ULM:⋆ A Core Programming Model for Global Computing
(Extended Abstract)

Gérard Boudol

INRIA Sophia Antipolis

Abstract. We propose a programming 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. To design the language, we use ideas and programming constructs from the synchronous programming style, that allow us to deal with the suspensive character of some operations, and to program reactive behaviour. We also introduce constructs for programming mobile agents, that move together with their state, which consists of a control stack and a store. This makes the access to references also potentially suspensive.

1 Introduction

According to the *global computing* vision, "*In the future most objects that we work with will be equipped with processors and embedded software* [...] *Many of these objects will be able to communicate with each other and to interact with the environment*" [15]. In a word: processors everywhere. This is gradually becoming a reality: besides computers and laptops, or personal assistants, nowadays telephones and smart cards also are equipped with processors, as well as cars, planes, audio and video devices, household appliances, etc. Then a question is: will we be able to exploit such a highly distributed computing power? How will we compute in such a context? There are clearly many new features to deal with, and many problems to address in order to be able to "compute globally". As pointed out by Cardelli [8,9], the global computing context introduces new observables, "*so radically different from the current computational norm that they amount to a new model of computation*".

In this paper we address one specific aspect of this global computing vision, namely the fact that "*The availability and responsiveness of resources* [...] *are unpredictable and difficult to control*" [15]. This is indeed something that everybody can already experiment while browsing the web: quite often we observe that an URL appears inaccessible, for various reasons which are not always clearly identified – failure of a node, insufficient bandwidth, congestion, transient

⋆ ULM is the acronym for the french words "*Un Langage pour la Mobilité*". Work partially supported by the MIKADO project of the IST-FET Global Computing Initiative, and the CRISS project of the ACI Sécurité Informatique.

D.A. Schmidt (Ed.): ESOP 2004, LNCS 2986, pp. 234–248, 2004.

disconnection, impossibility to cross a firewall, etc. We would like to have some ability to react and take actions to bypass these obstacles. This is what Cardelli provided for us, with service combinators for scripting the activity of web browsing [10]. Cardelli's combinators are quite specific, however: only the time and rate of transmission can be used as parameters for a reaction. We would like to have programming constructs to deal with more general situations where the availability and responsiveness of resources is not guaranteed. This is not provided by traditional computing models. For instance, in a LAN, the fact that some file is unreachable is considered as an error – unless this is prescribed by some security policy –, and a latency period that exceeds normal response times and communication delays is the sign of a failure. In global computing, by contrast, trying to access something which is, maybe temporarily, absent or unavailable appears to be the rule, rather than the exception. Such a failure should not be regarded as a fatal error. We should rather have means to detect and react to the lack of something.

A typical scenario where these partial failures occur is the one of *mobile code*, which is thought of as a general technique to cope with the new observable features of the global computing context [9,13]. For instance, a mobile code (which may be code embedded in a mobile device) may fail to link with some specific libraries it might need at a visited site, if the site does not provide them. In this case, a default behaviour should be triggered. Conversely, the mobile code may be unreachable from a site which is supposed to maintain connections with it. In such a case, the user who has delegated the mobile agent, or who tries to contact a remote mobile device, for instance, has to wait and, after a while, take some decisions.

The model we shall use to deal with absence, waiting, and reactions is basically that of *synchronous programming* [2,14]. This does not mean that, in our approach, something like the web is to be regarded as a synchronous system. We rather take the view that the global computing world is a (wide) GALS, that is, a "Globally Asynchronous, Locally Synchronous" network. Only at a local level does a notion of time make sense, which serves as the basis for taking decisions to react. As regards the way to program reactions, we follow the ESTEREL imperative (or control-oriented) style [3], which seems more appropriate for our purpose than the LUSTRE data flow style [14]. However, in ESTEREL a program is not always "causally correct", and a static analysis is needed to check correctness. This checking is not compositional, and more specifically, the parallel composition of causally correct programs is not always correct. From the point of view of global computing, this is unfortunate, because this makes it impossible to dynamically add new components, like mobile agents, to a system. Therefore, we shall use the *reactive* variant of ESTEREL designed by Boussinot [5,6], which is slightly less expressive, but is well suited for dynamically evolving concurrent systems. The main ingredients of the synchronous/reactive model, in its control-oriented incarnation – as we see it – are the following:

- *broadcast signals.* Program components react according to the absence or presence of signals, by computing, and emitting signals. These signals are broadcast to all the components of a synchronous area, which is a limited area where a local control over all the components exists.
- *suspension.* Program components may be in a suspended state, either because of the presence of a signal, or because they are waiting for a signal which is absent at the moment.
- *preemption.* There are means to abort the execution of a program component, depending on the presence or absence of a signal.
- *instants.* These are successive periods of the execution of the program, where the signals are consistently seen as present or absent by all the components.

