ReactiveML, a Reactive Extension to ML

Louis Mandel
louis.mandel@lip6.fr

Marc Pouzet
marc.pouzet@lip6.fr

Laboratoire d’Informatique de Paris 6
Université Pierre et Marie Curie
Paris, France

ABSTRACT
We present ReactiveML, a programming language dedicated to the implementation of complex reactive systems as found in graphical user interfaces, video games or simulation problems. The language is based on the reactive model introduced by Boussinot. This model combines the so-called synchronous model found in Esterel which provides instantaneous communication and parallel composition with classical features found in asynchronous models like dynamic creation of processes.

The language comes as a conservative extension of an existing call-by-value ML language and it provides additional constructs for describing the temporal part of a system. The language receives a behavioral semantics à la Esterel and a transition semantics describing precisely the interaction between ML values and reactive constructs. It is statically typed through a Milner type inference system and programs are compiled into regular ML programs. The language has been used for programming several complex simulation problems (e.g., routing protocols in mobile ad-hoc networks).

Categories and Subject Descriptors: F.4.1 [MATHEMATICAL LOGIC AND FORMAL LANGUAGES]: Mathematical Logic–Lambda calculus and related systems; D.3.1 [PROGRAMMING LANGUAGES]: Formal Definitions and Theory–Syntax, Semantics; D.3.2 [PROGRAMMING LANGUAGES]: Language Classification–Applicative (functional) languages, Concurrent, distributed, and parallel languages

General Terms: Languages

Keywords: Functional programming, reactive programming, semantics.

1. INTRODUCTION
Synchronous programming [4] has been introduced in the 80’s as a way to design and implement safety critical real-time systems. It is founded on the ideal zero delay model where communications and computations are supposed to be instantaneous. In this model, time is defined logically as the sequence of reactions of the system to input events. The main consequence of this model is to conciliate parallelism — allowing for a modular description of the system — and determinism. Moreover, techniques were proposed for this parallelism to be statically compiled, i.e., parallel programs are translated into purely sequential imperative code in terms of transition systems [5, 15].

Synchronous languages are restricted to the domain of real-time systems and their semantics has been specifically tuned for this purpose. In particular, they forbid important features like recursion or dynamically allocated data in order to ensure an execution in bounded time and memory. In the 90’s, Boussinot observed that it was possible to conciliate the basic principles of synchronous languages with the dynamic creation of processes if the system cannot react instantaneously to the absence of an event. In this way, logical inconsistencies which may appear during the synchronous composition of processes disappear as well as the need of complex causality analysis to statically reject inconsistent programs. This model was called the synchronous reactive model (or simply reactive) and identified inside SL [11], a synchronous reactive calculus derived from Esterel. Later on, the JUNIOR [16] calculus was introduced as a way to give a semantics to the SUGAR CUBES [12], this last one being an embedding of the reactive model inside JAVA. This model has been used successfully for the implementation of complex interactive systems as found in graphical user interfaces, video-games or simulation problems [13, 12, 1] and appears as a competitive alternative to the classical thread-based approach.

From these first experiments, several embedding of the reactive model have been developed [7, 12, 26, 28]. These implementations have been proposed in the form of libraries inside general purpose programming languages. The “library” approach was indeed very attractive because it gives access to all the features of the host language and it is relatively light to implement. Nonetheless, this approach can lead to confusions between values from the host language used for programming the instant and reactive constructs. This can lead to re-entrance phenomena which are usually detected by run-time tests. Moreover, signals in the reactive model are subject to dynamic scoping rules, making the reasoning on programs hard. Most importantly, implementations of the reactive model have to compete with traditional (mostly sequential) implementation techniques of complex simulation problems. This calls for specific compilation, optimization

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPDP’05, July 11–13, 2005, Lisbon, Portugal.
Copyright 2005 ACM 1-59593-090-6/05/0007 ...$5.00.

This work is supported by the French ACI Sécurité Alidecs.

*This work is supported by the French ACI Sécurité Alidecs.
and program analysis techniques which can be hardly done with the library approach.

The approach we choose is to provide concurrency at language level. We enrich a strict ML language with new primitives for reactive programming. We separate regular ML expressions from reactive ones through the notion of a process. An ML expression is considered to be an atomic (timeless) computation whereas a process is a state machine whose behavior depends on the history of its inputs. It is made of regular ML expressions and reactive expressions. Regular ML expressions are executed as is without any computational impact whereas reactive expressions are compiled in a special way. We introduce two semantics for the language. The first one is a behavioral semantics in the style of the logical behavioral semantics of Esterel. This semantics defines what is a valid reaction no matter how this reaction is actually computed. In order to derive an execution mechanism, we introduce a transition semantics and prove it to be equivalent. Compared to existing semantics for the reactive model (e.g., Junior), these two semantics express precisely the interaction between values from the host language and reactive constructs. Moreover, the language is statically typed through a Milner type system. Compared to the library approach, we believe that the language approach leads to a safer and a more natural programming.

In particular, the language provides a notion of signals with consideration to be instantaneous. Let us consider the approach leads to a safer and a more natural programming.

Section 2 illustrates the expressiveness of the language on some simple examples. A synchronous reactive calculus based on Boussinot’s model is defined in section 3. We embed this kernel inside a call-by-value ML kernel. Section 4 presents its behavioral semantics and establishes its two main properties: in a given environment, a program is deterministic and always progress. Section 5 presents a transition semantics and an equivalence theorem. Section 6 presents the type system which comes as a natural extension of the ML type system of the host language. Implementation issues are addressed in section 7. In section 8, we discuss related works and conclude.

2. LANGUAGE OVERVIEW

2.1 A Short Introduction to ReactiveML

ReactiveML is built above OCaml [19] such that every OCaml program (without objects, labels and functors) is a valid program and ReactiveML code can be linked to any OCaml library.

A program is a set of definitions. Definitions introduce, like in OCaml, types, values or functions. ReactiveML adds the process definition. Processes are state machines whose behavior can be executed through several instants. They are opposed to regular OCaml functions which are considered to be instantaneous. Let us consider the process `hello_world` that prints “hello” at the first instant and “world” at the second one (the `pause` statement suspends the execution until the next instant):

```ocaml
let process hello_world =
    print_string "hello;";
```

This process can be called by writing: `run hello_world`.

