Loops in Esterel

Olivier Tardieu and Robert de Simone
INRIA, Sophia Antipolis, France

ESTEREL is a synchronous design language for the specification of reactive systems. Thanks to its compact formal semantics, code generation for ESTEREL is essentially provably correct. In practice, due to the many intricacies of an optimizing compiler, an actual proof would be in order. To begin with, we need a *precise* description of an *efficient* translation scheme, into some lower-level formalism. We tackle this issue on a specific part of the compilation process: the translation of loop constructs. First, because of *instantaneous* loops, programs may generate runtime errors, which cannot be tolerated for embedded systems, and have to be predicted and prevented at compile time. Second, because of *schizophrenia*, loops must be partly unfolded, making C code generation as well as logic synthesis non-linear in general. Clever expansion strategies are required to minimize the unfolding. We first characterize these two difficulties w.r.t. the formal semantics of ESTEREL. We then derive very efficient, *correct-by-construction* algorithms to verify and transform loops at compile time, using static analysis and program rewriting techniques. With this aim in view, we extend the language with a new `gotopause` construct, which we use to encode loops. It behaves as a non-instantaneous jump instruction compatible with concurrency.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: synchronous languages, static analysis, code generation

1. INTRODUCTION

The design of modern complex embedded systems borrows from a number of engineering disciplines. Software, hardware, system, network, sometimes mechanical or scientific engineering for physical modeling, are customarily involved in the full system development. Due to the natural heterogeneity of components in the projected system, a first prototype software model is generally built, that serves for analysis, extensive simulation, and verification of many features beforehand.

Synchronous reactive formalisms [Benveniste et al. 2003] are becoming more and more attractive for the specification of reactive systems [Halbwachs 1992; Edwards 2000] as, in contrast with unconstrained "simulation" semantics of usual software models, they allow formal verification as exhaustive testing, as a result of their precise mathematical semantics.

Many synchronous formalisms and tools are available today, from industry and academia. There are imperative synchronous languages like ESTEREL or QUARTZ

email: olivier.tardieu@mines.org, robert.desimone@sophia.inria.fr.

[Schneider 2000], data-flow languages such as LUSTRE/SCADE [Halbwachs et al. 1991], and graphical description languages, for instance ARGOS [Maraninchi 1991] and SYNCCHARTS [André 1996a; 1996b]. Input languages to hardware model-checkers such as SMV [McMillan 1993], COSPAN/FORMALCHECK [Hardin et al. 1996], or VIS [Brayton et al. 1996] are mostly of synchronous nature. We shall focus throughout this paper on ESTEREL.

High-level synthesis and code generation tools provide implementation means for designs specified in these languages. Hopefully faithful to the semantics, implementation steps should provide a correct-by-construction design flow, thereby avoiding the all too often occurring problem of adequacy between simulation and synthesis semantics found in neighboring formalisms such as HDLs (VHDL, Verilog,...).

The primary concern of the current paper is to contribute to this "correct-by-construction" approach to code generation/synthesis, in our case by providing formal ground for some of the key transformation steps involved in the implementation of ESTEREL programs. While apparently utterly theoretical in nature, we feel this kind of work can greatly help gain confidence in the soundness and semantic preservation of the compiling process, with potential incidence on future certification procedures for highly safety-critical embedded domains where compiler correctness may have to be asserted at semantic level.

ESTEREL

ESTEREL [Berry and Gonthier 1992; Berry 2000a; 2000b; Boussinot and de Simone 1991] is an imperative synchronous parallel language dedicated to the programming control-oriented systems. Sophisticated controllers may be described using sequential and parallel compositions of behaviors, suspension and preemption mechanisms, cases, loops, and synchronizations by mean of instantly broadcast signals. Both synthesis and efficient simulation (via C code generation) are supported.

ESTEREL enjoys a full-fledged formal semantics under the form of Structural Operational Semantic rules [Plotkin 1981]. Many issues on transformational steps involved in the compilation process can thus be handled mathematically on these rules. In the current paper we shall consider a precise exercise of this nature. While specific, it seems to us of utter importance as it leads to a smooth transition from a definitional level, that we might call *dynamic*, in which successive execution steps create new program residuals to be further executed, to a lower *static* level, where memory resources for both data and control are assigned at compile time, in a way favored by embedded targets or hardware circuit implementations.

As ESTEREL disallows general recursive definitions and dynamic process creation, this static interpretation is feasible. The main problem which remains to be solved in our case goes with the treatment of "loop" iterative constructs.

