INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *The SL Synchronous Language*

Frédéric Boussinot , Robert de Simone

**N° 2510**

Mars 1995

PROGRAMME 2

Calcul symbolique,

programmation

et génie logiciel


ISSN 0249-6399

# The SL Synchronous Language

Frédéric Boussinot , Robert de Simone

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet MEIJE

Rapport de recherche n°2510 — Mars 1995 — 36 pages

**Abstract:** We present a new synchronous programming language named SL based on ESTEREL, in which hypothesis about signal presences or absences are not allowed. Thus, one can decide that a signal was absent during one instant only at the end of this instant, and so reaction to this absence is delayed. ESTEREL "causality problems" are avoided at the price of replacing strong preemptions by weak ones. An operational semantics based on rewriting rules is given and an implementation is described which allows either to directly execute programs, or to produce automata.

**Key-words:** Parallelism, concurrency, synchronous programming languages, automata, reactive systems.

*(Résumé : tsvp)*

Unité de recherche INRIA Sophia-Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : (33) 93 65 77 77 – Télécopie : (33) 93 65 77 65

# Le langage synchrone SL

**Résumé :** On présente le langage de programmation SL qui est un nouveau langage synchrone construit à partir d'ESTEREL. En SL, on interdit toute hypothèse sur la présence ou l'absence des signaux. On ne peut donc décider qu'un signal est absent durant un instant, qu'à la fin de cet instant, et ainsi la réaction à l'absence est reportée à l'instant suivant. Les "problèmes de causalité" d'ESTEREL sont ainsi évités, mais seule la préemption faible reste possible. Une sémantique opérationnelle à base de règles de réécritures est décrite, ainsi qu'une implémentation qui permet soit d'exécuter directement les programmes, soit de produire des automates.

**Mots-clé :** Parallélisme, concurrence, programmation synchrone, automate, systèmes réactifs

# Contents

# 1 Introduction

In reactive and synchronous languages, programs continuously react to activations coming from the environment. A program reacts to a given environmental input event by generating an output event, and this reaction defines what is called an instant.

In "*dataflow*" synchronous languages, such as LUSTRE[7] and SIGNAL[9], reaction of a program consists in the evaluation of a set of equations defining output variables from input ones, and from previous instant variable values.

In "*imperative*" languages, a reaction starts from a set of "control points" and finishes by reaching a set of new "control points". "*Reactive statements*" are provided to control execution. For example, in RC[5] the `Stop` statement finishes the current reaction, and at the next instant, execution will restart from that point. The ESTEREL[3] `watching` statement that "kills" its body as soon as a given signal becomes present, is an example of a more complex reactive statement.

Synchronous languages are based on the assumption that the environment does not interfere with the program during reactions[6]. For example, an input event used by a program is not allowed to change before the end of the current reaction. From the logical point of view, it is equivalent to consider that programs are always ready to accept new input events, or in other words, that reactions "take no time". This is the meaning of the "synchronous" word here: their outputs can be seen as synchronous with their inputs. One other major characteristic of these languages is that nondeterministic programs are rejected. Synchronous languages compilers use "symbolic execution" techniques to produce from programs, deterministic automata with the same input/output behaviors.

Very complex reactions can be expressed in the synchronous language ESTEREL, but there is a price to pay for this expressivity:

- There exist incoherent programs, that are programs without any possible behaviour, or with many possible behaviours (nondeterminism). These programs are said to exhibit "*causality problems*". The problem exists also for dataflow languages, when sets of equations cannot be sorted.
- Costly protocols are needed for distributed implementations, which are a natural concern for parallel languages. In particular, messages must be exchanged not only for present signals, but also for absent ones (so "messages" do not match "signals").

In this paper, we focus on imperative synchronous languages and consider a synchronous language called SL that is in fact a restriction of ESTEREL. Expressive power is weaken in order to avoid causality problems and to facilitate code distribution. The guideline is to forbid (unlike ESTEREL) the possibility of deciding that a given signal is absent during one instant, before the end of that instant. As a consequence, only *weak preemption* remains possible, but not the strong preemption induced by the `watching` statement.

The paper also describes an SL implementation allowing separate compiling and automata generation. This implementation is a lightweight one and is based on a translation into the reactive language RC.

## 2 The ESTEREL Language

ESTEREL provides a way to program parallel activities that communicate with broadcast signals. At each instant, a signal is present if it is in the input event, or if it is emitted (by execution of an "`emit`" statement); otherwise, it is absent. Broadcast means that at each instant a signal has one and only one presence status: either it is present, or it is absent during the instant. Thus, all those that are listening to a signal (by execution of a "`present S then ... end`" statement) have the same coherent information about its presence.

Here is an intuitive analogy: several people are placed in a room and can freely speak aloud and listen to other people. When everybody agrees, the current instant is declared terminated and the next instant can start. During one instant, people can have full dialogs (called also "*instantaneous dialogs*"). For example, people A may ask a question; people B hearing that question, may answer; according to the response, A may ask a new question, and so on. Notice that we do not describe a "client/server" communication: B may respond and also ask a new question that A may respond to. If A has spoken during one instant, then nobody can say that A has been mute during that instant (this is broadcast).

For example, consider the first instant of the following parallel statement:

```
   present S then emit T end
||
   emit S
||
   present T then emit U end
```

As the second branch of the parallel operator (written "||") emits `S`, it is present, thus, the "then" part of the first branch is executed, and `T` and `U` are both emitted. Therefore, the three signals `S, T` and `U` are all emitted during the first instant. Notice that signal broadcast implies that there is no other coherent solution.

The basic ESTEREL semantics (called "behavioral semantics"[8]) allows both signal presence and absence hypothesis.

**Presence Hypothesis.** In a presence hypothesis, a signal can be considered as present, provided it is actualy emitted in the current instant. Intuitively, this corresponds to anticipate one's speaking, for example as in: "as B will say that he agree with me, I, A, say ...".

Consider the following statement:

```
signal S in
   present S then emit T end; emit S
end
```

The "`signal S in ... end`" statement declares a local signal `S` whose scope is restricted to "`...`". In ESTEREL semantics, this statement emits both `S` and `T`. Although `S` is emitted in sequence after the test of its presence, the presence hypothesis is lawful as `S` is actualy emitted. However, ESTEREL compilers reject such solutions as they detect a *causal dependency* between the presence test of `S` and its emission[1].

**Absence Hypothesis.** In an absence hypothesis, a signal can be considered as absent, provided it is not actualy emitted in the current instant. One major ESTEREL characteristic is the possibility to *immediately* react to the absence of a signal. Intuitively, this corresponds to sentences like: "as B is mute, I say ..." where one declares that someone is mute during one instant, *before the end of the instant.*

Consider the following program that tests for the absence of a signal `S` (using the "`present S else ... end`" statement):

```
signal S in
   present S else emit T end
end
```

An absence hypothesis on `S` is justified as no emission of it can be done. Thus `T` is emitted. Notice that a presence hypothesis on `S` would be invalid and that an absence hypothesis on `S` is mandatory to give the statement a semantics.

Incoherent programs may result from signal hypothesis that cannot be justified. Intuitively for signal absences, this means that as far as one says that B is mute during the current instant, B must not begin to speak during that instant.