Communication between parallel processes is made by broadcasting signals. A signal can be emitted (emit), awaited (await) and we can test its presence (present). The following process emits the signal `z` every time `x` and `y` are synchronous.

```ocaml
let process together x y z =
    loop
        present x then present y then (emit z; pause)
    end
```

Unlike Esterel, it is impossible to react instantaneously to the absence of an event. Thus, the following program:

```ocaml
present x then () else emit x
```

which is incorrect in Esterel — `x` cannot be present and absent in the same instant and is thus rejected by a causality analysis — is perfectly valid in the reactive model. In this model, the absence of `x` is effective in the next instant. Thus, the previous program is equivalent to:

```ocaml
pause; emit x
```

Now, we can write the edge front detector, a typical construct appearing in control systems. The behavior of the process `edge` is to emit `s_out` when `s_in` is present and it was absent in the previous instant.

```ocaml
let process edge s_in s_out =
    loop
        present s_in then pause
            else (await immediate s_in;
                emit s_out)
    end
```

While `s_in` is present, the process emits no value. When `s_in` is absent, no value is emitted at that instant and the control passes through the else branch. At the next instant, the process awaits for the presence of `s_in`. When `s_in` is present then `s_out` is emitted (since `s_in` was necessary absent at the previous instant). The `immediate` keywords states that `s_in` is taking into account even if `s_in` appears at the very first instant.

We now introduce the two main control structures of the language: the construction `do e when s suspends the execution of a process e when the signal s is absent whereas do e until s interrupts the execution of e when s is present. We illustrate these two constructions on a suspend/resume process which control the instant where a process is executed.

We first define a process `sustain` parameterized by a signal `s`. `sustain` emits the signal `s` at every instant.

```ocaml
let process sustain s = loop emit s; pause end
```

We define now an other typical primitive. `switch` is a two states Moore machine which is parameterized by two signals, `s_in` and `s_out`. Its behavior is to start the emission
of a _out_ when _s_in_ is emitted and to sustain this emission while _s_in_ is absent. When _s_in_ is emitted again, the emission of _s_out_ is stopped and the process returns in its initial state.

\[
\text{let process switch } s\text{.in } s\text{.out} = \\
\text{loop} \\
\text{await immediate } s\text{.in}; \\
\text{pause;} \\
\text{do run (sustain } s\text{.out) until } s\text{.in done} \\
\text{end}
\]

We define now the process _suspend_resume_ parameterized by a signal _s_ and a process _p_. This process awaits the first emission of _s_ to start the execution of _p_. Then, each emission of _s_ alternatively suspends the execution of _p_ and resumes it. We implement this process with the parallel composition of (1) a _do/when_ construction that executes _p_ only when the signal _active_ is present and (2) the execution of a switch that controls the emission of _active_ with the signal _s_.

\[
\text{let process suspend_resume } s\text{.p} = \\
\text{signal active in} \\
\text{do run } p \text{ when active} \\
\text{end}
\]

Notice that _suspend_resume_ is an example of a higher-order process since it takes a process _p_ as a parameter.

REACTIVEML also provides valued signals. They can be emitted (_emit signal value_) or awaited to get the associated value (_await signal (pattern) in expression_). Different values can be emitted during an instant, it is called multi-emission. REACTIVEML adopts an original solution for that: when a valued signal is declared, we have to define how to combine values emitted during the same instant. This is achieved with the construction:

\[
\text{signal name default value gather function in expression}
\]

The behavior of multi-emission is illustrated in Fig. 1. We assume signal _s_ declared with the default value _d_ and the gathering function _f_. If values _v_1, ..., _v_n_ are emitted during an instant, then all the _await_ receive the value _v_ at the next instant. Getting the value associated to a signal is delayed

\[
v = (f \, v_1 \ldots \, (f \, v_2 \, (f \, v_1 \, d)) \ldots)
\]

Figure 1: Multi-emission on signal _s_, combined with function _f_, gives the value _v_ at the next instant.

to avoid causality problems. Indeed, as opposed to Esterel and following the reactive approach of Boussinot, the following program _await s(x) in emit s(x+1)_ is causal: the integer value _x_ of _s_ (potentially resulting from the combination of several values) is only available at the end of the instant. Thus, if _x_ = 42 during the current reaction, the program will emit _s_(43) in the following reaction. Notice that this is different from awaiting the signal presence which executes its continuation in the same instant.

The type of the emitted values and the type of the combination’s result can be different. This information is reported in the type of signals. If _τ_1 is the type of the emitted values on a signal _s_ and _τ_2 is the one of the combination, then _s_ has type (_τ_1, _τ_2)_event_.

If we want to define a signal _sum_ that computes the sum of the emitted values, then we can write:

\[
\text{signal sum default 0 gather (+) in ...}
\]

In this case, the program _await sum(x) in print_int x_ awaits the first instant in which _sum_ is emitted and then, at the next instant, prints the sum of the values emitted. _sum_ has type (_int, int)_event_.

An other very useful signal declaration is the one that collects all the values emitted during the instant which is written simply:

\[
\text{signal s in ...}
\]

as a short-cut for:

\[
\text{signal s default Multiset.empty gather Multiset.add in ...}
\]

Here, the default value is the empty set and the gathering function, the addition of an element in a multiset.

### 2.2 The Sieve of Eratosthenes

We consider the sieve of Eratosthenes as it can be found in [18] and is a classical in reactive calculus (see [8], for example). The Eratosthenes sieve is an interesting program because it combines signals, synchronous parallel composition and dynamic creation.

We first write the process _integers_ which generates the sequence of naturals from an integer value _n_.

\[
\text{let rec process integers n s\_out} = \\
\text{emit s\_out n;} \\
\text{pause;} \\
\text{run (integers (n+1) s\_out) }
\]

It is a recursive process that is parameterized by an integer _n_ and a signal _s_out_. Recursive calls are made through a _run_. We can notice that there is no instantaneous recursion because of the _pause_ statement. The type of the process is inferred by the compiler.

Now, we define the process _filter_ which removes all the multiple of some prime number. For this purpose, we define an auxiliary function _not_multiple_. _not_multiple_ is a regular OCAML function which can be used in any other OCAML expression or reactive process.

\[
\text{let not_multiple n p = n mod p <> 0}
\]

\[
\text{val not_multiple : int -> int -> bool}
\]

\[3]In the actual implementation, emitted values are gathered in a list.
let process filter prime s_in s_out =
  loop
    await s_in(n) in
    if not_multiple n prime then emit s_out n
  end
val filter : int -> ('a, int) event -> (int, 'a) event -> process

It is an error to write a reactive construction (such as pause) in a regular OCAML expression and the compiler rejects it. For example, the function let f x = pause; x is rejected.

Now, the process shift creates a new filter process for each newly discovered prime number. We can notice that dynamic creation is done through recursion. Therefore, as opposed to conventional synchronous programming languages, REACTIVEML does not ensure an execution in bounded time and memory but this is not a surprise.

let rec process shift s_in s_out =
  emit s_out prime; (* emit a discovered prime *)
  signal s default 0 gather fun x y -> x in
  run (filter prime s_in s) || run (shift s s_out)
val shift : (int, int) event -> (int, 'a) event -> process

Finally, we define the process output which prints the prime numbers and the main process sieve.

let process output s_in =
  loop await s_in (prime) in print_int prime end
val output : ('a, int) event -> process

let process sieve =
  signal nat default 0 gather fun x y -> x in
  signal prime default 0 gather fun x y -> x in
  run (integers 2 nat) ||
  run (shift nat prime) ||
  run (output prime)
val sieve : process

The gathering functions of the signals nat and prime keep only one of the emitted values.

