ReactiveML, a Reactive Extension to ML[*]

extended version

Louis Mandel and Marc Pouzet
Université Pierre et Marie Curie
LIP6 [†]

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).

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, 14].

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 [15] calculus was introduced as a way to give a semantics to the SugarCubes [13], 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 graphics user interfaces, video-games or simulation problems [12, 13, 2] 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, 13, 24, 26]. 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 languages 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 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 be-

[†] Laboratoire d'Informatique de Paris 6, Université Pierre et Marie Curie, 8 rue du Capitaine Scott, 75015 Paris, France. email: {Louis.Mandel, Marc.Pouzet}@lip6.fr

[*] This work is supported by the French ACI Sécurité Alidecs.

havior 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 and this is new. 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 regular scope properties. Moreover, some parts of a program can be compiled *vs* interpreted, leading to a far more efficient execution.

Section 2 illustrates the expressiveness of the language on some simple examples.[1] 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 establish 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 [18] 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):

```
let process hello_world =
  print_string "hello␣";
  pause;
  print_string "world"
```

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.

```
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:

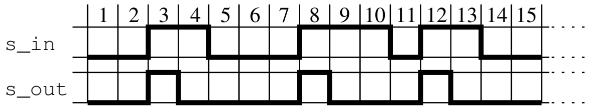
```
present x then () else emit x
```

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:

```
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.

```
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.

```
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 `s_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.

```
let process switch s_in s_out =
  loop
    await immediate s_in;
    pause;
    do run (sustain s_out) until s_in done
  end
```

[1] The distribution of REACTIVEML and complete examples can be found at `www-spi.lip6.fr/~mandel/rml`.

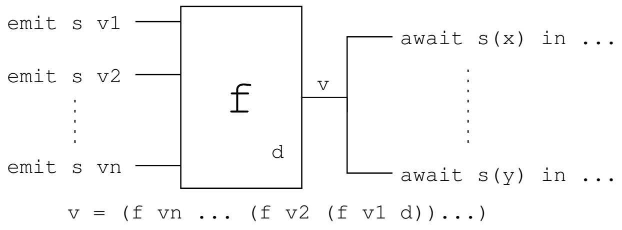
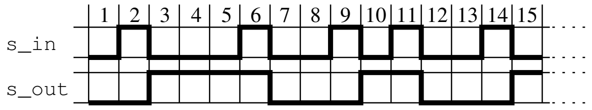


Figure 1: Multi-emission on signal s, combined with function f, gives the value v at the next instant.



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`.

```
let process suspend_resume s p =
  signal active in
  do run p when active
  ||
  run (switch s active)
```

Notice that `suspend_resume` is an example of a higher-order process since it takes a process `p` as a parameter.

ReactiveML also provides valuated 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:

`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 `v1`, ..., `vn` are emitted during an instant, then all the `await` receive the value `v` at the next instant. [2] Getting the value associated to a signal is delayed 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:

```
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:

```
signal s in ...
```

as a short-cut for:

```
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. [3]

2.2 The Sieve of Eratosthenes

We consider the sieve of Eratosthenes as it can be found in [17] 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`.

```
let rec process integers n s_out =
  emit s_out n;
  pause;
  run (integers (n+1) s_out)
val integers : int -> (int, 'a) event -> process
```