Suspension is what we need to signify the lack of something, and preemption allows us to program reactive behaviours, aborting suspended parts of a program in order to trigger alternative tasks[1]. Notice that a notion of time – the instants – is needed to give meaning to actions depending on the absence of a signal: otherwise, when could we decide that we have waited for too long, and that the signal is to be considered as really absent? This last notion of an instant is certainly the main innovation of the synchronous programming model, and also the less easy to grasp. In our formalization, an "instant", or, better, a *time slot* – or simply a slot –, is an interval in the execution of a program, possibly involving many (micro-)steps, the end of which is determined by the fact that all the components are either terminated or suspended, and therefore unable to activate any other component. Moreover, only at the very beginning of an "instant" interactions with the environment may take place. By contrast with ESTEREL, in the reactive programming style of Boussinot, the absence of a signal can only be determined at the end of the "instant", and therefore one cannot suspend a component on the presence of a signal (that is, run it only if the signal is absent), and one can only abort its execution at the end of a time slot. The following example illustrates the use of reactive programming in a global computing perspective:

$$\mathsf{agent}\ \lambda a.\mathsf{let}\ s = \mathsf{sig}\ \mathsf{in}\ \mathsf{thread}\ \lambda t.\mathsf{if}\ \mathsf{present}\ s\ \mathsf{then}\ ()\ \mathsf{else}\ (\mathsf{migrate\_to}\,\ell)a;\ P\,;\mathsf{emit}\ s$$

This is the code of an agent, named a, that, in some site, tries to executes a program P for one time slot. It spawns a thread (named t) that terminates if executing P succeeds, and otherwise moves the agent elsewhere. To trigger the migration, we use a local signal s which is only present when the task P terminates. One can also easily program a similar migration behaviour triggered by the presence of a failure signal emitted in a site:

$$\mathsf{agent}\ \lambda a.(\mathsf{thread}\ \lambda t.(\mathsf{when}\ \mathit{failure})(\mathsf{migrate\_to}\,\ell)a)\,;P$$

Here the migration instruction is suspended, by means of the when construct, until the *failure* signal is emitted.

[1] This is surely not new from a system or network programming point of view, where one needs to deal with suspended or stuck tasks. However, the relevant mechanisms are usually not integrated in high-level programming languages.

The main novelty in this paper is in the way we deal with the *state*, and more generally the context of a program. Apart from the reactive constructs, we adopt a fairly conventional style of programming, say the ML or SCHEME imperative and functional style, where a memory (or heap) is associated with a program, to store mutable values. Then, in the presence of mobile code, a problem arises: when a code migrates, what happens with the portion of the store it may share with other, mobile or immobile components? There are three conceivable solutions:

- *network references*. The owner of a reference (a pointer) keeps it with him, and remote accesses are managed by means of proxies, forwarding the requests through the network.
- *distributed references*. A shared reference is duplicated and each copy is directly accessible, by local operations. Its value is supposed to be the same in all the copies.
- *mobile references*. The owner of a reference keeps it with him, and all non local accesses to the reference fail.

All the three solutions have their own flaws, the last one being the simplest to implement, but also the worse, according to the current computing model: it amounts to systematically introducing dangling pointers upon migration, thus causing run-time errors. Indeed, as far as I can see, this solution has never been used. Regarding the first, the construction and use of chains of forwarders may be costly, but, more importantly, network references rely on the implicit assumption that the network is reliable, in the sense that a distant site is always accessible (otherwise this solution is as bad as the last one). As we have seen, this assumption does not hold in a global computing context, where it is, for instance, perfectly normal for a mobile device to be sometimes disconnected from a given network. Similarly, maintaining the coherence of replicated references in a large network may be very difficult and costly, especially if connections may be temporarily broken. Moreover, with mobile code, the conflicts among several values for a given reference are not easy to solve: if a user possesses such a reference, and launches several agents updating it, which is the right value?

We adopt mobile references in our programming model for global computing, but with an important new twist: trying to access an absent reference is not an error, it is a suspended operation, like waiting for an absent signal in synchronous/reactive programming. Then the reactive part of the model allows us to deal with programs suspended on absent references, exactly as with signals. Although we do not investigate them in this paper, we think that for many other aspects of distributed and global computing, a programming model taking into account suspension as an observable, and offering preemption constructs, is indeed relevant. For instance, we will not consider communications between nodes of a network (apart from the asynchronous migration of code), but it should be clear that, from a local point of view, remote communication operations should have a suspensive character. In particular, this should be the case in an implementation of network references. Similarly, we do not consider the dynamic linking phase that an agent should perform upon migration (we assume

that the sites provide the implementation of the constructs of our core model), but clearly an agent using some (supposedly ubiquitous) libraries for instance should involve some backup behaviour in case linking fails. In fact, in a global computing perspective, every operation engaging some code in an interaction with its environment should, from the point of view of this code, be considered as potentially suspensive. Therefore we think that, although they have been developed for radically different purposes – namely, hard real-time computations, and circuit design – the ideas of synchronous programming that we integrate in our ULM core programming model should prove also useful in the highly interactive and "odd-time" world of global computing.

