call is equivalent to its evaluation; the returned value is of no use, and the call is actually only evaluated for its side-effects (a wrong call does nothing and has no side-effect).

Task

$$\frac{t\,(v_1, \ldots, v_n)\Uparrow}{P \vdash \texttt{launch}\ t\,(v_1, \ldots, v_n) \xrightarrow{tt} \texttt{nothing}, \emptyset, \emptyset} \tag{15}$$

$$\frac{t\,(v_1, \ldots, v_n)\Downarrow}{P \vdash \texttt{launch}\ t\,(v_1, \ldots, v_n) \xrightarrow{ff} \texttt{await}\ e, \emptyset, \emptyset} \tag{16}$$

Three points should be noted:

- Rule 15 states that a wrong call of a task is equivalent to a `nothing` statement.
- In rule 16, e is a new event[1] which signals the termination of the launched task; it is automatically generated by the system when the call of t turns to be completely terminated.
- In case of real preemption, i.e. when rule 12 applies, the waiting for termination is abandonned and the task is not actually started.

Repeat

$$\frac{exp \rightsquigarrow n \quad P \vdash \overbrace{s; \ldots ; s}^{n \text{ times}} \xrightarrow{b} s', G, D}{P \vdash \texttt{repeat}\ exp\ \texttt{do}\ s\ \texttt{end} \xrightarrow{b} s', G, D} \tag{17}$$

Two points should be noted:

- Evaluation of exp is performed when the rule is applied, that is at execution time (not at compile time).
- In case exp is a wrong function call, n is equal to 0, and the sequence is equal to `nothing`[2]. The `repeat` statement is thus in this case equivalent to `nothing`.

If

$$\frac{exp \rightsquigarrow tt \quad P \vdash s_1 \xrightarrow{b} s_1', G, D}{P \vdash \texttt{if}\ exp\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end} \xrightarrow{b} s_1', G, D} \tag{18}$$

$$\frac{exp \rightsquigarrow ff \quad P \vdash s_2 \xrightarrow{b} s_2', G, D}{P \vdash \texttt{if}\ exp\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end} \xrightarrow{b} s_2', G, D} \tag{19}$$

Note that if exp is a wrong function call, its evaluation returns ff, and thus s_2 is chosen.

3.3 Least Fix-Point

The execution of scripts is deterministic:

if $P \vdash s \xrightarrow{b_1} s_1, G_1, D_1$ and $P \vdash s \xrightarrow{b_2} s_2, G_2, D_2$, then $s_1 = s_2$, $G_1 = G_2$, $D_1 = D_2$, and $b_1 = b_2$.

Let s be a script; the determinism property allows one to define the function f_s which, given a set P of present events, returns the set G of events generated by s:

$$f_s(P) = G \text{ where } P \vdash t \xrightarrow{b} s', G, D$$

The function f_s has two main characteristics: it is total and it is increasing. It is total because, for each script and each set of present events, there exists a (unique) rewriting:

$$\forall s, P, \exists s', G, D, b \quad P \vdash s \xrightarrow{b} s', G, D$$

The function f_s is increasing (for the set inclusion order):

$$\text{if } P_1 \subseteq P_2 \text{ then } f_s(P_1) \subseteq f_s(P_2)$$

The function f_s thus has a least fix-point μf_s (Kleene theorem) verifying:

$$f_s(\mu f_s) = \mu f_s$$

that is:

$$\mu f_s \vdash s \xrightarrow{b} s', \mu f_s, D$$

and:

$$f_s(Q) = Q \text{ implies } \mu f_s \subseteq Q$$

We know that the least fix-point μf_s is the limit of the sequence of approximations $X_0, X_1, \ldots$ defined by:

$$X_0 = \emptyset \text{ and } X_{n+1} = f_s(X_n)$$

which is noted:

$$\mu f_s = \bigcup f_s^n(\emptyset)$$

Finally, when the value of the least fix-point is not required, we write:

$$s \Rightarrow s', D$$

instead of:

$$\mu f_s \vdash s \xrightarrow{b} s', \mu f_s, D$$

3.4 Sites

A site is a couple $(site, s)$ made of a site name $site$ and a script s; it is noted $site : s$.

A program is a (finite) multiset of sites and of dropped scripts waiting to be incorporated into sites. A program is thus a multiset S where each element is either a site $site_i : s_i$ or a dropped script $site_i \downarrow s_i$. One supposes that there is at least one site and that all sites have distinct names:

$$\forall site_i : s_i, site_j : s_j \in S, i \neq j \Rightarrow site_i \neq site_j$$

[1] a mechanism to produce new fresh events is assumed.

[2] a sequence of $n \leq 0$ elements is by definition equal to `nothing`.

Note that the same dropped element can appear several time in a program (it is a multiset), as for example in:

$$\{site : \texttt{nothing}, site\downarrow f(), site\downarrow f()\}$$

The execution of a program S_0 is a sequence of rewritings of the form:

$$S_0 \mapsto S_1 \mapsto S_2 \mapsto ...$$

where the arrow $\mapsto$ is defined by rules 20, 21, and 22 defined below.

Site execution

$$\frac{s \Rightarrow s', D}{\{..., site : s, ...\} \mapsto \{..., site : s', ...\} \uplus D} \quad (20)$$

The dropped scripts resulting from a site execution are added in the program by rule 20; they are waiting to be absorbed by rule 21 below. In the definition of $\Rightarrow$, note that the least fix-point is not explicitly built: the semantics is not effective in this respect as it does not indicates *how* to compute it.

Absorbtion of dropped scripts

$$\{..., site : s, site\downarrow u, ...\} \mapsto \{..., site : s \parallel u, ...\} \quad (21)$$

Rule 21 represents the absorbtion of a dropped script u by the appropriate site *site*: the dropped script is simply put in parallel with the script s already present in *site*.

Inputs The dynamic adding of a script s in the site *site* of a program S is modelised by:

$$S \mapsto S \uplus \{site\downarrow s\} \quad (22)$$

Program inputs are dropped events: the input of the event e in the site *site* is simply modelised by the rewriting:

$$S \mapsto S \uplus \{site\downarrow \texttt{generate}\ e\}$$

3.5 Examples

We give several examples: the first shows the computation of the semantics by successive approximations; the second illustrates the links between the fix-point semantics and the notion of causality; the third example considers the `drop` primitive; the fourth consider the relation between the `watching` and `launch` instructions; finally, the last example shows the global execution of a program.

3.5.1 Approximations

Let us consider the following script s:

$$\texttt{generate}\ e; \texttt{await}\ f \parallel \texttt{await}\ e; \texttt{generate}\ f$$

Actually, one can prove that:

$$\{e, f\} \vdash s \xrightarrow{tt} \texttt{nothing} \parallel \texttt{nothing}, \{e, f\}, \emptyset$$

Let us show that this corresponds to the least fix-point μf_s of f_s (using the previous notations). Let $X_0 = \emptyset$. One has:

$$X_0 \vdash s \xrightarrow{ff} \texttt{await}\ f \parallel \texttt{await}\ e; \texttt{generate}\ f, \{e\}, \emptyset$$

Let $X_1 = \{e\}$. We have:

$$X_1 \vdash s \xrightarrow{ff} \texttt{await}\ f \parallel \texttt{nothing}, \{e, f\}, \emptyset$$

Let $X_2 = \{e, f\}$. Since:

$$X_2 \vdash s \xrightarrow{tt} \texttt{nothing} \parallel \texttt{nothing}, X_2, \emptyset$$

we get the result:

$$\mu f_s = \bigcup f_s^n(\emptyset) = X_2 = \{e, f\}$$

.

