SugarCubes Implementation of Causality

Frédéric Boussinot

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

SugarCubes Implementation of Causality

Frédéric Boussinot

N° 3487

Septembre 1998

THÈME 1

Rapport de recherche

ISSN 0249-6399

INRIA
SOPHIA ANTIPOLIS

SugarCubes Implementation of Causality

Frédéric Boussinot

Thème 1 — Réseaux et systèmes
Projets Meije

Rapport de recherche n° 3487 — Septembre 1998 — 31 pages

Abstract: Causality problems appear in synchronous formalisms basically when one emits an absent signal. Several strategies have been developed to reject programs with causality problems. Strategies based on potential functions are studied, and several formal semantics using these functions are defined for a fragment of the synchronous language Esterel. Implementation in Java of these strategies is also presented. This implementation is based on SugarCubes which is a set of Java classes for reactive programming.

Key-words: Parallelism, Reactive Synchronous Programming, Esterel, Java

With support from France Telecom-CNET

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 — Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

Implémentation de la causalité en SugarCubes

Résumé : Les problèmes de causalité apparaissent dans les formalismes synchrones lors de l'émission d'un signal absent. Plusieurs stratégies ont été développées pour rejeter les programmes ayant de tels problèmes. Dans ce texte, on étudie les stratégies à base de fonctions de potentiels et plusieurs sémantiques formelles utilisant ces fonctions sont définies pour un fragment du langage synchrone Esterel. L'implémentation en Java de ces stratégies est également présentée. Elle utilise les SugarCubes qui sont un ensemble de classes Java pour la programmation réactive.

Mots-clés : Parallélisme, Programmation réactive synchrone, Esterel, Java

1 Introduction

Reactive systems maintain a permanent close interaction with their environment [HP]. Embedded systems, as those used in airplanes, cars or cellular phones, are examples of such systems. Synchronous formalisms[Ha] have been designed for high level specification and programming of reactive systems.

Amongst the synchronous formalisms, is the Esterel language[BG] which adopts an imperative style. In Esterel programs, parallel components all share a global logical clock defining global instants and communicate using broadcast signals. Esterel is based on two fundamental hypothesis:

- the *perfect synchrony hypothesis* states that signal emissions and testings are instantaneous;
- the *determinism hypothesis* states that the behaviour of a program depends only on its inputs, and not on some internal choices.

As consequence of the perfect synchrony hypothesis is the existence of *incoherent programs*, in which there is no way to decide, while respecting the hypothesis, if a signal is present or absent. Basically, incoherency occurs when a signal is emitted only if it is absent.

In Esterel, incoherent or nondeterministic programs are said to have *causality problems*. Several solutions have been proposed to detect these programs at compile time [BG,Be]. However, the basic cause of causality problems is the possibility of *immediate reaction to a signal absence*. Without this possibility, causality problems do not exist anymore [BDS].

The reactive approach based on the Reactive-C language[Bo] comes from the synchronous approach with two main goals:

- to avoid causality problems by restricting the possible instantaneous program reactions (actually, by forbidding instantaneous reaction to absence; instantaneous reaction to presence remains allowed);
- to allow programmers to dynamically create parallel components (which is forbidden in synchronous formalisms) in order to get a more natural approach for dealing with systems which are basically dynamic.

Recently, a set of Java classes named SugarCubes[BS] has been designed for reactive programming in Java[GJS]. Roughly speaking, SugarCubes is to Java what Reactive-C is to C.

In this paper, one describes an experiment made with SugarCubes to deal with instantaneous reaction to absence, in order to get the synchronous and the reactive approaches closer. The flexibility of the Java language appears as a good point for experimenting with various possible solutions.

Causality problems also appear in other synchronous formalisms (for example in Lustre[HCRR], Signal[LBBG], or Statecharts[Har]); however, it is in the Esterel language that these problems appear in the purest form. This is the reason why, in this text, one chooses to put the focus on Esterel and to use its syntax.

The paper has the following structure: in section 2, causality problems and various ways to reject them are presented. The formal semantics of a fragment of Esterel which focus on causality is presented in section 3. Several solutions, based on potentially emitted signals, to reject programs with causality problems are presented in section 4. Comparison with Esterel is made in section 5. SugarCubes is presented in section 6, and is extended in section 7 to implement instantaneous reaction to absence.

2 Causality Problems

The synchronous hypothesis implies that testing a signal does not take time. In particular, one can test for the absence of a signal and, as the test takes no time, *react in the same instant* to this very absence. The basic Esterel statement to test for a signal presence has the form:

```
present S then
    <immediate reaction to presence of S>
else
    <immediate reaction to absence of S>
end
```

2.1 Incoherency

An *incoherent statement* is a statement in which there is no possibility to determine a signal presence status, while respecting the synchronous hypothesis. The following test is an example of an incoherent statement (`nothing` is the statement that does nothing and terminates immediately; `emit` is signal emission):

```
present S then
  nothing
else
  emit S
end
```

On one hand, signal S cannot be present as it is emitted only in case it is absent. On the other hand, signal S cannot be absent as, then, it is emitted and thus present.

In incoherent statements, the symptom is always the same: immediate reaction to the absence of a signal leads to negating this absence by emitting the signal.

Incoherency is related to parallelism as one can get an incoherent statement by putting in parallel two coherent statements. The following parallel statement is an example of an incoherent statement made of two coherent ones (absence of the `then` or of the `else` branch in a `present` statement simply means that this branch is `nothing`):

```
present S2 else          present S1 then
   emit S1          ||      emit S2
end                      end
```

If S2 is absent, then S1 is emitted by the left parallel branch; as S1 is present, S2 is emitted, which is contradictory with the absence of S2. Now, if S2 is present, then it is emitted by the right branch; but, this implies that S1 is also present; this is not possible because, as S2 is present, there is no emission of S1.

2.2 Nondeterminism

A nondeterministic statement can behave differently, while respecting the synchronous hypothesis. Consider the previous example, changing in the right parallel the `then` branch by a `else` branch:

```
present S2 else          present S1 else
   emit S1          ||      emit S2
end                      end
```

There are two possibilities: either S2 is absent and S1 present, or S1 is absent and S2 present, which lead to two different behaviours. An important point is that the two behaviours consist in an immediate reaction to a signal absence.

One now considers several general strategies proposed to reject statements with causality problems. The precise use of theses strategies by Esterel compilers is considered in section 5.

2.3 Static Cycle Detection

In static cycle detection, the basic idea is to consider the graph of signal emissions and tests; as any causality problem implies a cycle in this graph, rejecting statements with cyclic graphs implies rejecting all programs with causality problems.

In the previous example, there is an arrow from S2 to S1 produced by the left parallel branch, and conversely, there is an arrow from S1 to S2 produced by the right one. Thus, there is a cycle and the statement is rejected.

This approach is very restrictive as it also rejects a very large class of programs free from causality problem. Here is an example of a coherent rejected statement (`pause` is the Esterel statement to stop execution for the current instant):

```
present S1 else emit S2 end;
pause;
present S2 else emit S1 end
```

There is a cycle between S1 and S2 in the static graph constructed, despite the fact that the two tests are not executed at the same instant.

As another example, consider:

```
present S1 then
  present S2 else emit S2 end
end
```

It is rejected while, as no emission of S1 exist, the `then` branch of `present` S1 cannot be run.

The static cycle detection is used in the Esterel v4 compiler (considered in section 5.3) and in the Lustre compiler.