2.3 Higher Order and Scope Extrusion

We present now an example where processes are emitted on signals. We encode the construction JrDynapar("add", Jr.Halt()) of JUNIOR introduced in [2] for the programming of Agent systems. This process receives some processes on the signal add and executes them in parallel.

let rec process dynapar add =
  await add (p) in
  run p || run (dynapar add)

The emission of processes with free signals can lead to a scope-extrusion problem, a classical phenomenon in process calculi [23]. It can be illustrated on the typical example of a process which emits a process p1 and awaits an acknowledgment of its execution in order to execute a process p2.

let process send add p1 p2 =
  signal ack in
  emit add (process (run p1; emit ack));
  await immediate ack;
  run p2

The expression process (run p1; emit ack) is the definition of an anonymous process that executes p1 and emits ack. In this process, the signal ack is free when it is emitted on add. ack is a local signal and add has a bigger scope, so ack escapes its scope.

3. A SYNCHRONOUS REACTIVE CALCULUS IN ML

We introduce a reactive kernel in which programs given in the introduction can be translated easily. This kernel is built above a call-by-value functional language with an ML syntax. Expressions (e) are made of variables (x), immediate constants (c), pairs (e, e), abstractions (\lambda x.e), applications (e e), local definitions (let x = e in e), recursions (rec x = e), processes (proc e), a sequence (e; e), a parallel synchronous composition of two expressions (e|e), a loop (loop e), a signal declaration (signal x default e gather e2 in e) with a default value e1 and a combination function e2, a test of presence (present e then e else e), an emission of a valued signal (emit e), an instantiation of a process definition (run e), a preemption (do e until e), a suspension (do e when e) and the access to the value of a signal let e(x) in e.

\[
\begin{align*}
e &::= x | c | (e, e) | \lambda x.e | e.e | \text{rec } x = e | \text{proc } e \\
&| e; e | e|e | \text{loop } e | \text{present } e \text{ then } e \text{ else } e \\
&| \text{signal x default e gather e in e} | \text{emit e e} \\
&| \text{let } x = e \text{ in e} | \text{let } e(x) \text{ in e} | \text{run e} \\
&| \text{do e until e} | \text{do e when e}
\end{align*}
\]

In order to separate regular ML programs from reactive constructs, expressions (e) must verify some well formation rules given figure 2. For this purpose, we define the predicate k \vdash e where e is an expression and k \in \{0,1\}. We shall say that an expression e is instantaneous (or combinatorial) when 0 \vdash e can be derived whereas 1 \vdash e means that e is reactive (or sequential to follow classical circuit terminology). A sequential expression is supposed to take time. The rules are defined figure 2. A rule given in the context k (k \vdash e) is a short-cut for the two rules 0 \vdash e and 1 \vdash e. So, for example, it means that a variable or a constant can be used in any context. An abstraction (\lambda x.e) can also be used in an instantaneous expression or in a process but its body must be combinatorial. For a process definition (proc e) the body is typed with the context 1. All the ML expressions are well formed in any context and the expressions like run, loop, or present can be used only in a process. We can notice that there is no rules which conclude that an expression is well formed only in a context 0. Hence, all the combinatorial expressions can be used in a process.

This rules implies some choices in the design of the language. For example, we could allow reactive expressions to

\footnote{For example, the definition let process f x = e1 in e2 is a short-cut for let f = \lambda x. proc e1 in e2 and let f x = e1 in e2 stands for let f = \lambda x.e1 in e2.}
appear in a pair, and thus write:

\[
\frac{k \vdash e_1 \quad k \vdash e_2}{k \vdash (e_1, e_2)}
\]

but in this case, the expression \(\text{emit } s, \text{ pause}\) may have several semantics. If the evaluation order is from left to right, the signal \(s\) is emitted during the first instant while with an evaluation order from right to left the signal is emitted at the second instant. An other choice is to execute both expressions in parallel. We found it more clear to forbid the use of reactive expressions in a pair such that the evaluation order does not matter. A pair will only compose instantaneous computations.

This is essentially a two-level language, separating regular ML expressions used for describing instantaneous computations and reactive constructs for describing the reactive part of a system. In this way, regular ML program shall be executed as is without any computational impact whereas reactive programs will be treated specially. Compilation issues will be discussed in section 7.

Using this kernel, we can derive other operators like the following:

\[
\begin{align*}
\text{emit } e & \quad \frac{d e f}{\text{emit } e ()} \hfill \\
\text{present } e_1 \text{ then } e_2 & \quad \frac{d e f}{\text{present } e_1 \text{ then } e_2 ()} \hfill \\
\text{signal } s \text{ in } e & \quad \frac{d e f}{\text{signal } s \text{ default } \emptyset} \\
\text{pause} & \quad \frac{d e f}{\text{signal } x \text{ in present } x \text{ else } ()} \\
\text{await immediate } s & \quad \frac{d e f}{\text{do } () \text{ when } s} \\
\text{await } s(x) \text{ in } e & \quad \frac{d e f}{\text{await immediate } s; \text{let } s(x) \text{ in } e}
\end{align*}
\]

In \textsc{reactive}\textsc{ml}, signals are always valued. Thus, a pure signal (in the \textsc{esterel} sense) is implemented with a valued signal with value \(0\). At the declaration point of a signal, the programmer must provide a default value \(e_1\) and corresponding to the instants where the signal is not emitted and a combination function \(e_2\). This combination function is used to combine all the values emitted during the same reaction. The construction \text{signal } s \text{ in } p\) is a shortcut for the signal declaration that collects all the values emitted in a multiset. \(\emptyset\) stands for an empty multiset and \(\odot\) is the union (if \(m_1 = \{v_1, \ldots, v_n\}\) and \(m_2 = \{v_1', \ldots, v'_n\}\) then \(m_1 \odot m_2 = \{v_1, \ldots, v_n, v_1', \ldots, v'_n\}\)).

Indeed, as opposed to \textsc{esterel} and following SL \([11]\), the absence of a signal can only be decided at the end of the current reaction. Since \(x\) is not emitted, \texttt{present } x\) will evaluate to false at the end of the reaction so the instruction \(()\) will be executed during the next reaction. The \texttt{await/immediate}\) constructs awaits for the presence of a signal. Awaiting a valued signal can be written \texttt{await } \(s(x)\) in \(e\). The access construction \texttt{let } \(s(x)\) in \(e\) awaits for the end of the instant to get the value transmitted on the signal \(s\) and starts the execution of \(e\) on the next instant. When \(s\) is not emitted, \(x\) takes the default value of \(s\).

4. BEHAVIORAL SEMANTICS

In this section we formalize the execution of a \textsc{reactive}\textsc{ml} program. We base it on a behavioral semantics, in the style of the \textit{logical behavioral semantics} of \textsc{esterel} \([3]\). We define the semantics in two steps. We defines the semantics of instantaneous computations (for which \(0 \vdash e\)) before giving the semantics of sequential computations. Notice that sequential does not mean imperative but it is used like in the circuit terminology. An expression is sequential when its execution can take several instants.