3.5.2 Minimality

Minimality of fix-points is mandatory to reject "violations of causality". Indeed, consider the following script $s = \texttt{await}\ e; \texttt{generate}\ e$. Two fix-points, $\{e\}$ and $\emptyset$, exist:

1. $\{e\} \vdash s \xrightarrow{tt} \texttt{nothing}, \{e\}, \emptyset$
2. $\emptyset \vdash s \xrightarrow{ff} s, \emptyset, \emptyset$.

The least fix-point is thus $\emptyset$. Note that in the first rewriting, the generation of e "results" from the test of presence of e, and thus does not correspond to any "causal" execution. In a sense, the minimality of fix-points is a way to rule out non-causal executions.

3.5.3 Asynchrony

Let us consider the script:

```
drop
     generate e || await e; print ("ok")
in site1
```

The message will always be printed, because the dropped script is incorporated in `site1` as a whole. This would not be the case with:

```
drop
     generate e
in site1;
drop
     await e; print ("ok")
in site1
```

Indeed, `site1` may incorporate the first script and may react *before* the incorporation of the second script; in this case, the message is not printed because the generation of e is lost.

3.5.4 Task Abortion

Let us consider the immediate preemption of a task launched by the body of a `watching` statement:

```
generate e;
do
  launch t ()
watching e
```

If task `t` does not exist or is incorrectly called, then the global instruction terminates immediately (rules 15 and 11). Otherwise (`t` exists and is correctly called), the task is not launched by the executive system (last remark, rule 16), and the instruction will terminate at the next instant (rule 12).

3.5.5 Program Input

Let us consider the following program made of a unique site:

$$S = \{site : \texttt{await}\ e; print(msg)\}$$

The only rewriting that can be made is:

$$S \mapsto S$$

Suppose a new input is given to the program, which becomes S':

$$S \mapsto S' = S \uplus \{site \downarrow \texttt{generate}\ e\}$$

There are two possible rewritings for S':

$$S' \mapsto S'$$

and (rule 21):

$$S' \mapsto S'' = \{site : \texttt{await}\ e; print(msg) \parallel \texttt{generate}\ e\}$$

Now, one can prove that the only possible rewriting of S'' is:

$$S'' \mapsto \{site : \texttt{nothing}\}$$

During this rewriting, the function *print* is called and a message is printed as a side-effect of the call.

3.6 Variants of the semantics

In this section, we discuss 3 aspects of the semantics: instantaneous loops are first considered; then, a variant of the drop instruction is analysed; finally, the watching instruction is discussed.

3.6.1 Instantaneous Loops

The fact that the function f_s is total basically results from the rule 7 that "fixes" instantaneous loops. Note that without the fix, some loops could have no rewriting at all; this would be for instance the case with the rule:

$$\frac{P \vdash s; \texttt{loop}\ s\ \texttt{end} \xrightarrow{b} s', G, D}{P \vdash \texttt{loop}\ s\ \texttt{end} \xrightarrow{b} s', G, D} \quad (7')$$

in which the execution of a loop basically means to unfold it. The reactivity property of DSL would thus be lost by using this rule instead of rule 7.

3.6.2 Packed Drop

Let us consider a possible variant of the semantics in which the dropped scripts are grouped by destination. The idea is that, instead of dropping one after the other several scripts destinated to the same site, one drops the parallel composition of these scripts, in one unique drop action. This reduces the asynchrony of sites execution and thus makes the reasoning about programs easier. To model this variant, we first define the *pack* function which takes a multiset of dropped scripts and returns the set which is the compact version of it:

- $pack\ (D) = D$ if $\forall d_1 = (site_1, s_1) \in D, \forall d_2 = (site_2, s_2) \in D$, $d_1 \neq d_2$ implies $site_1 \neq site_2$

- $pack\ (D \uplus \{(site, s_1), (site, s_2)\}) = pack\ (D \uplus \{(site, s_1 \parallel s_2)\})$

The site execution rule 20 becomes:

$$\frac{s \Rightarrow s', D}{\{..., site : s, ...\} \mapsto \{..., site : s', ...\} \uplus pack\ (D)} \quad (20')$$

Note that the two `drop` scripts of 3.5.3 become equivalent in this variant of the semantics. To implement the variant, dropped scripts have to be stored, up to the end of the current instant, before being compacted and actually sent to remote sites.

3.6.3 Preemption Operator

The basic assumption of the model resides in the couple of rules 12 and 13 which state that the body of a `watching` instruction is executed in both cases of presence and of absence of e. The alternative proposed by Esterel, called "strong preemption", corresponds to the two following rules:

$$\frac{e \in P}{P \vdash \texttt{do}\ s\ \texttt{watching}\ e \xrightarrow{tt} \texttt{nothing}, \emptyset, \emptyset} \quad (12')$$

$$\frac{e \notin P \quad P \vdash s \xrightarrow{b} s', G, D}{P \vdash \texttt{do}\ s\ \texttt{watching}\ e \xrightarrow{b} \texttt{do}\ s'\ \texttt{watching}\ e, G, D} \quad (13')$$

With these rules, the body is immediately executed *in absence* of e (rule 13′), and it is not executed at all when e is present (rule 12′). One thus has an immediate reaction to the absence of e, which introduces "causality cycles" (e.g. `do generate` e `watching` e). Causality cycles are a major obstacle to the introduction of dynamic creation in Esterel. It is thus clear that strong preemption cannot be, in a way or another, introduced coherently in DSL.

We could have replaced rule 11 by the following:

$$\frac{P \vdash s \xrightarrow{tt} s', G, D}{P \vdash \texttt{do}\ s\ \texttt{watching}\ e \xrightarrow{ff} \texttt{nothing}, G, D} \quad (11')$$

This alternative rule gives a more uniform treatment of the preemption operator that actually would *never* terminate instantaneously. However, we prefer to keep on rule 11 because it entails the following intuitive invariant: `do` s `watching` e is strictly equivalent to t if e is never present. This invariant would be violated with the alternative rule 11′.

4 FunLoft Variant

In the FunLoft variant of DSL, a script is first translated into an instruction of the type `instruction_t` defined in FunLoft, before being compiled by the FunLoft compiler. The definition of FunLoft insures the reactivity and memory protection properties of the compiled code (actually, the static checks for bounded resource consumption are switched-off in the FunLoft compiler, but the remaining checks are sufficient to insure reactivity and memory protection).

The translation has the following characteristics:

- The notion of an instant is re-build: an instant of a script is made of several micro-steps of the target FunLoft program, each micro-step corresponding actually to one instant of the translated FunLoft program.
- Events are designed by strings. A hashtable (of the type `aa_t`) associating strings to events is available on each site.
- The generation of a DSL event is sustained during the following micro-steps, up to the end of instant.
- A valued event is used to deal with the dynamic adding of new scripts in a site (it has type `(instruction_t) event_t`).

The basic data structures defining the three recursive types `instruction_t`, `site_t`, and `engine_t`, the API for instructions, and the evaluation of scripts are defined in Annex B.