2.4 Possibly Emitted Signals

In approaches based on possibly emitted signals, the compiler can decide that a signal is absent only when there is no possibility for it to be emitted. Consider the previous example:

```
present S2 else          present S1 then
     emit S1                  emit S2
end                 ||   end
```

The compiler blocks on both branches of the parallel statement, as S1 and S2 are not present. At that moment, S1 cannot be decided to be absent because there is a potential emission of it in the `else` part of the left branch. S2 also cannot be decided as absent as there is a potential emission of it in the right branch. Thus, the statement is rejected as there is no way to proceed.

Several ways of computing possibly emitted signals are possible. In a simple solution, the compiler computes potential emissions without *using the signal environment* (this solution is used in the Esterel v3 compiler described in section 5.2). Consider for example:

```
emit S2;
present S1 then
  present S2 else emit S1 end
end
```

The compiler blocks when evaluating the test of S1 and computes the possibly emitted signals at that stage. As the compiler does not use the signal environment, it cannot deduce that, as S2 is already emitted, emission of S1 is actually impossible. Thus the statement is rejected, as S1 is possibly emitted.

In a more powerful solution (used in the Esterel v5 compiler described in section 5.3), the compiler can use the signal environment, but *without any possibility to add new information in it*, while computing potential emissions. The previous statement is accepted by such a solution, as the fact that emission of S1 is impossible can be deduced, when the test is analysed, from the fact that S2 is emitted. However, there still exist statements without causality problem which are rejected. Consider for example:

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

Emission of S is unreachable as the two tests of S are exclusive. However the compiler is not able to exclude this emission, when S is not already emitted, as, during potential analysis, S does not appear in the environment; thus S is seen as potentially emitted and the statement is rejected.

More powerful compilers can be designed that would be *allowed to increase their knowledge in the course of the computation of possible emissions*. It is a goal of this paper to study such approaches.

The present discussion is rather imprecise and it is necessary to be formal to eliminate possible ambiguities; this is done in the next section.

3 Formal Semantics

One considers a small subset of Esterel statements, sufficient to reflect the existence of causality problems and gives it a *basic semantics*, expressed in an operational style based on rewriting rules[Pl].

The basic semantics defined in this section rejects incoherent statements, but accepts nondeterministic ones. A restriction in the way signals are decided absent is introduced in section 4 in order to reject nondeterministic statements.

3.1 Basic Rules

One writes:

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

to means that statement t, executed in the signal environment E, transforms (one also says *rewrites*) in t', and E in E', and returns α as termination flag. There are 3 possible termination flags:

- TERM means that execution is terminated and that nothing remains to do;
- STOP means that execution is terminated but that something remains to do at next instant;
- SUSP means that execution must be resumed in the current instant.

An environment is a set which contains signals that are present or absent, but not both (environments are *coherent*). To note that an event S is present, one just puts its name in the environment; to note that it is absent, one puts its name with a bar on it ($\overline{S}$).

Environment E+$\overline{S}$ is obtained by adding $\overline{S}$ to E provided S is not in E (otherwise E+$\overline{S}$ would not be coherent); In the same way, E+S consists in adding S to E, provided that $\overline{S}$ is not in E; finally, E+E' is the union of E and E'.

As the focus is put on causality, we consider a very simple syntax which captures the core of the problem: sequence, parallel, pause, and signal emission and test.

One extends the syntax of the parallel statement: $_{\alpha}||_{\beta}$ is used to store the termination flags of the parallel branches (α is the termination flag of the left branch, β is the termination flag of the right one); α and β can be omitted when they are both equal to SUSP.

The BNF description of statements is:

$$\begin{array}{ll}
t = & \texttt{nothing} \\
| & \texttt{pause} \\
| & t \; ; \; t \\
| & t \;_{\alpha}||_{\beta}\; t \\
| & \texttt{emit}\ S \\
| & \texttt{present}\ S\ \texttt{then}\ t\ \texttt{else}\ t\ \texttt{end}
\end{array}$$

Compared to the Esterel kernel syntax, loops, boolean `if` tests, preemption, and traps are not considered here. One feels that their introduction does not really complicate the problem, which is deeply concentrated in the statements considered here, specially in the parallel and `present` ones. As a justification, let us recall that in Esterel the semantics of preemption is mainly captured by the `present` operator, and that the semantics of loops is basically captured by the ones of sequence.

Now, one considers each operator in turn and gives rewriting rules for it.

Nothing

The `nothing` statement does nothing and terminates:

$$\text{nothing,E} \xrightarrow{\text{TERM}} \text{nothing,E}$$

Pause

The `pause` statement stops execution for the current instant, and nothing remains to be done at next instant:

$$\mathrm{pause},E \xrightarrow{\mathrm{STOP}} \mathrm{nothing},E$$

Emit

The `emit` statement emits a signal and rewrites in `nothing`:

$$\mathrm{emit}\ S,E \xrightarrow{\mathrm{TERM}} \mathrm{nothing},E+S$$

Let us recall that S must not have the absent status in E ($\not{S}$ not in E), otherwise E+S would not be coherent.

Sequence

There are two rules for the sequence, depending on the termination of the left branch.

- If the left branch terminates, then the right one is immediately executed:

$$\frac{t,E \xrightarrow{\mathrm{TERM}} t',E' \quad u,E' \xrightarrow{\alpha} u',E''}{t;u,E \xrightarrow{\alpha} u',E''}$$

- If the left branch is stopped or suspended, then so is the sequence:

$$\frac{t,E \xrightarrow{\alpha} t',E' \quad \alpha \neq \mathrm{TERM}}{t;u,E \xrightarrow{\alpha} t';u,E'}$$

Signal Tests

There are three rules for the signal test, depending if the signal is present, absent, or unknown.

- the `then` branch is executed if the signal is present ($S \in E$):

$$\frac{S \in E \quad t,E \xrightarrow{\alpha} t',E'}{\mathrm{present}\ \ S\ \ \mathrm{then}\ \ t\ \ \mathrm{else}\ \ u\ \ \mathrm{end},E \xrightarrow{\alpha} t',E'}$$

- the `else` branch is executed if the signal is absent ($\not{S} \in E$):

$$\frac{\not{S} \in E \quad u,E \xrightarrow{\alpha} u',E'}{\mathrm{present}\ \ S\ \ \mathrm{then}\ \ t\ \ \mathrm{else}\ \ u\ \ \mathrm{end},E \xrightarrow{\alpha} u',E'}$$

- the test is suspended if the signal is unknown (neither present nor absent):

$$\frac{S \notin E \quad \not{S} \notin E}{\mathrm{present}\ \ S\ \ \mathrm{then}\ \ t\ \ \mathrm{else}\ \ u\ \ \mathrm{end},E \xrightarrow{\mathrm{SUSP}} \mathrm{present}\ \ S\ \ \mathrm{then}\ \ t\ \ \mathrm{else}\ \ u\ \ \mathrm{end},E}$$

Parallelism

Parallelism is synchronous: both branches run together in the same instant.

- If both branches are suspended (which is the initial situation at each instant), then they are both executed (let us remember that || means ${}_{\mathrm{SUSP}}||_{\mathrm{SUSP}}$):

$$\frac{t,E \xrightarrow{\alpha} t',E' \quad u,E \xrightarrow{\beta} u',E''}{t\,||\,u,E \xrightarrow{\mathrm{SUSP}} t' \;{}_{\alpha}||_{\beta}\; u',E'+E''}$$

Note that the two parallel branches are run in the same environment and that the produced environment E'+E'' must be coherent. This forbids emission of a signal in one branch, and decision of its absence in the other branch.