Loops

In synchronous formalisms execution is divided into (a discrete sequence of) behavioral reactions. While following the natural flow of control in a given reaction, loop constructs may have to be unrolled (in the dynamic version of the semantics). The following problem now comes as obvious: if a loop can be unrolled infinitely many times inside a given reaction, without execution halting to pause until the next reaction, then one may never reach completion of the behavior for the current

instant, ending in some kind of divergence. This phenomenon, called *instantaneous loop* in ESTEREL literature, has to be discarded as incorrect.

More subtly perhaps, without infinite unrolling, the body of a loop may contain elements whose multiple occurrences are executed simultaneously, which is referred to as *schizophrenia* in ESTEREL literature. These elements need to be duplicated appropriately at compile time. This is reminiscent of *Single Static Assignment* forms in classical compiler techniques. Here also the amount of code duplication should be minimized, while ensuring the desired property.

While current implementations already provide practical solutions for these two problems, they are not satisfactory in our view. For instance (first problem), compilers reject varying sets of correct programs, with no clear boundaries, i.e. arbitrarily as far as the user is concerned, because they rely on ad hoc loop analyses, that still need to be better understood. We shall see an example of such a program in Section 3.

Outline of the Paper

The paper starts with an introductory presentation of a kernel ESTEREL language in Section 2.

We tackle the first problem of incorrect loops [Tardieu and de Simone 2003] by extracting the relevant information from the semantic rules using abstract interpretation [Cousot and Cousot 1977]. This is the topic of Section 3, in which we provide a criterion, general enough, that ensures non-instantaneous divergence of loop structures. Importantly this criterion is correct by construction, as it is formally derived from the semantic rules themselves.

We describe and formalize the schizophrenia problem in Section 4. It has been remarked before [Mignard 1994] that schizophrenia can be solved by some amount of loop unrolling. A simple and naive solution would then be to unroll each loop construct once. But, because of loop nesting, this can produce an exponentially larger program. So one has to look for more selective expansions. This requires again clever static analysis, to identify as precisely as possible the potentially problematic locations. Here again our goal will be to specify this analysis step in a formal, provably correct fashion.

Then, in Section 5, in order to perform these code expansions as compactly as possible, we introduce a new `gotopause` construct in ESTEREL [Tardieu 2004a], so that several behaviors can be connected to the same halting point. Technically, we first rewrite the rules of the semantics using active halting points instead of program residuals, where active halting points act as *pc* program counters in a parallel setting. Defining `gotopause` is then straightforward.

We provide our solution [Tardieu and de Simone 2004] to the schizophrenia problem by program rewriting for our kernel language in Section 6. We start from the basic exponential expansion, called *reincarnation* in ESTEREL literature. We then describe a quadratic rewriting using `gotopause`. We improve this algorithm by selectively transforming the problematic statements located by our static analysis, thus achieving a quasi-linear rewriting in practice.

We extend our transformation to full ESTEREL in Section 7, and discuss our implementation in Section 8. It provides an experimental validation of our approach.

p, q	::=	`[`p`]`	
		`nothing`	does nothing and terminates instantly
		`pause`	suspends the execution for one instant
		p `;` q	executes p followed by q if/when p terminates
		p `||` q	executes p in parallel with q
		`loop` p `end`	repeats p forever
		`signal` S `in` p `end`	declares signal S in p
		`emit` S	emits signal S
		`present` S `then` p `else` q `end`	executes p if S is present, q otherwise
		`trap` T `in` p `end`	declares and catches exception T in p
		`exit` T_d	raises exception T of depth $d \in \mathbb{N}$
S, T	::=	*identifier*	

Fig. 1. Pure ESTEREL

Note: We tried our best to keep the paper self-contained, including complex definitions and comprehensive proofs. At times this was not completely possible, due to space considerations. In particular, the introduction of a new language feature in Section 5 would call for an adaptation of the contents and the proofs of previously stated results, most of which remain similar (but not all). These, and more generally the full technical material including lengthy proofs and complete definitions can be found in [Tardieu 2004b].

2. THE PURE ESTEREL KERNEL LANGUAGE

ESTEREL was born in the eighties [Berry and Cosserat 1984], and evolved since then. In this work, we consider the ESTEREL v5 dialect [Berry 1999; 2000a] endorsed by current academic compilers [INRIA et al. 2000; Edwards et al. 2004].

Pure ESTEREL is the fragment of the full ESTEREL language where data variables and data-handling primitives are abstracted away. Data abstraction makes loop-related issues decidable and is a first step toward efficient (conservative) heuristics to deal with them. We shall first concentrate on pure ESTEREL, then return to full ESTEREL in Section 7.