For example, consider:

```
signal S in
   present S else emit S end
end
```

If one makes an absence hypothesis, it would be invalidated since then, the signal has to be emitted and is thus present. On the other hand, if one makes a presence hypothesis it would also be infirmed (as no emission of `S` would take place). The program "has no solution" and is an example of an incoherent program. Notice that there exist a causal dependency between the presence test of `S` and its emission in the "`else`" branch.

Nondeterminism may result from the possibility at a given point, to make several distinct hypothesis leading to distinct behaviors. For example, consider:

[1] ESTEREL compilers are based on a semantics called "computational semantics", that does not exactly coincide with the behavioral semantics. See for example [8] for more details.

```
signal S in
   present S then
      emit S;
      emit T
   else
      emit U
   end
end
```

If a presence hypothesis is made on `S`, then `T` is emitted as the hypothesis is justified by `S` emission. On the other hand, if an absence hypothesis is made on `S`, then `U` is emitted as the hypothesis is justified because there is no emission of `S`. Thus, this statement is nondeterministic: for the same input, it has two possible distinct behaviours. Notice that now, there exist a causal dependency between the presence test of `S` and its emission in the "`then`" branch.

**Avoiding Presence Hypothesis** From a computational point of view, it may seem that signal presence hypothesis should be avoided. A strategy that consider a signal as present only after it is emitted is in fact implemented in the present ESTEREL v3 compiler. Unfortunately, to avoid presence hypothesis does not forbid incoherent nor nondeterministic programs. For example, consider:

```
   present S else emit T end
||
   present T else emit S end
```

Two distinct absence hypothesis leading to distinct behaviors are possible: if `S` is considered as absent, then `T` is emitted; on the other hand, if `T` is considered as absent, then `S` is emitted. Both solutions are valid and the program is thus nondeterministic. Notice that there is a causal dependency between `S` and `T` in the first branch, and a causal dependency between `T` and `S` in the second branch.

**Implementations.** An important point in ESTEREL is that, by only statically inspecting signal causal dependencies, it is possible to reject all incoherent or nondeterministic programs. This technique is used by present ESTEREL compilers. The v3 implementation[1] allows an absence hypothesis on a given signal only after verifying that this signal cannot be emitted from that point (when needed during symbolic execution, it computes "potentials" to determine what are the signal that remain potentially emitted [8]). Thus in this approach, absence hypothesis cannot be invalid. On the other hand, the v4 implementation[2] first translates programs into sets of equations (very similar to LUSTRE programs) and then sorts these sets in a way that variables defining signal emissions are evaluated before being tested. Thus, sorting implies absence of causality cycles.

Several problems arise with the causality cycles approach:

1. Due to weaknesses of control flow analysis, detection of causality cycles is not exact. Thus, there exists programs with one unique behaviour that are also rejected (in this case, one speaks of "*false causality cycles*"). For example, in v3 (and also in v4), the following program:

```
present T then
   present S else
      present T else emit S end
   end
end
```

   is rejected, although it has one unique solution that does not depend on `S` status. On the other hand, in v4, exclusive control flow can produce cycles. For example, v4 detects a cycle in:

```
if cond then
   present S then emit T end
else
   present T then emit S end
end
```

   Also, v4 does not take care of instants for cycle detection. For example, it rejects the following program:

```
await tick;
present S then emit T end
await tick;
present T then emit S end
```

   The predefined signal `tick` is present at each instant (it defines the "basic clock" of the program) and the `await tick` statement stops the reaction till the next instant. This program, although rejected, has one unique solution.

2. It is a very difficult task to interpret causality cycles. Several distant parts of a program can be involved in several cycles and it has been proved a very tedious task to understand from cycle informations, what is wrong with the program.

3. Causality cycles may appear only in very late development phases, after termination of sub-parts coding, during integration. As a consequence, modularity is difficult to achieve (futhermore, ESTEREL provides no way to specify signal dependencies when coding sub-parts of a program).

The question we address is thus the following: is there a possibility to avoid causality cycles, for example by restricting the primitives used ? What would be lost with such restrictions ? The SL language we are about to describe now is an attempt to give an answer to that question.

# 3 The SL Language

The reader will certainly have the feeling that ESTEREL complexity is in a great extend, a consequence of signal hypothesis. In fact, in SL we choose to forbid any signal hypothesis, and this greatly simplify the synchronous model.

**Forbidding Signal Hypothesis.** In SL, presence hypothesis are forbidden: one cannot decide that a given signal is present unless it has been emitted. On the other hand, absence hypothesis are also forbidden: the only moment one can decide that a signal is absent is the end of the current instant, as before that moment, the signal could always be later emitted. Thus, reaction to a signal absence is necessarily postponed to the *next instant*. In other words, unlike ESTEREL, SL forbids *immediate reactions to signal absences*.

Intuitively, in SL, one can conclude that someone is mute during one instant, only at the end of this instant. Consequently, one can react to this absence of speaking, only at the next instant. For example, one can say only things like: "as B has been mute during the previous instant, I can conclude ...". It should be clear that with this approach, every program has one and only one meaning, or in other words, that there is no incoherent nor nondeterministic programs. However, broadcast and instantaneous dialogs still exist. On the programming level, the difference with ESTEREL relies mainly on the absence of strong preemption, and on the fact that now, a reaction may be split into several ones. What programming style does this implies, is presently rather unclear.

We are now going to describe the SL language. First, we introduce the language kernel that defines the basic primitives, and we compare this kernel to ESTEREL. Then we give some examples, and finally we describe the whole language.

## 3.1 Syntax

SL syntax is very similar to ESTEREL syntax[2].

The syntax and informal semantics are defined by:

- "`nothing`" Does nothing and terminates.
- "`stop`" Ends the execution flow for the current instant. Execution will restart from that point at the next instant.
- "`t1 ; t2`" Behaves as `t1`, but when `t1` terminates, it then behaves as `t2`.
- "`t1 || t2`" Splits the control flow and executes `t1` and `t2`. The parallel terminates when both `t1` and `t2` terminate.

[2] In fact, we consider here only the so-called "*pure Esterel*" where signals have no value associated with.

- "`loop t end`" Behaves as `t`, but when `t` terminates, it is immediately re-executed. Important: `t` must not terminate instantaneously, otherwise the loop would never complete its reaction (such pathological loops are called "*instantaneous loops*")!
- "`signal S in t end`" Declares a local signal `S` in `t` and behaves as `t`.
- "`emit S`" Emits the signal `S` and terminates.
- "`when S then t1 else t2 end`" Tests for `S` presence. If `S` is present, the instruction behaves as `t1`. If `S` is absent, the instruction behaves as "`stop;t2`".
- "`wait S`" Terminates as soon as `S` becomes present.
- "`do t kill S`" Behaves as `t`, but after executing it and if `t` is not terminated, it tests for the presence of `S` and, if `S` is present, it becomes `stop`.

**Rationale for the kill primitive.** As a consequence of forbidding signal hypothesis, the strong preemption ESTEREL `watching` primitive cannot be allowed in SL. Indeed, a `watching` executes its body in case a given signal is absent. Thus, body execution is an example of immediate reaction to the absence of the watched signal. Noticed that `watching` may introduce incoherency, as for example in the following ESTEREL statement:

```
signal S in
   do
      await T;
      emit S
   watching S
end
```

When `T` becomes present, hypothesis that `S` is emitted is invalid, as if it is the case, the `watching` kills its body, and thus, `S` cannot be emitted. On the other hand, hypothesis that `S` is absent is also invalid as then, execution of the `watching` would emit `S`.