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.

```
let not_multiple n p = n mod p <> 0
val not_multiple : int -> int -> bool

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, 'b) 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.

[2] `v = (f vn ... (f v2 (f v1 d))...)`

[3] In the actual implementation, emitted values are gathered in a list.

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 =
  await s_in(prime) in
  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 `Jr.Dynapar("add", Jr.Halt())` of Junior introduced in [1] 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 [21]. 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.[4] 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 ($\mathtt{let}\ x\ \mathtt{=}\ e\ \mathtt{in}\ e$), recursions ($\mathtt{rec}\,x\,\mathtt{=}\,e$), processes ($\mathtt{proc}\ e$), a sequence ($e\mathtt{;}e$), a parallel synchronous composition of two expressions ($e\mathtt{||}e$), a loop ($\mathtt{loop}\,e$), a signal declaration ($\mathtt{signal}\,x\,\mathtt{default}\,e\,\mathtt{gather}\,e\,\mathtt{in}\,e$) with a default value e_1 and a combination function e_2, a test of presence ($\mathtt{present}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e$), an emission of a valued signal ($\mathtt{emit}\ e\ e$), an instantiation of a process definition ($\mathtt{run}\ e$), a preemption ($\mathtt{do}\ e\ \mathtt{until}\ e$), a suspension ($\mathtt{do}\ e\ \mathtt{when}\ e$) and the access to the value of a signal $\mathtt{let}\ e(x)\ \mathtt{in}\ e$.

$$
\begin{array}{rcl}
e & ::= & x \mid c \mid (e, e) \mid \lambda x.e \mid e\ e \mid \mathtt{rec}\,x\,\mathtt{=}\,e \mid \mathtt{proc}\ e \\
 & & \mid e\mathtt{;}e \mid e\mathtt{||}e \mid \mathtt{loop}\ e \mid \mathtt{present}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e \\
 & & \mid \mathtt{signal}\ x\ \mathtt{default}\ e\ \mathtt{gather}\ e\ \mathtt{in}\ e \mid \mathtt{emit}\ e\ e \\
 & & \mid \mathtt{let}\ x\ \mathtt{=}\ e\ \mathtt{in}\ e \mid \mathtt{let}\ e(x)\ \mathtt{in}\ e \mid \mathtt{run}\ e \\
 & & \mid \mathtt{do}\ e\ \mathtt{until}\ e \mid \mathtt{do}\ e\ \mathtt{when}\ e \\
 & & \\
c & ::= & \mathtt{true} \mid \mathtt{false} \mid \mathtt{()} \mid \mathtt{0} \mid \ldots \mid \mathtt{+} \mid \mathtt{-} \mid \ldots
\end{array}
$$

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 ($\mathtt{proc}\ e$) 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 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 `(emit s, 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.

[4] For example, the definition $\mathtt{let\ process}\ f\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2$ is a short-cut for $\mathtt{let}\ f\ \mathtt{=}\ \lambda x.\mathtt{proc}\ e_1\ \mathtt{in}\ e_2$ and $\mathtt{let}\ f\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2$ stands for $\mathtt{let}\ f\ \mathtt{=}\ \lambda x.e_1\ \mathtt{in}\ e_2$.

$$k \vdash x \qquad k \vdash c \qquad \frac{0 \vdash e}{k \vdash \lambda x.e} \qquad \frac{1 \vdash e}{k \vdash \texttt{proc}\ e} \qquad \frac{0 \vdash e_1 \quad 0 \vdash e_2}{k \vdash e_1\ e_2} \qquad \frac{0 \vdash e_1 \quad 0 \vdash e_2}{k \vdash (e_1, e_2)} \qquad \frac{0 \vdash e_1 \quad k \vdash e_2}{k \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2} \qquad \frac{0 \vdash e}{1 \vdash \texttt{run}\ e}$$

$$\frac{1 \vdash e}{1 \vdash \texttt{loop}\ e} \qquad \frac{0 \vdash e_1 \quad 0 \vdash e_2}{1 \vdash \texttt{emit}\ e_1\ e_2} \qquad \frac{0 \vdash e}{k \vdash \texttt{rec}\ x\,\texttt{=}\,e} \qquad \frac{1 \vdash e_1 \quad 1 \vdash e_2}{1 \vdash e_1 \texttt{||} e_2} \qquad \frac{k \vdash e_1 \quad k \vdash e_2}{k \vdash e_1 \texttt{;} e_2} \qquad \frac{0 \vdash e \quad 1 \vdash e_1 \quad 1 \vdash e_2}{1 \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2}$$

$$\frac{0 \vdash e_1 \quad 0 \vdash e_2 \quad 1 \vdash e}{1 \vdash \texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e} \qquad \frac{0 \vdash e_1 \quad 1 \vdash e_2}{1 \vdash \texttt{let}\ e_1(x)\ \texttt{in}\ e_2} \qquad \frac{0 \vdash e_1 \quad 1 \vdash e_2}{1 \vdash \texttt{do}\ e_2\ \texttt{until}\ e_1} \qquad \frac{0 \vdash e_1 \quad 1 \vdash e_2}{1 \vdash \texttt{do}\ e_2\ \texttt{when}\ e_1}$$

Figure 2: Well formation rules

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:

$$\begin{array}{lcl}
\texttt{emit}\ e & \overset{def}{=} & \texttt{emit}\ e\ \texttt{()} \\
\texttt{present}\ e_1\ \texttt{then}\ e_2 & \overset{def}{=} & \texttt{present}\ e_1\ \texttt{then}\ e_2\ \texttt{else ()} \\
\texttt{present}\ e_1\ \texttt{else}\ e_2 & \overset{def}{=} & \texttt{present}\ e_1\ \texttt{then ()}\ \texttt{else}\ e_2 \\
\texttt{signal}\ s\ \texttt{in}\ e & \overset{def}{=} & \texttt{signal}\ s\ \texttt{default}\ \emptyset \\
 & & \ \texttt{gather}\ \lambda x.\lambda y.\{x\} \uplus y\ \texttt{in}\ e \\
\texttt{pause} & \overset{def}{=} & \texttt{signal}\ x\ \texttt{in present}\ x\ \texttt{else ()} \\
\texttt{await immediate}\ s & \overset{def}{=} & \texttt{do () when}\ s \\
\texttt{await}\ s & \overset{def}{=} & \texttt{pause;await immediate}\ s \\
\texttt{await}\ s(x)\ \texttt{in}\ e & \overset{def}{=} & \texttt{await immediate}\ s\texttt{;let}\ s(x)\ \texttt{in}\ e
\end{array}$$

In ReactiveML, signals are always valued. Thus, a pure signal (in the Esterel sense) is implemented with a valued signal with value `()`. 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 `signal` s `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 $\uplus$ is the union (if $m_1 = \{v_1, ..., v_n\}$ and $m_2 = \{v'_1, ..., v'_k\}$ then $m_1 \uplus m_2 = \{v_1, ..., v_n, v'_1, ..., v'_k\}$).

The `pause` statement stops the execution for one instant. Indeed, as opposed to 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, `present` x will evaluate to false at the end of the reaction so the instruction `()` will be executed during the next reaction. The `await/immediate` constructs awaits for the presence of a signal. Awaiting a valued signal can be written `await` $s(x)$ `in` e. The access construction `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 ReactiveML program. We base it on a behavioral semantics, in the style of the *logical behavioral semantics* of Esterel [5]. 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 (v) such that:

$$v \quad ::= \quad c \mid n \mid (v, v) \mid \lambda x.e \mid \texttt{proc}\ e$$

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

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.

$$v \Downarrow v \qquad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{(e_1, e_2) \Downarrow (v_1, v_2)} \qquad \frac{e[x \leftarrow \texttt{rec}\ x\,\texttt{=}\,e] \Downarrow v}{\texttt{rec}\ x\,\texttt{=}\,e \Downarrow v}$$

$$\frac{e_1 \Downarrow \lambda x.e \quad e_2 \Downarrow v_2 \quad e[x \leftarrow v_2] \Downarrow v}{e_1\ e_2 \Downarrow v} \qquad \frac{e_1 \Downarrow v_1 \quad e_2[x \leftarrow v_1] \Downarrow v}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \Downarrow v}$$

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 $N_1.N_2$ 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:

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

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 $S^d(n_i) = d_i$, $S^g(n_i) = g_i$ and $S^v(n_i) = m_i$.

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

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

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

We take the convention that if $n \notin Dom(E)$ 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:

$$\forall n \in Dom(E_1) \cup Dom(E_2) : \; E(n) = E_1(n) \uplus E_2(n)$$

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

$$\forall n \in Dom(E_1) \cup Dom(E_2) : \; E(n) = E_1(n) \cap E_2(n)$$

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

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

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

$$\begin{array}{lll} E_1 \sqsubseteq E_2 & \text{iff} & \forall n \in Dom(E_1) : E_1(n) \subseteq E_2(n) \\ S_1 \sqsubseteq S_2 & \text{iff} & S_1^v \sqsubseteq S_2^v \end{array}$$

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

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