2.1 Syntax and Intuitive Semantics

Without loss of generality, we focus on a kernel language, which retains just enough of the pure ESTEREL language to attain its full expressive power. Figure 1 describes the grammar of our kernel language, as well as the intuitive behavior of its constructs. The non-terminals p and q denote *statements*, S *signals* and T *exceptions*.

The infix "`;`" operator binds tighter than "`||`". Brackets "`[`" and "`]`" may be used to group statements in arbitrary ways. In a `present` statement, `then` or `else` branches may be omitted. For example, "`present` S `then` p `end`" is a shortcut for "`present` S `then` p `else nothing end`".

In the sequel, the words *statement* and *program* are synonymous.

Instants and Reactions

An ESTEREL statement runs in steps called *reactions* in response to the *ticks* of a (possibly virtual) *global clock*. Each reaction takes one *instant*. Except for the `pause` instruction, primitive statements execute in zero-time, which is an ideal view of the fact that reactions are supposed to converge and terminate before the next clock tick occurs, so that no overlap is possible. Of course, just as gates in digital

circuits have delays, instruction in ESTEREL take physical time to execute. But at the formal semantics level, only "logical" time matters, that is to say instructions behave as if instantaneous[1].

When the clock ticks, a reaction occurs. It may either finish the execution instantly or delay part of it till the next instant, because it reached at least one `pause` instruction. In the latter case, the execution is resumed when the clock ticks again, and so on.

The statement "`emit A; pause; emit B; emit C; pause; emit D`" emits the signal `A` in the first instant of its execution, then emits `B` and `C` in the second instant, finally emits `D` and terminates in the third instant. It takes three instants to complete, i.e. proceeds by three reactions.

Synchronous Concurrency

Concurrency in ESTEREL is synchronous and deterministic. One reaction of "p `||` q" is made of exactly one reaction of each branch, until the termination of all branches. The statement "`pause; emit A; pause; emit B || emit C; pause; emit D`" emits `C` in the first instant of its execution, then emits `A` and `D` in the second instant, then emits `B` and terminates in the third instant.

Exceptions

Exceptions are lexically scoped, declared and caught by the "`trap` T `in` p `end`" construct, and raised by the "`exit` T_d" instruction. The integer d encodes the *depth* of "`exit` T":

—if "`exit` T_d" is enclosed in a declaration of T then d *must* be the number of exception declarations that have to be traversed before reaching that of T;

—if "`exit` T_d" is not enclosed in a declaration of T then d *must* be greater or equal to the number of exception declarations enclosing this `exit` statement.

For instance,

```
trap T in
  trap U in
    exit T1     has depth 1 because of the declaration of U
  ||
    exit U0     has depth 0
  ||
    exit V3     could have any depth greater or equal to 2
  end;
  exit T0       has depth 0
end
```

Such a "De Bruijn" encoding of exceptions for ESTEREL was first advocated for in [Gonthier 1988]. In a complete program all exceptions are declared so that depths are unambiguously inferred from scoping information by a traversal of the program. But depths are required to state the semantics of terms in general (i.e. incomplete pieces of programs). In the sequel, we shall only make depths explicit when necessary.

[1] As a result, the main goal of an ESTEREL compiler is to ensure by a proper scheduling of instructions in each instant that the implemented behavior complies with the idealized semantics.

In sequential code, the `exit` statement behaves as a goto to the end of the matching `trap` block. For example,

```
trap T in
  emit A; pause; emit B; exit T; emit C
end;
emit D
```

emits `A` in the first instant, then `B` and `D` and terminates in the second instant. The statement "`emit C`" is never executed.

An exception occurring in a parallel context causes it to terminate instantly. In

```
trap T in
  emit A; pause; emit B; pause; emit C
||
  emit D; pause; exit T
end;
emit E
```

`A` and `D` are emitted in the first instant, then `B` and `E` in the second and final one. Again, "`emit C`" is never executed. Exceptions implement *weak preemption*: "`exit T`" in the second branch does not prevent `B` to be simultaneously emitted in the first one.