Thus, only "weak" preemptions primitives may be allowed in SL, in which the body is always executed, even at the moment the killing signal becomes present. Moreover, the killing action must be postponed to the next instant. Indeed, suppose for a moment that we have a `instantkill` statement that kills its non terminating body and terminate immediately, as soon as the killing signal becomes present. Then, consider:

```
do
   emit K
||
   when S do emit T end
instantkill K;
emit S
```

Without the possibility to make a presence hypothesis on `S`, the body has to wait for the end of the instant to know if `T` must be emitted. But as execution of the body must be finished for the `instantkill` to terminate, it cannot terminate before the end of the instant, although `K` is present. That contradicts the `instantkill` definition.

Now, let us see some examples, to show the expressive power, and also some difficulties, of the language.

**Example 1**: Consider the following parallel statement:

```
      when S then emit T end;
      wait U;
      emit V
||
      emit S
||
      wait T; emit U
```

Signal `S` is emitted in the first instant by the second branch. It is tested as present by the first branch. So `T` is also emitted in the first instant. But now, in the third branch, `wait T` terminates, and thus `U` is emitted. As `U` is present, `wait U` terminates in the first branch, and `V` is emitted. Thus, signals `S`, `T`, `U` and `V` are synchronous and are all emitted in the first instant.

This example exhibits an "instantaneous dialog" between the first and the third branch.

**Example 2**: The following fragment emits signal `U` when signals `S` and `T` have been both received:

```
[
   wait S
||
   wait T
];
emit U
```

Notice that if `S` and `T` are synchronous, that is present in the same instant, then `U` is also emitted in that instant.

To let the same behaviour be cyclic, that is to emit `U` each time the two signals `S` and `T` have been both received, we put an external loop:

```
loop
   [
      wait S
   ||
      wait T
```

```
      ];
      emit U;
      stop
   end
```

Notice that a `stop` statement is mandatory, otherwise the loop would not finish to react when both `S` and `T` are present in the same instant (it would be an instantaneous loop).

**Example 3**:

The following fragment emits signal `U` when the first of the two signals `S` and `T` becomes present:

```
signal V in
   do
      wait S; emit V
   ||
      wait T; emit V
   kill V;
   emit U
end
```

As in the previous example, if `S` and `T` are synchronous, that is, present in the same instant, then `U` is also emitted in that instant. But now, if only one signal is present, then emission of `U` will be delayed to the next instant.

Notice that the following program has exactly the same behaviour:

```
signal V,W in
   do
      do
         wait S; emit V
      ||
         wait T; emit V
      kill V;
      emit W
   kill W;
   emit U
end
```

If only one of `S,T` is present, emission of `U` is also delayed to the next instant, in spite of the outer `kill`, as the body of the inner one becomes terminated.

The problem is that it is not so easy to obtain a cyclic behaviour, as emission of `U` may be delayed or not. Thus, one has to force it to be delayed, and the code becomes:

```
loop
   signal V in
      do
         wait S; emit V
      ||
         wait T; emit V
      ||
         loop stop end
      kill V;
      emit U
   end
end
```

The "`loop stop end`" forces the termination of the `kill` to be necessarily delayed to the next instant after `V` is emitted. Notice that now, there is no need of a `stop` statement to prevent the loop to be instantaneous.

**Example 4**: The following statement kills a `P` statement when a signal `K` becomes present, and in addition, it begins to execute a `Q` statement *in the same instant*:

```
   do
      P;
      emit K
   kill K
||
   await K;
   Q
```

Notice that, at the very instant `K` becomes present, both `P` and `Q` are executed. This code can be seen as defining a variant of the weak preemption, allowing to react instantaneously to the killing signal.

**Example 5**: Consider now a statement corresponding to an ESTEREL incoherent program:

```
when S else emit S end; emit U
```

If `S` is present in the first instant, then the `when` terminates instantaneously and `U` is emitted in that instant. On the other hand, if `S` is absent in the first instant, then `S` and `U` will be both emitted in the second instant.

Let us now change the `else` branch into a `then` one:

```
when S then emit S end; emit U
```

Now, if `S` is present in the first instant, then it is emitted again (which has no effect at all) and `U` is also emitted in that instant. On the other hand, if `S` is absent in the first instant, then the `when` statement will stop till the second instant. Then, at the second instant, it will terminates and `U` will be emitted at that instant.

### 3.2 Relation to ESTEREL

One could think to a correspondance between SL and ESTEREL,statements would always begins with "`await tick`"[3], and where the "strong watching" statement would be replaced by a kind of "weak watching" statement defined as previously. Unfortunately, this correspondance would not be satisfactoty to give SL a precise semantics, as there exists SL programs whose correspondants are nondeterministic. Consider for example, the following program:

```
signal S in
   when S then
      emit S;
      emit U
   end
end
```

This statement has one unique solution, where `U` is not emitted. Consider now its correspondant in ESTEREL:

```
signal S in
   present S then
      emit S;
      emit U
   else
      await tick
   end
end
```

This ESTEREL statement is rejected as being nondeterministic: in one solution `U` is emitted and the statement terminates, in the other, it `U` is not emitted and the statement is stopped.

Thus, we have to give SL a proper semantics, that will be done in section 4.

### 3.3 The Full Language

Syntactical extensions must be added to help in writing programs. Mainly, we introduce the notion of a module, and possibility to mix SL code and C code. This section ends with the example of a small reflex game program.

**Modules.** SL programs are structured into *modules*. A module declares input and output signals, and has a body that is a statement. For example, the following module `M` has two input signals `I1` and `I2`, and one output signal `O`. It first waits for `I1`, then for `I2`, and then `O` is emitted. After that, module `M` is terminated.

[3] In the new version of ESTEREL, `await tick` has been replaced by `stop`.

```
module M:
input I1,I2;
output O;

wait I1;
wait I2;
emit O;

end module
```

A main module can be executed; it is introduced by the `main` keyword. For example:

```
main module M:
input I1,I2;
output O;
...
end module
```

External modules defined in other files are declared as extern. For example, the following declaration declares two module `M1` and `M2`:

```
extern module M1, M2;
```

Module bodies can be considered as statements using the `run` keyword. Interface signals must be given as parameters. For example:

```
run M(Inp1,Inp2,Out)
```

runs module `M` body with `Inp1` and `Inp2` as input signals, and `Out` as output signal.

C **Statements and Extensions.** C code can be used freely to define objects, or as simple instantaneously terminating statements, or as `if` conditions. To be used, C code must be enclosed between "`-[`" and "`]-`".

A `repeat` loop executes its body only a limited number of times, defined as a `C` expression.

A "timeout" part added to a `kill` is executed only in case the body is not terminated while the watched signal becomes present.

**The Reflex Game.** The game we use is described in [4, 5]. It consists in measuring the time needed to an user to react to a light flash. There are three successive phases:

1. Waiting for the user to press a `READY` button.
2. After a random time, lightning on a `GO` lamp.

3. Counting time needed for the user to press a STOP button.

An error is detected, and a TILT lamp is light on, if the user abandon, or in case of cheating, when the user presses STOP before GO is light on. A bell rings when the user confuses the two buttons READY and STOP.