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 $\sqsubseteq$) such that:

$$N_i \vdash e_i \xrightarrow[S_i]{E_i,\, b} e'_i$$

where:

$$\begin{array}{l} O_i \sqsubseteq E_i, \text{ and } (I_i \sqcup E_i) \sqsubseteq S_i^v \\ S_i^d \subseteq S_{i+1}^d \text{ and } S_i^g \subseteq S_{i+1}^g \\ \forall n \in N_{i+1}.n \notin Dom(S_i) \end{array}$$

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 $S_i^d \subseteq S_{i+1}^d$ and $S_i^g \subseteq S_{i+1}^g$ 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 $\forall n \in N_{i+1}.n \notin Dom(S_i)$ 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 ($b = \mathit{true}$).

- 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 *false* guaranty that there is no instantaneous loop.

- `signal` x `default` e_1 `gather` e_2 `in` e declare a new signal. 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.

- `emit` 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.

- `let` $e(x)$ `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 *fold* is defined as follows:

$$\begin{array}{lcl} \mathit{fold}\ f\ (\{v_1\} \uplus m)\ v_2 & = & \mathit{fold}\ f\ m\ (f\ v_1\ v_2) \\ \mathit{fold}\ f\ \emptyset\ v & = & v \end{array}$$

  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 (`()`) 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/when` 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 absence 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:
  `do await s until s done; emit s`.

- `run` e evaluates the expression e into a process definition and executes it.

Now, we establish the main properties of the behavioral semantics stating that the reaction is deterministic: for a 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 $N \vdash e \xrightarrow[S]{E,\, b} e'$), then there exists a unique smallest signal environment in which it can react. The proofs are given in appendix A.

$$\frac{N \vdash e_1 \xrightarrow[S]{E_1,\mathit{false}} e_1'}{N \vdash e_1 \texttt{;} e_2 \xrightarrow[S]{E_1,\mathit{false}} e_1' \texttt{;} e_2} \qquad \frac{N_1 \vdash e_1 \xrightarrow[S]{E_1,\mathit{true}} e_1' \quad N_2 \vdash e_2 \xrightarrow[S]{E_2,b} e_2'}{N_1 \cdot N_2 \vdash e_1 \texttt{;} e_2 \xrightarrow[S]{E_1 \sqcup E_2,b} e_2'} \qquad \frac{N_1 \vdash e_1 \xrightarrow[S]{E_1,b_1} e_1' \quad N_2 \vdash e_2 \xrightarrow[S]{E_2,b_2} e_2'}{N_1 \cdot N_2 \vdash e_1 \texttt{||} e_2 \xrightarrow[S]{E_1 \sqcup E_2,b_1 \wedge b_2} e_1' \texttt{||} e_2'}$$

$$\frac{N \vdash e\texttt{;loop}\ e \xrightarrow[S]{E,\mathit{false}} e'}{N \vdash \texttt{loop}\ e \xrightarrow[S]{E,\mathit{false}} e'} \qquad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad S^d(n) = v_1 \quad S^g(n) = v_2 \quad N \vdash e[x \leftarrow n] \xrightarrow[S]{E,b} e'}{N.[n] \vdash \texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e \xrightarrow[S]{E,b} e'}$$

$$\frac{e_1 \Downarrow n \quad e_2 \Downarrow v}{\emptyset \vdash \texttt{emit}\ e_1\ e_2 \xrightarrow[S]{[\{v\}/n],\mathit{true}} \texttt{()}} \qquad \frac{e \Downarrow n \quad S(n) = (d,g,m) \quad v = \mathit{fold}\ g\ m\ d}{\emptyset \vdash \texttt{let}\ e(x)\ \texttt{in}\ e_1 \xrightarrow[S]{\emptyset,\mathit{false}} e_1[x \leftarrow v]} \qquad \frac{e_1 \Downarrow v \quad N \vdash e_2[x \leftarrow v] \xrightarrow[S]{E,b} e_2'}{N \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \xrightarrow[S]{E,b} e_2'}$$

$$\frac{}{\emptyset \vdash \texttt{()} \xrightarrow[S]{\emptyset,\mathit{true}} \texttt{()}} \qquad \frac{e \Downarrow n \quad n \in S \quad N \vdash e_1 \xrightarrow[S]{E,b} e_1'}{N \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow[S]{E,b} e_1'} \qquad \frac{e \Downarrow n \quad n \notin S}{\emptyset \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow[S]{\emptyset,\mathit{false}} e_2}$$

$$\frac{e \Downarrow n \quad n \notin S}{\emptyset \vdash \texttt{do}\ e_1\ \texttt{when}\ e \xrightarrow[S]{\emptyset,\mathit{false}} \texttt{do}\ e_1\ \texttt{when}\ n} \qquad \frac{e \Downarrow n \quad n \in S \quad N \vdash e_1 \xrightarrow[S]{E,\mathit{false}} e_1'}{N \vdash \texttt{do}\ e_1\ \texttt{when}\ e \xrightarrow[S]{E,\mathit{false}} \texttt{do}\ e_1'\ \texttt{when}\ n} \qquad \frac{e \Downarrow n \quad n \in S \quad N \vdash e_1 \xrightarrow[S]{E,\mathit{true}} e_1'}{N \vdash \texttt{do}\ e_1\ \texttt{when}\ e \xrightarrow[S]{E,\mathit{true}} \texttt{()}}$$

$$\frac{e \Downarrow n \quad n \in S \quad N \vdash e_1 \xrightarrow[S]{E,b} e_1'}{N \vdash \texttt{do}\ e_1\ \texttt{until}\ e \xrightarrow[S]{E,b} \texttt{()}} \qquad \frac{e \Downarrow n \quad n \notin S \quad N \vdash e_1 \xrightarrow[S]{E,b} e_1'}{N \vdash \texttt{do}\ e_1\ \texttt{until}\ e \xrightarrow[S]{E,b} \texttt{do}\ e_1'\ \texttt{until}\ n} \qquad \frac{e \Downarrow \texttt{proc}\ e_1 \quad N \vdash e_1 \xrightarrow[S]{E,b} e_1'}{N \vdash \texttt{run}\ e \xrightarrow[S]{E,b} e_1'}$$

Figure 3: Behavioral Semantics

The combination of this 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 $\forall n \in \mathit{Dom}(S) : S^g(n) = f$ *and* $f(x, f(y,z)) = f(y, f(x,z))$
and $N \vdash e \xrightarrow[S]{E_1,b_1} e_1'$ *and* $N \vdash e \xrightarrow[S]{E_2,b_2} e_2'$
then $(E_1 = E_2 \wedge b_1 = b_2 \wedge e_1' = e_2')$

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 $\mathcal{S}$ such that*

$$\mathcal{S} = \left\{ S \mid \exists E, N, b : N \vdash e \xrightarrow[S]{E,b} e' \right\}$$

then there exists a (unique) smallest signal environment ($\sqcap\mathcal{S}$) such that

$$\exists E, N, b : N \vdash e \xrightarrow[\sqcap\mathcal{S}]{E,b} e'$$

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 reaction to the absence of a signal. Indeed contrary to Esterel the absence of a signal can not 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: `present s1 then () else emit s2`.

Lemma 3. *Let S_1, S_2, S_3 and e such that $N_1 \vdash e \xrightarrow[S_1]{E_1,b_1} e_1$ and $N_2 \vdash e \xrightarrow[S_2]{E_2,b_2} e_2$ and $S_3^v = S_1^v \sqcap S_2^v$ then there exists E_3, N_3, b_3 and e_3 such that $N_3 \vdash e \xrightarrow[S_3]{E_3,b_3} e_3$ and $b_3 \Rightarrow (b_1 \wedge b_2)$ and $E_3 \sqsubseteq (E_1 \sqcap E_2)$ and $N_3 \subseteq (N_1 \cap N_2)$.*

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 ($\xrightarrow{\epsilon}$) 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 `()`.
- The `loop` duplicates its body.
- The `run` instruction applied to a process definition executes it.
- `emit` n v is reduced into `()` 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 in $\mathcal{N}$. 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 `()`.
- The `do/when` can be reduced into `()` only when its body is a value and when the signal is present.

From this axioms, we define the reduction $\rightarrow$:

$$\frac{e/S \xrightarrow{\epsilon} e'/S'}{\Gamma(e)/S \rightarrow \Gamma(e')/S'} \qquad \frac{e_{nv} \Downarrow v}{\Gamma(e_{nv})/S \rightarrow \Gamma(v)/S}$$

$$\frac{n \in S \quad e/S \rightarrow e'/S'}{\Gamma(\texttt{do } e \texttt{ when } n)/S \rightarrow \Gamma(\texttt{do } e' \texttt{ when } n)/S'}$$

where Γ 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:

$$\begin{array}{lll}\Gamma & ::= & [\,] \mid \texttt{let } x \texttt{ = } \Gamma \texttt{ in } e \mid \Gamma\texttt{;}e \\ & & \mid \Gamma \texttt{||} e \mid e \texttt{||} \Gamma \mid \texttt{run } \Gamma \mid \texttt{emit } \Gamma\ e \mid \texttt{emit } e\ \Gamma \\ & & \mid \texttt{let } \Gamma(x) \texttt{ in } e \mid \texttt{present } \Gamma \texttt{ then } e \texttt{ else } e \\ & & \mid \texttt{signal } x \texttt{ default } \Gamma \texttt{ gather } e \texttt{ in } e \\ & & \mid \texttt{signal } x \texttt{ default } e \texttt{ gather } \Gamma \texttt{ in } e \\ & & \mid \texttt{do } e \texttt{ until } \Gamma \mid \texttt{do } \Gamma \texttt{ until } n \mid \texttt{do } e \texttt{ when } \Gamma\end{array}$$

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: $S \vdash e \rightarrow_{eoi} e'$ 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 *fold g m d* 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 `()`.