4.1 Dynamic Adding of Instructions

A special event is associated to each site, used to add the scripts dropped in the site. The module `dynamic` awaits this event and collects its associated values using the `get_all_values` instruction of FunLoft; the collected instructions are processed by calling the function `incorporate`, defined below. The code of `dynamic` is:

```
let module dynamic (eng) =
  let inst_list = ref Nil_list in
    loop
      let add = Engine.add (eng) in
        begin
          await add;
          get_all_values add in inst_list;
          incorporate (eng,!inst_list);
```

```
        generate Engine.wakeup (eng);
        continue_instant (eng);
      end
```

The `continue_instant` function just sets the flag `move` of the engine:

```
let continue_instant (eng) =
   Engine.move (eng) := true
```

The function `incorporate` is recursively defined and the FunLoft compiler checks that it always terminates:

```
let incorporate (eng,inst_list) =
  match inst_list with
    Nil_list do ()
  | Cons_list (head,tail) do
      begin
        thread evaluate (eng,head,event);
        incorporate (eng,tail);
      end
  end
```

To drop an instruction in a site, basically means to generate the special event of the site engine with the instruction as value:

```
let module send_to (site,inst) =
  link Site.sched (site) do
    let engine = !Site.rengine (site) in
      generate Engine.add (engine) with inst
```

4.2 Reactive Engine

The reactive engine of a site basically sustains the generated events and decides when intants are terminated. Each time an event is generated, it is stored in the list `sustain` of the engine, in order to be regenerated at each micro-step, up to the end of instant. Moreover, the `move` flag of the engine is set (by calling `continue_instant`) to resume execution of scripts awaiting the event, if there are such scripts:

```
let dsl_generate (eng,evt) =
  let e = event_lookup (eng,evt) in
    begin
      generate e;
      let s = Engine.sustain (eng) in
        s := Cons_list (e,!s);
      continue_instant (eng);
      generate Engine.wakeup (eng);
    end
```

The algorithm of the engine is the following: micro-steps are executed cyclically while the `move` flag is set; when the `move` flag has not been set by the last micro-step, the `pre_eoi` flag is set to let `watching` statements proceed, in case of preemption. Indeed, in this case, a `watching` has to let its body react, in order to choose safely between rules 11 and 12. Cyclic execution is then resumed as previously, while there are new moves. The end of the current DSL instant is decided when the setting of `pre_eoi` does not produce any new move; in this case the `eoi` flag is set, to indicate the end of the current instant. This algorithm corresponds to the following code:

```
let module react (eng) =
  let move = Engine.move (eng) in
    loop
      begin
        move := false;
        sustain_all (eng);
        cooperate;
        if not !move then
          begin
            generate Engine.pre_eoi (eng);
            cooperate;
            if not !move then
              begin
                close_instant (eng);
                return;
              end
            end
          end
        end
      end
```

The `sustain_all` function maintains the generated events during the next micro-steps, up to the end of instant. The `close_instant` function generates `eoi`, stops the sustainment of generated events, and increments the instant counter:

```
let close_instant (eng,trace) =
  let instant = Engine.instant (eng) in
  begin
    generate Engine.eoi (eng);
    Engine.sustain (eng) := Nil_list;
    instant++;
 end
```

Note that the flag `pre_eoi` becomes useless if we replace the rule 11 by the alternative rule 11′. The alternative rule would thus simplify the implementation in FunLoft (however, this would not be the case for the other variants of DSL).

4.3 Functions and Tasks

Functions are called using the `Call` instruction. Functions and parameters are represented as character strings. The function `call_dispatch` analyses the first string passed to `Call` and calls the correct function. Functions that are used in a program must always be called through the call dispatcher. Here is an example of definition of it:

```
let call_dispatch (eng,fun,params) =
  if fun = "print_string" then
    let p1 = get_param (params,0) in
      print_string (p1)
  else if fun = "print_int" then
    let p1 = string2int(get_param(params,0)) in
      print_int (p1)
  else if fun = "quit" then
    quit (0)
  else
    warning ("unknown call")
```

Tasks are launched in sites with the `Launch` instruction. As with functions, tasks and parameters are represented as character strings. The module `task_dispatch` analyses the first string and launches the appropriate thread. An event given as parameter is generated at the end of the task execution. Here is an example of a task dispatcher:

```
let module task_dispatch (eng,task,params,evt) =
  begin
    if task = "print_getchar" then
      run print_getchar ()
    else if task = "getchar" then
      run getchar ()
    ...
    generate evt;
  end
```

Note that the compiler checks that non-cooperative FunLoft functions are always called while unlinked. Here is an example of use of the non-cooperative FunLoft function `fl_getchar` (which basically corresponds to the `getchar` function of C):

```
let getchar_result = ref ' '

let module getchar () =
  let loc = local ref ' ' in
    begin
      unlink loc := fl_getchar ();
      link main_scheduler do
        getchar_result := !loc;
    end
```

The compiler complains if the `unlink` statement is omitted. Note the use of a local reference to store the character read; an intermediate local reference is mandatory because it is not possible to access the global reference `getchar_result` while unlinked (otherwise, data-races while accessing it could occur).

Note that the executing engine is given as parameter to `call_dispatch` and `task_dispatch`; this allows the function `dsl_generate` to be called by functions and tasks. Communication through events can occur by this means from the host level, to the orchestration level.

4.4 Instantaneous Loops

The FunLoft compiler statically checks for the absence of instantaneous loops; more precisely, it rejects a program in which a loop body has the possibility to terminate instantly. In the present context, this means that there is no possibility for an instruction to cycle during the same micro-step. According to the semantics of DSL, loops that would cycle during the same (DSL) instant are dynamically detected and "corrected"; this is for example the case with:

```
Loop (Nothing)
```

The implementation proceeds as follows: the DSL instant is stored when the body of a loop starts to execute and the system checks that the instant when the body terminates is different from the stored instant. When the two instants are the same (the body thus terminates instantly), the system forces the body to wait for the next DSL instant (it is like if a `cooperate` statement is dynamically inserted at the end of the body when it terminates instantly).

4.5 Static Checks

The static checks performed by the compiler are the ones of FunLoft, excepted those insuring that the consumption of resources is bounded (actually, the execution engine defined does not run in bounded memory, basically because new scripts can always be dropped dynamically in sites).

4.5.1 Reactivity

Basically, the reactivity property comes from the fact that there is no way for a script to prevent the execution of the other scripts by exhausting the CPU. More precisely:

- It is not possible to define recursive scripts in DSL. A script can only launch an already defined task.
- Recursive tasks (recursive FunLoft modules) are allowed because they are needed by the implementation in FunLoft; however, execution of a launched task does not begin immediately, but at the next DSL instant; thus, there is no possibility for a recursively defined task to cycle forever during the same micro-step. Hence, no task could cycle forever during the same DSL instant.
- Functions are proved to always terminate. An example of correct function is:

```
let length(l) =
  match l with Nil_list -> 0
        | Cons_list (h,t) -> 1 + length(t)
  end
```

On the contrary, the following function is rejected:

```
let f(l) =
  match l with Nil_list -> 0
        | default -> f (l)
```

4.5.2 Memory Protection

Each site (scheduler) has its own memory, which is protected from accesses by instructions run by the other sites. As a consequence, no data-race can occur from the parallel execution of two scripts run on two distinct sites (thus, run by two distinct native threads).

Moreover, tasks may have a local private memory and the system verifies that this memory cannot be accessed by the other tasks. Basically, a task is forbidden to store one of its private reference into a public reference accessible by the other tasks.

