Reactive concurrent programming revisited

Roberto M. Amadio [a], Gérard Boudol [b], Frédéric Boussinot [b]
and Ilaria Castellani [b]

[a] *PPS, Université Denis Diderot, Case 7014, 2 Place Jussieu, 75251 PARIS Cedex 05*

[b] *INRIA Sophia Antipolis, BP 93, 06902 Sophia Antipolis Cedex, France*

Abstract

In this note we revisit the so-called *reactive* programming style, which evolves from the synchronous programming model of the ESTEREL language by weakening the assumption that the absence of an event can be detected instantaneously. We review some research directions that have been explored since the emergence of the reactive model ten years ago. We shall also outline some questions that remain to be investigated.

Key words: concurrency, semantics, cooperative scheduling, reactive systems

1 Introduction

In synchronous models the computation of a set of participants is regulated by a notion of *instant*. The *Synchronous Language* introduced in [12] belongs to this category. A *program* in this language generally contains sub-programs running in parallel and interacting via shared *signals*. By default, at the beginning of each instant a signal is absent and once it is emitted it remains in that state till the end of the instant. The model can be regarded as a relaxation of the ESTEREL model [6] where the *reaction to the absence* of a signal is delayed to the following instant, thus avoiding the difficult problems due to *causality cycles* in ESTEREL programs.

The model has gradually evolved into a programming language for concurrent applications and has been implemented in the context of various programming languages such as C, JAVA, SCHEME, and CAML (see Section 3 below). The design accommodates a dynamic computing environment with threads entering or leaving the synchronisation space. In this context, it seems natural to suppose that the scheduling of the threads is only determined at run time (as opposed to certain synchronous languages such as ESTEREL or LUSTRE).

The model is based on a *cooperative* notion of concurrency. This means that by default a running thread cannot be preempted unless it explicitly decides to return the control to the scheduler. This contrasts with the model of *preemptive* threads, where by default a running thread can be preempted at any point unless it explicitly requires that a series of actions is atomic. We refer to, e.g., [23] for an extended comparison of the cooperative and preemptive models. It appears that many typical "concurrent" applications such as event-driven controllers, data flow architectures, graphical user interfaces, simulations, web services, multiplayer games, are more effectively programmed in a cooperative (and possibly synchronous) model than in the preemptive one.

The purpose of this note is to revisit the basic model and to review some research directions that have been explored since the emergence of the model ten years ago. We shall also outline some questions that remain to be investigated.

2 The basic model

In this section, we introduce our basic model which is largely inspired by the original proposal [12], and, as regards concurrency, by the FairThreads model [10].

We assume a countable set of *signal names* $s, s', \ldots$ and we let *Int* be a finite set of signal names representing an observable *interface*. A *signal environment* E is a partial function from signal names to boolean values *true* and *false* whose domain of definition $dom(E)$ is finite and contains *Int*. Such an environment records the signals that have been emitted during the current instant, as well as the ones that exist but are still absent. The semantics should preserve the invariant that all signals defined in a program (see below) belong to the domain of definition of the related environment. In particular, all signal names which are not in the domain of definition of the environment are guaranteed to be fresh, i.e., not used elsewhere in the program.

We define a *thread* as an expression written according to the following grammar:

$$\begin{array}{rl} T ::= & () \mid (\mathsf{emit}\ s) \mid (\mathsf{local}\ s\ T) \mid (\mathsf{thread}\ T) \\ \mid & (\mathsf{when}\ s\ T) \mid (\mathsf{watch}\ s\ T) \mid A(\mathbf{s}) \mid (T;T) \end{array}$$

where $A(\mathbf{s}), B(\mathbf{s}), \ldots$ denote thread identifiers with parameters $\mathbf{s}$. As usual, each thread identifier is defined by exactly one equation $A(\mathbf{s}) = T$. A thread is executed in the context of a signal environment which is *shared* with other concurrent threads.