PHASE1 waits for READY and terminates when it becomes present. During the waiting, RING_BELL is emitted each time STOP is pressed, and the ABANDON procedure that detects abandon is executed:

```
module PHASE1:
input MS,READY,STOP;
output RING_BELL,ERROR;

   do
     loop wait STOP; emit RING_BELL; stop end
   ||
     run ABANDON(MS,ERROR)
   kill READY

end module
```

Notice that execution of ABANDON is terminated when READY appears.

The ABANDON procedure uses a repeat construct and is:

```
extern module DATA;

module ABANDON:
input MS;
output ERROR;
   repeat -[ LIMIT_TIME ]- times wait MS; stop end;
   emit ERROR
end module
```

The LIMIT_TIME variable is a C variable defined in the external DATA module.

PHASE2 waits a random number of time and emits GO. During the waiting, it detects an error if STOP is pressed:

```
module PHASE2:
input MS,STOP;
output GO,ERROR;

   do
     repeat -[ Random() ]- times wait MS; stop end
   kill STOP timeout emit ERROR end;
   emit GO
```

```
end module
```

PHASE3 counts the time, using the C variable TIME, while STOP is not pressed. It also detects abandon:

```
module PHASE3:
input MS,STOP;
output DISPLAY,ERROR;

   -[ TIME = 0;]-;
   do
     loop wait MS; -[TIME++;]- ; stop end
   ||
     run ABANDON(MS,ERROR)
   kill STOP;
   emit DISPLAY

end module
```

Module PLAY is a main module that can be executed. It schedules the three phases, and emits TILT in case of error:

```
main module PLAY:
input MS,READY,STOP;
output DISPLAY,RING_BELL,GO,GAME_OVER,TILT;

signal ERROR in
  do
    run PHASE1(MS,READY,STOP,RING_BELL,ERROR);
    [
      loop wait READY; emit RING_BELL; stop end
    ||
      run PHASE2(MS,STOP,GO,ERROR);
      run PHASE3(MS,STOP,DISPLAY,ERROR)
    ]
  kill ERROR
  timeout emit TILT end;
  emit GAME_OVER;

end module
```

# 4 SL Semantics

We are going to give a formal semantics to the SL kernel described in section 3.1. The approach we use comes from [11] and is called *Structural Operational Semantics*.

The basic idea is to build the meaning of a program fragment, called *term*, from the meanings of its sub-terms.

The meaning of a term $t$ depends on a signal *environment* which sets the signal values. We are going to describe how the environment is changed by executing $t$, and adopt an "arrow" notation $t, E \rightarrow t', E'$. It means that the reaction of $t$ in the environment $E$ transforms it into $E'$, and that "what remains to be done" is $t'$. Thus, in such a notation, $t$ and $E$ are "inputs" and $t'$ and $E'$ are "outputs". We shall say that $t, E \rightarrow t', E'$ is a *transition*, and that "$t$ rewrites in $t'$".

In a transition $t, E \rightarrow t', E'$, values of $t'$ and $E'$ may depend on the way some sub-terms of $t$ rewrite, or on the values of some signals in $E$ or in $F$. To express dependency, we use *deduction rules* of the form:

$$\frac{\ldots}{t, E \rightarrow t', E'}$$

It is read as: "if what is above the bar is verified, then what is under the bar is true". Thus, "..." is the assumption, and what is under the bar is the conclusion. For example, here is a rule that (partially) defines a binary operator written "#":

$$\frac{t_1, E \rightarrow u, F \quad P(F)}{t_1 \# t_2, E \rightarrow u \# t_2, F}$$

It means that if the sub-term $t_1$ rewrites in $u$ and changes $E$ into $F$, and if $F$ verifies predicate $P$, then one can conclude that the term $t_1 \# t_2$ rewrites in $u \# t_2$, and also changes $E$ into $F$.

When no assumption is needed in a rule, the bar is omitted, and the rule is called an *axiom*. We are now going to define what environments are, and give a set of rewrite rules that defines SL semantics.

## 4.1 Environments and Transitions

*Environment* are sets of signals. A signal put in an environment is considered as present. We are going now to introduce two kinds of transitions: *micro-transitions* and *macro-transitions*.

**Micro Transitions.** Micro-transitions are elementary steps of the semantics computing. We distinguish two kinds of micro-transitions: for *finished* micro-transitions, all that remains to be done is to be done at the next instant; for *unfinished* micro-transitions, what remains to be done is to be continued in the current instant. Two special values `Term` and `Stop`, called *termination status*, are used for finished micro-transitions: `Term` indicates termination of the term, and `Stop` indicates that the term is not terminated. If $T_1$ and $T_2$ are two termination status, $T_1 * T_2$ is defined to be `Term` if both are `Term`, otherwise it is `Stop`.

The rewrite formats we use are, for an unfinished micro-transition:

$$t, E \rightarrow t', E'$$

and for a finished micro-transition whose termination status is $T \in \{\textsc{Term}, \textsc{Stop}\}$:

$$t, E \xrightarrow{T} t', E'$$

**Macro Transitions.** A macro-transition is made from a sequence of micro-transitions, whose last one is finished. One writes the macro-transition:

$$t, E \stackrel{T}{\Longrightarrow} t', E'$$

if there exists a sequence:

$$t, E \rightarrow t_1, E_1 \rightarrow ... \rightarrow t_n, E_n \xrightarrow{T} t', E'$$

## 4.2 Rewrite Rules

We now give for each SL operator, a set of rewrite rules that defines its semantics.

**Axioms.** `nothing` does nothing and terminates.

$$\texttt{nothing}, E \xrightarrow{\textsc{Term}} \texttt{nothing}, E \tag{1}$$

`stop` does nothing, stops, and rewrites in `nothing`.

$$\texttt{stop}, E \xrightarrow{\textsc{Stop}} \texttt{nothing}, E \tag{2}$$

**Signal Emission.** A `emit` statement adds the signal to the environment and simply rewrites in `nothing`.

$$\texttt{emit S}, E \rightarrow \texttt{nothing}, E \cup \{S\} \tag{3}$$

Notice that to emit an already emitted signal has no effect) and that `emit` delays its termination to the one of `nothing`.