2 The Computing Model

2.1 Asynchronous Networks of Synchronous Machines

Since the focus in this paper is on local reactive programming, and mobility, we have only minimal requirements on the network model. We assume that computing takes place in a network which is a collection of named nodes (or sites, or localities). Besides the nodes, the network also contains "packets", which merely consist here of frozen programs asynchronously migrating to some destination node. Then, assuming given a set $\mathcal{L}$ of node names, we describe a network using the following syntax:

$$R \ ::= \ \ell[\mathcal{M}] \ \mid \ \ell\langle p\rangle \ \mid \ (R_0 \parallel R_1)$$

where ℓ is any node name. We denote by $\ell[\mathcal{M}]$ a site, named ℓ, containing the configuration $\mathcal{M}$ of a reactive machine, while $\ell\langle p\rangle$ is a packet, en route to the site called ℓ. The syntax of the content of packets, as well as machine configurations, will be given below, and we shall see in a next section how packets are introduced in the network. We denote by $(R_0 \parallel R_1)$ the juxtaposition of the nodes of the two networks R_0 and R_1. We make the assumption that the sets of node names – that is, ℓ in $\ell[\mathcal{M}]$ – are disjoint when joining two networks. This is supposed to be guaranteed by some name service.

As we said in the introduction, our model is the one of a GALS network, a concept that we formalize as follows. The behaviour exhibited by a network is represented by transitions $R \to R'$, and is determined by the behaviour of its nodes. To account for the asynchronous character of network computations, we assume that juxtaposition is associative and commutative. (This also means that all the nodes, and packets, are considered here as neighbours, with no security barrier filtering communication, for instance.) Then, denoting by $\equiv$ the least congruence on networks for which $\parallel$ is associative and commutative, we have:

$$\frac{R_0 \to R_0'}{(R_0 \parallel R_1) \to (R_0' \parallel R_1)} \ (\textsc{Ntwk1}) \qquad \frac{R_0 \to R_0' \quad R_1 \equiv R_0 \quad R_1' \equiv R_0'}{R_1 \to R_1'} \ (\textsc{Ntwk2})$$

Now let us say a few words about the "locally synchronous" aspect of a GALS network. In Section 2.3 we shall define transitions between machine configurations, and we shall see that the behaviour of a synchronous machine, starting from an initial configuration $\mathcal{M}_0$, can be described as a sequence of *time slots*:

$$\begin{array}{ll} \ell[\mathcal{M}_0] \to \cdots \to \ell[\mathcal{M}'_0] \to \ell[\mathcal{M}_1] & \textit{first slot} \\ \ell[\mathcal{M}_1] \to \cdots \to \ell[\mathcal{M}'_1] \to \ell[\mathcal{M}_2] & \textit{second slot} \\ \vdots & \textit{and so on.} \end{array}$$

where the transitions $\ell[\mathcal{M}_i] \stackrel{*}{\to} \ell[\mathcal{M}'_i]$ represent purely local (micro)-steps to a "stable" configuration, and the last step $\ell[\mathcal{M}'_i] \to \ell[\mathcal{M}_{i+1}]$ is a special "*tick*" transition. During the local steps of computation, the machine behaves in isolation from its environment. This normally ends up with a state $\mathcal{M}'_i$ where all the concurrent threads running in the machine are either terminated or suspended, waiting for some information, so that no further local computation may occur. Then all the interactions with the environment take place, during the "tick" transition, where the machine by itself does not perform any computation. Therefore one can say that during a time slot, the threads running in the machine have a coherent vision of the outside world. In this paper the interactions between a machine and its environment only consist in the management of migration – incorporating immigrant agents, and sending emigrating agents as packets in the network. However, one could easily imagine other forms of interactions, like receiving input signals or sending calls to remote procedures.

2.2 The Language: Syntax

As we suggested above, a reactive machine $\mathcal{M}$ runs a collection of concurrent threads – among which mobile threads, that we call agents –, organized in a queue which we denote by T. These threads share a common store S and a context (a set) of signals E. Moreover, the configuration of the machine records a set P of emigrating agents. That is, a machine configuration has the form

$$\mathcal{M} = (S, E, T, P)$$

In this section we introduce the core language used to program agents and threads. As in ML or SCHEME, this core language is a call-by-value λ-calculus, enriched with imperative and reactive programming constructs, and with constructs to create and manage threads, including mobile ones. The syntax is given in Figure 1. Let us comment briefly on the primitive constructs of the language.

• Regarding the λ-calculus part of the language, we shall use the standard abbreviations and notations. We denote $\lambda x M$ by $\lambda.M$ when x is not free in M. The capture avoiding substitution of N for x in M is denoted $\{x \mapsto N\}M$.

• The imperative constructs, that is ref, **?** and set are quite standard, except that we denote by **?** the dereferencing operation (which is denoted **!** in ML), to emphasize the fact that this is a potentially suspensive operation. As usual, we also write $M := N$ for $(\mathsf{set}\, M)N$.

• The intuitive semantics of the reactive constructs sig, emit, when and watch is as follows. The constant sig creates, much like the ref construct, a new signal, that is a name. The created signal is initially absent, and one uses the emit function to make a signal present for the current time slot. The function when

$$
\begin{array}{rcl}
M,\ N\ldots & ::= & V \mid (MN) \mid \mathsf{sig} \\
V,\ W\ldots & ::= & \ell \mid x \mid \lambda x M \mid () \\
& \mid & \mathsf{ref} \mid \mathbf{?} \mid \mathsf{set} \mid (\mathsf{set}\, V) \\
& \mid & \mathsf{emit} \mid \mathsf{when} \mid (\mathsf{when}\, V) \mid \mathsf{watch} \mid (\mathsf{watch}\, V) \\
& \mid & \mathsf{thread} \mid \mathsf{agent} \mid \mathsf{migrate\_to} \mid (\mathsf{migrate\_to}\, V)
\end{array}
$$

Fig. 1. Syntax

is intended to be applied to two arguments, the first one being a signal. It *suspends* the evaluation of its second argument when the first one is absent. The preemption function watch is also a function of a signal and an expression. At the end of the current slot, it *aborts* the execution of its second argument if the watched signal is present.