- 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 ($S_i^v = I_i$). The outputs are a subset of the signal environment at the end of the reaction ($O_i \sqsubseteq S_i'$). The default values and the gathering functions are kept for an instant to the other ($S_i'^d \subseteq S_{i+1}^d$ and $S_i'^g \subseteq S_{i+1}^g$).

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.

$$\frac{e_i/S_i \hookrightarrow e_i''/S_i' \quad S_i' \vdash e_i'' \rightarrow_{eoi} e_i'}{e_i/S_i \Rightarrow e_i'/S_i'}$$

Where $e/S \hookrightarrow e'/S'$ if $e/S \rightarrow^* e'/S'$ and $e'/S' \not\rightarrow$. The relation $\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow$.

$$\texttt{let } x \texttt{ = } v \texttt{ in } e/S \xrightarrow{\epsilon} e[x \leftarrow v]/S \qquad v\texttt{;}e/S \xrightarrow{\epsilon} e/S \qquad v_1 \texttt{||} v_2/S \xrightarrow{\epsilon} \texttt{()}/S \qquad \texttt{loop } e/S \xrightarrow{\epsilon} e\texttt{;loop } e/S$$

$$\texttt{run (proc } e\texttt{)}/S \xrightarrow{\epsilon} e/S \qquad \texttt{emit } n\ v/S \xrightarrow{\epsilon} \texttt{()}/S + [v/n] \qquad \texttt{present } n \texttt{ then } e_1 \texttt{ else } e_2/S \xrightarrow{\epsilon} e_1/S \text{ if } n \in S$$

$$\texttt{signal } x \texttt{ default } v_1 \texttt{ gather } v_2 \texttt{ in } e/S \xrightarrow{\epsilon} e[x \leftarrow n]/S[(v_1, v_2, \emptyset)/n] \text{ if } n \notin \mathit{Dom}(S)$$

$$\texttt{do } v \texttt{ until } n/S \xrightarrow{\epsilon} \texttt{()}/S \qquad \texttt{do } v \texttt{ when } n/S \xrightarrow{\epsilon} \texttt{()}/S \text{ if } n \in S$$

Figure 4: Head reduction

$$S \vdash v \rightarrow_{eoi} v \qquad \frac{S \vdash e_1 \rightarrow_{eoi} e_1' \quad S \vdash e_2 \rightarrow_{eoi} e_2'}{S \vdash e_1 \texttt{||} e_2 \rightarrow_{eoi} e_1' \texttt{||} e_2'} \qquad \frac{S \vdash e_1 \rightarrow_{eoi} e_1'}{S \vdash e_1 \texttt{;} e_2 \rightarrow_{eoi} e_1' \texttt{;} e_2}$$

$$\frac{n \notin S}{S \vdash \texttt{present } n \texttt{ then } e_1 \texttt{ else } e_2 \rightarrow_{eoi} e_2} \qquad \frac{S(n) = (d, g, m) \quad v = \mathit{fold}\ g\ m\ d}{S \vdash \texttt{let } n(x) \texttt{ in } e \rightarrow_{eoi} e[x \leftarrow v]} \qquad \frac{n \notin S \quad S \vdash e \rightarrow_{eoi} e'}{S \vdash \texttt{do } e \texttt{ until } n \rightarrow_{eoi} \texttt{do } e' \texttt{ until } n}$$

$$\frac{n \in S}{S \vdash \texttt{do } e \texttt{ until } n \rightarrow_{eoi} \texttt{()}} \qquad \frac{n \in S \quad S \vdash e \rightarrow_{eoi} e'}{S \vdash \texttt{do } e \texttt{ when } n \rightarrow_{eoi} \texttt{do } e' \texttt{ when } n} \qquad \frac{n \notin S}{S \vdash \texttt{do } e \texttt{ when } n \rightarrow_{eoi} \texttt{do } e \texttt{ when } n}$$

Figure 5: End of instant

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_{init} and e such that $e/S_{init} \Rightarrow e'/S$ then there exists N, b such that $N \vdash e \xrightarrow[S]{E, b} e'$ with $E = S^v \backslash S^v_{init}$.*

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

- If there is no $\rightarrow$ reduction possible, we have to prove that the reduction $\rightarrow_{eoi}$ 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 $S \vdash e \rightarrow_{eoi} e'$ then there exists N and b such that $N \vdash e \xrightarrow[S]{\emptyset, b} e'$.*

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* ($N \vdash e \xrightarrow[S]{E, \mathit{true}} e'$) then e' behaves as `()` ($\forall N', S'.\ N' \vdash e' \xrightarrow[S']{\emptyset, \mathit{true}} e'$). □

Lemma 6. *If $e/S_0 \rightarrow e_1/S_1$ and $N \vdash e_1 \xrightarrow[S]{E', b} e'$ with $S_1 \sqsubseteq S$ then $N' \vdash e \xrightarrow[S]{E, b} e'$ with $E = E' \sqcup (S^v_1 \backslash S^v_0)$*

Proof. The proof is made into two parts. First we prove the same property for the $\xrightarrow{\epsilon}$ 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_{init} and e such that:*

- *$N_1 \vdash e \xrightarrow[S_1]{E_1, b_1} e_1$ where S_1 is the small signal environment such that $S_{init} \sqsubseteq S_1$*
- *$e/S_{init} \Rightarrow e_2/S_2$*
- *$\forall n \in \mathit{Dom}(S_2) : S^g_2(n) = f$ and $f(x, f(y, z)) = f(y, f(x, z))$,*

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 $N_2 \vdash e \xrightarrow[S_2]{E_2, b_2} e_2$ and we can notice, by construction, S_2 is the smallest signal environment such that $S_{init} \sqsubseteq S_2$.

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 $N \vdash e \xrightarrow[S_1]{E_1, b_1} e_1$ and $N \vdash e \xrightarrow[S_2]{E_2, b_2} e_2$.

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 appendix B.

6. STATIC TYPING

We provide a type system as a conservative extension of the Milner type system of ML [20]. In doing so, we have to deal with signals and in particular values which can be transmitted on signals. The type language is:

$$\begin{array}{lll} \sigma & ::= & \forall \alpha_1, \ldots, \alpha_n . \tau \\ \tau & ::= & T \mid \alpha \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \texttt{process} \mid (\tau, \tau)\, \texttt{event} \\ T & ::= & \texttt{int} \mid \texttt{bool} \mid \ldots \end{array}$$

Types are separated in regular types (τ) and type schemes (σ). A type (τ) may be a basic type (T), a type variable (α), a function type ($\tau_1 \rightarrow \tau_2$), a product type ($\tau_1 \times \tau_2$) or a process type (`process`) or the type of a signal ((τ_1, τ_2) `event`). In the type of a signal, τ_1 is the type of the emitted value and τ_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:

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

The instantiation and generalization is defined like the following:

$$\begin{array}{l} \tau'[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n] \leq \forall \alpha_1, \ldots, \alpha_k . \tau \\ \mathit{Gen}(\tau, H) = \forall \alpha_1, ..., \alpha_n . \tau \\ \qquad\qquad \text{where } \{\alpha_1, ..., \alpha_k\} = FV(\tau) - FV(H) \end{array}$$

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

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

Expressions are typed by asserting the judgment $H \vdash e : \tau$ which states that the expression e has type τ 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 (τ, τ') `event`, x must have type τ'. The instantiation `run` e 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 [23].

7. IMPLEMENTATION AND 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 [18] (without objects, labels and functors) 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 [5], 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 $\mathcal{W}$ of actions waiting for the presence of a signal during one instant. When $\mathcal{W}$ 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 ($\mathcal{C}$). Therefore the execution of an instant consists in the execution of all the ready actions of $\mathcal{C}$. The end of the instant is decided when $\mathcal{C}$ 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 in order to obtain an efficient execution of programs. 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 the implementation of REACTIVEML compete favorably with the LOFT library written in C.

The main application written in REACTIVEML is a simulator of a complex network routing protocol for mobile ad-hoc

[5] Through 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.

$$\frac{\tau \leq H(x)}{H \vdash x : \tau} \qquad \frac{\tau \leq TC(c)}{H \vdash c : \tau} \qquad \frac{H \vdash e_1 : \tau_1 \quad H \vdash e_2 : \tau_2}{H \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{H \vdash e_1 : \tau_1 \quad H[x : Gen(\tau_1, H)] \vdash e_2 : \tau_2}{H \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

$$\frac{H[x : \tau_1] \vdash e : \tau_2}{H \vdash \lambda x.e : \tau_1 \rightarrow \tau_2} \qquad \frac{H \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad H \vdash e_2 : \tau_1}{H \vdash e_1\, e_2 : \tau_2} \qquad \frac{H \vdash e : \texttt{unit}}{H \vdash \texttt{proc}\ e : \texttt{process}} \qquad \frac{H \vdash e : \texttt{process}}{H \vdash \texttt{run}\ e : \texttt{unit}}$$

$$\frac{H \vdash e_1 : \tau_1 \quad H \vdash e_2 : \tau_2}{H \vdash e_1 \texttt{;} e_2 : \tau_2} \qquad \frac{H \vdash e_1 : \tau_1 \quad H \vdash e_2 : \tau_2}{H \vdash e_1 \texttt{||} e_2 : \texttt{unit}} \qquad \frac{H \vdash e : \tau}{H \vdash \texttt{loop}\ e : \texttt{unit}} \qquad \frac{H \vdash e_1 : (\tau_1, \tau_2)\,\texttt{event} \quad H \vdash e_2 : \tau_1}{H \vdash \texttt{emit}\ e_1\ e_2 : \texttt{unit}}$$