- If there is only one suspended branch, then the other one is run:

$$\frac{\beta \neq \mathrm{SUSP} \quad t,E \xrightarrow{\alpha} t',E'}{t \;{}_{\mathrm{SUSP}}||_{\beta}\; u,E \xrightarrow{\mathrm{SUSP}} t' \;{}_{\alpha}||_{\beta}\; u,E'} \qquad \frac{\alpha \neq \mathrm{SUSP} \quad u,E \xrightarrow{\beta} u',E'}{t \;{}_{\alpha}||_{\mathrm{SUSP}}\; u,E \xrightarrow{\mathrm{SUSP}} t \;{}_{\alpha}||_{\beta}\; u',E'}$$

- The parallel is terminated or stopped when both branches are:

$$\frac{\alpha \neq \mathrm{SUSP} \quad \beta \neq \mathrm{SUSP}}{t \;{}_{\alpha}||_{\beta}\; u,E \xrightarrow{\alpha\times\beta} t\,||\,u,E}$$

where $\alpha \times \beta$ equals TERM if both α and β are TERM, and equals STOP otherwise.

3.2 Absence Decision

Up to now, there is no possibility to make signals absent. This possibility is introduced with the notion of a *complete execution*: a complete execution of a statement t is a sequence of rewritings, starting from t and ending with a termination flag TERM or STOP. More precisely, a complete execution of statement t_0, with E_0 as input environment, is a sequence of the form:

$$t_0,E_0 \xrightarrow{\mathrm{SUSP}} t_1,E_1 \quad t_1,E_1+X_1 \xrightarrow{\mathrm{SUSP}} t_2,E_2 \quad \ldots \quad t_n,E_n+X_n \xrightarrow{\alpha} t_{n+1},E_{n+1}$$

where:

- α is different from SUSP;
- there is no absent signal in the input environment E_0;
- $X_1,\ldots,X_n$ are sets of absent signals.

One simply notes the previous sequence by:

$$t_0,E_0 \overset{\alpha}{\Rightarrow} t_{n+1},E_{n+1}$$

Signals present in E_0 are inputs, produced by the external context, which is not allowed to make signals absent. It is important to remark that absent signals in X_i cannot be emitted in E_i, in order E_i+X_i to be coherent.

Example

Let us prove that:

$$\text{present S1 then emit S2 end || emit S1, } \emptyset \overset{\text{TERM}}{\Longrightarrow} \text{nothing || nothing, S1,S2}$$

one has:

$$\frac{\begin{array}{l}\text{present S1 then}\\ \quad\text{emit S2}\\ \text{end, } \emptyset\end{array} \xrightarrow{\text{SUSP}} \begin{array}{l}\text{present S1 then}\\ \quad\text{emit S2}\\ \text{end, } \emptyset\end{array} \qquad \text{emit S1, } \emptyset \xrightarrow{\text{TERM}} \text{nothing, S1}}{\begin{array}{l}\text{present S1 then}\\ \quad\text{emit S2}\\ \text{end}\\ \text{||}\\ \text{emit S1, } \emptyset\end{array} \xrightarrow{\text{SUSP}} \begin{array}{l}\text{present S1 then}\\ \quad\text{emit S2}\\ \text{end}\\ \text{SUSP || TERM}\\ \text{nothing, S1}\end{array}}$$

and:

$$\frac{\dfrac{\text{emit S2, S1} \xrightarrow{\text{TERM}} \text{nothing, S1,S2}}{\begin{array}{l}\text{present S1 then}\\ \quad\text{emit S2}\\ \text{end, S1}\end{array} \xrightarrow{\text{TERM}} \text{nothing, S1,S2}}}{\begin{array}{l}\text{present S1 then}\\ \quad\text{emit S2}\\ \text{end}\\ \text{SUSP || TERM}\\ \text{nothing, S1}\end{array} \xrightarrow{\text{SUSP}} \text{nothing TERM || TERM nothing, S1,S2}}$$

which finally gives the result because:

$$\text{nothing }_{\text{TERM}}\text{||}_{\text{TERM}}\text{ nothing, S1,S2} \xrightarrow{\text{TERM}} \text{nothing||nothing, S1,S2}$$

No signal absence is needed in this proof.

Nondeterminism

The result of the rewriting may depend on the choice of signals that are decided absent. Consider:

```
  present S1 else emit S2 end
||
  present S2 else emit S1 end
```

If S1 is decided as absent, then S2 is emitted; on the contrary, if S2 is decided as absent, then S1 is emitted. Both choices are possible and such a statement is nondeterministic. Note that deciding both signals as absent in a single step would not lead to a terminated rewriting.

Nondeterminism will be rejected in section 4, by restricting the way signals are decided absent.

3.3 Definition of Incoherency

A statement t is incoherent if there exists an environment E without any absent signal in it such that no terminated or stopped rewriting apply to t,E:

$$\text{there is no } t',\ E' \text{ such that } t,E \overset{\alpha}{\Rightarrow} t',E'$$

Examples of Incoherent Statements

Let us try to execute the following statement in the empty environment:

```
present S else emit S end
```

The only possibility is to decide that S is absent; but then, `emit` S would have to be executed in the environment where S is absent, which is impossible. Thus, there is no way to rewrite the statement in the empty environment, which means that it is incohe`rent`.

In the same way, the following statement is shown to be incoherent:

```
present S then nothing end; emit S
```

Examples of Coherent Statements

From the previous example, it is easy to see that the following statement, which does not need any absence decision, is coherent:

```
present S1 then emit S2 end || emit S1
```

The following statement is also coherent:

```
present S then emit S end
```

In an environment E where S is present, the `then` branch is executed, which does not change E and rewrites in `nothing`. In an environment where S is not present, S must first be decided as absent and then the `else` branch, which is `nothing`, is executed.

Finally, let us prove that the following statement is coherent:

```
present S1 then
  emit S2;
  present S2 else emit S1 end
end
```

If the environment does not contain S1, the only possibility is to decide that S1 is absent; then, the implicit `else` branch is chosen and the statement rewrites in `nothing`. If

S1 is in the environment, then the statements becomes equivalent to:

```
emit S2;
present S2 else emit S1 end
```

which is coherent. Thus, the initial statement is coherent.

4 Potential Functions

One now restrict the possibility of deciding that signals are absent in order to reject nondeterministic statements. One uses a potential function Π which computes *potentially emitted* signals. Now, one considers sequences of rewritings of the form:

$$t_0,E_0 \xrightarrow{SUSP} t_1,E_1 \quad t_1,E_1+X_1 \xrightarrow{SUSP} t_2,E_2 \quad \ldots \quad t_n,E_n+X_n \xrightarrow{\alpha} t_{n+1},E_{n+1}$$

where, at each step, all signals that are not potentially emitted are decided absent: for all i, X_i is the set of $\overline{S}$ such that $S \notin \Pi(t_i, E_i)$. Intuitively, this is a correct strategy, provided Π is *correct*, that is indeed detects signals which are potentially emitted.

In the following subsections, one considers several Π functions. The three first functions correspond to the v3, v4, and v5 versions of the Esterel compiler; for this reason, one gives them the names Π_{v3}, Π_{v4}, and Π_{v5}. Two other functions are also considered; for simplicity, they are named Π_{v6} and Π_{v7}.

One adopts the two following notations:

- if X is a set, α:X=X if α=SUSP, and α:X=$\emptyset$ if $\alpha \neq$SUSP;
- if F is a boolean expression, α#F=F if α=SUSP, α#F=true if α=TERM, and α#F=false if α=STOP.