To illustrate the memory protection technique, let us consider the following task dispatcher:

```
let r = ref 0

let module task_dispatch (eng,task,params,evt) =
  if task = "tst1" then
    thread tst1 (r)
  else if task = "tst2" then
    thread tst2 (r)
  ...
```

An error is detected if both tasks `tst1` and `tst2` access the reference `r` while on different sites; indeed, in this case, there could be a data-race while accessing `r`. A way to fix the bug is to force the two tasks to be in the same site:

```
let module task_dispatch (eng,task,params,evt) =
  if task = "tst1" then
    link site1_sched do thread tst1 (r)
  else if task = "tst2" then
    link site1_sched do thread tst2 (r)
  else
    ...
```

4.6 Execution of Instructions

Let us return to the script of Section 2.5. An equivalent FunLoft program is:

```
#include "dsl3.fl"

let turns = 1000
let consume_value =
  Cons_list ("10000000",Nil_list)

let remote (from,target,msg,done) =
  Drop (target,
    Seq (Print (msg),
    Seq (Call ("consume",consume_value),
         Drop (from,Generate (done)))))

let parallel =
  Repeat (IntConst (turns),
    Seq (
      Par (remote (site1,site2,"0","done0"),
      Par (remote (site1,site3,"1","done1"),
      Par (Await ("done0"),Await ("done1")))),
      Cooperate) )

let module dsl_main () =
  drop_in_site1 (parallel)
```

The include directive of the file `dsl3.fl` defines the types and the constructors used for instructions, and three sites `site1`, `site2`, and `site3`. The function `remote` is called twice by the instruction `parallel`. Finally, the module `dsl_main` is defined; it is actually the program entry point. The body of `dsl_main` simply drops the instruction `parallel` in `site1`.

4.7 Translation in FunLoft

A translator of DSL in FunLoft is implemented; it translates the script of Section 2.5 into the following instruction:

```
Repeat (IntConst (1000),Seq (Par (Par (
Par (Drop (site2,Seq (Seq (Print ("0"),
Call ("consume",Cons_list ("10000000",
Nil_list))),Drop (site1,Generate ("done0")))),
Drop (site3, Seq (Seq (Print ("1"),
Call ("consume",Cons_list ("10000000",
Nil_list))), Drop (site1,Generate ("done1"))))),
Await ("done0")), Await ("done1")),Cooperate))
```

To execute it, one just replaces the definition of `parallel` by this instruction, in the FunLoft program of section 4.6.

5 SugarCubes Variant

The main points of the implementation of DSL in SugarCubes are:

- Sites are coded as reactive machines of the class `machine`, and the `drop` instruction is implemented as the `addProgram` method of `machine`.
- Scripts, except `drop` and `loop` are directly coded by their counterpart in SugarCubes; there is no need to make a distinction between DSL instants and SugarCubes instants.
- The `loop` instruction of DSL is implemented as a parallel instruction of the loop body with the `stop` instruction of SugarCubes.

5.1 Dispatcher

Dispatchers for calls, tasks and wrappers are coded in Java as a unique `JavaAction`. Here is an example of dispatcher:

```
class CallDispatcher implements JavaAction
{
  final String fun;
  final Vector args = new Vector();

  public CallDispatcher (final String fun,
                         final String arg)
  {
    this.fun = fun;
    this.args.add(arg);
  }
  public void execute (ReactiveEngine env)
  {
    if (fun.equals("quit")) {
      System.exit(0);
    }
    if (fun.equals("consume")) {
      int x = 0;
      int n =
        Integer.
          parseInt((String)args.elementAt(0));
      for (int k = 0; k < n; k++) x++;
    }
  }
}
```

5.2 Drop

The `drop` instruction calls the `addProgram` method of the class `Machine` of SugarCubes:

```
class Drop implements JavaAction
{
  final Machine target;
  final Program p;
  public Drop (final Machine target,
                       final Program p)
  {
    this.target = target;
    this.p = p;
  }
  public void execute (ReactiveEngine env)
  {
    target.addProgram (p);
  }
}
```

5.3 Translation in SugarCubes

A translator of DSL into SugarCubes has be implemented which translates scripts into SugarCubes programs. The following class `Parseq` uses the translation of the script of Section 2.5 and defines three sites as instances of the class `machine` of SugarCubes. The three sites are started and the program obttained with the translator is added in the first site. The code is the following:

```
public class Parseq
{
  public static void main (final String[] args)
  {
    final Machine site1 = SC.machine ();
    final Machine site2 = SC.machine ();
    final Machine site3 = SC.machine ();

    final Program p = SC.seq (SC.seq
      (SC.repeat (1000,
      SC.seq (SC.merge (SC.merge (SC.seq (
      SC.action(new Drop(site2,SC.seq (SC.seq
      (SC.print("0"),SC.action( new
      CallDispatcher("consume","10000000"))),
      SC.action(new Drop(site1,
```

```
      SC.generate ("done0")))))),
      SC.action(new Drop(site3,SC.seq
      (SC.seq (SC.print("1"),SC.action( new
      CallDispatcher("consume","10000000"))),
      SC.action(new Drop(site1,
      SC.generate ("done1")))))))),
      SC.await ("done0")),SC.await ("done1")),
      SC.stop())),SC.print("bye")),
      SC.action(new CallDispatcher("quit","0")));

    new Thread (site2).start ();
    new Thread (site3).start ();
    site1.addProgram (p);
    site1.run ();
  }
}
```

It is important to remind that the interference free property, automatically checked by the compiler in the FunLoft variant, is now of the programmers's responsability.

6 ReactiveML Variant

The main points of the implementation of DSL in ReactiveML are:

- Sites are coded in JoCaml. It provides high level constructs to implement the asynchronous communication between sites.
- Each site is executed in a process and the communication between sites is done over sockets. It allows to distribute the sites on different computers.
- Similarly to the FunLoft implementation, scripts are translated into a ReactiveML values of type `script` (that corresponds to the type `instruction_t` in FunLoft). Scripts are not directly translated to ReactiveML constructs as it is done in the SugarCubes implementation because it would require code mobility to implement the `drop` action.
- Scripts, except `drop` and `loop` have a direct counterpart in ReactiveML; there is no need to make a distinction between DSL instants and ReactiveML instants. The identification of the two notions of instants is analogous as in the SugarCubes variant of Section 5.
- The `loop` instruction of DSL is implemented as a parallel instruction of the loop body with the `pause` instruction of ReactiveML.

6.1 Evaluator in ReactiveML

In ReactiveML, a DSL script is represented as a value of type `script` which is similar to the type `instruction_t` in the FunLoft implementation. For example, the script of the Section 2.5 is translated by the compiler into the following value:

```
S_repeat
  (E_const (C_int 1000),
   S_seq
    (S_par (S_par (S_par
      (S_drop ("site2",
         S_seq (S_seq
          (S_print "0",
           S_call ("consume", [C_int 10000000])),
           S_drop ("main", S_generate "done0"))),
       S_drop ("site3",
         S_seq (S_seq
          (S_print "1",
           S_call ("consume", [C_int 10000000])),
           S_drop ("main", S_generate "done1")))),
       S_await "done0"),
       S_await "done1"),
     S_cooperate))
```

The evaluation of a script represented by a value of type `script` is defined by a ReactiveML process `eval_script`. Since the semantics of ReactiveML is very close to the semantics of DSL, the implementation of this process is straightforward. We define here the beginning of the process (the complete implementation and the definition of the type `script` is given in annex C).

```
let rec process eval_script script =
  match script with
  | S_nothing -> ()
  | S_cooperate -> pause
  | S_seq (s1, s2) ->
      run (eval_script s1);
      run (eval_script s2)
  | S_par (s1, s2) ->
      run (eval_script s1) ||
      run (eval_script s2)
  | ...
```

6.2 Dispatcher

Like in the other implementations, function calls are made through a dispatcher that dynamically links the function name with its definition. In the `eval_script` process, function calls are implemented as follows:

```
  | S_call (f_id, vl) ->
      let f = fun_of_fun_id f_id in
      f vl
```

where the function `fun_of_fun_id` is defined with the function `record_fun` as follows:

```
let fun_of_fun_id, record_fun =
  let tbl = Hashtbl.create 7 in
  let fun_of_fun_id f_id =
    try Hashtbl.find tbl f_id
    with Not_found ->
      prerr_endline ("unbound function "^f_id);
      (fun x -> C_unit)
  and record_fun f_id f =
    Hashtbl.add tbl f_id f
  in
  fun_of_fun_id, record_fun
```

The two functions `fun_of_fun_id` and `record_fun` share a common hash table (`tbl`). `fun_of_fun_id` returns the function associated in `tbl` to the string given in argument and `record_fun` allows to fill the hash table.