4.1 Instantaneous Computations

Instantaneous expressions (such that \(0 \vdash e\)) are regular ML expression which receive a standard operational semantics. For this purpose, we define the set of values \(\langle v \rangle\) such that:

\[
v ::= c \mid n \mid (v, v) \mid \lambda x.e \mid \text{proc } e
\]

A value can be an immediate constant \(c\), a signal value \(n\) (belonging to a numeric set \(\mathcal{N}\)), an abstraction \((\lambda x.e)\) or a value process \((\text{proc } e)\).

For every instantaneous expression \(e\), we define the predicate \(e \Downarrow v\) stating that \(e\) evaluates to the value \(v\). We use the notation \(e[x \leftarrow v]\) for the substitution of \(x\) by \(v\) in the expression \(e\).

\[
\begin{align*}
\frac{\downarrow v}{e_1 \Downarrow v \quad e_2 \Downarrow v \quad e_3 \Downarrow v} & \quad e_1 + e_2 \Downarrow e_3 \\
\frac{\downarrow v}{e_1 \Downarrow v \quad e_2 \Downarrow v} & \quad e_1 \odot e_2 \Downarrow v \\
\frac{\downarrow v}{e_1 \Downarrow v \quad e_2 \Downarrow v} & \quad e_1 \text{ rec } e_2 \Downarrow v \\
\frac{\downarrow v}{e_1 \Downarrow v \quad e_2 \Downarrow v} & \quad e_1 \odot e_2 \Downarrow v \\
\frac{\downarrow v}{\text{let } x = e_1 \text{ in } e_2 \Downarrow v} & \quad e_1 \Downarrow v \quad e_2 [x \leftarrow v_1] \Downarrow v
\end{align*}
\]

4.2 Sequential Computations

The behavioral semantics describes the reaction of a expression to some input signal. We start with some auxiliary definitions.
Let $\mathcal{N}$, a numerable set of names and $N_1 \subseteq \mathcal{N}$, $N_2 \subseteq \mathcal{N}$. The composition $N_1 \cup N_2$ is the union of the two set and is defined only if $N_1 \cap N_2 = \emptyset$.

A signal environment $S$ is a function:

$$S ::= ([d_1, g_1, m_1]/n_1, ..., (d_k, g_k, m_k)/n_k)$$

A name $n_i$ is associated to a triple $(d_i, g_i, m_i)$ where $d_i$ stands for the default value of $n_i$, $g_i$ stands for a combination function and $m_i$ is the multiset of values emitted during a reaction. If $S(n_i) = (d_i, g_i, m_i)$, we shall write $S^o(n_i) = d_i$, $S^g(n_i) = g_i$ and $S^m(n_i) = m_i$.

We use the notation $(n \in S)$ when the signal $n$ is present (that is, $S^o(n) \neq \emptyset$) and $(n \notin S)$ when the signal is absent (that is, $S^o(n) = \emptyset$).

An event $E$ is a function from names to multisets of values.

$$E ::= [m_1/n_1, ..., m_k/n_k]$$

We take the convention that if $n \notin \text{Dom}(E)$ then $E(n) = \emptyset$.

We define the union of two events $E_1$, $E_2$ as the event $E = E_1 \sqcup E_2$ such that:

$$\forall n \in \text{Dom}(E_1) \cup \text{Dom}(E_2): E(n) = E_1(n) \sqcup E_2(n)$$

And $E = E_1 \cap E_2$ is the intersection of $E_1$ and $E_2$:

$$\forall n \in \text{Dom}(E_1) \cup \text{Dom}(E_2): E(n) = E_1(n) \cap E_2(n)$$

The $+$ operator adds a value $v$ to the multiset of values associated to a signal $n$ in a signal environment $S$.

$$(S + [v/n])(n') = \begin{cases} S(n') & \text{if } n' \neq n \\ S^o(n), S^g(n), S^m(n) \cup \{v\} & \text{if } n' = n \end{cases}$$

And we define the order relation $\sqsubseteq$ on events and lift it to signal environments:

$$E_1 \sqsubseteq E_2 \iff \forall n \in \text{Dom}(E_1): E_1(n) \sqsubseteq E_2(n)$$

$$S_1 \sqsubseteq S_2 \iff \forall n \in \text{Dom}(S_1): S_1(n) \sqsubseteq S_2(n)$$

The reaction of an expression $e$ into $e'$ is defined in a transition relation of the form:

$$N \vdash e \xrightarrow{E, b, S} e'$$

$N$ stands for a set of fresh signal names, $S$ stands for a signal environment containing input, output and local signals and $E$ is the event made of signals emitted during the reaction. $b$ is a boolean value which is true if $e'$ has finished.

The execution of the program is a succession of reactions (potentially infinite). The execution is finished when the termination status $b$ is true. At each instant, the program reads some inputs $(I_i)$ and produces some outputs $(O_i)$ (and local signals). The execution of an instant is defined by the smallest signal environment $S_i$ (for the order $\subseteq$) such that:

$$N_i \vdash e_i \xrightarrow{E, b_i, S_i} e_i'$$

where:

- $O_i \subseteq I_i$, and $(I_i \cup E_i) \subseteq S_i^o$
- $S_i^o \subseteq S_{i+1}^o$ and $S_i^g \subseteq S_{i+1}^g$
- $\forall n \in N_{i+1}, n \notin \text{Dom}(S_i)$

The smallest $S_i$ denotes the signal environment in which the number of present signals is the smallest. This set contains input as well as output signals (this is the property of instantaneous broadcasting of events, that is, all the emitted signal are seen during the current reaction). The conditions $S_{i+1}^o \subseteq S_i^o$ and $S_{i+1}^g \subseteq S_i^g$ mean that the default value and gathering function associated to a signal stay the same during several reactions. We can notice that it is only necessary to keep this information for signals which are still alive at the end of the reaction (they do appear in $e_i$). The condition $\forall n \in N_{i+1}, n \notin \text{Dom}(S_i)$ means that $N_i$ is a set of fresh names.

The behavioral semantics is defined in figure 3. Let us comment the rules.