4.1 The v3 Semantics

The potential function Π_{v3} (corresponding to the Esterel v3 compiler described in section 5.2) does not use the signal environment. It is defined by:

$$\Pi_{v3}(t,E) = \pi_{v3}(t)$$

where π_{v3} is defined as follows:

- π_{v3}(`nothing`) = π_{v3}(`pause`) = $\emptyset$
- π_{v3}(`emit S`) = S
- π_{v3}(t $_{\alpha}||_{\beta}$ u) = α:π_{v3}(t) $\cup$ β:π_{v3}(u)
- π_{v3}(`present S then` t `else` u `end`) = π_{v3}(t) $\cup$ π_{v3}(u)
- π_{v3}(t;u) = π_{v3}(t) $\cup$ π_{v3}(u), if τ_{v3}(t),

$= \pi_{v3}(t)$ otherwise.

The function τ_{v3} returns true if execution of its parameter *can* terminate, false otherwise:

- τ_{v3}(`nothing`) = τ_{v3}(`emit` S) = **true**
- τ_{v3}(`pause`) = **false**
- $\tau_{v3}(t\ {}_{\alpha}||_{\beta}\ u) = \alpha\#\tau_{v3}(t)$ **and** $\beta\#\tau_{v3}(u)$
- $\tau_{v3}(t;u) = \tau_{v3}(t)$ **and** $\tau_{v3}(u)$
- τ_{v3}(`present` S `then` t `else` u `end`) = $\tau_{v3}(t)$ **or** $\tau_{v3}(u)$

4.2 The v4 Semantics

The potential function Π_{v4} (corresponding to the static analysis of the Esterel v4 compiler described in section 5.3) does not take instants into account. It is defined by:

$$\Pi_{v4}(t,E) = \pi_{v4}(t)$$

where π_{v4} is defined as follows:

- π_{v4}(`nothing`) = π_{v4}(`pause`) = $\emptyset$
- π_{v4}(`emit` S) = S
- $\pi_{v4}(t\ {}_{\alpha}||_{\beta}\ u) = \alpha{:}\pi_{v4}(t) \cup \beta{:}\pi_{v4}(u)$
- $\pi_{v4}(t;u)$ = π_{v4}(`present` S `then` t `else` u `end`)

  $= \pi_{v4}(t) \cup \pi_{v4}(u)$

The only difference with the definition of the potential function of v3 concerns the sequence operator. Basically, in v4 instants are not considered and the sequence and parallel operators are processed in the same way.

From the definition, it is clear that Π_{v4} is more restrictive than Π_{v3}: a coherent statement for Π_{v4} is also coherent for Π_{v3}. One writes: $\Pi_{v4} \subset \Pi_{v3}$.

4.3 The v5 Semantics

The potential function Π_{v5} (corresponding to the Esterel v5 compiler described in section 5.4) uses the environment but does not change it in any way. It is defined by:

- Π_{v5}(`nothing`,E) = Π_{v5}(`pause`,E) = $\emptyset$
- Π_{v5}(`emit` S,E) = S
- $\Pi_{v5}(t\ {}_{\alpha}||_{\beta}\ u,E) = \alpha{:}\Pi_{v5}(t,E) \cup \beta{:}\Pi_{v5}(u,E)$
- $\Pi_{v5}(t;u,E) = \Pi_{v5}(t,E) \cup \Pi_{v5}(u,E)$, if $\tau_{v5}(t,E)$

$= \Pi_{v5}(t,E)$, otherwise

- Π_{v5}(present S then t else u end,E)

  $= \Pi_{v5}(t,E)$, if $S \in E$

  $= \Pi_{v5}(u,E)$, if $\not{S} \in E$

  $= \Pi_{v5}(t,E) \cup \Pi_{v5}(u,E)$, otherwise

Definition of τ_{v5} is:

- τ_{v5}(nothing,E) = τ_{v5}(emit S,E) = **true**
- τ_{v5}(pause,E) = **false**
- $\tau_{v5}(t \ {}_{\alpha}|\,|_{\beta}\ u,E) = \alpha\#\tau_{v5}(t,E)$ **and** $\beta\#\tau_{v5}(u,E)$
- $\tau_{v5}(t;u,E) = \tau_{v5}(t,E)$ **and** $\tau_{v5}(u,E)$
- τ_{v5}(present S then t else u end,E)

  $= \tau_{v5}(t,E)$, if $S \in E$

  $= \tau_{v5}(u,E)$, if $\not{S} \in E$

  $= \tau_{v5}(t,E)$ **or** $\tau_{v5}(u,E)$, otherwise

The environment E is only used by present and always passed as it, without being changed.

It is clear from the definition that Π_{v3} is more restrictive than Π_{v5}: $\Pi_{v3} \subset \Pi_{v5}$.

4.4 The v6 Semantics

The potential function Π_{v6} is a variant of Π_{v5} in which the signal environment can be changed while analysing present statements, to keep track of the choosen branch. Π_{v6} is defined exactly as Π_{v5}, except for present:

Π_{v6}(present S then t else u end,E)

$= \Pi_{v6}(t,E)$, if $S \in E$

$= \Pi_{v6}(u,E)$, if $\not{S} \in E$

$= \Pi_{v6}(t,E+S) \cup \Pi_{v6}(u,E+\not{S})$, otherwise

with τ_{v6} defined exactly as τ_{v5}, except for:

τ_{v6}(present S then t else u end,E)

$= \tau_{v6}(t,E)$, if $S \in E$

$= \tau_{v6}(u,E)$, if $\not{S} \in E$

$= \tau_{v6}(t,E+S)$ **or** $\tau_{v6}(u,E+\not{S})$, otherwise

The difference with Π_{v5} only concerns the case where the signal status is unknown (neither S nor $\not S$ is in E); then, the two branches are considered, but they are analysed in the environment *augmented with the according signal status*.

It is clear from the definition that Π_{v5} is more restrictive than Π_{v6}: $\Pi_{v5} \subset \Pi_{v6}$.

Let us show that the following statement t is coherent for v6:

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

If S is in the starting environment E, then the rule for `present` with S present applies, and t rewrites in `nothing`. Let us suppose now that S is not in E. Then `present` can only suspend (remember: the starting environment does not contain any absent signal). Then, the only solution is to make S absent, which needs to compute the following:

Π_{v6}(t,E) = Π_{v6}(`present S else emit S end`,E+S) $\cup$ Π_{v6}(`nothing`,E+$\not S$)

= Π_{v6}(`nothing`,E+S) $\cup$ $\varnothing$ = $\varnothing$

Thus, S can safely be decided absent, and t is coherent.

4.5 The v7 Semantics

The potential function Π_{v7} is a variant of Π_{v6} with a finer analysis of sequences. In Π_{v7}, signals that are *necessarily* emitted by the left part are added to the environment of the right part. Π_{v7} is defined exactly as Π_{v6}, except for the sequence which is as follows:

Π_{v7}(t;u,E) = Π_{v7}(t,E) $\cup$ Π_{v7}(u,E+M_{v7}(t,E)), if τ_{v6}(t,E)

= Π_{v7}(t,E), otherwise

Signals that are necessarily emitted (computed by the function M_{v7}) in the left part of a sequence can be safely considered as present in the right part. Definition of M_{v7} is as follows:

- M_{v7}(`nothing`,E) = M_{v7}(`pause`,E) = $\varnothing$
- M_{v7}(`emit S`,E) = S
- M_{v7}(t $_\alpha$| |$_\beta$ u,E) = α:M_{v7}(t,E) $\cup$ β:M_{v7}(u,E)
- M_{v7}(`present S then` t `else` u `end`,E)