To do dynamic typing, DSL values are encapsulated in a type `const`:

```
type const =
  | C_unit
  | C_int of int
  | C_bool of bool
  | C_string of string
```

So if we want to register a function that take as input two integers and return a Boolean, we will use the following function:

```
let int_x_int_to_bool f_id f =
  record_fun f_id
    begin function
    | [ C_int n1 ; C_int n2 ] ->
        let b = f n1 n2 in C_bool b
    | _ ->
        prerr_endline "type error";
        C_bool false
    end
```

6.3 Drop

The `drop` instruction is implemented in the process `eval_script` as follows:

```
  | S_drop (site_id, script) ->
      Dsl_drop.put (site_id, script)
```

The `Dsl_drop.put` function is implemented with a join pattern of the JoCaml language:

```
let put, get =
  def put(site_id_x_script) & state(to_drop) =
    reply () to put &
    state(site_id_x_script :: to_drop)
  or get() & state(to_drop) =
    reply to_drop to get &
    state([])
in
spawn state([]);
put, get
```

The functions `put` and `get` share a common list. `put` adds an element to the list and `get` removes all the elements. The two functions can safely modify the list concurrently thanks to their join definition and their communication through the channel `state`.

6.4 Sites

In the ReactiveML implementation of DSL, a site is basically a JoCaml program that repeatedly executes the following three steps:

1. get the scripts dropped by the other sites
2. execute one instant of the DSL program with the new scripts in parallel
3. send the dropped scripts to the other sites.

The first step is implemented with a JoCaml function similar to the function `Dsl_drop.get`. The second step executes one instant of the `eval_script` process. The third step calls `Dsl_drop.get` and dispatches the dropped scripts to the corresponding sites.

7 Bigloo/Scheme Variant of DSL

The main points of the implementation of DSL in Scheme/Bigloo are:

- Sites are coded in Scheme/Bigloo and executed by a native thread.
- Script execution basically follows the semantics rules described in previous section.
- As there is no notion of instant in Scheme/Bigloo, we introduce the notion of reactive machine to implement them.
- In Scheme/Bigloo there is no difference between functions and tasks.

7.1 Site

In our implementation a reactive machine (site) is made of four lists:

- The first one contains the executing scripts.
- The second one contains the waiting scripts.
- The third one contains scripts whose execution is finished for the current instant.
- The last list is the event environment.

We also need a boolean to indicate if it the fix point is reached (`eoi`). A site is thus coded as:

```
(define site
   (list
      (list) (list) (list) (list)
       eoi))
```

The behaviour of the reactive machine is extremely simple: it executes the function `one_step` which defines one instant of DSL, upto the least fixed point. This function returns the new state of the reactive machine. The code of `one_step` is:

```
(define (one_step site)
 (when (event_generated site)
  (set! site (wakeup_waiting_script site)))

 (if (eoi site)
  site
  (let ((res (sem_action
       (first_script site)
       (get_env site)
       (get_eoi site))))

   (if (execution_finished res)
    (one_step
      (list (next_script site)
      (get_waiting_script site)
      (cons (get_script_res res)
            (get_finished_script site))
      (get_env_res res)
      (get_eoi res)))
    (one_step
      (list (next_script site)
      (cons (get_script_res res)
            (get_waiting_script site))
      (get_finished_script site)
      (get_env_res res)
      (get_eoi_res res)))))))
```

First, we check if there are generated events (`event_generated`). If it is the case, we wake-up all the waiting scripts (`wakeup_waiting_script`). Then, we should verify if the fixed point (*eoi*) is already reached. In that case, there is no need to continue; otherwise, we try to execute a new script (`sem_action`). If `execution_finished` returns true, the current script is terminated for the current instant. Otherwise, the script is waiting for an event. In both cases, we pass to the next script after storing the script in the approriated list (waiting script lists or finished scirpts list).

7.2 Functions and Tasks

In Scheme/Bigloo there is no different between function and tasks. In DSL, we require that functions are instantaneous, but tasks can take several instants. In the current implementation, the only way for the user to implement a task is to define a Scheme/Bigloo function executed as a native thread.

7.3 Translation in Bigloo

There are two possibilities in Scheme/Bigloo to execute a script: either it is translated in Scheme/Bigloo, then compiled, and executed as a standard Scheme/Bigloo program; or it is given as input to a toplevel interpretor that analyses the script, translates it in an abstract tree, and run it using a native Scheme/Bigloo thread.

From the script of Section 2.5, the DSL to Bigloo translator produces a code which is exactly the same as the one produced by the FunLoft variant:

```
Repeat (IntConstWrapper (1000),Seq (Par (Par (
Par (Drop (site2,Seq (Seq (Print ("0"),
Call ("consume",Cons_list ("10000000",
Nil_list))),Drop (site1,Generate ("done0")))),
Drop (site3, Seq (Seq (Print ("1"),
Call ("consume",Cons_list ("10000000",
Nil_list))), Drop (site1,Generate ("done1"))))),
Await ("done0")), Await ("done1")),Cooperate))
```

To execute the script, we drop it into the appropriated reactive machine. This execution is descripted in annex D.

8 Experiments

In this section, we execute the script defined in section 2.5 with the four variants of DSL and compare the results.

FunLoft

In FunLoft, the function `consume` called by the script to consume the computing resource can be defined by:

```
let consume_intern (n) =
  let x = local ref 0 in
     repeat n do x++
```

Alternatively, one could define `consume` as an extern C function, introduced in FunLoft by:

```
extern "C" consume : int -> unit
```

The definition of `consume` in C is then:

```
#include "val.h"
value consume (value vn)
{
    long n = val2int (vn);
    long k, x = 0;
    for ( k = 0; k < n; k++ ) x++;
    return val_unit;
}
```

The extern function is much more efficient than the function defined in FunLoft (see below). However, with the extern function, the compiler looses the possibility to detect an error in the C implementation (this is recalled to the user by a message issued at compile time).

The execution machine we use is a dual-core machine[3]. The execution time is obtained with the Unix commande `time`.

Results with the FunLoft variant are shown in Figure 1; on the left, the (intern) FunLoft definition of `consume` is used, while on the right it is defined in C (extern).

FL	intern	extern
real	2m58.616s	0m32.922s
user	5m47.278s	1m4.652s
sys	0m2.516s	0m0.208s

Figure 1: FunLoft variant

SugarCubes

Figure 2 shows the results with the SugarCubes variant (the `consume` method is directly implemented in Java). The counter used by `consume` is both implemented as an integer of type `int` and as a long integer of type `long`. The implementation with `int` is:

```
class FUN_consume implements Fun
{
   public void exec(final String arg){
      int len = Integer.valueOf(arg);
      int x = 0;
      for(int i = 0 ; i < len ;i++) x++;
   }
}
```

SC + JIT	long	int
real	0m30.658	0m1.899s
user	1m7.288s	0m3.692s
sys	0m0.160s	0m0.028

Figure 2: SugarCubes variant with JIT

In Figure 3 is shown the execution time when the JIT of Java is switched off (option `-Xint`).

SC - JIT	long	int
real	6m32.884s	3m34.355s
user	19m5.076s	10m25.387s
sys	0m13.617s	0m5.044s

Figure 3: SugarCubes variant without JIT

ReactiveML

Figure 4 shows the results with the ReactiveML variant (the `consume` method is directly implemented in ReactiveML).