**Signal Test.** A `when` statement on a present signal simply rewrites in its "then" branch.

$$\frac{S \in E}{\texttt{when S then } \texttt{t}_1 \texttt{ else } \texttt{t}_2 \texttt{ End}, E \rightarrow \texttt{t}_1, E} \tag{4}$$

A `when` statement on an absent signal stops, and rewrites in its "else" branch.

$$\frac{S \notin E}{\texttt{when S then } \texttt{t}_1 \texttt{ else } \texttt{t}_2 \texttt{ End}, E \xrightarrow{\textsc{Stop}} \texttt{t}_2, E} \tag{5}$$

**Sequence.** When the left branch of a sequence performs an unfinished transition, so does the sequence.

$$\frac{\mathtt{t}_1, E \rightarrow \mathtt{t}'_1, F}{\mathtt{t}_1; \mathtt{t}_2, E \rightarrow \mathtt{t}'_1; \mathtt{t}_2, F} \tag{6}$$

When the left branch of a sequence stops, so does the sequence.

$$\frac{\mathtt{t}_1, E \xrightarrow{\textsc{Stop}} \mathtt{t}'_1, F}{\mathtt{t}_1; \mathtt{t}_2, E \xrightarrow{\textsc{Stop}} \mathtt{t}'_1; \mathtt{t}_2, E} \tag{7}$$

When the left branch of a sequence terminates, the sequence simply rewrites in the right branch.

$$\frac{\mathtt{t}_1, E \xrightarrow{\textsc{Term}} \mathtt{t}'_1, F}{\mathtt{t}_1; \mathtt{t}_2, E \rightarrow \mathtt{t}_2, E} \tag{8}$$

**Loop.** A loop simply rewrites in a sequence whose left part is the loop body, and whose right part is the loop itself.

$$\texttt{loop t end}, E \rightarrow \mathtt{t}; \texttt{loop t end}, E \tag{9}$$

**Parallelism.** The parallel can behave as its right or left branch provided it performs an unfinished transition.

$$\frac{\mathtt{t}_1, E \rightarrow \mathtt{t}'_1, F}{\mathtt{t}_1 \parallel \mathtt{t}_2, E \rightarrow \mathtt{t}'_1 \parallel \mathtt{t}_2, F} \tag{10}$$

$$\frac{\mathtt{t}_2, E \rightarrow \mathtt{t}'_2, F}{\mathtt{t}_1 \parallel \mathtt{t}_2, E \rightarrow \mathtt{t}_1 \parallel \mathtt{t}'_2, F} \tag{11}$$

When both branches perform finished transitions, so does the parallel. The termination status is Term only if both branches status are also Term. The environment is left unchanged, and the parallel rewrites into the parallel of the two rewritings.

$$\frac{\mathtt{t}_1, E \xrightarrow{T_1} \mathtt{t}'_1, F \qquad \mathtt{t}_2, E \xrightarrow{T_2} \mathtt{t}'_2, G}{\mathtt{t}_1 \parallel \mathtt{t}_2, E \xrightarrow{T_1 * T_2} \mathtt{t}'_1 \parallel \mathtt{t}'_2, E} \tag{12}$$

Notice that distributed termination of parallel components comes from this rule.

**Waiting** A `wait` statement rewrites in `nothing` when the awaited signal is present.

$$\frac{S \in E}{\texttt{wait S}, E \rightarrow \texttt{nothing}, E} \tag{13}$$

A `wait` statement stops and rewrites in itself when the awaited signal is absent.

$$\frac{S \notin E}{\texttt{wait S}, E \xrightarrow{\textbf{Stop}} \texttt{wait S}, E} \tag{14}$$

**Kill** If its body performs an unfinished transition, so does the `kill`.

$$\frac{t, E \rightarrow t', F}{\texttt{do t kill S}, E \rightarrow \texttt{do t}' \texttt{ kill S}, F} \tag{15}$$

If its body terminates, the `kill` simply rewrites in `nothing`.

$$\frac{t, E \xrightarrow{\textbf{Term}} t', F}{\texttt{do t kill S}, E \rightarrow \texttt{nothing}, E} \tag{16}$$

If its body stops, and if the watched signal is absent, the `kill` stops and rewrites in itself.

$$\frac{t, E \xrightarrow{\textbf{Stop}} t', F \quad S \notin F}{\texttt{do t kill S}, E \xrightarrow{\textbf{Stop}} \texttt{do t}' \texttt{ kill S}, E} \tag{17}$$

If its body stops, and if the watched signal is present, the `kill` stops and rewrites in `nothing`.

$$\frac{t, E \xrightarrow{\textbf{Stop}} t', F \quad S \in F}{\texttt{do t kill S}, E \xrightarrow{\textbf{Stop}} \texttt{nothing}, E} \tag{18}$$

**Signal Declaration** The syntax is slihtly extended (in the spirit of [3]) by allowing statements the form `signal_E in ... end` where `E` is either the empty set, either a singleton {`S`} whose element is a signal. This extra information is needed to store the status the signal have outside the scope of the declaration.

First, a `signal` statement stores the value of the defined signal to be able to restore it if the body terminates in the current instant.

$$\texttt{signal S in t end}, E \rightarrow \texttt{signal\_(E} \cap \{\texttt{S}\}) \texttt{ S in t end}, E - S \tag{19}$$

When its body performs an unfinished transition, so does the `signal` statement.

$$\frac{t, E \rightarrow t', F}{\texttt{signal\_X S in t end}, E \rightarrow \texttt{signal\_X S in t}' \texttt{ end}, F} \tag{20}$$

When its body terminates, the `signal` statement rewrites in `nothing` and restores the signal if it was present.

$$\frac{t, E \xrightarrow{\text{Term}} t', F}{\texttt{signal\_X S in t end}, E \rightarrow \texttt{nothing}, E \cup X} \tag{21}$$

When its body stops, the `signal` statement also stops and rewrites in itself.

$$\frac{t, E \xrightarrow{\text{Stop}} t', F}{\texttt{signal\_X S in t end}, E \xrightarrow{\text{Stop}} \texttt{signal S in t' end}, E} \tag{22}$$

Notice that the only choice that exists in using these rules, concern the parallel operator that can interleave unfinished execution of its branches (rules 11 and 10).

### 4.3 Examples

Here are some examples to show how the semantics works.

**Example 6**: Consider the following fragment $t_1$:

```
signal S in
   when S else emit S end
end
```

We have:

$$t_1, \phi \rightarrow t_2, \phi$$

where $t_2$ is:

```
signal_φ S in
  when S else emit S end
end
```

Now as:

$$\texttt{when S else emit S end}, \phi \xrightarrow{\text{Stop}} \texttt{emit S}, \phi$$

we have:

$$t_2, \phi \xrightarrow{\text{Stop}} t_3, \phi$$

where $t_3$ is:

```
signal S in
   emit S
end
```

Thus, we have shown that:

$$t_1, \phi \stackrel{\text{Stop}}{\Longrightarrow} t_3, \phi$$

Notice that this is the only provable transition.

**Example 7**: In the same way, one can show that:

$$u_1, \phi \stackrel{\text{Stop}}{\Longrightarrow} u_2, \phi$$

where $u_1$ is:

```
signal S in
   when S then emit S end
end
```

and $u_2$ is:

```
signal S in
   nothing
end
```

**Example 8**: Consider $v$:

```
   when S then emit T end
||
   emit S
||
   wait T; emit U
```

The only possible transition is: $v, \phi \rightarrow v_1, \{S\}$ where $v_1$ is:

```
   when S then emit T end
||
   nothing
||
   wait T; emit U
```

Then one has: $v_1, \{S\} \rightarrow v_2, \{S\}$ where $v_2$ is:

```
   emit T
||
   nothing
||
   wait T; emit U
```

Then one has: $v_2, \{S\} \rightarrow v_3, \{S, T\}$ where $v_3$ is:

```
   nothing
||
   nothing
||
   wait T; emit U
```

And then: $v_3, \{S,T\} \rightarrow v_4, \{S,T\}$ where $v_4$ is:

```
   nothing
||
   nothing
||
   emit U
```

And finally: $v_4, \{S,T\} \rightarrow v_5, \{S,T,U\}$ where $v_5$ is:

```
   nothing
||
   nothing
||
   nothing
```

Thus we have shown that: $v, \phi \stackrel{\mathbf{Term}}{\Longrightarrow} v_5, \{S,T,U\}$

### 4.4 Coherency and Determinism

We are now going to show that there is no incoherent or nondeterministic SL program.