  = M_{v7}(t,E), if S $\in$ E

  = M_{v7}(u,E), if $\not S$ $\in$ E

  = M_{v7}(t,E+S) $\cap$ M_{v7}(u,E+$\not S$), otherwise

- $M_{v7}(t;u,E) = M_{v7}(t,E) \cup M_{v7}(u,E+M_{v7}(t,E))$, if $\mu_{v7}(t,E)$
  $= M_{v7}(t,E)$, otherwise

The function μ_{v7} returns true if execution of its parameter *must* terminate, and false otherwise:

- μ_{v7}(`nothing`,E) = μ_{v7}(`emit` S,E) = **true**
- μ_{v7}(`pause`,E) = **false**
- $\mu_{v7}(t\ {}_{\alpha}||_{\beta}\ u,E) = \alpha\#\mu_{v7}(t,E)$ **and** $\beta\#\mu_{v7}(u,E)$
- $\mu_{v7}(t;u,E) = \mu_{v7}(t,E)$ **and** $\mu_{v7}(u,E)$
- μ_{v7}(`present` S `then` t `else` u end,E)
  $= \mu_{v7}(t,E)$, if $S \in E$
  $= \mu_{v7}(u,E)$, if $\mathbb{S} \in E$
  $= \mu_{v7}(t,E+S)$ **and** $\mu_{v7}(u,E+\mathbb{S})$, otherwise

The only difference with τ_{v6} is that **and** replaces **or** in the last equality. This reflects the fact that, in order to be sure that a `present` statement terminates, one must be sure that both branches do.

The following statement is accepted by the v7 semantics:

```
present S1 then
  emit S2;
  present S2 else emit S1 end
end
```

It is clear from the definition that Π_{v6} is more restrictive than Π_{v7}: $\Pi_{v6} \subset \Pi_{v7}$.

4.6 Others Possible Semantics

We have seen in the previous sections the inclusions of semantics:

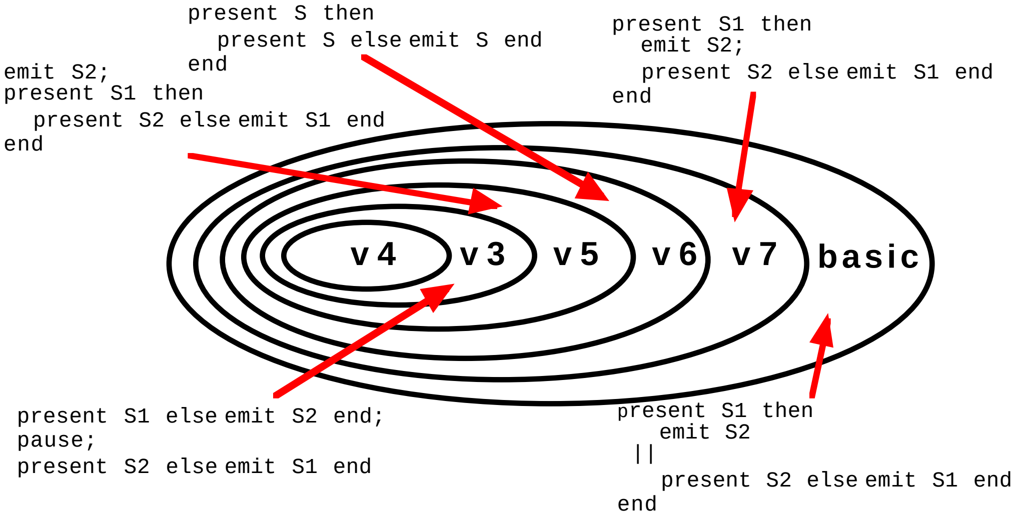
$$\Pi_{v4} \subset \Pi_{v3} \subset \Pi_{v5} \subset \Pi_{v6} \subset \Pi_{v7}$$

Examples of coherent but rejected statements have been given for each of v3-v6; here is an example of a coherent statement rejected by v7:

```
present S1 then
   emit S2
 ||
   present S2 else emit S1 end
end
```

The value returned by Π_{v7} when computing `present` S1 is the union of the values returned by the two parallel branches in the same environment. Thus, there is no way to see that, as S2 is necessarily emitted by the left branch, emission of S1 is actually impossible.

The following figure sums up the situation:



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

```
present S1 then
  emit S2;
  present S2 else emit S1 end
end
```

```
emit S2;
present S1 then
  present S2 else emit S1 end
end
```

```
present S1 else emit S2 end;
pause;
present S2 else emit S1 end
```

```
present S1 then
   emit S2
 ||
   present S2 else emit S1 end
end
```

It is possible to imagine semantics even more powerful than v7; consider for example a variant of v7 with the following change for parallel:

$$\Pi_{v7}(t\ {}_{\alpha}||_{\beta}\ u,E) = \alpha{:}\Pi_{v7}(t,E+M_{v7}(u,E)) \cup \beta{:}\Pi_{v7}(u,E+M_{v7}(t,E))$$

Each branch sees the signals that must be emitted by the other. Such a semantics would accept the previous statement.

However, one shall not consider such semantics in more details as they would be very close from the basic semantics.

4.7 Correctness

In potential based semantics, there is no choice at any step in the rule to apply (it is defined by the structure of the statement) nor in the signals to be decided as absent (they are defined by the potential function).

In the basic semantics, a nondeterministic situation always has the form:

$$\begin{tikzcd} & t1,E1 \\ t,E \arrow[ur, Rightarrow, "\alpha"] \arrow[dr, Rightarrow, "\beta"'] & \\ & t2,E2 \end{tikzcd}$$

where there exists a signal S which is present in one of E1, E2 but is absent in the other. One says that t,E is *nondeterministic*.

One shows that, if t,E is nondeterministic, then no terminating rewriting sequence can be built from t,E in a potentially based semantics. To get this result, one first shows the correctness of the potential functions, which intuitively means that all emitted signals are indeed detected.

Definition of Correctness

A potential function Π is *correct* if all signals that are emitted are indeed detected:

$$\text{if } t,E \xrightarrow{\alpha} t',E' \text{ then for all S emitted in E'-E, } S \in \Pi(t,E)$$

One first proves correction of v5 (the simpler cases of v3 and v4 are left to the reader), then correction of v7 (the case of v6 is very similar to the one of v5). Proofs are by induction on the structure of statements.

Correction of v5

In the following, Π means Π_{v5} and τ means τ_{v5}. One has the following properties:

Property 1. Rewriting always makes the environment more precise:

$$t,E \xrightarrow{\alpha} t',E' \text{ implies } E \subset E'$$

Property 2. Impossibility of termination cannot disappear as the environment becomes more precise:

$$E1 \subset E2 \text{ and } \tau(t,E2) \text{ implies } \tau(t,E1)$$

Property 3. The more precise the environment is, the smaller the potentially emitted signals set is:

$$\text{if } E1 \subset E2 \text{ then } \Pi(t,E2) \subset \Pi(t,E1).$$

Property 4. The function τ reflects termination:

$$t,E \xrightarrow{\text{TERM}} t',E' \text{ implies } \tau(t,E)$$

Property 5. Π_{v5} is correct:

$$\text{if } t,E \xrightarrow{\alpha} t',E' \text{ then S emitted in E'-E implies } S \in \Pi_{v5}(t,E)$$

The proof is by induction.

- The property is of course true for `nothing` and `pause` that do not change the environment.
- Suppose t is `emit S`. Then E'-E = S = Π(`emit S`,E).
- Suppose t is `present S then t1 else t2 end`. One gets the result by inspecting the

three cases.