• The thread function spawns a new thread, as well as the agent function which, in addition, creates an agent name. Agent names are used by the migrate_to function to cause the migration of the designated thread.

As we have suggested in these informal explanations, the evaluation of several constructs of the language, namely ref, sig and agent (and also thread) will create various kinds of names at run-time. These names are values in an extended, run-time language. They are defined as follows. First, we assume given a denumerable set $\mathcal{N}$ of names, disjoint from $\mathcal{L}$. Then, besides node names in $\mathcal{L}$, used in the source language, we shall use three kinds of names:

(i) Simple names, belonging to $\mathcal{N}$ and ranged over by u, v, $w\ldots$. These are used to name immobile threads, that is, the threads created using the thread construct.

(ii) Compound names of the form $\ell.u$, thus belonging to $\mathcal{L}.\mathcal{N}$. They are used to name signals, immobile references (that is, references created by an immobile thread), and agents. All these names may be imported in a node by a mobile agent, and this is why we use the name ℓ of the creation node as a prefix, to avoid conflicts whith names created elsewhere.

(iii) Compound names of the form $\ell.u.v$, belonging to $\mathcal{L}.\mathcal{N}.\mathcal{N}$. These are the names of references created by agents. As we will see, an agent will carry with it its own portion of the memory upon migration, and it is therefore convenient to distinguish the references created by an agent, by prefixing their name v with the name $\ell.u$ of their creator.

2.3 Operational Semantics

We have already described the semantics of networks, which relies on machine transitions. To introduce the latter, let us first define more precisely the components of a configuration:

• As usual, the *store* S is a mapping from a finite set $\mathsf{dom}(S) \subseteq \mathcal{L}.\mathcal{N} \cup \mathcal{L}.\mathcal{N}.\mathcal{N}$ of memory addresses to values.

• The signal environment $E \subseteq \mathcal{L}.\mathcal{N}$ is the set of names of signals that are *present* during the current time slot.
• The threads queue T is a sequence of named programs M^t, where M is an expression of the run-time language and t is the name of a thread executing M (so that $t \in \mathcal{N} \cup \mathcal{L}.\mathcal{N}$). We denote by $T \cdot T'$ the concatenation of two threads sequences, and by ε the empty sequence.
• The set P of emigrating agents consists in named programs together with a destination, denoted $(\mathsf{to}\,\ell)M^t$.

To run an expression M of the language in a locality ℓ, we start from an initial configuration where M is the only thread in the queue, with an arbitrary (simple) name, and everything else is empty, that is:

$$\ell[\emptyset, \emptyset, M^u, \emptyset]$$

For the sake of exposition, we distinguish four kinds of machine transitions. These transitions generally concern the head of the thread queue (on the left), that is the code M if $T = M^t \cdot T'$. First, we have purely functional transitions that occur in the active code part M of a configuration, and are essentially independent from the rest, like β-reduction. Then we have the imperative and reactive behaviour, where the code M interacts with the store S or the signal environment E. Next there are transitions having an effect on the thread queue, like thread creation, migration, and scheduling. Last but not least, there are transitions from one time slot to the next. As we said, all interactions with the "outside world" take place at this moment, and in particular the P part of the configuration is only considered at this point. The functional transitions, denoted $M \to_f M'$, are given by the following axioms:

$$(\lambda x M V) \;\to_f\; \{x \mapsto V\}M \qquad (\beta_{\mathrm{v}})$$
$$((\mathsf{when}\, W)V) \;\to_f\; V \qquad (\textsc{When})$$
$$((\mathsf{watch}\, W)V) \;\to_f\; () \qquad (\textsc{Watch})$$

The two transitions regarding when and watch mean that, when the code they control terminates, then the whole expression terminates.

To define the imperative and reactive transitions, acting on tuples (S, E, M^t), we need some auxiliary notions. First, we shall use the standard notion of an *evaluation context*, analogous to a "control stack". Evaluation contexts in our language are defined by the following grammar:

$$\mathbf{E} \;::=\; [] \;\mid\; (\mathbf{E}N) \;\mid\; (V\mathbf{E})$$

Besides the evaluation contexts, we need the predicate $(S, E, M)\,\dagger$ of *suspension*. In our language, there are two ways in which a code M may be suspended in the context of a store S and a signal environment E: either M tries to access, for reading or writing it, a reference which is not in the domain of the store, or M waits for a signal which is absent. This is easily formalized, see [4].