- The rules for the sequence illustrate the use of the termination status $b$. The expression $e_2$ is executed only if $e_1$ terminates instantaneously ($b = \text{true}$).
- The behavior of the parallel composition is to execute $e_1$ and $e_2$ and to terminate when both branches have terminated.
- The loop is defined by unfolding. The termination status $\text{false}$ guaranty that there is no instantaneous loop.
- The parallel composition is defined in figure 3. Let us comment the rules.
- The default value $(e_1)$ and the gathering function $(e_2)$ associated to $x$ are evaluated at the signal declaration. The name $x$ is substituted by a fresh name $n$ in $e$.
- $e_1$ $e_2$ evaluates $e_1$ into a signal $n$ and adds the result of the evaluation of $e_2$ to the multiset of emitted values on $n$.
- $e(x)$ in $e_1$ is used to get the value associated to a signal. $e$ must be evaluated in a signal $n$ and $v$ is the combination of all the values emitted on $n$ during the instant. The function $\text{fold}$ is defined as follows:
  - $\text{fold} f ([v_1] \cup m) v_2 = \text{fold} f m (f v_1 v_2)$
  - $\text{fold} f \emptyset v = v$

The reaction of the program substitutes $x$ by $v$ in $e_1$. The body is executed at the next instant. This instruction takes one instant because, in the reactive approach, all the emitted signal are known at the end of instant only.

- Let $x = e_1$ in $e_2$ evaluates $e_1$ into $v$ and substitutes $x$ by $v$ in $e_2$. Then it evaluates $e_2$.
- The unit expression $(\emptyset)$ does nothing and terminates instantaneously.

- In a present test, if the signal is present the then branch is executed in the instant, otherwise the else branch is executed at the next instant.
- The do/until corresponds to the suspend construction of Esterel. The difference is that the suspension is not made on the presence of a signal but on the absence. This is due to the reactive approach: the reaction of a signal cannot depend instantaneously on the presence of a signal.
- The behavior of do/until is the same as the kill of SL. This is a weak preemption that takes one instant. Indeed, we cannot have strong preemption to avoid causality problems. For example with a strong preemption the following expression is not causal: $\text{do \; await \; s \; until \; s \; done; \; emit \; s}$.
\[
\begin{align*}
N &\vdash e_1 \quad \frac{E_1, false \quad e_1}{S} \\
N &\vdash e_1; e_2 \quad \frac{E_1, false \quad e_1; e_2}{S} \\
N &\vdash e \quad \frac{E, false \quad e}{S} \\
N &\vdash \text{loop } e \quad \frac{E, false \quad e \cdot \text{loop } e}{S} \\
e_1 \downarrow n \quad e_2 \downarrow v \\
\emptyset \vdash \text{emit } e_1 \quad \frac{[(e/v)/n], \text{true} \quad e_1}{S} \\
\emptyset \vdash () \quad \frac{\text{true} \quad ()}{S} \\
e \downarrow n \quad \emptyset \vdash () \quad \frac{\text{false} \quad ()}{S} \\
\emptyset \vdash \text{do } e \quad \frac{\text{false} \quad \text{do } e}{S} \\
e \downarrow n \quad \emptyset \vdash \text{do } e \quad \frac{\text{false} \quad \text{do } e}{S} \\
N &\vdash e_1 \quad \frac{E_1, true \quad e_1}{S} \\
N &\vdash e_2 \quad \frac{E_2, b_2 \quad e_2}{S} \\
N &\vdash e_1 \quad \frac{E_1, true \quad e_1}{S} \\
N &\vdash e_2 \quad \frac{E_2, b_2 \quad e_2}{S} \\
N &\vdash e_1 \quad \frac{E_1, b_1 \quad e_1}{S} \\
N &\vdash e_2 \quad \frac{E_2, b_2 \quad e_2}{S} \\
N &\vdash e_1 || e_2 \quad \frac{E_1 \lor E_2, b_1 \land b_2 \quad e_1 \land e_2}{S} \\
e_i \downarrow v_1 \quad e_2 \downarrow v_2 \quad S^0(n) = v_1 \quad S^0(n) = v_2 \\
N &\vdash e[x \leftarrow n] \quad \frac{E, b \quad e'}{S} \\
N &\vdash \text{signal } x \quad \frac{e \quad \text{default } e_1 \quad \text{gather } e_2 \quad e \quad S \quad \text{in } e_2 \quad \frac{E, b \quad e'}{S} \\
e \downarrow n \quad \emptyset \vdash \text{let } x = e \quad \frac{E_1, b_1 \quad e_1; e \quad e \quad S \quad \text{in } e_2 \quad \frac{E, b \quad e'}{S} \\
N &\vdash \text{proc } e_1 \quad \frac{E_1, b_1 \quad e_1}{S} \\
N &\vdash \text{run } e \quad \frac{E_1, b_1 \quad e_1}{S} \\
\end{align*}
\]

\[\text{Figure 3: Behavioral Semantics}\]

- **run** \(e\) evaluates the expression \(e\) into a process definition and executes it.

Now, we establish the main properties of the behavioral semantics: that the reaction is deterministic: for any given signal environment there is only one way a program can react. And if a program is reactive [5] (there exists one \(S\) such that \(N \vdash e \rightarrow e'\)), then there exists a unique smallest signal environment in which it can react. The proofs are given in the extended version of the paper [21].

The combination of these properties insures that every reactive programs can be executed in REACTIVEML. Notice, this property is not verified a priori in ESTEREL and needs a causality analysis.

**Lemma 1.** For every expression \(e\), the behavioral semantics of \(e\) is deterministic, i.e.: \(\forall e, \forall S, \forall N:\) if \(n \in \text{Dom}(S) : S^0(n) = f(f(x, f(y, z))) = f(y, f(x, z))\) and \(N \vdash e \quad \frac{E_1, b_1 \quad e_1 \quad E_2, b_2 \quad e_2}{S} \) then \((E_1 = E_2 \land b_1 = b_2 \land e_1 = e_2)\).

The associativity and commutativity of the gathering function expresses the fact that the order of emissions during an instant is not specified. It is a strong constraint. But even if it is not satisfied a program can be deterministic. For example if there is no multi--emissions the gathering function does not have to be associative and commutative.

**Lemma 2.** For every expression \(e\), let \(S\) such that \(S = \{ S \mid \exists E, N, b : N \vdash e \quad \frac{E, b \quad e'}{S} \} \).

then there exists a (unique) smallest signal environment \((\forall S)\) such that \(\exists E, N, b : N \vdash e \quad \frac{E, b \quad e'}{S} \).

The proof of this lemma is based on the following lemma which states that if an expression can react in two different environments then it can react in the intersection of these environments. This lemma is based on the absence of instantaneous reactions to the absence of a signal. Indeed contrary to ESTEREL the absence of a signal cannot generate the emission of other signals. For example, in ESTEREL, the following program emits \(s2\) if \(s1\) is absent, but in REACTIVEML the emission of \(s2\) is delayed to the next instant such that the absence can emit signals during the instant:

**Lemma 3.** Let \(S_1, S_2, S_3\) and \(e\) such that \(N_1 \vdash e \quad \frac{E_1 \quad \text{false} \quad e_1}{S_1} \)

\(N_2 \vdash e \quad \frac{E_2 \quad \text{false} \quad e_2}{S_2} \) and \(S^0 = S_1 \cap S_2 \) then there exists \(S_3\) such that \(N_3 \vdash e \quad \frac{E_3 \quad \text{false} \quad e_3}{S_3} \).