- Suppose t is $t1_{\alpha}||_{\beta}t2$. One gets the result by inspecting the various cases.
- Suppose t is t1;t2. If t1 does not terminate, one gets the result by induction. Now, suppose that t1 terminates:

$$\frac{t_1,E \xrightarrow{TERM} t',E' \quad t_2,E' \xrightarrow{\alpha} t'',E''}{t_1;t_2,E \xrightarrow{\alpha} t'',E''}$$

Suppose $S \in E'-E$; then, one gets the result by induction. Suppose now that $S \in E''-E'$; by induction, $S \in \Pi(t2,E')$. By property 4, $\tau(t1,E)$ is true; thus $\Pi(t1;t2,E) = \Pi(t1,E) \cup \Pi(t2,E)$. But, as $E \subset E'$, $S \in \Pi(t2,E)$, by property 3, which gives the result.

Correction of v7

One needs the following auxiliary property concerning M_{v7}.

Property 6. Signals that must be emitted are indeed emitted by a terminated rewriting:

$$t,E \xrightarrow{\alpha} t',E' \text{ with } \alpha \neq SUSP \text{ implies } M_{v7}(t,E) \subset E'$$

Property 7. Π_{v7} is correct.

The only operator to consider is the sequence. Suppose t is t1;t2. If t1 does not terminate, one has the result by induction. Suppose now t1 terminates:

$$\frac{t_1,E \xrightarrow{TERM} t',E' \quad t_2,E' \xrightarrow{\alpha} t'',E''}{t_1;t_2,E \xrightarrow{\alpha} t'',E''}$$

Suppose $S \in E'-E$; then, one gets the result by induction. Otherwise, $S \in E''-E'$. As $\tau_{v7}(t1,E)$ is true (property 4), one has $\Pi_{v7}(t1;t2,E) = \Pi_{v7}(t1,E) \cup \Pi_{v7}(t2,E+M_{v7}(t,E))$. By induction, $S \in \Pi_{v7}(t2,E')$. By property 6, $M_{v7}(t1,E) \subset E'$. As $E \subset E'$, $E+M_{v7}(t1,E) \subset E'$. Thus, $S \in \Pi_{v7}(t2,E+M_{v7}(t1,E))$ which proves that $S \in \Pi_{v7}(t1;t2,E)$.

Extension of Correctness to Sequences

In the following, Π means one of Π_{v3} -Π_{v7}. One has the following property:

Property 8. Non-terminated rewritings make potentially emitted signals sets more precise:

$$\text{if } t,E \xrightarrow{SUSP} t',E' \text{ then } \Pi(t',E') \subset \Pi(t,E)$$

Correction of Π extends to sequences. Suppose:

$$t_0,E_0 \xrightarrow{SUSP} t_1,E_1 \quad t_1,E_1+X_1 \xrightarrow{SUSP} t_2,E_2 \quad \ldots \quad t_n,E_n+X_n \xrightarrow{\alpha} t_{n+1},E_{n+1}$$

with a signal S present in E_{n+1} but not in E_0. Then, there exist j such that $S \in E_{j+1}-E_j$. As Π is correct, $S \in \Pi(t_j,E_j+X_j)$. Thus, by property 3, $S \in \Pi(t_j,E_j)$. By applying property 8 and property 3 repeatedly, one gets the result: $S \in \Pi(t_0,E_0)$.

4.8 Determinism

Now one can prove that if t,E is nondeterministic in the basic semantics, then it is rejected by potential based semantics.

Let us consider the nondeterministic situation:

$$\begin{tikzcd} & t1,E1 \\ t,E \arrow[ur, Rightarrow, "\alpha"] \arrow[dr, Rightarrow, "\beta"'] & \\ & t2,E2 \end{tikzcd}$$

and a signal S1 which is present in one of E1, E2 but is absent in the other. Let us write X->Y if emission of Y results from choosing one branch of a present X statement, and absence of Y results from choosing the other branch. One build a set G, starting from S1. There exists a signal S2 such that S2->S1. If S2 is equal to S1, one terminates the construction of G because of the cycle S1<->S1. Otherwise, one continues with S2 and, as the number of signals is finite, one eventually falls in the first case. At the end of the construction, each signal in G can be emitted by deciding absent an other signal of G. Now, considering the potential based semantics, on one hand, none of the elements in G can be decided absent as it is potentially emitted. On the other hand, there is no possibility to get a terminated rewriting without making one element of G absent. Thus, there is a step with no possibility of progression, which implies that t,E is rejected.

5 Comparison with Esterel

In this section, one compares the semantics previously defined with the Esterel semantics. One considers four Esterel semantics: the behavioural semantics, and the semantics corresponding to versions v3, v4, and v5 of the Esterel compilers.

5.1 Behavioural Semantics

As the basic semantics, the Esterel *behavioural semantics*[BG] does not reject nondeterministic statements but only incoherent ones. In the behavioural semantics, the key rule for causality is the one of local signal declaration. It is the place where one can make hypothesis on signal presence or absence. Hypothesis concern local signals and must be validated in the produced environment: a signal supposed absent must not be emitted, and a signal supposed present must be emitted.

On the opposite, in the basic semantics, the only rule that can make a signal present is the `emit` rule; there is no possibility to suppose a signal present. An advantage is that Esterel *non-causal* executions, in which a presence hypothesis is needed, are impossible in the basic semantics. Consider:

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

It is accepted by the behavioural semantics of Esterel (while rejected by potential-based semantics), with S emitted, but it is rejected by the basic semantics.

Note that the difference on non-causal executions has a consequence on the definition of non-deterministic statements. Consider, for example:

```
present S then emit S end
```

- It is non deterministic in Esterel, because, starting from the empty environment, one has a (non-causal) transition with S emitted and (a causal) one without it (one assumes S is local).
- It is deterministic in the basic semantics because there is a unique solution, with S absent.

However, in both cases, the statement is rejected by potential based semantics.

To sum up the relations between the behavioural and the basic semantics:

- a program rejected by the behavioural semantics is also rejected by the basic semantics, but the converse is false (`present S then nothing end; emit S`).
- there exists programs which are nondeterministic with the behavioural semantics, but are deterministic with the basic semantics (`present S then emit S end`).
- there exists programs which are deterministic for the behavioural semantics, but are rejected by the basic semantics (`present S then nothing end; emit S`).

5.2 The v3 Compiler

The Esterel v3 compiler[BG] uses a potential function which does not take in account the signal environment.

There are actually two distinct Esterel v3 semantics. The first one, which is implemented as the default in the Esterel v3 compiler, is more restrictive than the v3 semantics presented previously. The second one, which is available with the *oldcausality* flag, corresponds to the one presented here.

The default Esterel v3 semantics allows one to take the `then` branch of a `present` S statement only when no potential emission of S remain. Thus, signal presence is treated exactly in the same way as absence (this symmetry is the basic reason why this semantics was introduced, while more restrictive than the *oldcausality* one). Here is an example of a statement which is rejected by the default Esterel v3 semantics:

```
emit S; present S else emit S end
```

The emission of S in the `else` branch of the `present` statement is considered as possible, although S is already emitted.

On the contrary, the *oldcausality* v3 compiler accepts the previous statement, as it can execute the `then` branch of `present` S as soon as S is emitted.

The Esterel v3 compiler translates programs in finite states machines (*automata*) in both versions.

5.3 The v4 Compiler

The v4 Esterel compiler is based on the static cycle detection. This approach, which is the most restrictive, has been introduced to overcome the problem of automata states explosion in the Esterel v3 compiler.

In the Esterel v4 compiler, a program is translated into a system of equations the size of which is linear in the size of the program code. A program is accepted by the compiler if the associated system is cycle free, and thus can be sorted.