**Lemma 1** *Suppose* $t, E \stackrel{\mathbf{Term}}{\longrightarrow} t', E'$ *and* $E \subset F$. *Then,* $t, F \stackrel{\mathbf{Term}}{\longrightarrow} t', F$.

*Suppose* $t, E \rightarrow t', E'$ *and* $E \subset F$. *Then,* $t, F \rightarrow t', E' \cup F$.

Proof by structural induction on $t$. □

**Proposition 1** *There is no incoherent program:*

$\forall t, E, \quad \exists T, t'E', \quad t, E \stackrel{T}{\Longrightarrow} t', E'$

Proof by structural induction on $t$.

Two cases are of interest. The first one concerns the sequence operator. Suppose $t$ is $u; v$. By induction, we have $u, E \stackrel{T_1}{\Longrightarrow} u_1, E_1$. If $T_1 = stop$, then we have $u; v, E \stackrel{T_1}{\Longrightarrow} u_1; v, E_1$. Suppose now that $T_1 = term$. Then, by induction one has $v, E_1 \stackrel{T_2}{\Longrightarrow} v_1, E_2$. But then $u; v, E \stackrel{T_2}{\Longrightarrow} v_1, E_2$.

Consider now, the case where $t$ is $u \parallel v$. Then, by lemma 1, we can construct a maximal sequence $t, E_0 \rightarrow ... \rightarrow t_n, E_n$. Now in one step, one gets $t_n, E_n \stackrel{T}{\longrightarrow} t', F$, and the result follows. □

**Lemma 2** *If* $t, E \rightarrow t_1, E_1$, *then there exist no* $t_2$ *such that* $t, E \stackrel{T}{\longrightarrow} t_2, E_2$. *Conversely, if* $t, E \stackrel{T}{\longrightarrow} t_2, E_2$, *then there exist no* $t_1$ *such that* $t, E \rightarrow t_1, E_1$.

Moreover, there is only one way to finish one instant:

$t, E \stackrel{T_1}{\longrightarrow} t_1, E_1$ and $t, E \stackrel{T_2}{\longrightarrow} t_2, E_2$ implies $t_1 = t_2$, $T_1 = T_2$, and $E_1 = E_2$.

Proof by structural induction on $t$. □

**Lemma 3** *The rules are strongly confluent:*

*Suppose* $t, E \rightarrow t_1, E_1$, *and* $t, E \rightarrow t_2, E_2$. *Then,* $\exists t', E'$ *such that* $t_1, E_1 \rightarrow t', E'$ *and* $t_2, E_2 \rightarrow t', E'$.

Proof by structural induction on $t$.

Two cases are of interest. The first one concerns the sequence operator. Suppose $t$ is $u; v$ and $t, E \rightarrow t_1, F_1$ and $t, E \rightarrow t_2, F_2$. Then, by lemma 2, there are only two cases: first, $t_1 = t_2 = v$ and $E_1 = E_2 = E$, and the result follows; second, $t_1 = u_1; v$ and $t_2 = u_2; v$. Then, by hypothesis, one has: $u_1, F_1 \rightarrow w, H$ and $u_2, F_2 \rightarrow w, H$. Then, $u_1; v, F_1 \rightarrow w; v, H$ and $u_2; v, F_2 \rightarrow w; v, H$.

The second case concerns parallelism. Suppose $t_1 \parallel u_1, E \rightarrow t_2 \parallel u_1, F$ and $t_1 \parallel u_1, E \rightarrow t_1 \parallel u_2, G$, with $t_1, E \rightarrow t_2, F$ and $u_1, E \rightarrow u_2, G$. By lemma 1, one has $u_1, F \rightarrow u_2, F \cup G$ and $t_1, G \rightarrow t_2, G \cup F$. So, $t_2 \parallel u_1, F \rightarrow t_2 \parallel u_2, F \cup G$ and $t_1 \parallel u_2, G \rightarrow t_2 \parallel u_2, F \cup G$. □

**Proposition 2** SL *programs are deterministic:*

$t, E \stackrel{T_1}{\Longrightarrow} u_1, F_1$ *and* $t, E \stackrel{T_2}{\Longrightarrow} u_2, F_2$ *implies* $u_1 = u_2$, $T_1 = T_2$ *and* $F_1 = F_2$.

Suppose we have the situation: $t, E \rightarrow ... \rightarrow t_1, E_1$ and $t_1, E_1 \stackrel{T_1}{\longrightarrow} u_1, F_1$, and $t, E \rightarrow ... \rightarrow t_2, E_2$ and $t_2, E_2 \stackrel{T_2}{\longrightarrow} u_2, F_2$. Then, by successive applications of lemma 3, one has $t_1, E_1 \rightarrow ... \rightarrow t_2, E_2$. Then, by lemma 2, one must have $t_1 = t_2$ and $E_1 = E_2$, and thus, $u_1 = u_2$, $T_1 = T_2$ and $F_1 = F_2$. □

**Corollary:** There is no incoherent nor nondeterministic SL program.

# 5 SL Implementation

In this section, we describe an implementation of SL. SL programs are first translated into RC, then, compiled to be either executed or to produce automata.

Execution consists in a sequence of micro steps in which decisions concerning signal absences are delayed as far as possible. In fact, it exactly corresponds to what is done in proof of proposition 1. Of course, there is no problem to delay absence decisions, as immediate reactions to signal absences are not possible in SL. The RC `suspend` primitive allows to have micro steps and thus, it provides the way to delay absence decisions: an execution flow that has to test a signal which is not emitted, becomes suspended. Thus, the signal may be emitted by other sub parts of the program. When all execution flows are suspended or finished, all signals that are not already emitted can be considered as absent. Thus, execution may resume and the current instant may complete.

SL modules are translated into RC reactive procedures. Reactive procedures have a way to store their environments, that is where execution flows are stopped, and values of persistent variables (whose values are preserved from one instant to

the next). Thus, to make a reactive procedure react means to execute it in its environment. Reactive procedure environments give a way to produce automata: states are environment values, and transitions are environment changes resulting from procedure activations.

We are going first to overview the RC language, then describe more precisely how SL programs are translated into RC.

## 5.1 The RC Language

RC has been design to allow a reactive programming style in C. The basic notion is that of execution of piece of code up to reach `stop` statements that finishes the current reaction. Execution will restart at the next activation, from these `stop` statements.

**Reactive Procedures.** Reactive procedures are the reactive counterpart of C functions, but they maintain an environment to store control points from where execution starts. Control points can be seen as local static variables that keep their values from an instant (that is a reaction) to the next. For example, consider the following reactive procedure `R`:

```
rproc void R(){
   printf("first reaction\n");
   stop;
   printf("second reaction\n");
   stop;
   printf("termination of R\n");
}
```

The `rproc` keyword introduces reactive procedures. `R` has no parameter and does not return any result. When executed for the first time, it prints "`first reaction`" and stops on the first `stop` statement that defines the first control point value. At the second instant, execution starts from the first `stop`, prints "`second reaction`", and stops on the second `stop` statement. At the third activation, "`termination of R`" is printed and the procedure is terminated. The value of the control point becomes the end of the procedure and it will not change anymore during next activations.