Exception declarations may be nested. In the following program, `A` is never emitted. The outermost exception `T` has priority over `U`. In general, the exception of greater depth ($\mathtt{T}_1$) has priority over the exception of lower depth ($\mathtt{U}_0$).

```
trap T in
  trap U in
    exit T1
  ||
    exit U0
  end;
  emit A
end
```

Loops

The statement "`loop emit S; pause end`" emits `S` at each instant and never terminates. Finitely iterated loops may be obtained by combining `loop`, `trap` and `exit` statements, as in the kernel expansions[2] of "`await` S" and "`await_not` S":

```
await S  =def=  trap T in
                  loop
                    pause;
                    present S then exit T end
                  end
                end
```

[2]Since the `pause` and `present` statements are sequentially executed in this order, the status of `S` in the first instant of execution of "`await` S" or "`await_not` S" is ignored. This is true as well of suspension and abortion instructions (defined below).

$$\texttt{await\_not S} \stackrel{def}{=} \begin{array}{l}\texttt{trap T in}\\ \quad\texttt{loop}\\ \quad\quad\texttt{pause;}\\ \quad\quad\texttt{present S else exit T end}\\ \quad\texttt{end}\\ \texttt{end}\end{array}$$

Loop bodies may not be *instantaneous*. For example "`loop emit S end`" is not a correct program. As explained before, such a pattern would prevent the reaction to reach completion. Therefore, loop bodies are required to raise an exception or retain the control for at least one instant, i.e. execute a `pause` or an `exit` statement.

Signals

The instruction "`signal` S `in` p `end`" declares the *local* signal S in p. The free signals of a statement are said to be *interface* signals for this statement. Full Esterel defines *modules* and distinguishes `input`, `output` and `inputoutput` signals. In this work however, we have no need for such definitions. Basically, all interface signals are "inputoutput" signals.

In an instant, the *status* of a signal S is either *present* or *absent*. If S is present then all "`present` S `then` p `else` q `end`" statements executed in this instant, execute their "`then` p" branch in this instant; if S is absent they all execute their "`else` q" branch. A local signal S is present iff it is explicitly emitted in this instant by some "`emit` S" statement, absent otherwise. Its status is not maintained for the next instant. Interface signals are present iff locally emitted or generated by the *environment*. For example,

—In "`signal S in emit S; pause; present S then emit O end end`", signal `S` is present in the first instant of execution only, thus `O` is not emitted by this statement, as `S` is absent at the time of the "`present S`" test.

—In "`present A then emit B end || emit A`", both `A` and `B` are emitted.

—In "`signal S in present I then emit S end end`", the status of `I` depends on the environment, and the status of `S` follows from that of `I`.

Suspension and Abortion

Our kernel instructions are those of [Berry 1999] except for its "`suspend` p `when` S" statement. It can nevertheless be encoded by substituting all `pause` instructions of p by "`await_not` S" instructions:

$$\texttt{suspend}\ p\ \texttt{when}\ S \stackrel{def}{=} p[\texttt{pause} \rightarrow \texttt{await\_not}\ S]$$

Thanks to `trap` and `suspend` statements, we can encode *weak* and *strong abortion* (described in [Berry 1999]):

$$\texttt{weak abort}\ p\ \texttt{when}\ S \stackrel{def}{=} \begin{array}{l}\texttt{trap}\ T\ \texttt{in}\\ \quad p\texttt{;}\\ \quad\texttt{exit}\ T\\ \texttt{||}\\ \quad\texttt{await}\ S\texttt{;}\\ \quad\texttt{exit}\ T\\ \texttt{end}\end{array} \quad \text{(where } T \text{ is a fresh exception name)}$$

$$\texttt{abort } p \texttt{ when } S \stackrel{def}{=} \begin{array}{l}\texttt{weak abort}\\ \quad \texttt{suspend } p \texttt{ when } S\\ \texttt{when } S\end{array}$$

2.2 Logical Behavioral Semantics

The *logical behavioral semantics* of ESTEREL [Gonthier 1988; Berry 1999] defines the reactions of a statement p via a labeled transition system:

$$p \xrightarrow[E]{E', k} p'$$

where:

—the set E is the set of *present signals*,

—the set E' is the set of *emitted signals*,

—the integer k is the *completion code of the reaction*,

—the statement p' is the *residual* of the reaction.

Each transition represents one possible reaction of p (i.e. one instant of execution).

Figure 2 expresses the logical behavioral semantics of ESTEREL as a set of facts and deduction rules in a structural operational style [Plotkin 1981]. We now comment on the use of the various rule ingredients, in their most interesting cases.

Signals

The sets E of *present signals* and E' of *emitted signals* are meant to encode the I/Os of the reaction. By construction, all reactions defined by this semantics are such that $E' \subset E$. The statement p reacts to *inputs* I with *outputs* O iff:

$$p \xrightarrow[I \cup O]{O, k} p'$$

By construction, the set E' is a subset of the set of interface signals of p. In addition, one can prove by structural induction on p that if S is not an interface signal of p then:

$$p \xrightarrow[E]{E', k} p' \Leftrightarrow p \xrightarrow[E \cup \{S\}]{E', k} p'$$

The *signal coherence law* (a signal is present iff emitted or generated by the environment for an interface signal) is enforced by the form of the following rules, from Figure 2:

(present+)	if S is present then execute p in "`present` S `then` p `else` q `end`".
(present−)	if S is absent then execute q in "`present` S `then` p `else` q `end`".
(emit)	if S is emitted then S is present.
(signal+)	if the local signal S is present then S is emitted.