The Esterel v4 compiler starts by building the dependency graph of signals emissions and tests, and it rejects the program is there is a cycle in it. This method is even more restrictive than the v4 semantics described in this paper. For example, the Esterel v4 compiler rejects the following program accepted by the previous v4 semantics:

```
emit S; present S else emit S end
```

The Esterel v4 compiler has been felt as too restrictive and is presently replaced by the v5 compiler.

5.4 The v5 Compiler

The Esterel v5 compiler is based on the constructive approach[Be]; two functions are defined: *can* which computes what signals are possibly emitted by a statement, and *must* which computes what signals are necessarily emitted. The two functions can only proceed by constructive steps, “deducing facts from facts”.

The Esterel v5 semantics of a statement t in an environment E is defined in two steps:

1. First, the environment produced E’ is incrementally computed (as a fix-point), using *can* and *must*.
2. Second, if all signals are determined (the statement is *constructive* and accepted by the semantics), then the new statement t’ is computed using the behavioural semantics, starting from E’ (assuming that there is no instantaneous loop in t).

The function *can* is actually equivalent to Π_{v5}. The function *must* is very close to M_{v7} and differs from it in the way unknown signals are processed. In the case of an unknown signal, *must* returns the empty set, instead of the intersection of the two branches returned by M_{v7}. As a consequence, the following program is, for example, rejected by the Esterel constructive semantics, while accepted by v7:

```
present S1 then
   present S2 then emit S3 else emit S3 end;
   present S3 else emit S1 end
end
```

6 The SugarCubes Framework

The two main notions of SugarCubes are the one of *reactive instruction* whose semantics refer to instants, and the one of *reactive machine* whose purpose is to execute reactive instructions in an environment made of instantaneously broadcast events.

6.1 Instructions

The `Instruction` class implements reactive instructions. A reactive instruction is activated by a call to its method `activ` which returns as result one of the three following values:

- TERM (for *terminated*) means that the instruction is completely terminated; nothing remains to do for the current instant and also for future ones. Thus, to activate one more time an instruction returning TERM has no effect and also returns TERM.
- STOP (for *stopped*) means that execution of the instruction is over for current instant, but that some code remains to be executed at next instant.
- SUSP (for *suspended*) means that execution of the instruction has not reached a stable state and must be resumed during current instant. This is for example the case for the instruction that waits for a not yet generated event (see below): execution is suspended to let the others components the possibility to generate the event during current instant.

The basic reactive instructions of SugarCubes are:

- `Stop`, which stops execution for the current instant;
- `Seq` to put one reactive instruction in sequence with another one;
- `Merge` to put two reactive instructions in parallel;
- atoms to execute basic Java statements such as printing messages;
- `Loop` and `Repeat`, for cyclic executions;
- `Generate` to generate an event, `Await` to wait for it, and `When` to test for an event.

The correspondence between SugarCubes and the syntax previously introduced is as follows:

- SugarCubes events are analogous to Esterel signals;
- `Stop` corresponds to `pause`;

- `Merge` corresponds to the parallel statement;
- `Seq` corresponds to the sequence;
- `Generate` corresponds to `emit`.

However, `When` does not correspond to `present`, as immediate reaction to absence is forbidden in SugarCubes. Actually, the rules for `When` are the following:

$$\frac{S \in E \quad t,E \xrightarrow{\alpha} t',E'}{\text{when } S \text{ then } t \text{ else } u \text{ end},E \xrightarrow{\alpha} t',E'}$$

$$\frac{S \notin E \quad \mathbb{S} \notin E}{\text{when } S \text{ then } t \text{ else } u \text{ end},E \xrightarrow{\text{SUSP}} \text{when } S \text{ then } t \text{ else } u \text{ end},E}$$

$$\frac{\mathbb{S} \in E}{\text{when } S \text{ then } t \text{ else } u \text{ end},E \xrightarrow{\text{STOP}} u,E}$$

The only difference with `present` is the last rule, which forbids immediate reaction to the signal absence.

Now, one briefly describes reactive machines and the `Merge` and `When` instructions.

6.2 Machines

The class `EventMachine` implements reactive machines. A reactive machine executes a program which is a reactive instruction. It has two main tasks to perform: first, to decide the end of instants, and second, to broadcast events. Initially, the program is the `Nothing` instruction which does nothing and terminates instantaneously. New instructions are dynamically added to the program (by calling the machine method `add`) and executed in parallel with the previous ones.

Basically, a reactive machine detects the end of the current instant, when all parallel instructions of the program are terminated or stopped. The behaviour is as follows:

- The program is cyclically activated while there are suspended instructions in it (while activation returns SUSP).
- The end of the current instant is effective when all the parallel instructions in the program are terminated or stopped (no suspended instruction remains).
- At the end of each program activation, the machine tests if some new events were generated during this execution. If it was not the case, then there is no hope that future program activations will change the situation. Then, a flag is set to let suspended instructions stop, knowing from that point that events which are not emitted are actually absent.

Two fields `move` and `endOfInstant` are used to implement this behaviour. Field `move` is set to true to indicate that a new event is generated (`Generate` statement); in this case, the end of the current instant is postponed to allow the suspended receivers awaiting the event (`Await` instruction) or testing it (`When` instruction) to resume. Field `endOfInstant` is set to true when the end of the current instant is decided by the machine, to let suspended receivers know that awaited or tested events are absent.

Method `activ` of `EventMachine` implements this behaviour; the code is the following:

```
endOfInstant = move = false;
while (program.activ(this) == SUSP){
   if (move) move = false; else endOfInstant = true;
}
newInstant();
```

Parameter `this` is the machine, which is the execution context of the program.

6.3 Merge Instruction

The `Merge` instruction actually implements the rules for the parallel operator introduced in section 3. Statuses of branches are coded in two fields `leftStatus` and `rightStatus` whose initial values are SUSP. The code of method *activ* is:

```
if (leftStatus == SUSP) leftStatus = left.activ(machine);
if (rightStatus == SUSP) rightStatus = right.activ(machine);
if (leftStatus == TERM && rightStatus == TERM){ return TERM; }
if (leftStatus == SUSP || rightStatus == SUSP){ return SUSP; }
leftStatus = rightStatus = SUSP;
return STOP;
```

6.4 Reaction to Absence

The `When` instruction behaves as the `present` statement except that absence is only decided at the end of the the current instant. The `activ` method of `When` is (`left` is the `then` branch and `right` is the `else` one):

```
if (!testEvaluated){
   if (event.presence(machine) == UNKNOWN) return SUSP;
   value = event.isPresent(machine);
   testEvaluated = true;
   if(machine.isEndOfInstant()) return STOP;
}
return value ? left.activ(machine) : right.activ(machine);
```

The return code is STOP if the test cannot be evaluated before the end of the instant; this is for example the case when one tests for an absent event. As a consequence, reaction to absence is always postponed to the next instant.

7 Extensions to SugarCubes

Now, one extends SugarCubes to deal with immediate reaction to absence. Basically, one gives instructions a new method `explore` which implements potential events computing. This method adds potentially generated events to a vector, and returns a boolean which indicates if control can go in sequence, or not. So, `explore` actually implements both Π and τ functions of the semantics.

In potential based approaches, the natural moments for absence decisions are blocked situations, when there is no other way to proceed. That is, absence decisions are delayed as far as possible.

The proposed implementation exactly mimics the semantics of parallel and `present`. Execution of a parallel branch is suspended when one tries to test a signal which is neither emitted nor absent. When all parallel branches are either terminated or suspended, potential analysis can start to decide if some signals can be made absent. An incoherent program is detected if all signals on which execution is blocked are potentially emitted. Otherwise, signals that are not potentially emitted are decided as absents and execution can proceed with this new information.