5. OPERATIONAL SEMANTICS

The previous semantics is not operational since it express what the reaction should verify and not how reactions are computed. In particular, the signal environment has to be guessed. We present now a small step semantics where the reaction build the signal environment. An instant is made into two steps. The first one is an extension of the reduction
semantics of ML. The second one, name the end of instant’s reaction, prepares the next instant’s reaction.

5.1 Reduction Semantics

The reaction of an instant starts with a sequence of reactions of the form:

\[ e/S \rightarrow e'/S' \]

Contrary to the previous semantics, the signal environment \( S \) is built during the reaction.

To define the reaction \( \rightarrow \), we start with the axioms for the relation of head reduction (\( \rightarrow \)) figure 4.

- The let’s axiom substitutes \( x \) by \( v \) in \( e \).
- The rule of the sequence remove the left branch when this is a value.
- When the two branches of a parallel are values, the parallel is reduced into the value (\( \circ \)).
- The loop duplicates its body.
- The run instruction applied to a process definition executes it.
- emit \( n \) \( v \) is reduced into (\( \circ \)) and adds \( v \) to the multiset of values emitted on \( n \).
- The present construction can be reduced only if the signal is present in the environment.
- The declaration of a signal \( x \) substitutes \( x \) by \( n \) in \( e \). \( n \) is a fresh name taken the set \( \mathcal{N} \). \( n \) is added to the signal environment with the default value \( v_1 \) and the gathering function \( v_2 \). Initially, the multiset of values associated to \( n \) is empty.
- When the body of a do/until construct is a value, it means that it reaction is finished. So, the do/until can be reduced into (\( \circ \)).
- The do/when can be reduced into (\( \circ \)) only when its body is a value and when the signal is present.

From this axioms, we define the reduction (\( \rightarrow \)):

\[
\begin{align*}
\frac{e/S \rightarrow e'/S'}{\Gamma(e)/S \rightarrow \Gamma(e')/S'} & \quad e_{nv} \vdash v \\
\frac{\Gamma(e)/S \rightarrow \Gamma(e)/S'}{\Gamma(e_{nv})/S \rightarrow \Gamma(v)/S} \\
\frac{n \in S \quad e/S \rightarrow e'/S'}{\Gamma(\text{do } e \text{ when } n)/S \rightarrow \Gamma(\text{do } e' \text{ when } n)/S'}
\end{align*}
\]

where \( \Gamma \) is a context with one hole. With the first rule, if an expression \( e \) head reduces to \( e' \), then \( e \) can be reduced in any context. The second rule defines the execution of combinatorial expressions. \( e_{nv} \) must be an expression which is not a value to avoid infinite reductions. The last rule is the suspension. The body of a do/when can be executed only if the signal is present.

The contexts are defined as follow:

\[
\Gamma ::= \emptyset \mid \text{let } x = \Gamma \text{ in } e \mid \Gamma ; e \\
\Gamma; e \mid e; \Gamma \mid \text{run } \Gamma \mid \text{emit } \Gamma \mid e \mid \text{emit } e \Gamma \\
\text{let } \Gamma(x) \text{ in } e \mid \text{present } \Gamma \text{ then } e \text{ else } e \\
\text{signal } x \text{ default } \Gamma \text{ gather } e \text{ in } e \\
\text{signal } x \text{ default } e \text{ gather } \Gamma \text{ in } e \\
\text{do } e \text{ until } \Gamma \mid \text{do } \Gamma \text{ until } n \mid \text{do } e \text{ when } \Gamma
\]

The contexts for the parallel composition show that the evaluation order is not specified. In the implementation of REACTIVEML, the scheduling is fixed such that the execution is always deterministic but this is not specified.

5.2 End of Instant’s Reaction

The reactive model is based on the absence of instantaneous reaction to the absence of a signal such that the treatment of the absence to prepare the reaction for the next instant can only be done at the end of instant.

The reaction of an instant is stopped when there is no more (\( \rightarrow \)) reductions possible. From this point, the signal environment cannot change, there is no more signal emission. So, all the signals not emitted are consider to be absent.

The rules for the end of instant’s reaction are of the form: \( S \vdash e \rightarrow e' \) and are defined figure 5. We can notice that the rules are not given for all the expressions because they are applied only when the program cannot be reduced with (\( \rightarrow \)). Let’s comment the rules of figure 5:

- Values do not change at the end of an instant.
- The reaction of the parallel composition is the reaction of the two branches.
- Only the left branch of the sequence reacts because the right branch is not activated during the instant.
- If there is a present instruction, the signal is considered to be absent. So the else branch has to be executed at the next instant.
- The let \( n(x) \) in \( e \) gets the values associated to the signal \( n \) and combines them with the function \( f \) to obtain the value \( v \). Then \( x \) is substituted by \( v \) in \( e \) for the next instant. If \( n \) has not been emitted \( v \) is equal to \( d \)
- The preemption occurs at the end of instant. If the signal that control the do/until is present, the expression has to be preempted. In this case, the do/until is rewritten into (\( \circ \)).
- For the do/when, if the signal is present then the body must be activated at the end of instant. If the signal is absent, the body is not activated because it has not been activated during the instant.

5.3 Execution of a Program

The reaction of an instant is defined by the relation:

\[
e_i/S_i \Rightarrow e'_i/S'_i
\]

If we note \( I_i \) the inputs of the reaction and \( O_i \) the outputs, the signal environment have the following properties. All the signals that are not in \( I_i \) are initially absent (\( S_i^0 = I_i \)). The outputs are a subset of the signal environment at the end of the reaction (\( O_i \subseteq S_i^0 \)). The default values and the gathering functions are kept for an instant to the other (\( S_i^{old} \subseteq S_{i+1}^{old} \) and \( S_i^{new} \subseteq S_{i+1}^{new} \)).

The execution of an instant is made of two steps. The reduction of \( e_i \) until a fix point is reached. Then there is the end of instant’s reaction.

\[
\frac{e_i/S_i \Rightarrow e_i''/S''_i \quad S_i' \vdash e_i'' \rightarrow e_i'}{e_i/S_i \Rightarrow e_i''/S''_i}
\]

Where \( e/S \rightarrow e'/S' \) if \( e/S \rightarrow^* e''/S'' \) and \( e'/S' \neq \). The relation \( \rightarrow^* \) is the reflexive and transitive closure of \( \rightarrow \).
5.4 Equivalence

In this section we show the equivalence between the two semantics.

We start with the proof that if an expression $e$ reacts into an expression $e'$ with the small step semantics then it can react in the same signal environment with the big step semantics.

**Lemma 4.** For every $S_{\text{init}}$ and $e$ such that $e / S_{\text{init}} \Rightarrow e' / S$ then there exists $N, b$ such that $N \vdash e \xrightarrow{S} e'$ with $E = S^{v} \setminus S_{\text{init}}^{v}$.