The intended semantics is as follows: $()$ is the terminated thread; $(\mathsf{emit}\ s)$ emits s, *i.e.* sets it to *true* and terminates, $(\mathsf{local}\ s\ T)$ creates a fresh signal which is local to the thread T and executes T (this construct is a binder for the name s in T); $(\mathsf{thread}\ T)$ spawns a thread T which will be executed in parallel and terminates; $(\mathsf{when}\ s\ T)$ allows the execution of T whenever the signal s is present and suspends its execution otherwise; $(\mathsf{watch}\ s\ T)$ allows the execution of T but kills whatever is left of T at the end of the first instant where the signal s is present, $T;T$ is the usual sequentialisation. This operational intuition is formalised in the following rules, where the predicate $(T,E) \Downarrow^P (T',E')$ means that the thread T in the environment E executes an *atomic* sequence of instructions (possibly none) resulting in the thread T', the environment E', and the spawning of the *multi-set* of threads P.

$$(T_1)\ \frac{}{((),E) \Downarrow^{\emptyset} ((),E)}$$

$$(T_2)\ \frac{}{(\mathsf{emit}\ s,E) \Downarrow^{\emptyset} ((),E[s := \mathit{true}])}$$

$$(T_3)\ \frac{([s'/s]T, E \cup \{s' \mapsto \mathit{false}\}) \Downarrow^P (T',E') \quad s' \notin \mathit{dom}(E)}{(\mathsf{local}\ s\ T,E) \Downarrow^P (T',E')}$$

$$(T_4)\ \frac{}{(\mathsf{thread}\ T,E) \Downarrow^{\{\!|T|\!\}} ((),E)}$$

$$(T_5)\ \frac{([\mathbf{s}/\mathbf{x}]T,E) \Downarrow^P (T',E') \quad A(\mathbf{x}) = T}{(A(\mathbf{s}),E) \Downarrow^P (T',E')}$$

$$(T_6)\ \frac{E(s) = \mathit{false}}{(\mathsf{when}\ s\ T,E) \Downarrow^{\emptyset} (\mathsf{when}\ s\ T,E)}$$

$$(T_7)\ \frac{E(s) = \mathit{true} \quad (T,E) \Downarrow^P ((),E')}{(\mathsf{when}\ s\ T,E) \Downarrow^P ((),E')}$$

$$(T_8)\ \frac{E(s) = \mathit{true} \quad (T,E) \Downarrow^P (T',E') \quad T' \neq ()}{(\mathsf{when}\ s\ T,E) \Downarrow^P (\mathsf{when}\ s\ T',E')}$$

$$(T_9)\ \frac{(T,E) \Downarrow^P ((),E')}{(\mathsf{watch}\ s\ T,E) \Downarrow^P ((),E')}$$

$$(T_{10})\ \frac{(T,E) \Downarrow^P (T',E') \quad T' \neq ()}{(\mathsf{watch}\ s\ T,E) \Downarrow^P (\mathsf{watch}\ s\ T',E')}$$

$$(T_{11})\ \frac{(T_1,E) \Downarrow^{P_1} ((),E_1) \quad (T_2,E_1) \Downarrow^{P_2} (T',E')}{(T_1;T_2,E) \Downarrow^{P_1 \cup P_2} (T',E')}$$

$$(T_{12})\ \frac{(T_1, E) \Downarrow^P (T', E') \quad T' \neq ()}{(T_1; T_2, E) \Downarrow^P (T'; T_2, E')}$$

It can be seen from this description of the operational semantics that whenever $(T, E) \Downarrow^P (T', E')$ then the execution of T is either *terminated*, that is $T' = ()$, or *suspended*, that is T' is an expression where one has to execute a subexpression of the form $(\mathsf{when}\ s\ S)$, but $E'(s) = \mathit{false}$ (see the rule T_6). In other words, in our cooperative framework, the when instruction is the only one that may cause the interruption of the execution of a thread.