As a consequence, the two rules for "`signal` S `in` p `end`" encode:

(signal+)	S is present in p and emitted by p.
(signal−)	S is absent in p and not emitted by p.

Remark that these rules are in general neither exclusive, nor complementary, without further requirements (see below, Section 2.3).

Completion Code and Residual

The completion code k and the residual p' encode the status of the execution:

—If $k = 1$ then this reaction does not complete the execution of p.
It has to be continued by the execution of p' in the next instant.

—If $k \neq 1$ then this reaction ends the execution of p, and the residual p' is `nothing`:
—$k = 0$ if the execution completes *normally*, that is to say without exception;
—$k = d + 2$ if an exception of depth d *escapes* from p.

In particular, the completion code of "`exit` T_d" is "$2 + d$". If p terminates with completion code k and q with completion code l then "p `||` q" terminates with code "$\max(k, l)$", hence the encoding of exceptions.

In order to compute the completion code "$\downarrow k$" of "`trap` T `in` p `end`" from the completion k of p, we define:

$$\downarrow k = \begin{cases} 0 & \text{if } k = 0 \text{ or } k = 2 \\ 1 & \text{if } k = 1 \\ k - 1 & \text{if } k > 2 \end{cases}$$

For example,

$$\texttt{trap T in trap U in exit T}_1 \texttt{ || exit V}_5 \texttt{ end end} \xrightarrow[\emptyset]{\emptyset, 3} \texttt{nothing}$$

In order to normalize the residual p' when k is not 1, we define:

$$\delta^k(p') = \begin{cases} \texttt{nothing} & \text{if } k \neq 1 \\ p' & \text{if } k = 1 \end{cases}$$

Execution

An *execution* of the statement p is a potentially infinite chain of reactions, such that all completion codes are equal to 1, except for the last one in the finite case:

—finite execution: $p \xrightarrow[I_1 \cup O_1]{O_1, 1} p_1 \xrightarrow[I_2 \cup O_2]{O_2, 1} \ldots \xrightarrow[I_n \cup O_n]{O_n, k} p_n$ with $k \neq 1$

—infinite execution: $p \xrightarrow[I_1 \cup O_1]{O_1, 1} p_1 \xrightarrow[I_2 \cup O_2]{O_2, 1} \ldots \xrightarrow[I_n \cup O_n]{O_n, 1} \ldots$

For example, the statement "`emit A; pause; emit B`" with input `I` emits `A` and does not terminate instantly ($k = 1$), with the residual "`nothing; emit B`" remaining to be executed. In the second and final instant of execution, `B` is emitted.

$$\texttt{emit A; pause; emit B} \xrightarrow[\texttt{\{I,A\}}]{\texttt{\{A\}}, 1} \texttt{nothing; emit B} \xrightarrow[\texttt{\{J,B\}}]{\texttt{\{B\}}, 0} \texttt{nothing}$$

We say that the statement q *derives from* the statement p, and note $p \twoheadrightarrow q$, if q may be generated in the course of the execution of p, that is to say iff there exits $n \geq 0$ and $(p_k, E_k, E'_k)_{0<k\leq n}$ such that:

$$p = p_0, q = p_n, \forall k \in I\!N : 0 < k \leq n \Rightarrow p_{k-1} \xrightarrow[E_k]{E'_k, 1} p_k$$

By definition, the $\twoheadrightarrow$ relation is reflexive: $\forall p : p \twoheadrightarrow p$.

$$\texttt{nothing} \xrightarrow[E]{\emptyset, 0} \texttt{nothing} \qquad \text{(nothing)}$$

$$\texttt{pause} \xrightarrow[E]{\emptyset, 1} \texttt{nothing} \qquad \text{(pause)}$$

$$\texttt{exit}\ T_d \xrightarrow[E]{\emptyset, d+2} \texttt{nothing} \qquad \text{(exit)}$$

$$\frac{S \in E}{\texttt{emit}\ S \xrightarrow[E]{\{S\}, 0} \texttt{nothing}} \qquad \text{(emit)}$$