To abbreviate the notations, in the rules below we will use the predicate $(E, \mathbf{E})\,\lightning$, meaning that, given the signal environment E, the evaluation context $\mathbf{E}$ is not suspensive, that is, if $\mathbf{E} = \mathbf{E}_0[((\mathsf{when}\, s)\mathbf{E}_1)]$ then $s \in E$. In the rules we

also need, in some cases, the knowledge of the name of the locality where they occur. Therefore the imperative and reactive transitions are of the form

$$\vdash_\ell (S, E, M^t) \to_{ir} (S', E', M'^t)$$

For lack of space, we omit here some of the rules of the operational semantics. The complete design has to be found in [4]. For instance, there is a rule by which functional transitions are allowed in any non suspensive context. Then we have rules to create a reference and enlarge the store, and to get or update the value of a reference. These are more or less standard, except that dereferencing and updating are suspensive operations when the reference is not in the domain of the store (again, see [4] for the details). The creation of a signal is similar to the creation of a reference, that is, a fresh signal name is returned as a value:

$$\frac{(E, \mathbf{E}) \not\downarrow \qquad \ell.u \text{ fresh}}{\vdash_\ell (S, E, \mathbf{E}[\mathsf{sig}]^t) \to_{ir} (S, E, \mathbf{E}[\ell.u]^t)} \quad (\textsc{Sig})$$

Notice that the signal is still absent, that is $\ell.u \notin E$. The signal is present, for the rest of the current slot, when it has been emitted:

$$\frac{(E, \mathbf{E}) \not\downarrow}{\vdash_\ell (S, E, \mathbf{E}[(\mathsf{emit}\, s)]^t) \to_{ir} (S, E \cup \{s\}, \mathbf{E}[()]^t)} \quad (\textsc{Emit})$$

One can see that, following these transitions, it may happen that the evaluation of M gets suspended at some point. In this case, we have to look for other threads to execute, or to notice that the current slot has ended. This is decribed by the next rules, for transitions of the form

$$\ell[\mathcal{M}] \to \ell[\mathcal{M}']$$

that we mentioned in Section 2.1. A first rule says that an imperative or reactive transition is also a machine transition, if this transition is performed by the first thread in the queue (see [4]). Then, when the first thread is suspended, it is put at the end of the queue, provided that there is still some computation to perform, that is, there is another thread in the queue which is not terminated, and not suspended. By abuse of notation, we shall denote by $(S, E, T) \not\downarrow$ the fact that there is a thread in the queue T which is active in the context of S and E. The rule for scheduling the queue of threads is as follows:

$$\frac{\neg(S, E, M^t) \not\downarrow \qquad (S, E, T) \not\downarrow}{\ell[S, E, M^t \cdot T, P] \to \ell[S, E, T \cdot M^t, P]} \quad (\textsc{Sched})$$

We have adopted a round-robin scheduling, placing the current thread at the end of the queue when it is terminated or suspended. We could obviously use any other (fair) strategy, but we think it is important, in a language with effects, to have a deterministic strategy, in order to have some understanding of the semantics. One may notice that if a thread is suspended, waiting for a signal on a $(\mathsf{when}\, s)$ statement, and if this signal is emitted during the current time slot, then the thread will always be activated during this slot.

It remains to see how threads are created, and how migration is managed. The thread function takes a thunk – that is, a frozen expression $\lambda.M$ – as argument, creates a new thread, the body of which is the unfrozen thunk, and terminates. The body of the created thread is subject to the control (by signals, that is by means of when and watch functions) exercized by the context that created it. To formulate this, let us define the control context $\mathbf{E}^w$ extracted from an evaluation context $\mathbf{E}$ as follows:

$$\begin{aligned}
[]^w &= [] \\
(\mathbf{E}N)^w &= \mathbf{E}^w \\
(V\mathbf{E})^w &= \begin{cases} (V\mathbf{E}^w) \text{ if } V = (\mathsf{when}\, W) \text{ or } V = (\mathsf{watch}\, W) \\ \mathbf{E}^w \qquad \text{otherwise} \end{cases}
\end{aligned}$$

One can see that $\mathbf{E}^w$ is a sequence of $(\mathsf{when}\, W)$ and $(\mathsf{watch}\, W)$ statements. The rule for thread creation is as follows:

$$\frac{(E, \mathbf{E}) \not\downarrow}{\ell[S, E, \mathbf{E}[(\mathsf{thread}\, V)]^t \cdot T, P] \to \ell[S, E, \mathbf{E}[()]^t \cdot T \cdot \mathbf{E}^w[(V\,())]^u, P]} \quad (\textsc{Thread})$$

Notice that the newly created thread is put at the end of the queue, and therefore it cannot prevent the existing threads to execute, since the thread queue is scanned from left to right by the scheduler.

The agent function also creates threads, but with a different semantics. It is intended to have as argument a function of an agent name, say λxN. Then $(\mathsf{agent}\, \lambda xN)$ creates a new agent name, returned as a value, and a new thread, the body of which is N where x is instantiated by the created agent name. Unlike with threads created by thread, the control exercized by the creating context is not transferred on the created agent. The idea is that a thread, created by thread, is intended to cooperate with its creation context in the achievement of some task, while an agent is thought of as detached from a main task, and is intended to move (maybe just to escape from abortion of the main task). The rule is:

$$\frac{(E, \mathbf{E}) \not\downarrow \qquad \ell.u \text{ fresh}}{\ell[S, E, \mathbf{E}[(\mathsf{agent}\, V)]^t \cdot T, P] \to \ell[S, E, \mathbf{E}[\ell.u]^t \cdot T \cdot (V\ell.u)^{\ell.u}, P]} \quad (\textsc{Agent})$$