The implementation of both the when and the watch instructions requires a *stack*. For instance, in $(\mathsf{when}\ s_1\ (\mathsf{when}\ s_2\ T))$ the computation of T may progress only if both the signals s_1 and s_2 are present. In

$$(\mathsf{watch}\ s_1\ (\mathsf{watch}\ s_2\ T_1); T_2); T_3,$$

we start executing T_1. Assuming that at the end of the instant, the execution of T_1 is not completed, the computation in the following instant resumes with T_3 if s_1 was present at the end of the instant, with T_2 if s_1 was absent and s_2 was present at the end of the instant, and with the residual of T_1, otherwise. Note that whenever we spawn a new thread we start its execution with an empty stack of signals, as in the FairThreads model [10].

A *program* P is a finite non-empty multi-set of threads. We denote with $\mathit{sig}(T)$ (resp. $\mathit{sig}(P)$) the set of signals free in T (resp. in threads in P). To execute a program P in an environment E during one instant, we proceed as follows: first schedule (non-deterministically) the atomic executions of the threads that compose it as long as some progress is possible and second transform all active $(\mathsf{watch}\ s\ T)$ instructions where the signal s is present into the terminated thread $()$. To say that a thread T in an environment E is stuck we write $(T, E)\ddagger$. This is defined as

$$(T, E) \ddagger \ \text{ if } (T, E) \Downarrow^{\emptyset} (T, E) \tag{1}$$

Notice that if $(T, E)\ddagger$ then T is either terminated or suspended in the context of E. To perform the *abort* operation associated with the watch construct at the end of the instant, we rely on the function $\lfloor \_ \rfloor_E$ defined as follows:

$$\lfloor P \rfloor_E = \{\!|\lfloor T \rfloor_E \mid T \in P|\!\} \quad \lfloor () \rfloor_E = () \quad \lfloor T; T' \rfloor_E = \lfloor T \rfloor_E; T'$$

$$\lfloor \mathsf{when}\ s\ T \rfloor_E = \begin{cases} (\mathsf{when}\ s\ \lfloor T \rfloor_E) & \text{if } E(s) = \mathit{true} \\ (\mathsf{when}\ s\ T) & \text{otherwise} \end{cases}$$

$$\lfloor \mathsf{watch}\ s\ T \rfloor_E = \begin{cases} () & \text{if } E(s) = \mathit{true} \\ (\mathsf{watch}\ s\ \lfloor T \rfloor_E) & \text{otherwise} \end{cases}$$

We then formalise as follows the execution during an instant of a program P

in the environment E, where we rely on a multi-set notation.

$$(P_1)\ \frac{\forall T \in P\ \ (T,E)\ddagger}{(P,E) \Downarrow (\lfloor P \rfloor_E, E)} \qquad (P_2)\ \frac{\begin{array}{c}\exists T \in P\ \neg(T,E)\ddagger \quad (T,E) \Downarrow^{P'} (T',E') \\ (P\backslash\{\!|T|\!\} \cup \{\!|T'|\!\} \cup P', E') \Downarrow (P'',E'')\end{array}}{(P,E) \Downarrow (P'',E'')}$$

Finally, the input-output behaviour of a program is described by labelled transitions $P \stackrel{I/O}{\rightarrow} P'$ where $I, O \subseteq \mathit{Int}$ are the signals in the interface which are present at the beginning and at the end of the instant, respectively. As in Mealy machines, the transition means that from program (state) P with "input" signals I we move to program (state) P' with "output" signals O. This is formalised by the rule:

$$(I/O) \quad \frac{(P, E_{I,P}) \Downarrow (P',E') \quad O = \{s \in \mathit{Int} \mid E'(s) = \mathit{true}\}}{P \stackrel{I/O}{\rightarrow} P'}$$

$$\text{where:}\quad E_{I,P}(s) = \begin{cases} \mathit{true} & \text{if } s \in I \\ \mathit{false} & \text{if } s \in (\mathit{Int} \cup \mathit{sig}(P)) - I \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

Note that we insist on having all free signals of the program in the domain of definition of the environment.