$$\frac{H \vdash e_1 : \tau_2 \quad H \vdash e_2 : \tau_1 \rightarrow \tau_2 \rightarrow \tau_2 \quad H[s : (\tau_1, \tau_2)\,\texttt{event}] \vdash e : \tau}{H \vdash \texttt{signal}\ s\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e : \tau} \qquad \frac{H \vdash e_1 : (\tau_1, \tau_2)\,\texttt{event} \quad H[x : \tau_2] \vdash e : \tau}{H \vdash \texttt{let}\ e_1(x)\ \texttt{in}\ e : \tau}$$

$$\frac{H \vdash e : (\tau, \tau')\,\texttt{event} \quad H \vdash e_1 : \tau \quad H \vdash e_2 : \tau}{H \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau} \qquad \frac{H \vdash e_1 : (\tau_1, \tau_2)\,\texttt{event} \quad H \vdash e : \tau}{H \vdash \texttt{do}\ e\ \texttt{until}\ e_1 : \texttt{unit}} \qquad \frac{H \vdash e_1 : (\tau_1, \tau_2)\,\texttt{event} \quad H \vdash e : \tau}{H \vdash \texttt{do}\ e\ \texttt{when}\ e_1 : \texttt{unit}}$$

Figure 6: The Type System

% of active cells	0 %	4 %	42 %	60 %	83 %
OCAML	0.74 *s*	0.75 *s*	0.76 *s*	0.77 *s*	0.77 *s*
LOFT	0.02 *s*	0.11 *s*	0.93 *s*	1.57 *s*	2.09 *s*
REACTIVEML	0.05 *s*	0.08 *s*	0.89 *s*	1.46 *s*	1.94 *s*

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

networks [3, 19], 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 simulation tools (e.g., NS, OPNET) inadequate and network protocols appear to be very hard to program efficiently in conventional programming languages. This experiment showed that the reactive model as introduced by Boussinot strongly matters for programming those systems. It provides adequate programming constructs — namely synchronous parallel composition, broadcast communication and dynamic creation — which allow for a natural implementation of the hard part of the simulation. The complete REACTIVEML implementation (with graphical interface, statistics) is about 1000 lines. Experimental results show that the REACTIVEML implementation 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 twice faster that the ad-hoc version directly programmed in NAB [22]. 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 [16].

8. CONCLUSION AND RELATED WORKS

In this paper, we have presented an extension of an existing strict ML language with reactive constructs. The result language is dedicated to the implementation of complex dynamic 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 [1], 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, 26] are an extension of the reactive approach that allows to mix cooperative and preemptive scheduling. In this model several synchronous schedulers can be executed in an asynchronous way. The threads can move from a scheduler to an other dynamically or can be executed asynchronously out of all schedulers. The threads that can be executed alone must be implemented over the system threads, it limits the number of such threads and it leads to efficiency problems. Contrary to REACTIVEML, in the FAIR THREADS, there is only top-level concurrency: we cannot write `(`e_1`||`e_2`);`e_3, and there is no hierarchical control structures.

ULM [6] is a language dedicated to mobility. It also borrows the principles of synchronous reactive programming introduced by Boussinot and embed it inside a call-by-value λ-calculus. In ULM, references are encoded like signals: accessing a reference which is not local is delayed until it becomes present. We did not address mobility issues and thus, accessing a reference is instantaneous. In REACTIVEML, synchronization can only be done through the use of a signal and reactive construct must appear in particular places of the program. In comparison, ULM allows to insert reactive constructs (e.g., `pause`) anywhere in an expression. As 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 [25] is a language that support concurrent programming and functional programming. As opposed to REACTIVEML, it is based on a preemptive scheduling. 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 we have the guaranty by the compilation of the concurrency that instantaneous actions are atomic. In REACTIVEML, there is a sequential run-time (there is no concurrency during the execution).

The language is very young and several extensions can be considered. In particular, an important work must be done concerning the efficiency of the execution 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 Boussinot model, the scope extrusion phenomena (which is absent in existing synchronous languages) make this compilation difficult and calls for new program analysis.

9. REFERENCES

[1] R. Acosta-Bermejo. *Rejo Langage d'Objets Réactifs et d'Agents.* PhD thesis, Ecole des Mines de Paris, 2003.

[2] Raul Acosta-Bermejo. Reactive operating system, reactive java objects. In *NOTERE'2000*, Paris, November 2000. ENST.

[3] F. Benbadis, M. Dias de Amorim, and S. Fdida. ELIP: Embedded location information protocol. In *IFIP Networking 2005 Conference*, 2005.

[4] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The Synchronous Languages Twelve Years Later. *Proceedings of the IEEE*, 2003.

[5] G. Berry. The constructive semantics of esterel, 1998.

[6] Gérard Boudol. ULM a core programming model for global computing. In *Proceedings of the European Symposium on Programming*, 2004.

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

[8] Frédéric Boussinot. Concurrent programming with Fair Threads: The LOFT language, 2003.

[9] Frédéric Boussinot. FairThreads: mixing cooperative and preemptive threads in C. Research report 5039, INRIA, 2003.

[10] Frédéric Boussinot. Reactive programming of cellular automata. Technical Report 5183, INRIA, 2004.

[11] Frédéric Boussinot and Robert de Simone. The SL synchronous language. *Software Engineering*, 22(4):256–266, 1996.

[12] Frédéric Boussinot, Jean-Ferdinand Susini, Frédéric Dang Tran, and Laurent Hazard. A reactive behavior framework for dynamic virtual worlds. In *Proceedings of the sixth international conference on 3D Web technology*, pages 69–75. ACM Press, 2001.

[13] Frédéric Boussinot and Jean-Ferdinant Susini. The sugarcubes tool box - a reactive java framework. *Software Practice and Experience*, 28(14):1531–1550, 1998.

[14] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.

[15] Laurent Hazard, Jean-Ferdy Susini, and Frédéric Boussinot. The Junior reactive kernel. Research report 3732, INRIA, 1999.

[16] E. Jahier, P. Raymond, and P. Baufreton. Case studies with Lurette V2. In *Proceedings of the First International Symposium on Leveraging Applications of Formal Method*, 2004.

[17] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP 74 Congress.* North Holland, Amsterdam, 1974.

[18] Xavier Leroy. The Objective Caml system release 3.08. Documentation and user's manual. Technical report, INRIA, 2004.

[19] Louis Mandel and Farid Benbadis. Simulation of mobile ad hoc network protocols in ReactiveML. Submitted to publication.

[20] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:348–375, 1978.

[21] Robin Milner. *Communicating and Mobile Systems: The π-Calculus.* Cambridge University Press, 1999.

[22] Network in A Box. `http://nab.epfl.ch/`.

[23] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists.* MIT Press, 1991.

[24] Riccardo Pucella. Reactive programming in Standard ML. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 48–57. IEEE Computer Society Press, 1998.

[25] John H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.

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

APPENDIX

A. PROOFS ON BEHAVIORAL SEMANTICS

A.1 Proof of lemma 1

We first start with an auxiliary property. The semantics of ML is deterministic.

Lemma 8. *For every expression e, if $e \Downarrow v_1$ and $e \Downarrow v_2$ then $v_1 = v_2$.*

Now we prove the determinism of the behavioral semantics by structural induction on the derivations. We just present the most interesting case.

Case `let` $e(x)$ `in` e': There is only one rule that can be applied.

$$\frac{e \Downarrow n_1 \quad S(n_1) = (d_1, g_1, m_1) \quad v_1 = \mathit{fold}\ g_1\ m_1\ d_1}{N \vdash \texttt{let}\ e(x)\ \texttt{in}\ e' \xrightarrow[S]{\emptyset,\, \mathit{false}} e'[x \leftarrow v_1]}$$