As one can see, the created name $\ell.u$ is known from both the creating context, and the agent itself. This name could be used to control the behaviour of the agent in various ways. We could for instance have primitive constructs to kill, or resume/suspend the execution of an agent. In this paper, the only construction of this kind we consider is the statement causing the migration of an agent. When we have to execute a $(\mathsf{migrate\_to}\, \ell')t$ instruction, where ℓ' is a node name and t is a thread name (it should be guaranteed by typing that migrate_to only applies to such values), we look in the thread queue for a component named t, and put it, with destination ℓ', in the set of emigrating agents:

$$\frac{(E, \mathbf{E}) \not\downarrow \qquad \mathbf{E}[()]^r \cdot T = T' \cdot N^t \cdot T''}{\ell[S, E, \mathbf{E}[((\mathsf{migrate\_to}\, \ell')t)]^r \cdot T, P] \to \ell[S, E, T' \cdot T'', P \cup \{(\mathsf{to}\, \ell')N^t\}]} \quad (\textsc{Migr})$$

(There is also a rule in the case where the agent is not there, which is omitted.) If $t = r$, we have $N = \mathbf{E}[()]$, and we say that the migration is subjective, since it is performed by the agent itself. Otherwise, that is if the migration instruction is exercized by another thread, we say that the migration is objective. We also observe that (subjective) migration in our model may be qualified as "strong mobility" [13], since the "control stack", that is the evaluation context $\mathbf{E}$ of the migration instruction, is moving (together with the memory of the agent, as we shall see).

The last rule describes what happens at the end of a time slot, which is also the beginning of the next one – we call this the *tick* transition. One can see that no rule applies if the thread queue only consists of terminated or suspended threads. In this case, that is when $\neg(S, E, T) \not\downarrow$, the current slot is terminated, and the computation restarts for a new one, after the following actions have been done:

• The signal environment is reset to the empty set

Note 1. One could also imagine that signals could be emitted from the external environment. In this case, they would be incorporated as input signals for the next slot, but only during the "tick" transition..

• The preemption actions take place. That is, a sub-expression $((\mathsf{watch}\, s)M)$ (in a non suspensive context) terminates if the signal s was present in the signal environment at the end of the slot. Otherwise – that is, if s is absent –, this expression is resumed for the next slot, as well as the when guards.

• The immigrant agents, that is the contents of network packets $\ell\langle p\rangle$, are incorporated. In a more realistic language, one should have a buffering mechanism, involving a security check, to receive the asynchronously incoming agents. The contents p of a packet is a pair (S, M^t) of a store and a named program. The intuition is that M is the code of a migrating agent, and S is its own, local store. Then the code M is put at the end of the threads queue, and the S store is added to the one of the host machine (thus possibly activating threads suspended on trying to access an absent reference).

• The emigrant agents, in P, are sent in the outside world. More precisely, an element $(\mathsf{to}\, \ell')N^t$ of P is transformed into a packet $\ell'\langle S', N^t\rangle$ where S' is the portion of the store that is owned by the agent N^t. This portion is removed from the store of the locality where the tick transition is performed. Notice that a machine waits until it reaches a stable configuration to send the emigrating agents in the network. This is to maintain a coherent view of the store, as far as the presence or absence of references is concerned, during each time slot (notice also that the other threads may modify the store carried by the agent, even after it has been put in the P part of the configuration).

The precise formulation of the "tick" transition is given in [4].

3 Some Examples

In this section, we examine some examples, to illustrate the expressive power of the constructs of the language. We begin with examples related to the reac-

tive programming primitives. There has been many variations, sometimes just notational, in the choice of primitives in synchronous, imperative programming. To obtain a "minimal" language, suited for formal reasoning, we have chosen to have only two primitives for reactive programming in ULM, one for introducing suspension, and the other for programming preemption. However, the other constructs, functional and imperative, of the language allow us to encode most of the standard constructs of synchronous/reactive programming. Let us see this now.

First we notice that our preemption construct $(\mathsf{watch}\, s)M$ exists in some versions of ESTEREL [2], where it is written $(\mathsf{abort}\ M\ \mathsf{when}\ s)$, and also in SL [6], where it is written $(\mathsf{do}\ M\ \mathsf{kill}\ s)$. In synchronous programming, there is a statement, called stop in [6] and pause in [2] (it is not a primitive, but is easily derived in the language of [3]), which lasts just one instant, that is, in our setting, which is suspended until the end of the current time slot, and terminates at the tick transition. To express this in ULM, it is convenient to first introduce a now construction which, when applied to a thunk, executes it for the current time slot only, so that $(\mathsf{now}\, M)$ always terminates at the end of the slot, at the latest. This is easy to code: we just have to preempt the argument of now on a signal that is immediately emitted:

$$\mathsf{now} =_{\mathrm{def}} \lambda x(\mathsf{let}\ s = \mathsf{sig}\ \mathsf{in}\ (\mathsf{watch}\, s)(\mathsf{emit}\, s\, ;\, x()))$$