To conclude this section we give some examples of derived constructions, which are frequently used in the programming practice. In what follows $(\mathsf{local}\ s_1 \cdots (\mathsf{local}\ s_n\ T)\cdots)$ abbreviates as $(\mathsf{local}\ s_1,\ldots,s_n\ T)$, and a similar convention is used for when and watch. Moreover, we assume that the signals that are introduced in the following encodings (i.e. s in now, etc.) are fresh, that is they do not occur in the parameters (i.e. $s \notin \mathit{sig}(T)$, etc.).

$$\begin{aligned}
(\mathsf{await}\ s) &= (\mathsf{when}\ s\ ()) \\
(\mathsf{loop}\ T) &= A(\mathbf{s}) \quad \text{where } \{\mathbf{s}\} = \mathit{sig}(T),\ A(\mathbf{s}) = T; A(\mathbf{s}) \\
(\mathsf{now}\ T) &= (\mathsf{local}\ s\ (\mathsf{emit}\ s); (\mathsf{watch}\ s\ T)) \\
\mathsf{pause} &= (\mathsf{local}\ s\ (\mathsf{now}\ (\mathsf{await}\ s))) \\
(\mathsf{exit}\ s) &= (\mathsf{emit}\ s); \mathsf{pause} \\
(\mathsf{trap}\ s\ T) &= (\mathsf{local}\ s\ (\mathsf{watch}\ s\ T))
\end{aligned}$$

and finally

$$\begin{aligned}(\mathsf{present}\ s\ T\ T') = (\mathsf{local}\ t\ &(\mathsf{thread}\ (\mathsf{watch}\ s\ \mathsf{pause}; (\mathsf{thread}\ T'; (\mathsf{emit}\ t))));\\ &(\mathsf{now}\ (\mathsf{await}\ s); (\mathsf{thread}\ T; (\mathsf{emit}\ t)));\\ &(\mathsf{await}\ t))\end{aligned}$$

The instruction ($\mathsf{await}\ s$) suspends the computation till the signal s is present. The instruction ($\mathsf{loop}\ T$) can be thought of as $T; T; T; \cdots$. Note that in ($\mathsf{loop}\ T$)$; T'$, T' is *dead code*, *i.e.*, it can never be executed. The instruction ($\mathsf{now}\ T$) runs T for the current instant, *i.e.*, if the execution of T is not completed within the current instant then it is terminated. The instruction pause suspends the execution of the thread for the current instant and resumes it in the following one. We may rely on this instruction to guarantee the termination of the computation of each thread within an instant. The constructs trap/exit provide an elementary exception mechanism. The instruction ($\mathsf{present}\ s\ T\ T'$) branches on the presence of a signal. More precisely, if s is emitted during the current instant, this construction spawns the thread T for execution, and blocks T' (which is thrown away at the next instant), while if s is not emitted, the thread T' is executed in the next instant, and T never gets performed.

Remark 2.1 [comparison with [12]] The model we have introduced is largely inspired by the original proposal [12]. The main novelties or variations are: replacing parallel composition, the await and the loop instructions with, respectively, the thread and when constructs, and recursive definitions, and relying on a "big step" operational semantics. We also remark that in the definition of the conditional branching ($\mathsf{present}\ s\ T\ T'$) the expressions T and T' are under a thread instruction. This implies that their execution does *not* depend on when or watch signals that may be on top of them. If this must be the case, then we may prefix T and T' with suitable when and watch instructions.

3 Implementations and applications

Several implementations related to the model described in the previous section have been proposed over the years. Here, we briefly review some of them (in a more or less chronological order), highlighting their main features.