$$\frac{p \xrightarrow[E]{E', k} p' \quad k \neq 0}{\texttt{loop}\ p\ \texttt{end} \xrightarrow[E]{E', k} \delta^k(p'\texttt{;}\ \texttt{loop}\ p\ \texttt{end})} \qquad \text{(loop)}$$

$$\frac{p \xrightarrow[E]{E', k} p' \quad q \xrightarrow[E]{F', l} q' \quad m = \max(k, l)}{p\ \texttt{||}\ q \xrightarrow[E]{E' \cup F', m} \delta^m(p'\ \texttt{||}\ q')} \qquad \text{(parallel)}$$

$$\frac{S \in E \quad p \xrightarrow[E]{E', k} p'}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \xrightarrow[E]{E', k} p'} \qquad \text{(present+)}$$

$$\frac{S \notin E \quad q \xrightarrow[E]{E', k} q'}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \xrightarrow[E]{E', k} q'} \qquad \text{(present−)}$$

$$\frac{p \xrightarrow[E]{E', k} p'}{\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end} \xrightarrow[E]{E', \lfloor k} \delta^k(\texttt{trap}\ T\ \texttt{in}\ p'\ \texttt{end})} \qquad \text{(trap)}$$

$$\frac{p \xrightarrow[E]{E', k} p' \quad k \neq 0}{p\texttt{;}\ q \xrightarrow[E]{E', k} \delta^k(p'\texttt{;}\ q)} \qquad \text{(sequence-p)}$$

$$\frac{p \xrightarrow[E]{E', 0} p' \quad q \xrightarrow[E]{F', k} q'}{p\texttt{;}\ q \xrightarrow[E]{E' \cup F', k} q'} \qquad \text{(sequence-q)}$$

$$\frac{p \xrightarrow[E \cup \{S\}]{E', k} p' \quad S \in E'}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \xrightarrow[E]{E' \setminus \{S\}, k} \delta^k(\texttt{signal}\ S\ \texttt{in}\ p'\ \texttt{end})} \qquad \text{(signal+)}$$

$$\frac{p \xrightarrow[E \setminus \{S\}]{E', k} p'}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \xrightarrow[E]{E', k} \delta^k(\texttt{signal}\ S\ \texttt{in}\ p'\ \texttt{end})} \qquad \text{(signal−)}$$

Fig. 2. Logical Behavioral Semantics

2.3 Logical Correctness

A statement p is said to be *logically correct* iff the logical behavioral semantics defines a unique reaction for it at any stage of any execution (that is to say after any number of reactions and for any sequence of inputs):

$$p \text{ is logically correct iff } \forall q, \forall I : p \twoheadrightarrow q \Rightarrow \left[\exists! O, \exists! k, \exists! q' : q \xrightarrow[I\cup O]{O,k} q'\right]$$

This is not always so. For example,

(1) "`loop nothing end`" is not logically correct as no reaction is defined.

(2) "`present S else emit S end`" is not logically correct since no reaction is defined in the absence of inputs. On the one hand (signal−), if we suppose `S` absent for the duration of the reaction then it is emitted, which contradicts the hypothesis. On the other hand (signal+), if we suppose `S` present then it has no emitter.

(3) "`present S then emit S end`" is not logically correct. The signal `S` may be both present or absent in the absence of inputs. For the empty set of inputs there exist two possible sets of outputs: $\emptyset$ and $\{\texttt{S}\}$:

$$\texttt{present S then emit S end} \xrightarrow[\emptyset\cup\emptyset]{\emptyset,0} \texttt{nothing}$$

$$\texttt{present S then emit S end} \xrightarrow[\emptyset\cup\{\texttt{S}\}]{\{\texttt{S}\},0} \texttt{nothing}$$

Logical correctness characterizes programs that have *reactive* (at least one possible behavior), and *deterministic* (at most one possible behavior) reactions. These are the minimal requirements that any ESTEREL compiler must enforce.

2.4 Causality

Logical correctness does not take into account *causality*. The following program, while being logically correct (`S` can only be present), is not *causal*, since the emission of `S` *depends* on a test on `S`:

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

The logical behavioral semantics of ESTEREL can be refined into various semantics formalizing causal dependencies, such as the *constructive semantics* [Berry 1999]. This semantics is based on a notion of constructive causality, and heavily relies on causality analysis techniques, that have been extensively studied in the context of digital circuit description at gate netlist level in order to allow descriptions that are safe but might contain static combinatorial dependency cycles between wires [Malik 1993; Shiple et al. 1996; Namjoshi and Kurshan 1999].