Another useful notation, that exists in ESTEREL (see [2]) is

$$\mathsf{await} =_{\mathrm{def}} \lambda s.(\mathsf{when}\, s)()$$

Then the pause construct is easily defined, as waiting for an absent signal, for the current slot only:

$$\mathsf{pause} =_{\mathrm{def}} (\mathsf{let}\ s = \mathsf{sig}\ \mathsf{in}\ (\mathsf{now}\, \lambda.\mathsf{await}\, s))$$

Using a pause statement, a thread suspends itself until the next time slot. We can also define a statement by which a thread just gives its turn, as if it were put at the end of the queue. This is easily done by spawning a new thread (thus put at the end of the queue) that emits a signal on which the "self-suspending" thread waits:

$$\mathsf{yield} =_{\mathrm{def}} (\mathsf{let}\ x = \mathsf{sig}\ \mathsf{in}\ (\mathsf{thread}\ \lambda.\mathsf{emit}\ x)\ ;\ \mathsf{await}\ x)$$

We can also use the now construct to define a present predicate whose value, when applied to a signal, is true (assuming that the language is extended with truth values *tt* and *ff*) if the signal is present, and false otherwise. Notice that since we can only determine that a signal is absent at the end of the current time slot, the value false can only be returned at the beginning of the next slot. The code is, using a reference to record the presence or absence of the signal:

$$\begin{aligned}\mathsf{present} =_{\mathrm{def}}\ & \lambda s\, \mathsf{let}\ r = \mathsf{ref}\, \mathit{ff} \\ & \quad \mathsf{in}\ (\mathsf{now}\, \lambda.(\mathsf{when}\, s)\, r := \mathit{tt})\ ;\ ?r\end{aligned}$$

In a conditional construct $\mathsf{if}\ \mathsf{present}\, s\ \mathsf{then}\ M\ \mathsf{else}\ N$, the code M is started in the current slot if s is present, and otherwise N is executed at the next one.

To finish with the examples, let us briefly discuss the mobile code aspect of the language. Imagine that we want to send, from a main program at site ℓ, an agent to a site ℓ', with the task of collecting some information there and coming back to communicate this to the sender (like we would do with a proxy for a network reference, for instance). We cannot write this as

$$\mathsf{let}\ r = (\mathsf{ref}\ X)\ \mathsf{in}\ \mathsf{migrate\_to}\,\ell'(\mathsf{agent}\,\lambda a(\cdots r := Y\ ;\ (\mathsf{migrate\_to}\,\ell)a))\ ;\ \cdots\ ?r\ \cdots$$

(which is an example of both objective and subjective migration) for two reasons: after some steps, where r is given a value $\ell.u$, that is an address in the store at site ℓ containing the value X, and where a name $\ell.v$ is given for a, the agent runs at ℓ' the code

$$\cdots\ \ell.u := Y\ ;\ (\mathsf{migrate\_to}\,\ell)\ell.v$$

Since the address $\ell.u$ is created by an immobile thread, it will always remain in the store at ℓ, and will never move. Therefore, the agent is blocked in ℓ' waiting for the reference $\ell.u$ to be there. On the other hand, the "main thread" at ℓ may perform $?\ell.u$, but it will get the value X, which is certainly not what is intended. We have to use a signal to synchronize the operations, like with:

$$\begin{array}{l}\mathsf{let}\ r = (\mathsf{ref}\ X)\ \mathsf{and}\ s = \mathsf{sig}\\ \mathsf{in}\ \mathsf{let}\ \mathit{write}\ =\ \lambda x.r := x\ ;\ \mathsf{emit}\ s\ \mathsf{and}\\ \qquad\quad \mathit{read}\ =\ \mathsf{await}\ s\ ;\ ?r\\ \quad \mathsf{in}\ \cdots\end{array}$$

writing the body of the agent as:

$$\cdots\,\mathsf{let}\ y = Y\ \mathsf{in}\ (\mathsf{migrate\_to}\,\ell)a\ ;\ \mathit{write}\,y$$

One can see that the main thread cannot read a value from the reference r before the signal s has been set as present, which only occurs (provided that r and s are private to the functions *write* and *read*) when a *write* operation has occurred. This example shows that mobile computing should be "location aware", at least at some low level, as envisioned by Cardelli [9]. This example also shows that an object-oriented style of programming would be useful to complement mobile agent programming, to define synchronization and communication structures, such as the one we just introduced in this example, of a signal with a value, that we may call an *event*. We are currently investigating this.

4 Conclusion

In this paper we have introduced a computing model to deal with some new observables arising in a global computing context, and especially the unreliability of resource accesses resulting from the mobility of code. For this purpose, we have used some ideas of synchronous programming, and most notably the idea that the behaviour of a program can be seen as a succession of "instants", within which a reaction to the suspension of some operations can be elaborated. This is what we need to give a precise meaning to statements like "abort the computation

C if it gets suspended for too long, and run C' instead", that we would like to encode as an algorithmic behaviour.