Reactive-C [9] was proposed as a preprocessor of C for assembly-like reactive programming, and it has been used to implement SL. There also exists a reactive library very close to Reactive-C written in Standard ML [24]. Two sets of Java classes have been designed for reactive programming in Java: SugarCubes [13] and Junior [17]. In these implementations, reactive threads are not mapped on Java threads and thus the problems raised by the latter (for example, the limitation on their number or their memory footprints) are avoided. Icobjs [14] is a framework for graphical reactive programming, built on top of SugarCubes. Icobjs have been used for video games, simulations in physics and simulations of the Ambient calculus. Both Java and ML have been extended with reactive primitives, respectively in Rejo [1] and ReactiveML [21]. FairThreads [10,26] and Loft [20] define a thread-based framework in which reactive cooperative threads and preemptive threads can be used jointly. Finally, ULM [8,16] proposes to use reactive programming, enriched with migration primitives, for global computing over the Web. This takes advantage of the fact that reactive programming, as opposed to the synchronous model of ESTEREL for instance, is well-suited for applications involving dynamic concurrency.

Starting from the work initiated by Laurent Hazard on Junior, a lot of effort has been devoted to designing efficient implementations of reactive frameworks. Efficiency mainly comes from the absence of busy-waiting of suspended threads waiting for an event, and from scheduling techniques allowing direct access to the next thread to execute. As examples of efficiency-critical applications recently implemented using the reactive style, we may mention the simulation of a complex network routing protocol for mobile ad-hoc networks described in ReactiveML [21], the implementation of a Web server in Scheme [26], and the implementation of cellular automata in [11], which we shall now describe in some details.

Cellular automata (CA) are used in various simulation contexts, for example, physical simulations, fire propagation, or artificial life. These simulations basically consider large numbers of small-sized identical components, called cells, with local interactions and a global synchronized evolution. Conceptually, the evolution of a CA is decomposed into couples of steps: during the first step, cells get information about the states of their neighbours and during the second step they change their own state according to the information obtained from the previous step. Usually, CA are coded as sequential programs, basically made of a single main loop which considers all cells in turn. Using the reactive style to program cellular automata, where each cell is a reactive thread, has the following advantages:

- Instants naturally represent steps: at each instant, each cell changes its state according to the neighbours states at the previous instant, signals its new state, and then waits for the information about the state of its neighbours.
- The behaviour of cells coded as look-up tables in usual CA implementations is rather opaque. This is generally not felt as a big issue because cells behaviours are often very simple. However, in some contexts, for example artificial life, one may ask for more complex cell behaviours. In these cases, the modularity obtained with reactive programming is an advantage.
- One can obtain efficient implementations of CA spaces in which each cell is implemented as a thread. To improve efficiency, cells can be created only when needed. Note that quiescent cells (with no active neighbour) are just waiting for an activation signal; their presence thus does not introduce any

overhead at execution.

Reactive programming focusses on behaviours rather than on data. Entities found in video games can thus be naturally coded using reactive primitives. Similarly, we have also used the reactive model for interactive simulation of physical systems. Indeed, the reactive style provides us with a very simple and modular way to describe the evolution of complex physical systems. The main features of this approach are simplicity of model construction and high modularity of components. This approach allows us to express both continuous and discrete aspects of a model. For example, consider a planet/meteor system. A planet is implemented with a behaviour which, at each instant, emits a gravity signal with its coordinates. A meteor, at each instant, waits for the gravity signal and moves accordingly. One thus gets systems made of interacting components in which new components can be dynamically added. Applets illustrating this approach, coded in SugarCubes, are available on the Web [25].

4 Some issues

In this section we briefly discuss some issues related to reactive programming.