The `activate` statement allows to execute a reactive procedure in its actual environment. On the contrary, the `exec` statement creates an new copy of the environment of a procedure, before executing it from the beginning.

Variable that have to keep their values from one activation to the next must be declared as `rauto`. These variables are also stored in environments. Notice that they are local to reactive procedures, on the contrary to `static` variables.

**Reactive Statements.** Reactive statements are used to code reactions to activations. The `stop` statement is of course the basic reactive statement.

Another important reactive statement is the `rif` statement that chooses at each activation, a statement to execute, accordingly to a boolean condition. For example, consider:

```
rproc void R(int i){
   rif(i)
      for(;;){ printf("True."); stop; }
   else
      for(;;){ printf("False."); stop; }
}
```

Each time it is called, `R` chooses to continue the first branch of the `rif` statement if `i` is true, or the second branch otherwise. Thus, "`True.`" or "`False.`" is printed accordingly to the parameter value.

The "`merge t1 t2`" statement allows to make `t1`, then `t2`, both react in the same instant. For example, consider:

```
rproc void R(){
   merge
      for(;;){ printf("True."); stop; }
      for(;;){ printf("False."); stop; }
}
```

Then "`True.False.`" is printed at each activation. Notice that order of `merge` branches execution is syntactically fixed. The `merge` operator is of course essential to implement parallelism.

**Suspension.** RC give a way to break one instant into several micro-instants. The `suspend` statement stops the control, but on the contrary to `stop`, execution can be resumed (using a `close` statement) during the same instant. Thus, to suspend one branch of a `merge` statement, allows the other to react, but execution of the first branch may be resumed in the same instant.

When the name of reactive statement that tests a condition at each instant, is terminated by `susp`, the statement also performs the test at each micro-instant. For example, the `rifsusp` statement corresponding to `rif`, chooses a branch to continue, accordingly to a boolean condition evaluated at each micro-instant.

## 5.2 Translation into RC

Execution proceeds as follows: parallel statements are executed as far as possible; control leaves one branch for the next when either, is has finished for the current instant (`Stop`), or it has terminated, or when it has to test a signal which is not already emitted. In this last case, execution will be continued either when the signal will be emitted, or at the end of the instant, and in this case, the signal will be considered as absent. The end of the current instant is detected when, after executing all parallel statements, there is no new signal emitted.

**Signals.** Each signal is translated into a "`rauto int*`" variable. To emit signal `i` means to change the corresponding value to 1, and to set a global variable `_move` used to detect the end of the current instant: `((*i)=1,_move=1)`. To reset signal `i` simply means `(*i)=0`. The macro `_PRES` returns 1 if a signal is present, and 0 if it is not.

The following reactive procedure `_Fix` tests for the presence of a signal: it terminates as soon as the signal is present, or at the end of the current instant; otherwise it suspends its execution:

```
rproc void _Fix(s)
int *s;
{
   for(;;){
      if(_PRES(s)||_endOfInstant)break;else suspend;
   }
}
```

**Instants.** The following reactive procedure resets the two global variables `_endOfInstant` and `_move` at the beginning of each instant.

```
rproc void _InitInstants(){
   for(;;){
      _endOfInstant = _move = 0;
      stop;
   }
}
```

The `_TerminateInstants` reactive procedure detects end of instants by setting `_endOfInstant` when `_move` remains unset after the program being executed. When `_move` is set, the procedure unsets it and suspends its execution to let the program reexecute. When `_move` is unset, the procedure sets `_endOfInstant` and suspends its execution so the program can consider as absent signals on which execution was blocked.

```
rproc void _TerminateInstants(){
   rifsusp (_move)
      for(;;){ _move = 0; suspend; }
   else
      for(;;){ _endOfInstant = 1; suspend; stop; }
}
```

**Reactive Statements.** SL statements are easily translated into RC statements. For example:

- SL `stop` statement is translated into RC `stop`, `run` into `exec`, and $\|$ into `merge`.

- "`when S then t1 else t2 end`" is translated into:

```
{
   exec _Fix(S);if(_PRES(S)){t1}else{stop;t2}
}
```

  First, signal status is fixed, then presence test is performed and the corresponding branch is chosen.

- The `kill` case is a little more complex and uses the RC `trap` construct. Translation of "`do t kill S`" is:

```
catch("E"){
  catch("TO"){
    merge
      {t; raise "E";}
      for(;;){exec _Fix(S);if(_PRES(S))raise "TO";stop;}
  }handle stop;
}
```

Notice that using `merge` to implement parallelism corresponds in the semantics, to always perform left micro-transition rules before right ones. Proposition 2 assures that this is a valid strategy to get the result. This shows an example of a commutative parallel implementation using `merge`.

### 5.3 The `sl2rc` command

Now we describe the implementation by means of a small example. Consider the following module `TST1` put in a file named `test1.sl`:

```
main module TST1:
input I;
output O;

wait I;
emit O

end module
```

Translation into RC is obtained by the command:

```
sl2rc < test1.sl > test1.rc
```

Now, file `test1.rc` contains:

```
#include "/p4/rc/fb/SL/7/sl.h"

rproc void TST1(I,O)
int *I,*O;
{
  for(;;){_FORK(I);exec _Fix(I);if(_PRES(I)) break; stop;}
  _EMIT(O);
}
```

The _FORK(I) statement is used for automata production and will be explained later. The _EMIT(O) statement correspond to O emission. Notice that the only way to emit O is to test I as present, after I status being fixed.

As TST1 is a main module, a file named TST1.comp.rc has also been produced by the sl2rc command. It contains the main reactive procedure that allows to execute the module. Its contents is:

```
extern int I,O;
rproc void rmain(){
   rauto int *I__,*O__;
   close
      merge
         terminate(((*I__)=I,0)) activate _InitInstants();
      merge
         exec TST1(I__,O__);
         do
            activate _TerminateInstants();
         terminate(( _RST(I__),O=(*O__),_RST(O__),0));
}
```

## 5.4 Execution

An environment to execute the program is to be given by the user. It could be simply the following env.rc file:

```
int I=0, O=0;
extern rprocType rmain;

main(){
   int i;
   for(i=1; i<5; i++){
     I = (i==2);
     printf("* ");
     react rmain();
     if (O){ printf("O!");O=0; }
     printf("\n");
```

```
   }
}
```

The `main` function calls the global program `rmain` (using the SL `react` primitive) four times, with `I` present at the second instant.