$$\frac{e \Downarrow n_2 \quad S(n_2) = (d_2, g_2, m_2) \quad v_2 = \mathit{fold}\ g_2\ m_2\ d_2}{N \vdash \texttt{let}\ e(x)\ \texttt{in}\ e' \xrightarrow[S]{\emptyset,\, \mathit{false}} e'[x \leftarrow v_2]}$$

with lemma 8, $n_1 = n_2$ so $S(n_1) = S(n_2) = (d, g, v)$. With the property of associativity and commutativity of the gathering functions, we are sure that *fold* is deterministic, thus $v_1 = v_2$. Hence $e'[x \leftarrow v_1] = e'[x \leftarrow v_2]$.

A.2 Proof of lemma 3

By structural induction on the derivations.

Case $e_1 \,||\, e_2$: For the parallel composition there is only one rule that can be applied:

$$\frac{N_{1_1} \vdash e_1 \xrightarrow[S_1]{E_{1_1},\, b_{1_1}} e'_{1_1} \quad N_{1_2} \vdash e_2 \xrightarrow[S_1]{E_{1_2},\, b_{1_2}} e'_{2_1}}{N_{1_1} \cdot N_{1_2} \vdash e_1 \,||\, e_2 \xrightarrow[S_1]{E_{1_1} \sqcup E_{1_2},\, b_{1_1} \wedge b_{1_2}} e'_{1_1} \,||\, e'_{2_1}}$$

$$\frac{N_{2_1} \vdash e_1 \xrightarrow[S_2]{E_{2_1},\, b_{2_1}} e'_{1_2} \quad N_{2_2} \vdash e_2 \xrightarrow[S_2]{E_{2_2},\, b_{2_2}} e'_{2_2}}{N_{2_1} \cdot N_{2_2} \vdash e_1 \,||\, e_2 \xrightarrow[S_2]{E_{2_1} \sqcup E_{2_2},\, b_{2_1} \wedge b_{2_2}} e'_{1_2} \,||\, e'_{2_2}}$$

By the induction hypothesis,

$$N_{3_1} \vdash e_1 \xrightarrow[S_3]{E_{3_1},\, b_{3_1}} e'_{1_3} \qquad N_{3_2} \vdash e_2 \xrightarrow[S_3]{E_{3_2},\, b_{3_2}} e'_{2_3}$$

and $b_{3_1} \Rightarrow b_{1_1} \wedge b_{2_1}$ and $b_{3_2} \Rightarrow b_{1_2} \wedge b_{2_2}$ and $E_{3_1} \sqsubseteq E_{1_1} \sqcap E_{2_1}$ and $E_{3_2} \sqsubseteq E_{1_2} \sqcap E_{2_2}$ and $N_{3_1} \subseteq N_{1_1} \cap N_{2_1}$ and $N_{3_2} \subseteq N_{1_2} \cap N_{2_2}$.

To apply the rule of the parallel in the environment S_3, we first have to prove that $N_{3_1} \cap N_{3_2} \subseteq \emptyset$.
From the hypothesis, we have $N_{3_1} \cap N_{3_2} \subseteq (N_{1_1} \cap N_{2_1}) \cap (N_{1_2} \cap N_{2_2})$. By associativity and commutativity $N_{3_1} \cap N_{3_2} \subseteq (N_{1_1} \cap N_{1_2}) \cap (N_{2_2} \cap N_{2_2})$. With the definition of "$\cdot$", we have $N_{1_1} \cap N_{1_2} = \emptyset$ and $N_{2_2} \cap N_{2_2} = \emptyset$. Hence $N_{3_1} \cap N_{3_2} \subseteq \emptyset$.

So we have

$$\frac{N_{3_1} \vdash e_1 \xrightarrow[S_3]{E_{3_1},\, b_{3_1}} e'_{1_3} \quad N_{3_2} \vdash e_2 \xrightarrow[S_3]{E_{3_2},\, b_{3_2}} e'_{2_3}}{N_{3_1} \cdot N_{3_2} \vdash e_1 \,||\, e_2 \xrightarrow[S_3]{E_{3_1} \sqcup E_{3_2},\, b_{3_1} \wedge b_{3_2}} e'_{1_3} \,||\, e'_{2_3}}$$

The proof of $(b_{3_1} \wedge b_{3_2}) \Rightarrow ((b_{1_1} \wedge b_{1_2}) \wedge (b_{2_1} \wedge b_{2_2}))$ is made from the hypothesis $b_{3_1} \Rightarrow b_{1_1} \wedge b_{2_1}$ and $b_{3_2} \Rightarrow b_{1_2} \wedge b_{2_2}$.

We can prove easily that $N_{3_1} \cup N_{3_2} \subseteq (N_{1_1} \cup N_{1_2}) \cap (N_{2_1} \cup N_{2_2})$

The last point that we have to prove is:

$$E_{3_1} \sqcup E_{3_2} \sqsubseteq (E_{1_1} \sqcup E_{1_2}) \sqcap (E_{2_1} \sqcup E_{2_2})$$

From the hypothesis $E_{3_1} \sqsubseteq (E_{1_1} \sqcap E_{2_1})$ and $E_{3_2} \sqsubseteq (E_{1_2} \sqcap E_{2_2})$ we have

$$E_{3_1} \sqsubseteq E_{1_1} \tag{1}$$
$$E_{3_1} \sqsubseteq E_{2_1} \tag{2}$$
$$E_{3_2} \sqsubseteq E_{1_2} \tag{3}$$
$$E_{3_2} \sqsubseteq E_{2_2} \tag{4}$$

The union of (1) and (3) and the union of (2) and (4) give

$$E_{3_1} \sqcup E_{3_2} \sqsubseteq E_{1_1} \sqcup E_{1_2} \tag{5}$$
$$E_{3_1} \sqcup E_{3_2} \sqsubseteq E_{2_1} \sqcup E_{2_2} \tag{6}$$

the intersection of (5) and (6) is

$$(E_{3_1} \sqcup E_{3_2}) \sqcap (E_{3_1} \sqcup E_{3_2}) \sqsubseteq (E_{1_1} \sqcup E_{1_2}) \sqcap (E_{2_1} \sqcup E_{2_2})$$

which is equal to

$$E_{3_1} \sqcup E_{3_2} \sqsubseteq (E_{1_1} \sqcup E_{1_2}) \sqcap (E_{2_1} \sqcup E_{2_2})$$

Case $e_1 ; e_2$: In the two rules of the sequence, the left branch reacts. So, by hypothesis

$$\frac{N_{1_1} \vdash e_1 \xrightarrow[S_1]{E_{1_1},\, b_{1_1}} e'_{1_1} \quad ?}{N_1 \vdash e_1 ; e_2 \xrightarrow[S_1]{E_1,\, b_1} e'_1} \qquad \frac{N_{2_1} \vdash e_1 \xrightarrow[S_2]{E_{2_1},\, b_{2_1}} e'_{2_1} \quad ?}{N_2 \vdash e_1 ; e_2 \xrightarrow[S_2]{E_2,\, b_2} e'_2}$$

with $N_{1_1} \subseteq N_1$, $E_{1_1} \sqsubseteq E_1$ and $N_{2_1} \subseteq N_2$, $E_{2_1} \sqsubseteq E_2$.
By induction hypothesis,

$$N_{3_1} \vdash e_1 \xrightarrow[S_3]{E_{3_1},\, b_{3_1}} e'_{3_1}$$

and $E_{3_1} \sqsubseteq E_{1_1} \sqcap E_{2_1}$, $N_{3_1} \subseteq N_{1_1} \cap N_{2_1}$ and $b_{3_1} \Rightarrow b_{1_1} \wedge b_{2_1}$.
We examine the different cases following the status of b_{3_1}.

- Case $b_{3_1} = \mathit{false}$: The semantics of the sequence gives $N_{3_1} \vdash e_1 ; e_2 \xrightarrow[S_3]{E_{3_1},\, \mathit{false}} e'_{3_1} ; e_2$.

  We first have to prove $b_{3_1} \Rightarrow (b_1 \wedge b_2)$. Here $b_{3_1} = \mathit{false}$, so by definition of "$\Rightarrow$", we have $\mathit{false} \Rightarrow (b_1 \wedge b_2)$.

  Now we prove that $E_{3_1} \sqsubseteq E_1 \sqcap E_2$.
  By hypothesis $E_{1_1} \sqsubseteq E_1$, so $(E_{1_1} \sqcap E_{2_1}) \sqsubseteq (E_1 \sqcap E_{2_1})$, and $(E_{2_1} \sqsubseteq E_2) \Rightarrow ((E_1 \sqcap E_{2_1}) \sqsubseteq (E_1 \sqcap E_2))$. So by transitivity $(E_{1_1} \sqcap E_{2_1}) \sqsubseteq (E_1 \sqcap E_2)$.

We have notice that by induction $E_{3_1} \sqsubseteq (E_{1_1} \sqcap E_{2_1})$, therefore $E_{3_1} \sqsubseteq (E_1 \sqcap E_2)$ by transitivity.

We finish this case with the proof of $N_{3_1} \subseteq N_1 \cap N_2$. $N_{1_1} \subseteq N_1$ and $N_{2_1} \subseteq N_2$ implies $N_{1_1} \cap N_{2_1} \subseteq N_1 \cap N_2$. We have $N_{3_1} \subseteq N_{1_1} \cap N_{2_1}$ so by transitivity $N_{3_1} \subseteq N_1 \cap N_2$.

- Case $b_{3_1} = \mathit{true}$: By induction, $\mathit{true} \Rightarrow (b_{1_1} \wedge b_{2_1})$, so $b_{1_1} = b_{2_1} = \mathit{true}$ and

$$\frac{N_{1_1} \vdash e_1 \xrightarrow[S_1]{E_{1_1},\, \mathit{true}} e'_{1_1} \quad N_{1_2} \vdash e_2 \xrightarrow[S_1]{E_{1_2},\, b_{1_2}} e'_{2_1}}{N_{1_1} \cdot N_{1_2} \vdash e_1\, \texttt{;}\, e_2 \xrightarrow[S_1]{E_{1_1} \sqcup E_{1_2},\, b_{1_2}} e'_{2_1}}$$

$$\frac{N_{2_1} \vdash e_1 \xrightarrow[S_2]{E_{2_1},\, \mathit{true}} e'_{1_2} \quad N_{2_2} \vdash e_2 \xrightarrow[S_2]{E_{2_2},\, b_{2_2}} e'_{2_2}}{N_{2_1} \cdot N_{2_2} \vdash e_1\, \texttt{;}\, e_2 \xrightarrow[S_2]{E_{2_1} \sqcup E_{2_2},\, b_{2_2}} e'_{2_2}}$$

By induction $N_{3_1} \vdash e_1 \xrightarrow[S_3]{E_{3_1},\, \mathit{true}} e'_{1_3}$, $N_{3_2} \vdash e_2 \xrightarrow[S_3]{E_{3_2},\, b_{3_2}} e'_{2_3}$ and $b_{3_2} \Rightarrow b_{1_2} \wedge b_{2_2}$ and $N_{3_1} \subseteq N_{1_1} \cap N_{2_1}$, $N_{3_2} \subseteq N_{1_2} \cap N_{2_2}$.

So $N_{3_1} \cap N_{3_2} \subseteq (N_{1_1} \cap N_{2_1}) \cap (N_{1_2} \cap N_{2_2})$, by associativity and commutativity $N_{3_1} \cap N_{3_2} \subseteq (N_{1_1} \cap N_{1_2}) \cap (N_{2_2} \cap N_{2_2})$. $N_{1_1} \cap N_{1_2} = \emptyset$ and $N_{2_2} \cap N_{2_2} = \emptyset$ by definition of "$\cdot$". Hence $N_{3_1} \cap N_{3_2} \subseteq \emptyset$.
Therefore we can apply the second sequence rule:

$$\frac{N_{3_1} \vdash e_1 \xrightarrow[S_3]{E_{3_1},\, \mathit{true}} e'_{1_3} \quad N_{3_2} \vdash e_2 \xrightarrow[S_3]{E_{3_2},\, b_{3_2}} e'_{2_3}}{N_{3_1} \cdot N_{3_2} \vdash e_1\, \texttt{;}\, e_2 \xrightarrow[S_3]{E_{3_1} \sqcup E_{3_2},\, b_{3_2}} e'_{2_3}}$$

As in the case $b_{3_1} = \mathit{false}$, $E_{3_1} \sqsubseteq (E_{1_1} \sqcap E_{2_1})$ and $E_{3_2} \sqsubseteq (E_{1_2} \sqcap E_{2_2})$ so like in the parallel composition we can prove that

$$E_{3_1} \sqcup E_{3_2} \sqsubseteq (E_{1_1} \sqcup E_{1_2}) \sqcap (E_{2_1} \sqcup E_{2_2})$$

Case `present` e `then` e_1 `else` e_2: With the determinism of ML (lemma 8), we have that $e \Downarrow n$.

- Case $n \notin S_3$: In this case the expression can react

$$\frac{e \Downarrow n \quad n \notin S_3}{\emptyset \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow[S_3]{\emptyset,\, \mathit{false}} e_2}$$

and this is trivial to check the properties $\mathit{false} \Rightarrow b_1 \wedge b_2$, $\emptyset \sqsubseteq E_1 \sqcap E_2$ and $\emptyset \subseteq N_1 \cap N_2$.