4.1 Values

Practical programming languages that have been developed on top of the basic reactive model include *data types* beyond pure signals. For instance, we may have the inductive type of booleans $bool = \mathsf{t} \mid \mathsf{f}$, and the inductive type of natural numbers in unary notation $nat = \mathsf{z} \mid \mathsf{s}\ of\ nat$. At the very least, the reactive kernel embedded in a general purpose language should include ways of using the values manipulated in this language. There are two main approaches to adding values to the model: (1) to introduce references as in the ML language, and (2) to assume that signals carry values and that the last emission "covers" in a sense the previous ones (if any). In the latter case, an important design choice to make is to decide what is "the" value associated with a signal at a given instant, and what is the corresponding construct for consulting this value. The simplest model is to regard the value of a signal as ephemeral. That is, the value is updated, as for a reference, by the next emission of the given signal. However, this is not quite compatible with the idea that a signal is broadcast, and that all the running threads have a consistent view of it – either present or absent – at each instant. Therefore, some other mechanisms have been designed. In ESTEREL for instance, one assumes for each type of signal value a function for combining the various values emitted on that signal, and the actual value carried by the signal at some instant is the combination of all the values emitted during this instant (in ESTEREL, with the strong synchrony hypothesis, the combination function should be associative and commutative, since the result should be independent of any scheduling). A similar approach has been followed in SugarCubes [13] and ReactiveML [21]. Notice that in the reactive model, where one cannot statically predict that a signal will or will not be emitted, one has to collect the value of a signal only at the beginning of the next instant. One may also trigger a processing mechanism each time a value is emitted on a signal. Another possibility that is considered in some implementations is to specify, in a receive statement, the rank of the value (in the emission order) in which one is interested.

4.2 Reactivity

A first property that we would like to ensure regarding reactive programs is that they should indeed be reactive, in the following (coinductive) sense:

Definition 4.1 A program P is *reactive* if for every choice I of the input signals there are O, P' such that $P \stackrel{I/O}{\rightarrow} P'$ and P' is reactive.

The reactivity property is not for free. For instance, the thread $A = (\mathsf{await}\ s); A$ may potentially loop within an instant. Whenever a thread loops within an instant the computation of the whole program is blocked as the instant never terminates. One approach to ensure reactivity is to produce a static analysis that guarantees that all loops that may occur within an instant traverse a pause instruction.

While reactivity is a necessary property, it does not guarantee that in practice the program will react for arbitrarily many instants and that this will happen within reasonable time and/or space. A first problem has to do with the implementation of the when and watch instructions. Consider, the thread $A = (\mathsf{local}\ s\ (\mathsf{watch}\ s\ \mathsf{pause}; A))$. Every time the execution crosses the watch instruction it causes the insertion of a new signal s which may potentially abort the execution (although this is not the case with this particular program). Thus the execution of this program may potentially cause a stack overflow. This kind of pathological programs can be removed by a static analysis that checks that there is no loop in the program (possibly going through several instants) that may cause an increase of the stack.

A second problem is due to the fact that the number of (active) threads and signals may grow without limit. Indeed, it can be shown that our basic language is Turing complete. In practice, we need to control the number of threads, and in this respect an interesting feature of the language is the watch instruction which allows to terminate explicitly the execution of a thread (at the end of an instant).

Finally, a third problem, as regards reactivity, is caused by the introduction of data values. The size of the values we are interested in, like lists or trees, is usually not a priori bounded. What does it mean to ensure reactivity in this case? We have in [3,4] considered three increasingly ambitious goals in this respect. A first one is to ensure that every instant terminates. A second one is to guarantee that the computation of an instant terminates within feasible bounds which depend on the size of the parameters of the program at the beginning of the instant. A third one is to guarantee that the parameters of the program stay within certain bounds, and thus the resources needed for the execution of the system are controlled for arbitrarily many instants. In particular, we have been adapting and extending techniques developed in the framework of (first-order) functional languages. The general idea is that polynomial time or space bounds can be obtained by combining traditional termination techniques for term rewriting systems with an analysis of the size of computed values based on the notion of quasi-interpretation ([2,7]). Thus, in a nutshell, ensuring "feasible reactivity" requires a suitable termination proof and bounds on data size.