Now, executable code is obtained by:

```
rcc -c test1.rc
rcc -c TST1.comp.rc
rcc -c env.rc
rcc -o test1 test1.o env.o TST1.comp.o -lsl
```

As expected, execution of `test1` gives:

```
fisher$ tst1
*
* O!
*
*
```

## 5.5 Separate Compilation

Separate compilation is available using `.o` codes generated from `.sl` files. As example, consider the reflex game program described in section 3.3. One can compile it by:

```
sl2rc<basic.sl>basic.rc
rcc -c basic.rc
rcc -c PLAY.comp.rc
rcc -o basic basic.o PLAY.comp.o env.o data.o -lsl
```

Then, suppose we want to use the previous basic program within a more complex one that calls it several times:

```
Extern Module PLAY;

main module GAME:
input COIN,MS,READY,STOP;
output DISPLAY,RING_BELL,GAME_OVER,TILT;

-[ PrintOut("Put a coin to start."); ]-;
wait COIN;
stop;
loop
 do
   do
```

```
     repeat -[ 4 ]- times
       signal OVER in
         run PLAY(MS,READY,STOP,DISPLAY,RING_BELL,OVER,TILT);
       end
     end
   kill TILT;
   -[ PrintOut("Put a coin to start."); ]-;
   halt
 kill COIN timeout Stop end
end

end module
```

Now, we can compile the new program using `basic.o`, by:

```
sl2rc<game.sl>game.rc
rcc -c game.rc
rcc -c GAME.comp.rc
rcc -o game game.o GAME.comp.o basic.o env.o data.o -lsl
```

Notice that parallelism "remains" in the `.o` files produced, as it is translated into `merge` statements. Thus, the suspension mechanism and the signal absence hypothesis delay are executed at run time.

Automata production described in the next section, is a way to avoid run time overhead generated by parallelism.

### 5.6 Translation into Automata

Compilation of a SL program without giving any `main` entry point, produces a RC program implementing an automaton. Automaton states are values of the program environment, and the `FORK` macro is used to process both cases where an input signal is emitted by the outside, and where it is not (to do such a job, `FORK` has to make a copy of the program, that is to copy its environment using the RC `rprocDup` function).

Automata format uses the following macros:

- `_state(n)` defines a state numbered by `n`.
- `_next(n)` completes the reaction. Next starting point is state `n`.
- `_over` means that the program is terminated.
- `_go_if_absent(I,n)` tests for the presence of `I`. If `I` is present, then execution continues in sequence, otherwise it goes to state `n`.

For example, suppose the content of the `tst2.sl` file is:

```
main module TST2:
input I;
output O;

signal S in
   when I then emit S end
||
   when S then emit O end
end

end module
```

Signals used are defined in file `tst2sig.rc` whose content is:

```
int I=0, O=0;
```

Then, `TST2` is compiled by the command:

```
sl2rc<tst2.sl>tst2.rc
rcc -o tst2 tst2.rc TST2.comp.rc tst2sig.rc -lsl
```

Executing `tst2` produces an automaton whose body is the following C code:

```
_state(1)_go_if_absent(I,2);_emit(O);_over;
_state(2)_next(3);
_state(3)_over;
```

The initial state is always state 1. State 1 means simply to go in state 2 if `I` is absent otherwise, to emit `O` is emitted and to terminate. If `I` is absent, then the current reaction is finished, and next reaction will start from state 3. In state 3 the module simply terminates.

Notice that now, parallelism and communication through local signals have completely disapeared.

**Minimality.** As another example, consider the following module (the body of which is considered in example 3):

```
main module TST3:
input S,T;
output O;

signal V in
   do
      wait S; emit V
   ||
```

```
      wait T; emit V
   kill V;
   emit O
end

end module
```

The automaton produced is:

```
_state(1)_go_if_absent(S,2);_go_if_absent(T,3);_emit(O);_over;
_state(2)_go_if_absent(T,4);_next(5);
_state(3)_next(6);
_state(4)_next(7);
_state(5)_emit(O);_over;
_state(6)_emit(O);_over;
_state(7)_go_if_absent(S,8);_go_if_absent(T,9);_emit(O);_over;
_state(8)_go_if_absent(T,10);_next(5);
_state(9)_next(6);
_state(10)_next(7);
```

By looking at this code, it is clear that automata produced are not minimal neither in the number of states, nor in the number of transitions. In fact, one can distinguish between micro states (e.g. 2) that correspond to unfinished transitions, and macro states (e.g 5) corresponding to finished transitions. More work has to be done to eliminate micro states, and to identify similar macro states (identification may have the meaning of "bisimulation"[10]; for example, states 5 and 6 are equivalent in this sense).

**The game example.** To show what is gained by automata production, consider the basic reflex game program of section 3.3. One can compile it by:

```
sl2rc<basic.sl>basic.rc
rcc -c basic.rc
rcc -c PLAY.comp.rc
rcc -o basic basic.o PLAY.comp.o env.o data.o -lsl
```

To play with, just type:

```
fisher$ basic
Press r.
Go!
score: 77
It's more fun to compete ...
fisher$
```

Now, to compile the game into an automaton, we just have to suppress the `main` function that is in `env.rc`:

```
rcc -o basiccomp basic.o PLAY.comp.o data.o -lsl
```

Now `basiccomp` produces an automaton that is equivalent to `PLAY`. To see what is gained, first replace the `basic.rc` code by the automaton:

```
basiccomp>basic.rc
```

Then compile the automaton:

```
rcc -c basic.rc
rcc -o basic basic.o PLAY.comp.o env.o data.o -lsl
```

Now, this is a session:

```
ress r.
Go!
score: 509
It's more fun to compete ...
fisher$
```

One sees that the automaton runs faster than the parallel code (supposing the same user reaction time, the greater the score is, the faster the program runs).

# 6 Conclusion

We have presented a new synchronous language based on ESTEREL. In fact, language design and precise syntax were not primary concerns. The idea was to study how specific problems resulting from the synchrony hypothesis and known as causality cycles, could be eliminated while preserving expressive power as far as possible. We show that what has to be rejected are reactions in the current instant to signal absences. The formal semantics given allows us to show that no incoherent nor non-deterministic program does exist anymore. To forbid immediate reactions to signal absences, also simplifies implementation in a very large extend (the implementation described is less than one thousand of RC lines).

Some main features of the synchronous approach still remains: parallelism, signal broadcast, and instantaneous dialogs for example. However, what does rejection of immediate reactions to absence imply for programming style, is not clear and has to be investigated further.

## References

[1] R. Bernhard, G. Berry, F. Boussinot, G. Gonthier, A. Ressouche, J.P. Rigault, and J.M. Tanzi. A quick survey of the Esterel v3 compiler. Rapport technique, EMP, 1988.

[2] G. Berry. A hardware implementation of pure Esterel. *Sadhana, Academy Proceedings in Engineering Sciences, Indian Academy of Sciences*, 17(1):95–130, 1992. Rapport Centre de Mathématiques Appliquées de l'Ecole des Mines de Paris, numéro 06/91.

[3] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[4] F. Boussinot. Programming a reflex game in Esterel v3_2. Rapport de recherche 07/91, Centre de Mathématiques Appliquées, Ecole des Mines de Paris, Sophia-Antipolis, 1991.

[5] F. Boussinot. Reactive c: An extension of c to program reactive systems. *Software Practice and Experience*, 21(4), 1991.

[6] F. Boussinot and R. de Simone. The Esterel language. *Another Look at Real Time Programming, Proceedings of the IEEE*, 79:1293–1304, 1991.

[7] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: a declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages*, january 1987.

[8] G. Gonthier. Sémantique et modèles d'exécution des langages réactifs synchrones; application à Esterel. Thèse d'informatique, Université d'Orsay, 1988.

[9] P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMaire. Programming real time applications with signal. In *IEEE*, september 1991.

[10] R. Milner. Concurrent processes and their syntax, April 1979.

[11] G.D. Plotkin. A structural approach to operational semantics, September 1981.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399