- Case $n \in S_3$: By hypothesis $n \in S_3 \Rightarrow (n \in S_1 \wedge n \in S_2)$. So

$$\frac{e \Downarrow n \quad n \in S_1 \quad N_1 \vdash e_1 \xrightarrow[S_1]{E_1,\, b_1} e'_{1_1}}{N_1 \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow[S_1]{E_1,\, b_1} e'_{1_1}}$$

$$\frac{e \Downarrow n \quad n \in S_2 \quad N_2 \vdash e_1 \xrightarrow[S_2]{E_2,\, b_2} e'_{1_2}}{N_2 \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow[S_2]{E_2,\, b_2} e'_{1_2}}$$

By induction and with the rule of present

$$\frac{e \Downarrow n \quad n \in S_3 \quad N_3 \vdash e_1 \xrightarrow[S_3]{E_3,\, b_3} e'_{1_3}}{N_3 \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow[S_3]{E_3,\, b_3} e'_{1_3}}$$

and $b_3 \Rightarrow b_1 \wedge b_2$ and $E_3 \sqsubseteq E_1 \sqcap E_2$ and $N_3 \subseteq (N_1 \cap N_2)$.

Case `loop` e:

$$\frac{N_1 \vdash e \xrightarrow[S_1]{E_1,\, \mathit{false}} e'_1}{N_1 \vdash \texttt{loop}\ e \xrightarrow[S_1]{E_1,\, \mathit{false}} e'_1\texttt{;loop}\ e}$$

$$\frac{N_2 \vdash e \xrightarrow[S_2]{E_2,\, \mathit{false}} e'_2}{N_2 \vdash \texttt{loop}\ e \xrightarrow[S_2]{E_2,\, \mathit{false}} e'_2\texttt{;loop}\ e}$$

then by induction $N_3 \vdash e \xrightarrow[S_3]{E_3,\, b_3} e'_3$ and $b_3 = \mathit{false}$ because $b_3 \Rightarrow \mathit{false}$, and $E_3 \sqsubseteq E_1 \sqcap E_2$ and $N_3 \subseteq N_1 \cap N_2$ so

$$N_3 \vdash \texttt{loop}\ e \xrightarrow[S_3]{E_3,\, \mathit{false}} e'_3\texttt{;loop}\ e$$

Case `run` e: With the property 8, we known that the evaluation of ML expressions is deterministic. So

$$\frac{e \Downarrow \texttt{proc}\ e' \quad N_1 \vdash e' \xrightarrow[S_1]{E_1,\, b_1} e'_1}{N_1 \vdash \texttt{run}\ e \xrightarrow[S_1]{E_1,\, b_1} e'_1}$$

$$\frac{e \Downarrow \texttt{proc}\ e' \quad N_2 \vdash e' \xrightarrow[S_2]{E_2,\, b_2} e'_2}{N_2 \vdash \texttt{run}\ e \xrightarrow[S_2]{E_2,\, b_2} e'_2}$$

By induction, $N_3 \vdash e' \xrightarrow[S_3]{E_3,\, b_3} e'_3$ and $b_3 \Rightarrow b_1 \wedge b_2$ and $E_3 \sqsubseteq E_1 \sqcap E_2$ and $N_3 \subseteq N_1 \cap N_2$.
So

$$\frac{e \Downarrow \texttt{proc}\ e' \quad N_3 \vdash e' \xrightarrow[S_3]{E_3,\, b_3} e'_3}{N_3 \vdash \texttt{run}\ e \xrightarrow[S_3]{E_3,\, b_3} e'_3}$$

Case `let` x `=` e_1 `in` e_2: We suppose that $e_1 \Downarrow v$. So

$$\frac{e_1 \Downarrow v \quad N_1 \vdash e_2[x \leftarrow v] \xrightarrow[S_1]{E_1,\, b_1} e'_{2_1}}{N_1 \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \xrightarrow[S_1]{E_1,\, b_1} e'_{2_1}}$$

$$\frac{e_1 \Downarrow v \quad N_2 \vdash e_2[x \leftarrow v] \xrightarrow[S_2]{E_2,\, b_2} e'_{2_2}}{N_2 \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \xrightarrow[S_2]{E_2,\, b_2} e'_{2_2}}$$

By induction and with the rule of let,

$$\frac{N_3 \vdash e_2[x \leftarrow v] \xrightarrow[S_3]{E_3,\, b_3} e'_{2_3}}{N_3 \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \xrightarrow[S_3]{E_3,\, b_3} e'_{2_3}}$$

and $b_3 \Rightarrow b_1 \wedge b_2$ and $E_3 \sqsubseteq E_1 \sqcap E_2$ and $N_3 \subseteq (N_1 \cap N_2)$.

Case $\texttt{emit}\ e_1 e_2$: With lemme 8,

$$\frac{e_1 \Downarrow n \quad e_2 \Downarrow v}{\emptyset \vdash \texttt{emit}\ e_1\ e_2 \xrightarrow[S_1]{[\{v\}/n],\, \mathit{true}} \texttt{()}}$$

$$\frac{e_1 \Downarrow n \quad e_2 \Downarrow v}{\emptyset \vdash \texttt{emit}\ e_1\ e_2 \xrightarrow[S_2]{[\{v\}/n],\, \mathit{true}} \texttt{()}}$$

so

$$\frac{e_1 \Downarrow n \quad e_2 \Downarrow v}{\emptyset \vdash \texttt{emit}\ e_1\ e_2 \xrightarrow[S_3]{[\{v\}/n],\, \mathit{true}} \texttt{()}}$$

Case $\texttt{let}\ e(x)\ \texttt{in}\ e_1$: This is immediate

$$\frac{e \Downarrow n \quad S(n) = (d, g, v) \quad r = \mathit{fold}\ g\ v\ d}{\emptyset \vdash \texttt{let}\ e(x)\ \texttt{in}\ e_1 \xrightarrow[S_3]{\emptyset,\, \mathit{false}} e_1[x \leftarrow r]}$$

Case $\texttt{signal}\ s\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e$:

$$\frac{\begin{array}{c} e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad {S'}_1^{d}(n_1) = v_1 \quad {S'}_1^{g}(n_1) = v_2 \\ N_1 \vdash e[x \leftarrow n_1] \xrightarrow[S'_1]{E'_1,\, b_1} e''_1 \end{array}}{N_1.[n_1] \vdash \texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e \xrightarrow[S'_1]{E'_1,\, b_1} e''_1}$$

$$\frac{\begin{array}{c} e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad {S'}_2^{d}(n_2) = v_1 \quad {S'}_2^{g}(n_2) = v_2 \\ N_2 \vdash e[x \leftarrow n_2] \xrightarrow[S'_2]{E'_2,\, b_2} e''_2 \end{array}}{N_2.[n_2] \vdash \texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e \xrightarrow[S'_2]{E'_2,\, b_2} e''_2}$$

if $n \notin \mathit{Dom}(S_1) \cup \mathit{Dom}(S_2)$, we can rename n_1 and n_2 into n such that

$$\frac{\begin{array}{c} e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad S_1^{d}(n) = v_1 \quad S_1^{g}(n) = v_2 \\ N_1 \vdash e[x \leftarrow n] \xrightarrow[S_1]{E_1,\, b_1} e'_1 \end{array}}{N_1.[n] \vdash \texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e \xrightarrow[S_1]{E_1,\, b_1} e'_1}$$

$$\frac{\begin{array}{c} e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad S_2^{d}(n) = v_1 \quad S_2^{g}(n) = v_2 \\ N_2 \vdash e[x \leftarrow n] \xrightarrow[S_2]{E_2,\, b_2} e'_2 \end{array}}{N_2.[n] \vdash \texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e \xrightarrow[S_2]{E_2,\, b_2} e'_2}$$

with $S_1 = S'_1[n_1 \leftarrow n]$, $E_1 = E'_1[n_1 \leftarrow n]$, $e'_1 = e''_1[n_1 \leftarrow n]$, and $S_2 = S'_2[n_2 \leftarrow n]$, $E_2 = E'_2[n_2 \leftarrow n]$, $e'_2 = e''_2[n_2 \leftarrow n]$.

So by induction, $N_3 \vdash e[x \leftarrow n] \xrightarrow[S_3]{E_3,\, b_3} e'_3$ and $b_3 \Rightarrow b_1 \wedge b_2$, and $E_3 \sqsubseteq E_1 \sqcap E_2$ and $N_3 \subseteq (N_1 \cap N_2)$.

And $n \notin N_1 \cup N_2$ so $n \notin N_3$. Hence we can apply the rule of the sequence:

$$\frac{\begin{array}{c} e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad S_3^{d}(n) = v_1 \quad S_3^{g}(n) = v_2 \\ N_3 \vdash e[x \leftarrow n] \xrightarrow[S_3]{E_3,\, b_3} e'_3 \end{array}}{N_3.[n] \vdash \texttt{signal}\ x\ \texttt{default}\ e_1\ \texttt{gather}\ e_2\ \texttt{in}\ e \xrightarrow[S_3]{E_3,\, b_3} e'_3}$$

Case $\texttt{do}\ e_1\ \texttt{when}\ e$: We suppose that $e \Downarrow n$.

- Case $n \notin S_3$:

$$\frac{e \Downarrow n \quad n \notin S_3}{N_3 \vdash \texttt{do}\ e_1\ \texttt{when}\ e \xrightarrow[S_3]{\emptyset,\, \mathit{false}} \texttt{do}\ e_1\ \texttt{when}\ n}$$

  $\mathit{false} \Rightarrow b_1 \wedge b_2$ and $\emptyset \sqsubseteq E_1 \sqcap E_2$.
- Case $n \in S_3$: By hypothesis, $n \in S_3 \Rightarrow ((n \in S_1) \wedge (n \in S_2))$. So

$$\frac{e \Downarrow n \quad n \in S_1 \quad N_1 \vdash e_1 \xrightarrow[S_1]{E_1,\, b_1} e'_{1_1}}{N_1 \vdash \texttt{do}\ e_1\ \texttt{when}\ e \xrightarrow[S_1]{E_1,\, b_1} e'_1}$$

$$\frac{e \Downarrow n \quad n \in S_2 \quad N_2 \vdash e_1 \xrightarrow[S_2]{E_2,\, b_2} e'_{1_2}}{N_2 \vdash \texttt{do}\ e_1\ \texttt{when}\ e \xrightarrow[S_2]{E_2,\, b_2} e'_2}$$

  where $e'_1 = \begin{cases} \texttt{()} & \text{if } b_1 = \mathit{true} \\ \texttt{do}\ e'_{1_1}\ \texttt{when}\ n & \text{if } b_1 = \mathit{false} \end{cases}$
  and $e'_2 = \begin{cases} \texttt{()} & \text{if } b_2 = \mathit{true} \\ \texttt{do}\ e'_{1_2}\ \texttt{when}\ n & \text{if } b_2 = \mathit{false} \end{cases}$
  By induction,

$$N_3 \vdash e_1 \xrightarrow[S_3]{E_3,\, b_3} e'_{1_3}$$

  and $b_3 \Rightarrow b_1 \wedge b_2$, and $E_3 \sqsubseteq E_1 \sqcap E_2$ and $N_3 \subseteq (N_1 \cap N_2)$. And with the rules of $\texttt{do/when}$:
  - if $b_3 = \mathit{true}$ then
    $N_3 \vdash \texttt{do}\ e_1\ \texttt{when}\ e \xrightarrow[S_3]{E_3,\, \mathit{true}} \texttt{()}$
  - if $b_3 = \mathit{false}$ then
    $N_3 \vdash \texttt{do}\ e_1\ \texttt{when}\ e \xrightarrow[S_3]{E_3,\, \mathit{false}} \texttt{do}\ e'_{1_3}\ \texttt{when}\ n$

Case $\texttt{do}\ e_1\ \texttt{until}\ e$:

$$\frac{e \Downarrow n \quad N_1 \vdash e_1 \xrightarrow[S_1]{E_1,\, b_1} e'_{1_1}}{N_1 \vdash \texttt{do}\ e_1\ \texttt{until}\ e \xrightarrow[S_1]{E_1,\, b_1} e'_1} \qquad \frac{e \Downarrow n \quad N_2 \vdash e_1 \xrightarrow[S_2]{E_2,\, b_2} e'_{1_2}}{N_2 \vdash \texttt{do}\ e_1\ \texttt{until}\ e \xrightarrow[S_2]{E_2,\, b_2} e'_2}$$