4.3 Determinism

We say that two programs P, P' are equal up to renaming if there is a bijection from $sig(P)$ to $sig(P')$ that is the identity on the observable signal names in the interface *Int* and that when applied to P produces P'. As usual, an inspection of the semantics shows that the observable behaviour of a program does not depend on the specific choice of its internal signal names. First we define deterministic programs. As with the notion of reactivity, determinism should hold at every instant, and therefore our definition is coinductive.

Definition 4.2 A program P is *deterministic* if for every choice I of the input signals if $P \overset{I/O_1}{\rightarrow} P_1$ and $P \overset{I/O_2}{\rightarrow} P_2$ then $O_1 = O_2$ and $P_1 = P_2$ up to the same renaming, and P_1 is deterministic.

It is immediate to verify that the evaluation of a thread T in an environment E is deterministic. Therefore the only potential source of non-determinism comes from the scheduling of the threads. The basic remark is that the emission of a signal can never block the execution of a statement within an instant. The more we add signals the more the computation of a thread can progress within an instant. Of course, this property relies on the fact that we cannot detect the absence of a signal before the end of the instant.

Proposition 4.3 *All programs are deterministic.*

Clearly, this property is likely to be lost when adding values to the model. Assuming that we have valued signals, consider for instance the program $P = \{\!|(\mathsf{emit}\ s\ \mathsf{t}), (\mathsf{emit}\ s\ \mathsf{f})|\!\}$ where two threads emit the boolean values t and f, respectively, on the signal s. The value which is observed on the signal at the end of the instant depends on the scheduling of the threads (unless the values are combined using an associative and commutative function, as in Esterel). So it seems that we have to accept the idea that when introducing data types the result of the program depends on the scheduler. In practice, one may assume that the scheduler is *deterministic* in the program and the input.

This is a significant difference with preemptive concurrency. In preemptive concurrency, the scheduling policy may depend on factors such as the current workload which are *independent* from the program and the input. Assuming a deterministic scheduler has a positive effect on the process of testing, tracing, and debugging concurrent programs. Besides determinism, it might be reasonable to put additional constraints on the scheduler. One such constraint is the following: if a thread suspends its execution during an instant then all the threads that are ready to run at the moment of the suspension will be given a chance to progress before the computation of the suspended thread is resumed (if ever). With such a scheduler in mind, it makes sense to define:

$$\mathsf{yield} = (\mathsf{local}\ s\ (\mathsf{thread}\ (\mathsf{emit}\ s)); (\mathsf{await}\ s))$$

4.4 Program equivalence

We have described the operational semantics of reactive and deterministic programs as a reaction to a given input, producing a unique output and continuation. Looking for a more abstract, extensional semantics, one possibility is to consider that it is determined by the set $tr(P)$ of infinite traces associated with the possible runs of the program P. Namely:

$$tr(P) = \{(I_1/O_1)(I_2/O_2)\cdots \mid P \stackrel{I_1/O_1}{\rightarrow} P_1 \stackrel{I_2/O_2}{\rightarrow} P_2 \cdots\}$$

Another possibility could be to define a notion of bisimulation. Namely, consider the largest (symmetric) relation R on programs that satisfies the following condition: for every $(P, P') \in R$ and input I, if $P \stackrel{I/O}{\rightarrow} P_1$ then $P' \stackrel{I/O}{\rightarrow} P'_1$ and $(P_1, P'_1) \in R$. It is important to notice that for our deterministic language these two notions coincide.

Proposition 4.4 *Two reactive and deterministic programs are trace equivalent iff they are bisimilar.*

Of course, this reduces considerably the debate on what the right notion of program equivalence is. The notion of weak bisimulation – another familiar concept in the semantics of concurrency – is also missing. However, we must point out that, although the problem of defining program equivalence has an obvious solution, little work has been done so far on the problem of defining and characterising a suitable notion of *thread equivalence* which is preserved by program contexts. Moreover, as we have seen, adding values to the language turns it into a non-deterministic model, for which no notion of equivalence has been investigated so far.