In ESTEREL constructive issues are typically dealt with in compilers after the loop related issues considered in the present paper. The formulation of constructiveness as semantic rules across statements is rather involved, requiring recursive application of auxiliary functions. Its match against the simpler and more direct formulation at circuit level is also a delicate matter. Altogether this strongly suggests to deal with loops independently from causality considerations, which we shall leave for "further work". As a consequence, we shall study loops in the single framework of the logical semantics.

3. INSTANTANEOUS LOOPS

The logical behavioral semantics provides a unique rule for `loop` statements:

$$\frac{p \xrightarrow[E]{E',k} p' \qquad k \neq 0}{\texttt{loop}\ p\ \texttt{end} \xrightarrow[E]{E',k} \delta^k(p'\texttt{; loop}\ p\ \texttt{end})} \qquad \text{(loop)}$$

As expected, the side condition $k \neq 0$ disallows loop bodies to terminate instantly without raising an exception, making programs such as "`loop nothing end`" logically incorrect. This requirement is of dynamic nature, as it is enforced in each reaction, and may be responsible for runtime errors at any instant. For example, "`await I; loop nothing end`" fails upon the reception of the signal `I`.

In the context of embedded systems however, runtime errors cannot be tolerated. As a consequence, ESTEREL compilers have to prevent them. Intuitively, they may only compile *safe* programs, that is to say programs for which the runtime check $k \neq 0$ can be safely ignored, because it cannot fail (under any circumstances). Reciprocally, *unsafe* programs must be rejected.

Compilers must replace a runtime check by a compile time analysis of programs. As already mentioned in the introduction, we believe that this issue is a typical example of a well known complexity of ESTEREL, already solved by existing implementations, but not yet fully understood. For example, depending on the compiler, the following program may be (correctly) compiled or rejected:

```
loop
  trap T in
    trap U in
      trap V in
        exit U || exit V
      end;
      exit T
    end;
    pause
  end
end
```

Intuitively, establishing the logical correctness of this program requires to take into account the relative priority of `U` over `V`, which shows that "`exit T`" is unreachable, which in turn implies that "`pause`" is executed in each instant [Tardieu and de Simone 2003]. Several implementations, such as SAXO-RT [Closse et al. 2002], do lack this power of analysis.

Our goal in this section is to specify such a conservative filtering of programs:

—in a formal, provably correct fashion,

—of reasonable computational complexity,

—while keeping the user's satisfaction in mind.

We shall formalize the algorithm implemented in the ESTEREL compiler of [INRIA et al. 2000], which, in our view, provides a reasonable trade off between these

last two points. In particular, it handles the above example. To the best of our knowledge, no available ESTEREL compiler achieves a more powerful analysis, that is to say accepts more programs than the algorithm we shall formalize.

3.1 (Non-)Instantaneous Loops and (Un)safe Programs

We define for a statement p:

(1) *p may be instantaneous*, in other words *p is potentially instantaneous* iff:

$$\exists E, \exists E', \exists p' : p \xrightarrow[E]{E', 0} p'$$

(2) *p cannot be instantaneous*, in other words *p is non-instantaneous* iff:

$$\forall E, \forall E', \forall k, \forall p' : p \xrightarrow[E]{E', k} p' \Rightarrow k \neq 0$$

(3) p is *safe* iff every loop body of p cannot be instantaneous, *unsafe* otherwise.

THEOREM 1. *If p is safe and q derives from p ($p \twoheadrightarrow q$) then q is safe.*

PROOF. If a reaction of the statement p produces the residual p' then the bodies of the loops of p' occur in p, by definition of the (loop) rule. □

Removing the runtime check $k \neq 0$ for loop bodies means replacing the (loop) rule of the logical behavioral semantics with the following (unsafe-loop) rule:

$$\frac{p \xrightarrow[E]{E', k} p'}{\texttt{loop}\ p\ \texttt{end} \xrightarrow[E]{E', k} \delta^k(p'\texttt{;}\ \texttt{loop}\ p\ \texttt{end})} \qquad \text{(unsafe-loop)}$$

THEOREM 2. *If p is safe then the reference and revised semantics define the same executions for p.*

PROOF. If p is safe then all subterms of p are safe. Structural induction makes sense here. A simple induction on p shows that $p \xrightarrow[E]{E', k} p'$ in the revised semantics iff $p \xrightarrow[E]{E', k} p'$ in the reference semantics. Hence, the two semantics define the same reactions for p, thus same executions, thanks to Theorem 1. □

As a consequence, rejecting unsafe programs, that is to say potentially instantaneous loop bodies, removes the need for runtime check of completion codes of loop bodies, thus the risk of corresponding runtime errors.