The code of `consume` is:

```
let consume n =
  let x = ref 0 in
  for i = 1 to n do
    x := !x + 1
  done
```

RML	
real	1m27.292s
user	0m39.946s
sys	0m0.584s

Figure 4: ReactiveML variant

[3]machine characteristics: Dell Latitude, Linux 2.6.35 processor Intel Core i5, 2.4 GHz, 4GB of memory

Scheme/Bigloo

The results for the Scheme/Bigloo variant are shown on figure 5. We have considered the two cases, where the counter is implemented as an integer and as a long.

Scheme	long	int
real	47m34.820s	4m40.714s
user	76m53.744s	8m0.614s
sys	4m9.536s	0m4.572s

Figure 5: Scheme/Bigloo variant

Interpretation

With the FunLoft, SugarCubes and Scheme/Bigloo variants, we see that the user time is (more or less) twice the real time, which shows that the two cores are running simultaneously at full speed. The two instances of `consume` are indeed executed in real parallelism by the two cores. The ReactiveML variant does not use the two cores in an optimal way; it seems to be slow down by the presence of JoCaml which introduces a communication overhead.

The efficiency of the SugarCubes variant heavilly depends on the JIT of Java. We see also that it depends on the use of integers instead of longs. Note that in the FunLoft variant, integers are coded as `long long` integers of C.

The dependence on the choice between integers and longs is also clear in the Scheme/Bigloo variant. Note that in this variant, no optimization is performed in the reactive engine; optimization were not the focus and the efficiency of this variant could be clearly improved in that matter.

9 Related Work

9.1 Esterel

DSL differs from Esterel [7] in two main aspects: first, it allows dynamicity. The main idea which makes this possible is to forbid the immediate reaction to absence of signals (called events, in DSL); this is the basis of reactive programming [4]. Second, DSL introduces asynchrony through the notion of site. Asynchrony is not present in Esterel[4], thus the issue of memory protection against asynchronous concurrent accesses is not addressed.

[4]except in the first versions of the language, in which the notion of an `exec` task was defined.

9.2 Reactive Scripts

A proposal very close to the one presented here, called Reactive Scripts, has been made in [13]. Reactive Scripts define migration means based on RPC. Reactive Scripts is defined independently of the precise definition of atomic actions (function calls). Thus, the main issues considered in DSL were not considered with Reactive Scripts: termination of instants and absence of interferences. Reactive Scripts however proposes the notion of an object, which could be interesting to introduce in DSL (this notion basically relies on a `control` primitive that is currently not present in DSL).

9.3 FunLoft

DSL is of course strongly linked to FunLoft [12], which is based on the work of [15]. FunLoft adopts a thread-based approach, while DSL uses a parallel operator. Actually, the implementation of DSL in FunLoft shows how a parallel primitive can be implemented with threads. Moreover, resources consumption is checked for boundness by the FunLoft compiler; this feature is switched-off for the DSL compiler. Resource control is actually totally absent from DSL.

9.4 SugarCubes

A correct program in DSL should be proved to be reactive (i.e. the flow of instants is infinite) and free from data-races. These properties should be valid in SugarCubes [14] (and in the related framework Junior [16]), but they are not checked by the implementations. In SugarCubes, instantaneous loops are dealt with, but non-terminating atomic functions are not. However, SugarCubes offers a much more rich set of instructions, in particular for distributed programming; it is also object-based unlike DSL.

9.5 ReactiveML

The language ReactiveML [17] gives a possibility to execute scripts which are standard ReactiveML programs, compiled "on-the-fly". This feature introduces high flexibility in programming. Based on ML, ReactiveML is safe (no crash can occur at run-time). However, the termination of instants is not checked by the compiler. Moreover, ReactiveML, as ML, is

not presently able to benefit from the real parallelism offered by multiprocessor architectures.

9.6 S-π Calculus

The theoretical approach of the S-π Calculus [5] is related to DSL, as it is based on the same synchronous model. The focus in S-π Calculus is resource control: it provides a way to checks that resources are *polynomially* bounded. This is not the case in DSL which does not consider resource control at all.

9.7 Cooperative Multithreading

The use of static analysis techniques in the context of cooperative multithreading is considered in several recent papers [8], [19]. The case of synchronous parallelism is not however covered in these works.

9.8 ORC

ORC [18] is an orchestration language dedicated to the Web. The objectives of ORC are close to the ones of DSL. We plan to compare ORC and DSL and to implement ORC in DSL.

10 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: how to be sure that the program is indeed reactive? how to execute it efficiently on a multicore architecture? how to be sure that there is no harmful interference between parallel computations (e.g. data-races)? Our proposal gives answers to these questions. In the FunLoft variant of DSL, the basic properties of DSL (reactivity and data-race freeness) are statically checked. However, sites cannot be run by different machines (they are confined to the cores of the same machine). In the ReactiveML, SugarCubes, or Scheme/Bigloo variants, the basic properties of DSL are not automatically checked but are of the programmer's responsibility. The ReactiveML and SugarCubes variants give access to distribution of sites across the network (using JoCaml for ReactiveML, and Java RMI for SugarCubes). Note that in all the variants of DSL, sites can be run in full parallelism (either cores or processors).

We envision the following tracks for future work:

- Introduction of means for distribution of sites in the FunLoft version.
- Implementation of a communication protocol, based on Hop, allowing the different variants of DSL to run as a unique distributed program.
- Design and implementation of a top-level for DSL adapted to each version.
- Extension of DSL to allow the dynamic mapping of programs onto cores, in order to maximise the use of cores at execution time.

References

[1] Bigloo. http://www-sop.inria.fr/indes/fp/Bigloo.

[2] Hop. http://hop.inria.fr.

[3] JoCaml. http://jocaml.inria.fr.

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

[5] R. M. Amadio. A Synchronous pi-Calculus. *Journal of Information and Computation*, 205(9):1470–1490, 2007.

[6] A. Benveniste and G. Berry. The Synchronous Approach to Reactive and Real-Time System. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[7] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[8] G. Boudol. Fair Cooperative Multithreading. In *CONCUR 2007*, pages 272–286, 2007.

[9] F. Boussinot. Reactive C: An Extension of C to Program Reactive Systems. *Software Practice and Experience*, 21(4):401–428, april 1991.

[10] F. Boussinot. FairThreads: Mixing Cooperative and Preemptive Threads in C. *Concurrency and Computation: Practice and Experience*, vol 18 pp 445-469, 2006.

[11] F. Boussinot. *Safe Reactive Programming. The FunLoft Language*. Lambert Academic Pub., 2010.

[12] F. Boussinot and F. Dabrowski. Safe Reactive Programming: the FunLoft Proposal. In *Proc. of MULTIPROG – First Workshop on Programmability Issues for Multi-Core Computers*. Göteborg, January 2008.

[13] F. Boussinot and L. Hazard. Reactive Scripts. In *Proc. International Conference on Real-Time Computing Systems and Applications, RTCSA'96*, pages 267–274, Seoul, October 1996.

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

[15] F. Dabrowski. *Programmation réactive synchrone - Langage et contrôle des ressources*. PhD thesis, Université Paris 7, 2007.

[16] L. Hazard, J-F. Susini, and F. Boussinot. The Junior Reactive Kernel. *Inria Research Report, RR-3732*, july 1999.

[17] L. Mandel and M. Pouzet. ReactiveML, a Reactive Extension to ML. In *ACM International conference on Principles and Practice of Declarative Programming (PPDP'05)*, Lisbon, Portugal, July 2005.