References

[1] Raul Acosta-Bermejo. Reactive Operating System, Reactive Java Objects. *Proc. NOTERE'2000, ENST, Paris*, November 2000.

[2] R. Amadio. Synthesis of max-plus quasi-interpretations. In *Fundamenta Informaticae*, 65(1-2):29–60, 2005.

[3] R. Amadio, S. Dal-Zilio. Resource control for synchronous cooperative threads. In *Proc. CONCUR*, Springer LNCS 3170, 2004.

[4] R. Amadio, F. Dabrowski. Feasible reactivity for synchronous cooperative threads. In preparation.

[5] F. Benbadis and L. Mandel. *Simulation of mobile ad hoc networks in reactive ML*. Pre-print Université Paris 6, available from the authors, 2004.

[6] G. Berry and G. Gonthier, The Esterel synchronous programming language. *Science of computer programming*, 19(2):87–152, 1992.

[7] G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. On termination methods with space bound certifications. In *Proc. Perspectives of System Informatics*, Springer LNCS 2244, 2001.

[8] G. Boudol, ULM, a core programming model for global computing. In *Proc. of ESOP*, Springer LNCS 2986, 2004.

[9] F. Boussinot. Reactive C: An extension of C to program reactive systems. *Software Practice and Experience*, 21(4):401–428, 1991.

[10] Frédéric Boussinot. Fairthreads: mixing cooperative and preemptive threads in C. *Inria research report, RR-5039*, December 2003, to appear in *Concurrency and Computation: Practice & Experience*.

[11] F. Boussinot, Reactive programming of cellular automata. *Rapport de Recherche INRIA 5183*, 2004.

[12] F. Boussinot and R. De Simone, The SL Synchronous Language. *IEEE Trans. on Software Engineering*, 22(4):256–266, 1996.

[13] F. Boussinot and J-F. Susini. The SugarCubes tool box - a reactive Java framework. *Software Practice and Experience*, 28(14):1531–1550, 1998.

[14] Ch. Brunette. A visual reactive framework for dynamic behavior creation. *2nd Workshop on Domain Specific Visual Languages, OOPSLA, Seattle*, 2002.

[15] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *Proc. ACM Conf. on Functional Programming*, 1996.

[16] S. Epardaud. Mobile reactive programming in ULM. *SCHEME Workshop*, 2004.

[17] L. Hazard, J-F. Susini, and F. Boussinot. The Junior reactive kernel. *Inria Research Report*, (3732), 1999.

[18] J. Hopcroft and J. Ullman. Introduction to automata theory, languages, and computation. Prentice-Hall, 1989.

[19] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. IFIP Congress, North-Holland*, 1974.

[20] Loft, `http://www-sop.inria.fr/mimosa/rp/LOFT`.

[21] Louis Mandel and Marc Pouzet. ReactiveML, a reactive extension to ML. In *ACM International conference on Principles and Practice of Declarative Programming (PPDP'05)*, Lisbon, Portugal, July 2005.

[22] R. Milner. Communication and Concurrency. Prentice-Hall, 1989.

[23] J. Ousterhout. Why threads are a bad idea (for most purposes). Invited talk at the USENIX Technical Conference, 1996.

[24] Riccardo Pucella. Reactive programming in Standard ML. *Proceedings of the IEEE International Conference on Computer Languages (ICCL'98)*, pages 48–57, 1998.

[25] Reactive Programming, INRIA, Mimosa Project. `http://www-sop.inria.fr/mimosa/rp`.

[26] Manuel Serrano, Frédéric Boussinot, and Bernard Serpette. Scheme fair threads. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 203–214, New York, NY, USA, 2004. ACM Press.