**Proof.** By induction on the number of $\Rightarrow$ reductions in $e / S_{\text{init}} \Rightarrow e' / S$.

- If there is no $\Rightarrow$ reduction possible, we have to prove that the reduction $\Rightarrow_{\text{coi}}$ is the same that the big step semantics (cf. lemma 5).

- If there is at least one $\Rightarrow$ reduction, we have to prove that one $\Rightarrow$ reduction followed by a big step reaction is equivalent to one big step reaction (cf. lemma 6).

□

The proof is based on the following properties:

**Lemma 5.** If $e / S \not\Rightarrow$ and $S \vdash e \Rightarrow_{\text{coi}} e'$ then there exists $N$ and $b$ such that $N \vdash e \xrightarrow{S} e'$.

**Proof.** The proof is made by structural induction. We have just to notice that if an expression $e$ reacts with the big step semantics into $e'$ and the termination status is true ($N \vdash e \xrightarrow{S} e'$) then $e'$ behaves as $\top$ ($\forall N', S'. N' \vdash e \xrightarrow{S} e'$).

□

**Lemma 6.** If $e / S_{0} \not\Rightarrow e_{1} / S_{1}$ and $N \vdash e \xrightarrow{S} e'$ with $S_{1} \subseteq S$ then $N \vdash e \xrightarrow{S} e'$ with $E = E' \cup (S_{1} \setminus S_{0})$.

**Proof.** The proof is made into two parts. First we prove the same property for the $\Rightarrow$ reduction. Then we show that this is true in any context. □

Now the following lemma shows that if an expression can react with the two semantics then the signal environment and the expression obtained at the end of the reaction are the same.

**Lemma 7.** For every $S_{\text{init}}$ and $e$ such that:

- $N_{1} \vdash e \xrightarrow{S_{1}} e_{1}$ where $S_{1}$ is the small signal environment such that $S_{\text{init}} \subseteq S_{1}$

- $e / S_{\text{init}} \Rightarrow e_{2} / S_{2}$

- $\forall n \in \text{Dom}(S_{2}) : S_{2}^{v}(n) = f$ and $f(x, y, z) = f(y, f(x, z))$, then $e_{1} = e_{2}$ and $S_{1} = S_{2}$.

**Proof.** With lemma 4, there exists $N_{2}, E_{2}$ and $b_{2}$ such that $N_{2} \vdash e \xrightarrow{S_{2}} e_{2}$ and we can notice, by construction, $S_{2}$ is the smallest signal environment such that $S_{\text{init}} \subseteq S_{2}$.

$N_{1}$ and $N_{2}$ are the sets of fresh names use during the reactions. With some renaming, we can have a set $N$ such that $N \vdash e \xrightarrow{S_{1}} e_{1}$ and $N \vdash e \xrightarrow{S_{2}} e_{2}$.

With lemma 2, we know that there is a unique smallest signal environment in which an expression can react with the big step semantics so $S_{1} = S_{2}$. Now with the determinism (lemma 1) we have $E_{1} = E_{2}, b_{1} = b_{2}$ and $e_{1} = e_{2}$. □

The details of the proofs are given in an extended version of the paper [21].

$^5$It is available at www-spi.lip6.fr/~mandel/rml.
6. STATIC TYPING

We provide a type system as a conservative extension of the Milner type system of ML [22]. In doing so, we have to deal with signals and in particular values which can be transmitted on signals. The type language is:

\[ \sigma ::= \forall \alpha_1, \ldots, \alpha_n. \tau \]
\[ \tau ::= T \mid \alpha \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \text{process} \mid (\tau, \tau) \text{event} \]
\[ T ::= \text{int} \mid \text{bool} \mid \ldots \]

Types are separated in regular types (\(\tau\)) and type schemes (\(\sigma\)). A type (\(\tau\)) may be a basic type (\(T\)), a type variable (\(\alpha\)), a function type (\(\tau_1 \rightarrow \tau_2\)), a product type (\(\tau_1 \times \tau_2\)) or a process type (\(\text{process}\)) or the type of a signal ((\(\tau_1, \tau_2\) event)).

In the type of a signal, \(\tau_1\) is the type of the emitted value and \(\tau_2\) is the type of the read value (obtained after collecting all the emitted values during an instant).

A typing environment \(H\) has the following form:

\[ H ::= [x_1 : \sigma_1; \ldots; x_k : \sigma_k] \]

The instantiation and generalization is defined like the following:

\[ r^*[(\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n)] \leq \forall \alpha_1, \ldots, \alpha_n. \tau \]

\[ \text{Gen}(\tau, H) = \forall \alpha_1, \ldots, \alpha_n. \tau \]

where \(\{\alpha_1, \ldots, \alpha_k\} = FV(\tau) - FV(H)\)

Expressions are typed in an initial typing environment \(TC\) such that:

\[ TC = [\text{true} : \text{bool}; \text{fst} : \forall \alpha, \beta. \alpha \times \beta \rightarrow \alpha; \ldots] \]

Expressions are typed by asserting the judgment \(H \vdash e : \tau\) which states that the expression \(e\) has type \(\tau\) in the typing environment \(H\). The predicate is defined in figure 6.

The typing rules for ML expressions are not modified. In the typing of signal, the default value \((e_1)\) has the type of the associated value and the gathering function \((e_2)\) is a function of an emitted value and of the combination of the previous emitted values and returns the new combination. The rule for emit checks that the first argument has a signal type, and that the first parameter of this type and the type of the value emitted are the same. let \(e_1(x)\) in \(e\) gets the value associated to a signal. So, if \(e_1\) has type \((\tau, \tau)\) event, \(x\) must have type \(\tau^*\). The instantiation run is applied to a process. Finally, the present, until and when constructions can be applied to any signal.

The safety of the type system is proved with standard techniques [25].

7. IMPLEMENTATION, EXPERIMENTS

We followed a very pragmatic approach in the design of the language and efficiency was one of our major concern. We built REACTIVEML as an extension of a subset of OCAML (without objects, labels and functions) which can mix reactive processes and regular OCAML expressions. We choose OCAML with the following idea in mind: OCAML will provide modular data and control structures for programming the algorithmic part of the system whereas reactive constructs will provide modular control structures for describing the temporal aspect.

The compilation of a REACTIVEML program processes as follows: programs are first typed before being translated into OCAML code. This code can in turn be linked with other REACTIVEML programs or OCAML libraries. This translation leaves unchanged regular ML expressions (only the type information is used) whereas every reactive construction is translated into a combinator defined in OCAML. Reactive programs can finally be executed by linking them with an ad-hoc OCAML library.

As opposed to classical synchronous programs, reactive programs are no more statically scheduled. Programs are rather scheduled dynamically or interpreted according to the actual dependences between instructions reading or emitting signals in the programs. The scheduling strategy we have implemented is a greedy strategy reminiscent to a technique introduced by Hazard 6, known as one of the most efficient scheduling technique for JUNIOR. The precise description of the scheduling technique we have implemented in REACTIVEML is outside the scope of this paper. Let us give an intuitive presentation.