One first describes the new algorithm for machines, then the `Present` class which implements the `present` instruction; finally, one describes the `explore` method which implements potential functions.

7.1 IraMachine

The extended machine class is the class `IraMachine` (Ira stands for *Immediate Reaction to Absence*). Two new fields are introduced:

- `lockOn` is the vector of events on which execution is blocked because they are unknown;
- `possible` is the vector of potentially generated events.

Method `makeAllAbsent` sets to absent all events not in `possible`; it returns false if no event can be set to absent, and true otherwise.

The code for method `activ` is the following:

```
endOfInstant = move = false;
while (program.activ(this) == SUSP){
   if (move) move = false;
    else if (lockOn.isEmpty()) endOfInstant = true;
   else{
       body.explore(this);
       if (!makeAllAbsent()){
           System.out.println("causality error");
          break;
       }
   }
    if (!lockOn.isEmpty()) lockOn = new Vector();
```

```
        if (!possible.isEmpty()) possible = new Vector();
    }
    newInstant();
```

When the body is suspended while there is no new generated event (`move` is false) but while there exist events on which execution is blocked (`lockOn` is not empty), then potentially generated events are computed by calling the method `explore`.

A causality error is detected when there is no possibility to set any event to absent (all events on which execution is suspended are potentially generated). This corresponds in the semantics to situations where the program is suspended while no non-empty X_i can be added to the environment; in this case, no terminated rewriting exist.

7.2 Present Instruction

There are two differences between the `activ` method of `When` and the one of `Present`:

- when the event is unknown, it is added to the `lockOn` vector;
- when the event is absent, control immediately goes to the right (`else`) branch.

Method `activ` of `Present` is:

```
if (!testEvaluated){
  if (event.presence(machine) == UNKNOWN){
     if (!machine.lockOn.contains(name)) machine.lockOn.addElement(name);
     return SUSP;
  }
  value = event.isPresent(machine);
  testEvaluated = true;
}
return value ? left.activ(machine) : right.activ(machine);
```

7.3 Method explore for v5

One describes the method `explore` for the v5 semantics.

Generate

When explored, `Generate` adds the event to the vector `possible` of possibly generated events; it returns true indicating that the control goes in sequence.

```
machine.possible.addElement(eventName);
return true;
```

Merge

Both branches are explored and `explore` returns true if both branches return true (possible termination only if both branches possibly terminate).

```
boolean bl = (leftStatus == TERM), br = (rightStatus == TERM);
if (leftStatus == SUSP) bl = left.explore(context);
if (rightStatus == SUSP) br = right.explore(context);
return bl && br;
```

Note that the two branches are explored in the same context.

Seq

The left branch is first explored; if it terminates, then the right branch is also explored. Exploration returns true if both branches return true.

```
if (left.isTerminated()) return right.explore(context);
boolean b = left.explore(context);
if (b) return right.explore(context);
return false;
```

Present

If the event is present, only the `then` (`left`) branch is explored; if it is absent, only the `else` (`right`) branch is explored; if it is unknown, both branches are explored and the control can go in sequence as soon as one branch can. Thus, in this case the two sets of events possibly generated by the two branches are added to `possibly`.

```
if (testEvaluated)
   return value ? left.explore(context) : right.explore(context);
if (event.isPresent(machine)) return left.explore(machine);
if (event.isAbsent(machine)) return right.explore(machine);
boolean b1 = left.explore(machine);
boolean b2 = right.explore(machine);
return b1 || b2;
```

7.4 Method explore for v6

One describes now the method `explore` of `Present` for the v6 semantics:

```
if (testEvaluated)
   return value ? left.explore(context) : right.explore(context);
Event event = machine.getEvent(name);
if (event.isPresent(machine)) return left.explore(machine);
if (event.isAbsent(machine)) return right.explore(machine);

AccessToEvent access = machine.getAccessToEvent(name);
Event save = access.event();             // save the event
Event newEvent = (Event)save.clone();   // make a copy of the event

newEvent.generate(machine);              // make the presence hypothesis
access.setEvent(newEvent);
boolean b1 = left.explore(machine);     // explore the then branch

newEvent.makeAbsent(machine);            // make the absence hypothesis
access.setEvent(newEvent);
boolean b2 = right.explore(machine);    // explore the else branch

access.setEvent(save);                   // restore the event
return b1 || b2;
```

7.4 Method explore for v7

For v7, one needs to implement both Π_{v7} and M_{v7} functions with the `explore` method.

One introduces a new field in machines to determine which of the two functions is actually computed: Π_{v7} when `possibleFlag` is true and M_{v7} otherwise.

A new method `getNecessary` is added to `Seq`, `Present`, and `Generate` to compute M_{v7}. As it is standard code, one does not give it here.

8 Conclusion

In this paper, one adopts a non-standard point of view upon Esterel semantics. There are several benefits in doing so:

- One gets a "software-based" description, complementary to the standard "circuit-based" one of [Be].
- One gets a very simple description of the Esterel v5 semantics, in which only the *can* function appears. Signals that must be emitted need not to be explicitly computed.
- Semantics and implementation are very close; in particular, the SugarCubes implementation is straightforward.
- One gets a very clear characterisation of the Esterel v5 semantics: it is the one allowed to use the signal environment, *but not to extend it in any way*, when computing potentially emitted signals.

If the Esterel v5 semantics is, in some sense, the "end of the story", when one adopts the circuit point of view (it is shown in [Be] that the constructive semantics implemented in the Esterel v5 compiler exactly reflects the circuit semantics), this could not be the same, when adopting the software point of view. Experiments with SugarCubes can be a way to investigate this question.

Acknowledgments

Thanks to Robert de Simone for his comments on a first version of this paper.

Bibliography

[BG] G. Berry, G. Gonthier, *The Esterel Synchronous Language: Design, Semantics, Implementation,* Science of Computer Programming, **19**(2), 1992.

[Be] G. Berry, *The Constructive Semantics of Esterel*, 1995, available at URL `http://www.inria.fr/meije/esterel/Documentation`.

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

[BDS] F. Boussinot, R. De Simone, *The SL Synchronous Language*, IEEE Trans. Software Engineering, **22**(4), 1996.

[BS] F. Boussinot, J-F Susini, *The SugarCubes Tool Box - Definition*, INRIA Research Report 3247, available at URL `http://www.inria.fr/meije/rc/SugarCubes/`, 1997 (to appear in Software Practice & Experience).

[GJS] J. Gosling, B. Joy, G. Steele, *The Java Language Specification*, Addison-Wesley, 1996.

[HCRR] N. Halbwachs, P. Caspi, P. Raymond, Ch. Ratel, *The Synchronous Dataflow Programming Language Lustre*, Proc. IEEE, **79**(9), 1991.

[Ha] N. Halbwachs, *Synchronous Programming of Reactive System,* Kluwer Academic Pub., 1993.

[HP] D. Harel, A. Pnueli, *On the Development of Reactive Systems*, NATO ASI Series F, Vol. 13, Springer-Verlag, 1985.

[Har] D. Harel, *StateCharts: A Visual Approach to Complex Systems*, Science of Computer Programming, **8**(3), 1987.

[LBBG] P. Leguernic, A. Benveniste, P. Bournai, T. Gautier, *SIGNAL: A Dataflow Oriented Language for Signal Processing*, IEEE-ASSP, **34**(2), 1986.

[Pl] G. Plotkin, *A Structural Approach to Operational Semantics*, Report DAIMI FN-19, Aarhus University, 1981.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399