where $e'_1 = \begin{cases} \texttt{()} & \text{if } n \in S_1 \\ \texttt{do}\ e'_{1_1}\ \texttt{until}\ n & \text{if } n \notin S_1 \end{cases}$
and $e'_2 = \begin{cases} \texttt{()} & \text{if } n \in S_2 \\ \texttt{do}\ e'_{1_2}\ \texttt{until}\ n & \text{if } n \notin S_2 \end{cases}$
By induction,

$$N_3 \vdash e_1 \xrightarrow[S_3]{E_3,\, b_3} e'_{1_3}$$

and $b_3 \Rightarrow b_1 \wedge b_2$, and $E_3 \sqsubseteq E_1 \sqcap E_2$ and $N_3 \subseteq (N_1 \cap N_2)$.

And with the rules of `do/until`:

- if $n \in S_3$ then
  $N_3 \vdash \texttt{do } e_1 \texttt{ until } e \xrightarrow[S_3]{E_3, b_3} \texttt{()}$
- if $n \notin S_3$ then
  $N_3 \vdash \texttt{do } e_1 \texttt{ until } e \xrightarrow[S_3]{E_3, b_3} \texttt{do } e'_{1_3} \texttt{ until } e$

B. EQUIVALENCE PROOF

B.1 Proof of lemma 6

To prove the lemma 6, we first have to prove that this is true for $\xrightarrow{\epsilon}$ reductions.

LEMMA 9. *If $e/S_0 \xrightarrow{\epsilon} e_1/S_1$ and $N \vdash e_1 \xrightarrow[S]{E', b} e'$ with $S_1 \sqsubseteq S$ then $N' \vdash e \xrightarrow[S]{E, b} e'$ with $E = E' \sqcup (S_1^v \backslash S_0^v)$*

PROOF. Let examine each case of $\xrightarrow{\epsilon}$ reduction.

Case $\texttt{let } x \texttt{ = } v \texttt{ in } e$**:** By hypothesis we have

$$\texttt{let } x \texttt{ = } v \texttt{ in } e/S_0 \xrightarrow{\epsilon} e[x \leftarrow v]/S_0$$

and $N \vdash e[x \leftarrow v] \xrightarrow[S]{E, b} e'$ with $S_0 \sqsubseteq S$.

By definition of $\Downarrow$, $v \Downarrow v$.

Hence we can have the following derivation:

$$\dfrac{v \Downarrow v \quad N \vdash e[x \leftarrow v] \xrightarrow[S]{E, b} e'}{N \vdash \texttt{let } x \texttt{ = } v \texttt{ in } e \xrightarrow[S]{E, b} e'}$$

Notice that $S_0^v \backslash S_0^v = \emptyset$.

Case $v\texttt{;}e$**:** We must prove the derivation:

$$\dfrac{\emptyset \vdash v \xrightarrow[S]{\emptyset,\, true} v \quad N \vdash e \xrightarrow[S]{E, b} e'}{N \vdash v\texttt{;}e \xrightarrow[S]{E, b} e'}$$

$\emptyset \vdash v \xrightarrow[S]{\emptyset,\, true} v$ is always true and $N \vdash e \xrightarrow[S]{E, b} e'$ is true by hypothesis.

Case $v_1 \,||\, v_2$**:** The derivation of $v_1 \,||\, v_2$ with the big step semantics is:

$$\dfrac{\emptyset \vdash v_1 \xrightarrow[S]{\emptyset,\, true} v_2 \quad \emptyset \vdash v_1 \xrightarrow[S]{\emptyset,\, true} v_2}{\emptyset \vdash v_1 \,||\, v_2 \xrightarrow[S]{\emptyset,\, true} v_1 \,||\, v_2}$$

and $v_1 \,||\, v_2$ is equivalent to `()`. So the big step reaction is equivalent to $v_1 \,||\, v_2/S \xrightarrow{\epsilon} \texttt{()}/S$ followed by $\emptyset \vdash \texttt{()} \xrightarrow[S]{\emptyset,\, true} \texttt{()}$

Case $\texttt{loop } e$**:** By hypothesis $\texttt{loop } e/S_0 \xrightarrow{\epsilon} e\texttt{;loop } e/S_0$ and $N \vdash e\texttt{;loop } e \xrightarrow[S]{E,\, false} e'$

Hence

$$\dfrac{N \vdash e\texttt{;loop } e \xrightarrow[S]{E,\, false} e'}{N \vdash \texttt{loop } e \xrightarrow[S]{E,\, false} e'}$$

Case $\texttt{run (proc } e\texttt{)}$**:** Trivial.

Case $\texttt{emit } nv$**:** By hypothesis $\texttt{emit } n\ v/S \xrightarrow{\epsilon} \texttt{()}/S + [v/n]$ and $\emptyset \vdash \texttt{()} \xrightarrow[S]{\emptyset,\, true} \texttt{()}$.

By definition of the behavioral semantics:

$$\dfrac{n \Downarrow n \quad v \Downarrow v}{\emptyset \vdash \texttt{emit } n\ v \xrightarrow[S]{[\{v\}/n],\, true} \texttt{()}}$$

And $[\{v\}/n] = S^v \backslash (S + [v/n])^v$.

Case $\texttt{present } n \texttt{ then } e_1 \texttt{ else } e_2$**:** The `present` construction can be reduced with a $\xrightarrow{\epsilon}$ reaction only if n is present. So by hypothesis we have

$$\texttt{present } n \texttt{ then } e_1 \texttt{ else } e_2/S_0 \xrightarrow{\epsilon} e_1/S_0$$

and $N \vdash e_1 \xrightarrow[S]{E, b} e'_1$.

Therefore we have

$$\dfrac{n \Downarrow n \quad n \in S \quad N \vdash e_1 \xrightarrow[S]{E, b} e'_1}{N \vdash \texttt{present } n \texttt{ then } e_1 \texttt{ else } e_2 \xrightarrow[S]{E, b} e'_1}$$

Case $\texttt{signal } x \texttt{ default } v_1 \texttt{ gather } v_2 \texttt{ in } e$**:** By hypothesis,

$$\texttt{signal } x \texttt{ default } v_1 \texttt{ gather } v_2 \texttt{ in } e/S_0 \xrightarrow{\epsilon} e[x \leftarrow n]/S_0[(v_1, v_2, \emptyset)/n]$$

with $n \notin Dom(S_0)$ and $N \vdash e[x \leftarrow n] \xrightarrow[S]{E, b} e'$.

Hence

$$\dfrac{v_1 \Downarrow v_1 \quad v_2 \Downarrow v_2 \quad S^d(n) = v_1 \quad S^g(n) = v_2 \quad N \vdash e[x \leftarrow n] \xrightarrow[S]{E, b} e'}{N.[n] \vdash \texttt{signal } x \texttt{ default } e_1 \texttt{ gather } e_2 \texttt{ in } e \xrightarrow[S]{E, b} e'}$$

Case $\texttt{do } v \texttt{ until } n$**:** This case is similar to the one of $v_1 \,||\, v_2$.

Case $\texttt{do } v \texttt{ when } n$**:** In this case n must be present so

$$\dfrac{n \Downarrow n \quad n \in S \quad N \vdash v \xrightarrow[S]{E,\, true} v}{N \vdash \texttt{do } v \texttt{ when } n \xrightarrow[S]{E,\, true} \texttt{()}}$$

□

Now we prove by structural induction on the contexts the lemma 6.

Case $\Gamma = []$**:** By hypothesis we have $e/S_0 \rightarrow e_1/S_1$ and $N \vdash e_1 \xrightarrow[S]{E', b} e'$. There is three cases for the $\rightarrow$ reduction.

- The reduction can be a head reduction

$$\dfrac{e/S_0 \xrightarrow{\epsilon} e_1/S_1}{e/S_0 \rightarrow e_1/S_1}$$

In this case we can apply the lemma 9.

- The expression is an ML expression

$$\frac{e \Downarrow v}{e/S_0 \rightarrow v/S_0}$$

Such that by hypothesis, $\emptyset \vdash v \xrightarrow[S]{\emptyset,\, \mathit{true}} v$. And the behavioral semantics gives

$$\frac{e \Downarrow v}{\emptyset \vdash e \xrightarrow[S]{\emptyset,\, \mathit{true}} v}$$

- The expression can be a `do/when`

$$\frac{n \in S_0 \quad e/S_0 \rightarrow e_1/S_1}{\texttt{do } e \texttt{ when } n/S_0 \rightarrow \texttt{do } e_1 \texttt{ when } n/S_1}$$

In this case the hypothesis gives

$$\frac{n \Downarrow n \quad n \in S \quad N \vdash e_1 \xrightarrow[S]{E,\, b} e'}{N \vdash \texttt{do } e_1 \texttt{ when } n \xrightarrow[S]{E,\, b} e'}$$

By induction we have

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

Hence we can prove in the behavioral semantics

$$\frac{n \Downarrow n \quad n \in S \quad N \vdash e \xrightarrow[S]{E,\, b} e'}{N \vdash \texttt{do } e \texttt{ when } n \xrightarrow[S]{E,\, b} e'}$$

Case `let` x `=` Γ `in` e: By hypothesis

$$\frac{e_1/S_0 \rightarrow e_1'/S_1}{\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2/S_0 \rightarrow \texttt{let } x \texttt{ = } e_1' \texttt{ in } e_2/S_1}$$

and

$$\frac{e_1' \Downarrow v \quad N \vdash e_2[x \leftarrow v] \xrightarrow[S]{E,\, b} e_2'}{N \vdash \texttt{let } x \texttt{ = } e_1' \texttt{ in } e_2 \xrightarrow[S]{E,\, b} e_2'}$$

Such that by induction $e_1 \Downarrow v$. So

$$\frac{e_1 \Downarrow v \quad N \vdash e_2[x \leftarrow v] \xrightarrow[S]{E,\, b} e_2'}{N \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \xrightarrow[S]{E,\, b} e_2'}$$

Case $\Gamma\,;e$: By hypothesis

$$\frac{e_1/S_0 \rightarrow e_1'/S_1}{e_1\,;e_2/S_0 \rightarrow e_1'\,;e_2/S_1}$$

and

$$\frac{N \vdash e_1' \xrightarrow[S]{E_1,\, b} e_1'' \quad D_2}{N \vdash e_1'\,;e_2 \xrightarrow[S]{E,\, b} e'}$$

where D_2 is the derivation of e_2 if $b = \mathit{true}$.

By induction $N \vdash e_1 \xrightarrow[S]{E,\, b} e_1''$. Therefore

$$\frac{N \vdash e_1 \xrightarrow[S]{E_1,\, b} e_1'' \quad D_2}{N \vdash e_1\,;e_2 \xrightarrow[S]{E,\, b} e'}$$

Case $\Gamma\,||\,e$ or $e\,||\,\Gamma$: By hypothesis

$$\frac{e_1/S_0 \rightarrow e_1'/S_1}{e_1\,||\,e_2/S_0 \rightarrow e_1'\,||\,e_2/S_1}$$

and

$$\frac{N_1 \vdash e_1' \xrightarrow[S]{E_1,\, b_1} e_1'' \quad N_2 \vdash e_2 \xrightarrow[S]{E_2,\, b_2} e_2'}{N_1 \cdot N_2 \vdash e_1'\,||\,e_2 \xrightarrow[S]{E_1 \sqcup E_2,\, b_1 \wedge b_2} e_1''\,||\,e_2'}$$

By induction $N_1 \vdash e_1 \xrightarrow[S]{E,\, b} e_1''$. Therefore

$$\frac{N_1 \vdash e_1 \xrightarrow[S]{E_1,\, b_1} e_1'' \quad N_2 \vdash e_2 \xrightarrow[S]{E_2,\, b_2} e_2'}{N_1 \cdot N_2 \vdash e_1\,||\,e_2 \xrightarrow[S]{E_1 \sqcup E_2,\, b_1 \wedge b_2} e_1''\,||\,e_2'}$$

All other cases are similar to the `let/in`.