During the last few years of the last century, a lot of proposals have been made to offer support for mobile code (see for instance the surveys [13,19], and the articles in [20]). As far as I can see none of these proposals addresses the unreliablity issue as we did (and most often they lack a clear and precise semantics). Most of these proposals are assuming, like Cardelli's *OBLIQ* [7], that a distributed scope abstraction holds, in the sense that the access to a remote resource is transparent, and does not differ, from the programmer's point of view, from a local access. The JoCaml language for instance [12] is based on this assumption. However, we think that this abstraction can only be maintained in a local network. Indeed, Cardelli later argued in [9] that global computing should be location aware, and should also take into account the fact that in a global context, failures become indistinguishable from long delays. The Sumatra proposal [1] acknowledges the need "to be able to react to changes in resource availability", a feature which is called "agility", and is implemented by means of exception handling mechanisms. However, as far as I can see, no precise semantics is given for what is meant by "react quickly to asynchronous events" for instance. That is, a precise notion of time is missing.

Softwares and languages that are based on Linda, such as JavaSpaces [18], Lime [16] or Klaim [11], or on the π-calculus, like the JoCaml language [12] or NomadicPict [17], all involve an implicit notion of suspension: waiting for a matching tuple to input in the case of Linda, waiting for a message on an input channel in the case of π. However, they all lack preemption mechanisms, and a notion of time (indeed, the semantics of these models is usually asynchronous). Summarizing, we can conclude that our proposal appears to be the first (apart from [10]) that addresses, at the programming language level, and with a formal semantics, the issue of reacting to an unreliable computing context. Clearly, a lot of work is to be done to develop this proposal. In the full version of this paper [4], we show how to predict, by a static analysis, that some uses of references may be considered as safe, that is, they do not need testing the presence of the reference. Then, we should extend ULM into a more realistic language, and provide a proof-of-concept implementation. We should also investigate the network communication aspect (network references, RPC for instance), and the dynamic linking of mobile agents, that we did not consider. Last but not least, we should investigate (some of) the (numerous) security issues raised by the mobility of code. (A formal semantics for the mobile code, as the one we designed, is clearly a prerequisite to tackling seriously these issues.) One may have noticed for instance that in our model, a thread may prevent the other components to compute, simply by running forever, without terminating or getting suspended. Clearly, this is not an acceptable behaviour for an incoming agent. We are currently studying means to restrict the agents to always have a cooperative behaviour, in the sense that they only perform, at each time slot, a finite, predictable number of transitions.

References

[1] A. Acharya, M. Ranganathan, J. Saltz, *Sumatra:* a language for resource-aware mobile programs, in [20] (1997) 111–130.

[2] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, R. de Simone, *The synchronous languages twelve years later*, Proc. of the IEEE, Special Issue on the Modeling and Design of Embedded Software, Vol. 91 No. 1 (2003) 64–83.

[3] G. Berry, G. Gonthier, *The ESTEREL synchronous programming language: design, semantics, implementation*, Sci. of Comput. Programming Vol. 19 (1992) 87–152.

[4] G. Boudol, ULM, *A core programming model for global computing*, available from the author's web page (2003).

[5] F. Boussinot, *La Programmation Réactive – Application aux Systèmes Communicants, Masson*, Coll. Technique et Scientifique des Télécommunications (1996).

[6] F. Boussinot, R. de Simone, *The SL synchronous language*, IEEE Trans. on Software Engineering Vol. 22, No. 4 (1996) 256–266.

[7] L. Cardelli, *A language with distributed scope*, Computing Systems Vol. 8, No. 1 (1995) 27–59.

[8] L. Cardelli, *Global computation*, ACM SIGPLAN Notices Vol. 32 No. 1 (1997) 66–68.

[9] L. Cardelli, *Wide area computation*, ICALP'99, Lecture Notes in Comput. Sci. 1644 (1999) 10–24.

[10] L. Cardelli, R. Dawies, *Service combinators for web computing*, IEEE Trans. on Software Engineering Vol. 25 No. 3 (1999) 303–316.

[11] R. De Nicola, G. Ferrari, R. Pugliese, KLAIM: *a kernel langage for agents interaction and mobility*, IEEE Trans. on Software Engineering Vol. 24, No. 5 (1998) 315–330.

[12] C. Fournet, F. Le Fessant, L. Maranget, A. Schmitt, JoCaml: *a language for concurrent, distributed and mobile programming*, Summer School on Advanced Functional Programming, Lecture Notes in Comput. Sci. 2638 (2003) 129–158.

[13] A. Fuggetta, G.P. Picco, G. Vigna, *Understanding code mobility*, IEEE Trans. on Software Engineering, Vol. 24, No. 5 (1998) 342–361.

[14] N. Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer (1993).

[15] The IST-FET Global Computing Initiative, `http://www.cordis.lu/ist/fet/gc.htm (2001)`.

[16] G.P. Picco, A.L. Murphy, G.-C. Roman, *LIME: Linda meets mobility*, ACM Intern. Conf. on Software Engineering (1999) 368–377.

[17] P. Sewell, P. Wojciechowski, *Nomadic Pict: language and infrastructure design for mobile agents*, IEEE Concurrency Vol. 8 No. 2 (2000) 42–52.

[18] Sun Microsystems, *JavaSpaces Service Specification* `http://www.sun.com/jini/specs` (2000).

[19] T. Thorn, *Programming languages for mobile code*, ACM Computing Surveys Vol. 29, No. 3 (1997) 213–239.

[20] J. Vitek, Ch. Tschudin, Eds, *Mobile Object Systems: Towards the Programmable Internet*, Lecture Notes in Comput. Sci. 1222 (1997).