[18] J. Misra and W. Cook. Computation orchestration. *Software and Systems Modeling*, 6:83–110, 2007. 10.1007/s10270-006-0012-1.

[19] Jaeheon Yi and C. Flanagan. Effects for Cooperable and Serializable Threads. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation, TLDI 2010*, 2010.

A DSL Grammar

This appendix describes the grammar of DSL expressed in a YACC format.

```
script:
   basic.script
|  script ';' script
|  script '||' script
|  '(' script ')'

basic.script:
  NOTHING
| COOPERATE
| if
| loop
| repeat
| generate
| await
| watching
| call
| launch
| drop

if:
  IF bool.test THEN script ELSE script END
| IF bool.test THEN script END

loop:
  LOOP script END
| WHILE bool.test DO script END

repeat:
  REPEAT int.val DO script END

generate: GENERATE event

await: AWAIT event

watching: WATCHING event DO script END

call:
  IDENTIFIER '(' value.list ')'

launch:
  LAUNCH IDENTIFIER '(' value.list ')'

drop: DROP script IN site

site: IDENTIFIER

bool.test: BOOLEAN | call

int.val: NUMBER | call

event: IDENTIFIER

value:
  NUMBER | BOOLEAN | STRING

value.list:
  /* empty */
| value
| value ',' value.list
```

B Evaluation in FunLoft

The basic data structures define the three recursive types `instruction_t`, `site_t`, and `engine_t`:

```
type
  instruction_t =
    _Nothing
  | _Print of string
  | _Call of string * string list
  | _Launch of string * string list
  | _Cooperate
  | _Seq of instruction_t * instruction_t
                          * ref bool
                          * ref thread_t
  | _If of bool_wrapper_t * instruction_t
                          * instruction_t
                          * ref bool
```

```
                          * ref thread_t
  | _Par of instruction_t * instruction_t
                          * ref thread_t
                          * ref thread_t
  | _Loop of instruction_t * ref thread_t
  | _Repeat of int_wrapper_t * instruction_t
                             * ref thread_t
  | _Generate of string
  | _Await of string
  | _Watching of string * instruction_t
                        * ref thread_t
                        * ref thread_t
  | _Drop of ref site_t * instruction_t
and
 site_t =
    Site of
        rengine: ref engine_t
      * sched: scheduler_t
and
 engine_t =
    Engine of
        eoi: (unit) event_t
      * pre_eoi: (unit) event_t
      * move: ref bool
      * instant: ref int
      * sustain: ref[array_size] (unit) event_t
      * sustain_count:ref int
      * add: (instruction_t) event_t
      * event_env: ((unit) event_t) aa_t
      * wakeup: (unit) event_t
      * next: ref bool
```

The API for scripts is:

```
let rn () = ref null_thread
let Nothing = _Nothing
let Print (s) = _Print (s)
let Cooperate = _Cooperate
let Par (i1,i2) = _Par (i1,i2,rn(),rn())
let If (e,i1,i2) = _If (e,i1,i2,ref true,rn())
let Seq (i1,i2) = _Seq (i1,i2,ref true,rn())
let Loop (i) = _Loop (i,rn())
let Repeat (n,i) = _Repeat (n,i,rn())
let Generate (e) = _Generate (e)
let Await (e) = _Await (e)
let Watching (e,i) = _Watching (e,i,rn(),rn())
let Call (fun,params) = _Call (fun,params)
let Launch (task,params) = _Launch (task,params)
let Drop (site,i) = _Drop (site,i)
```

The evaluation of atomic scripts is defined by the following function:

```
let eval_atomically (eng,inst) =
  match inst with
    _Nothing do ()
 |  _Print (s) do
      begin print_string (s); flush (); end
 |  _Call (fun,params) do
      call_dispatch (eng,fun,params)
 |  _Generate (evt) do
      dsl_generate (eng,evt)
 |  _Seq (i1,i2,_,_) do
      begin
        eval_atomically (eng,i1);
        eval_atomically (eng,i2);
      end
 |  _If (w,i1,i2,_,_) do
      if bool_wrapper (w) then
        eval_atomically (eng,i1)
      else
        eval_atomically (eng,i2)
 |  _Par (i1,i2,_,_) do
      begin
        eval_atomically (eng,i1);
        eval_atomically (eng,i2);
      end
 |  _Repeat (w,body,_) do
      repeat int_wrapper (w) do
        eval_atomically (eng,body)
 |
    default do warning ("internal error")
```

For the non-atomic scripts, one first defines a synchronisation barrier module, needed by the parallel operator:

```
let module barrier (engine,reach,go) =
  let threshold = 2 in
  let count = local ref 0 in
  let b = local ref true in
    while !b do
      begin
        await reach;
        continue_instant (engine);
        for_all_values reach with _ do count++;
        if !count = threshold then
          begin
            continue_instant (engine);
            generate go;
            b := false;
          end
        end;
     end
```

The following macro is used to shorten the code:

```
#define EVAL(r,inst,term)\
begin\
  continue_instant (eng);\
  r := thread evaluate (eng,inst,term);\
end
```

The evaluation of scripts is defined by the following module:

```
let module evaluate (eng,inst,term) =
  let eoi = Engine.eoi (eng) in
    begin
      Engine.next (eng) := true;
      if is_atom (inst) then
        begin
          eval_atomically (eng,inst);
          generate term;
        end
      else
        match inst with
         _Launch (t,p) do
           if not known_task (t) then
             generate term
           else
             let e = event in
               begin
                 await eoi;
                 continue_instant (eng);
                 thread task_dispatch (eng,t,p,e);
                 await e;
                 generate Engine.wakeup (eng);
                 generate term;
               end
      |
        _Cooperate do
          begin
            await eoi;
            generate term;
          end
      |
        _Seq (i1,i2,b,r) do
          let e = event in
            begin
              b := true;
              EVAL (r,i1,e);
              await e;
              b := false;
              EVAL (r,i2,term);
            end
      |
        _If (w,i1,i2,b,r) do
          begin
            b :=  bool_wrapper (w);
            if !b then
              EVAL (r,i1,term)
            else
              EVAL (r,i2,term);
          end
      |
        _Par (i1,i2,r1,r2) do
          let e = event in
            begin
              thread barrier (eng,e,term);
              EVAL (r1,i1,e);
              EVAL (r2,i2,e);
            end
      |
        _Loop (body,r) do
          let e = event in
          let k = Engine.instant (eng) in
            loop
              let initial = !k in
                begin
                  EVAL (r,body,e);
                  await e;
                  continue_instant (eng);
                  cooperate;
                  if initial = !k then
                    await eoi
                  end
                end
      |
        _Repeat (w,body,r) do
          let e = event in
            begin
              repeat int_wrapper (w) do
                begin
                  EVAL (r,body,e);
                  await e;
                  continue_instant (eng);
                  cooperate;
                end;
              generate term;
            end
      |
        _Await (evt) do
          let e = get_event (eng,evt) in
            begin
              await e;
              generate term;
            end
      |
        _Watching (evt,body,b_th,w_th) do
          let b_evt = event in
          let e = event in
          let r = ref false in
            begin
              thread await_mod (b_evt,e,r);
              EVAL (b_th,body,b_evt);
              EVAL (w_th,Await (evt),e);
              await e;
              await Engine.pre_eoi (eng);
              killing (eng,inst);
              if !r then
                begin
                  generate term;
                  continue_instant (eng);
                end
              else
                begin
                  await eoi;
```

```
              generate term;
            end
        end
    |
      _Drop (site,script) do
        begin
          run send_to (site,script);
          generate term;
        end
    |
      default do ()
  end
```