The scheduling is based on the use of waiting queues such that an action is fired only when the signal it is waiting for is emitted. During the execution, the interpreter keeps track of the set \(W\) of actions waiting for the presence of a signal during one instant. When \(W\) is not empty at the end of the instant, pertinent informations are transfered to the next instant.

In order to implement a greedy scheduling technique, we associate two waiting queues for every signal. One queue is used for instructions waiting only one instant (e.g., present) and the other queue is used for instructions that can wait for more than one instant (e.g., do/when). Thus, if the execution of some code is stopped on the test of a signal then the code to be executed is recorded in the appropriate waiting queue. Otherwise, its continuations are put in the set of actions to be executed in the current instant (C). Therefore the execution of an instant consists in the execution of all the ready actions of C. The end of the instant is decided when C is empty. Instructions which are in the short-term waiting queues can be treated to prepare the next instant.

With this scheduling strategy, a fast access to signals (for presence information and waiting queues) is crucial. Almost all implementations of the reactive approach use dedicated hash tables during the execution for representing the signal environment. In our implementation signals are represented as regular values which are automatically garbage collected by OCAML when possible. Moreover, the presence information and associated waiting queues is done in constant time. The efficient representation of signals together with the absence of busy waiting during the execution are central in order to be able to program real-size problems.

Several applications have been written in REACTIVEML, ranging from simple graphical systems to complex simulation problems. In particular, we have rewritten classical cellular automata programs written in LOFT by Boussinot [10] to serve as benchmarks for testing the efficiency of our implementation. This example puts emphasis on the absence of busy waiting. Quiescent cells are stopped on the waiting of an activation signal such that only active cells are executed. Figure 7 compares the execution times given for LOFT, REACTIVEML and an imperative version written in OCAML. The imperative version scans the array of cells with for loops. The numbers show that REACTIVEML and the LOFT library written in Care both as fast.

The main application written in REACTIVEML is a simula-

---

6Through being well known in the synchronous community, this technique has unfortunately never been published so far and can only be appreciated through a careful reading of the code.
tor of a complex network routing protocol for mobile ad-hoc networks [3, 20], done in collaboration with F. Benbadis and the need of complex data-structure | combined to the
ery node is able to move, nodes evolve concurrently and characteristics | concurrency with many communications
in NAB [24]. A project is under way for using
conventional programming languages. The
imulation tools (e.g., NS, OPNET) inadequate and network
implementation showed that the reactive model introduced
by Boussinot provides adequate programming constructs | namely synchronous parallel composition, broadcast com-
munication and dynamic creation — which allow for a natural im-
plementation of the hard part of the simulation. The complete implementation (with graphical interface, statistics) is about 1000 lines. Experiments show that the REAC-
tiveML version is two order of magnitude faster than the original C version; it was able to simulate more that 1000 nodes where the original C version failed (after 200 nodes) and is faster than the ad-hoc version directly programmed in NAB [24]. A project is under way for using REACTIVEML for simulating network sensors, taking into account the temporal aspects of nodes (e.g., energy consumption or failure) and to connect REACTIVEML with automatic test sequences generators such as LURETTE [17].

8. CONCLUSION AND RELATED WORKS
In this paper, we have presented an extension of an exist-
8. CONCLUSION AND RELATED WORKS
In this paper, we have presented an extension of an exist-

ing strict ML language with reactive constructs. The result
language is dedicated to the implementation of complex dy-
namic systems as found in graphical interfaces, video games and simulation problems.

Compared to existing embedding of the reactive approach in either an imperative language [8] or an object-oriented language [2], the present work provides a complete semantics of the embedding. This allows a precise understanding of the communication between the two levels and reveals, in particular, classical problems appearing in process calculi such as scope-extrusion phenomena.

The Fair Threads [9, 28] are an extension of the reac-
tive approach that allows to mix cooperative and preemptive
executing a reference which is not local is delayed until it be-
comes present. We did not address mobility issues and thus,

accessing a reference is instantaneous. In

Figure 6: The Type System

Figure 7: Average of execution time of one instant for a 500x500 Fredkin's cellular automata.

or of a complex network routing protocol for mobile ad-hoc networks [3, 20], done in collaboration with F. Benbadis (from the Network team at LIP6, Paris). Mobile ad-hoc networks are highly dynamic networks characterized by the absence of physical infrastructure. In such networks, every node is able to move, nodes evolve concurrently and synchronize continuously with their neighbors. Due to mobility, connections in the network can change dynamically and nodes can be added or removed at any time. All these characteristics — concurrency with many communications and the need of complex data-structure — combined to the routing protocol specifications make the use of standard sim-
ulation tools (e.g., NS, OPNET) inadequate and network protocols appear to be very hard to program efficiently in conventional programming languages. The REAC-
tiveML implementation showed that the reactive model introduced by Boussinot provides adequate programming constructs — namely synchronous parallel composition, broadcast com-
munication and dynamic creation — which allow for a natural im-
plementation of the hard part of the simulation. The complete implementation (with graphical interface, statistics) is about 1000 lines. Experiments show that the REAC-
tiveML version is two order of magnitude faster than the original C version; it was able to simulate more that 1000 nodes where the original C version failed (after 200 nodes) and is faster than the ad-hoc version directly programmed in NAB [24]. A project is under way for using REACTIVEML for simulating network sensors, taking into account the temporal aspects of nodes (e.g., energy consumption or failure) and to connect REACTIVEML with automatic test sequences generators such as LURETTE [17].

Figure 6: The Type System

Figure 7: Average of execution time of one instant for a 500x500 Fredkin’s cellular automata.
a consequence, some overhead is imposed on the execution on regular ML expressions. Indeed, reactive code is transformed into continuation-passing style by CPS transformation, whereas OCaml code does not have to be modified. We know that ML code cannot be interrupted, so we do not have to introduce some mechanism to save the execution context.

**ConcurrentML** [27] is a language that support concurrent programming and functional programming. As opposed to **REACTIVEML**, it is asynchronous. The communication between processes is made by communication channels or shared memory. To control concurrent access to the memory, **ConcurrentML** uses semaphores, mutex locks and condition variables, whereas in **REACTIVEML** we do not have to use them because instantaneous actions are atomic.

**Functional Reactive Programming** [29] and **Lucid Synchrone** [14] combines reactive and functional programming. Compared to **REACTIVEML**, they are based on a data flow approach which leads to a very different style of programming.

The language is still young and several extensions can be considered. One of them concerns efficient implementation techniques in order to use **REACTIVEML** for programming real-size simulation problems and to be a convincing alternative to traditional methods. For example, the recognition of subparts of a reactive program which can be compiled (that is, statically scheduled) is still open. Whereas causality inconsistencies are eliminated in the model of Boussinot, the scope extrusion phenomena (which is absent in existing synchronous languages) make this compilation difficult and calls for new program analysis.

## 9. REFERENCES