C Evaluation in ReactiveML

The DSL programs are represented with the following abstract syntax tree in ReactiveML[5]:

```
type fun_id = string
type module_id = string
type event_id = string
type site_id = string

type const =
  | C_unit
  | C_int of int
  | C_bool of bool
  | C_string of string

type expr =
  | E_const of const
  | E_call of fun_id * expr list

type script =
  | S_nothing
  | S_cooperate
  | S_seq of script * script
  | S_par of script * script
  | S_if of expr * script * script
  | S_loop of script
  | S_repeat of expr * script
  | S_generate of event_id
  | S_await of event_id
  | S_watching of event_id * script
  | S_call of fun_id * const list
  | S_launch of module_id * const list
  | S_drop of site_id * script
```

Script are interpreted as follows:

```
let rec eval_expr expr =
  match expr with
  | E_const c -> c
  | E_call (f_id, vl) ->
      let f = fun_of_fun_id f_id in
      f vl

and eval_expr_bool e =
  match eval_expr e with
  | C_bool b -> b
  | _ ->
      prerr_endline "Type error";
      false

and eval_expr_int e =
  match eval_expr e with
  | C_int n -> n
  | _ ->
      prerr_endline "Type error";
      0

let rec process eval_script script =
  match script with
  | S_nothing -> ()
  | S_cooperate -> pause
  | S_seq (s1, s2) ->
      run (eval_script s1);
      run (eval_script s2)
  | S_par (s1, s2) ->
      run (eval_script s1) ||
      run (eval_script s2)
  | S_if (e, s1 ,s2) ->
      if eval_expr_bool e then
        run (eval_script s1)
      else run (eval_script s2)
  | S_loop s ->
      loop
        pause ||
        run (eval_script s)
      end
  | S_repeat (e, s) ->
      let n = eval_expr_int e in
      for i = 1 to n do
        run (eval_script s)
      done
  | S_generate ev_id ->
      let ev = event_of_event_id ev_id in
      emit ev
  | S_await ev_id ->
      let ev = event_of_event_id ev_id in
      await immediate ev
  | S_watching (ev_id, s) ->
      let ev = event_of_event_id ev_id in
      do
        run (eval_script s)
      until ev done
  | S_call (f_id, vl) ->
      let f = fun_of_fun_id f_id in
      ignore (f vl)
  | S_launch (m_id, vl) ->
      let m = module_of_module_id m_id in
```

[5] a `while` instruction is added to DSL.

```
    run (m vl)
| S_drop (site_id, script) ->
    Dsl_drop.put (site_id, script)
```

D Evaluation in Scheme/Bigloo

The basic data structure in Scheme/Bigloo is the lists. A site principaly is a set of list as we explained it before (section 7). Each script is identified by an integer and its arguements by a list, if there is no arguments it will be an empty list. The evaluation of scripts is defiend by the fallowing function, where `sem_action` is a function which evaluate a script in exact semantics rules; all the function start with `get_` try to acquire an object, for example: `get_env_res` get the current environement for the result that we obtained before.

```
(define (sem_action s env eoi)
 (case (car s)
   ;; b = true(0) | falase(1) | Bot (2)
   ((13);;PRINT
    (print (first_elm s))
    (list (list NOTHING) env '() #f 0))
   ((0);;NOTHING
    (list (list NOTHING) env (list) #f 0))
   ((1);;COOP
    (list (list NOTHING) env (list) #f 1))
   ((2);;IF
    (if (string=? (first_elm s) "true")
        (sem_action (get_true s) env eoi)
        (sem_action (get_false s) env eoi)))
   ((3);;SEQ
    (let ((res (sem_action (first_elm s) env eoi)))
       (if (execution_succed res)
           (let ((res2
                   (sem_action (get_second s)
                               (get_env_res res)
                               eoi)))
              (list (car res2)
                    (get_env_res res2)
                    (get_drop_res res2)
                    (get_move_res res2)
                    (get_state_res res2)))
           (list (list SEQ (get_script_res res)
                           (car (cddr s)))
                 (get_env_res res)
                 (get_drop_res res)
                 (get_move_res res)
                 (get_state_res res)))))
   ((4);;PARA
    (let* ((s1 (sem_action (first_elm s) env eoi))
           (s2 (sem_action (get_second s)
                           (get_env_res s1)
                           (get_state_res s1))))
      (list (if (equal? (car s1)
                        (list NOTHING))
                (car s2)
                (if (equal? (car s1)
                            (list NOTHING))
                    (car s1)
                    (list PARA
                          (car s1)
                          (car s2))))
            (get_env_res s2)
            (get_drop_res s2)
            (or (get_move_res s1)
                (get_move_res s2))
            (set_state (get_state_res s2)
                       (get_state_res s1)))))
   ((5);;LOOP
    (let ((s1 (sem_action (first_elm s) env eoi)))
       (list (list SEQ (car s1) s)
             (get_env_res s1)
             (get_drop_res s1)
             (get_move_res s1)
             (get_state_res s1))))
   ((6);;REPEAT
    (let ((count (first_elm s)))
       (if (string? count)
           (let ((proc (funcall-resolve count)))
              (if proc
                  (set! count (proc))
                  (set! count
                        (string->number count)))))
       (sem_action
         (list SEQ
               (create_repeat (get_second s)
                              count)
               (get_second s))
          env
          eoi)))
   ((7);;GENERATE
    (if (member (string->symbol (get_evt s)) env)
        (list (list NOTHING) env (list) #t 0)
        (list (list NOTHING)
              (cons (string->symbol (get_evt s))
                    env)
              (list) #t 0)))
   ((8);;AWAIT
    (if (member (string->symbol (get_evt s)) env)
        (list (list NOTHING) env (list) #f 0)
        (list s env (list) #f 2)))
   ((9);;WATCHING
    (let ((s1 (sem_action (first_elm s) env eoi)))
       (if (execution_succed s1)
           (list (list NOTHING) (get_env_res s1)
                                (get_drop_res s1)
                                (get_move_res s1)
```

```
                          0)
        (if (member (get_evt_watching s) env)
            (list (list NOTHING)
                  (get_env_res s1)
                  (get_drop_res s1)
                  (get_move_res s1)
                  (get_state_res s1))
            (list (list WATCHING
                        (car s1)
                        (get_evt s))
                  (get_env_res s1)
                  (get_drop_res s1)
                  (get_move_res s1)
                  (get_state_res s1))))))
((10);;CALL
 (let ((proc (first_elm s))
       (args (get_second s)))
    (when (procedure? proc) (apply proc args))
    (list (list NOTHING) env (list) #f 0)))
((11);;LAUNCH
 (let ((proc (first_elm s))
       (args (get_second s)))
    (when (procedure? proc) (apply proc args))
    (list (list NOTHING) env (list) #f 1)))
((12);;DROP
 (let ((script (get_dropped_script s))
       (site
        (site-resolve
         (get_dropped_site s))))
    (when site
     (if (site-is-remote? site)
         (cond-expand
            (hop-server
             (let ((host
                    (site-remote-host site))
                   (port
                    (site-remote-port site))
                   (remote
                    (site-remote-site site)))
                (with-hop
                   (DSL/drop
                    :script script
                    :site remote)
                   :host host
                   :port port
                   :sync #f
                   :user "admin"
                   :password "admin"
                   (lambda (r) #t))
             )
            (bigloo
             (list (list NOTHING)
                   env
                   script
                   #f
                   0)))
            (site 'drop! script)))
       (list (list NOTHING)
             env
             (list)
             #t
             0)))
   (else
    (error "sem_action"
           "Illegal instruction"
           s))))
```

Contents