Unsafe programs may nevertheless be logically correct as illustrated by the program "`loop emit S; pause end || loop present S then pause end end`" in which the left loop enforces the correct execution of the right one, through the continuous emission of `S`. Similarly, "`trap T in exit T; loop nothing end end`" is unsafe but logically correct, since the potentially instantaneous loop is unreachable.

But these intricate patterns serve no purpose. Rejecting such programs is not an issue; it even enforces a good coding style. Indeed, the subterms of a safe program are all safe, whereas the subterms of a logically correct program may well be logically incorrect, as shown by the above examples. As a result, as far as instantaneous loops are concerned, safe programs may be arbitrarily (de)composed safely, whereas, in general, programs with non-instantaneous loops cannot be.

Therefore, the restriction to safe programs is a conservative approach which makes sense both from the compiler's and from the user's perspective.

3.2 Exact Analysis

Deciding whether a statement may be instantaneous requires to take into account all its possible executions, thus 2^n possible valuations of its n interface signals, which is unreasonable in practice. In fact, SAT (boolean satisfiability in propositional logic) can be expressed in term of instantaneous termination in ESTEREL, via a polynomial reduction, as shown by the example of Figure 3. Therefore, compilers rely on conservative static analysis techniques, such as the one formalized below.

$(A \vee \neg B \vee C) \wedge (\neg A \vee C \vee \neg D) \wedge (\neg B \vee \neg C \vee D)$ is satisfiable

$\Updownarrow$

```
present A else present B then present C else pause end end end;
present A then present C else present D then pause end end end;
present B then present C then present D else pause end end end
```

may be instantaneous

Fig. 3. Reducing SAT to Instantaneous Termination

$$\texttt{nothing} \hookrightarrow 0 \qquad \text{(nothing)}$$

$$\texttt{pause} \hookrightarrow 1 \qquad \text{(pause)}$$

$$\texttt{exit}\ T_d \hookrightarrow d+2 \qquad \text{(exit)}$$

$$\texttt{emit}\ S \hookrightarrow 0 \qquad \text{(emit)}$$

$$\frac{p \hookrightarrow k \quad k \neq 0}{\texttt{loop}\ p\ \texttt{end} \hookrightarrow k} \qquad \text{(loop)}$$

$$\frac{p \hookrightarrow k \quad q \hookrightarrow l}{p\ \texttt{||}\ q \hookrightarrow \max(k,l)} \qquad \text{(parallel)}$$

$$\frac{p \hookrightarrow k}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \hookrightarrow k} \qquad \text{(present+)}$$

$$\frac{q \hookrightarrow k}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \hookrightarrow k} \qquad \text{(present−)}$$

$$\frac{p \hookrightarrow k}{\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end} \hookrightarrow \downarrow k} \qquad \text{(trap)}$$

$$\frac{p \hookrightarrow k \quad k \neq 0}{p\texttt{;}\ q \hookrightarrow k} \qquad \text{(sequence-p)}$$

$$\frac{p \hookrightarrow 0 \quad q \hookrightarrow k}{p\texttt{;}\ q \hookrightarrow k} \qquad \text{(sequence-q)}$$

$$\frac{p \hookrightarrow k}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \hookrightarrow k} \qquad \text{(signal+)}$$

$$\frac{p \hookrightarrow k}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \hookrightarrow k} \qquad \text{(signal−)}$$

Fig. 4. Abstract Semantics

p	Γ_p
`nothing`	$\{0\}$
`pause`	$\{1\}$
`exit` T_d	$\{d+2\}$
`emit` S	$\{0\}$
`loop` p `end`	$\Gamma_p \backslash \{0\}$
p `||` q	$\{m \in I\!N \ s.t.\ \exists k \in \Gamma_p, \exists l \in \Gamma_q, m = \max(k,l)\}$ i.e. $\max(\Gamma_p, \Gamma_q)$
`present` S `then` p `else` q `end`	$\Gamma_p \cup \Gamma_q$
`trap` T `in` p `end`	$\{l \in I\!N \ s.t.\ \exists k \in \Gamma_p, l = \downarrow k\}$ i.e. $\downarrow \Gamma_p$
p; q	if $0 \in \Gamma_p$ then $(\Gamma_p \backslash \{0\}) \cup \Gamma_q$ else Γ_p
`signal` S `in` p `end`	Γ_p

Fig. 5. Potential Completion Codes of Reactions

3.3 Static Analysis

By making abstraction of signals (E and E') and residuals (p') in the logical behavioral semantics of Esterel we obtain the abstract semantics of Figure 4, where:

$$p \xrightarrow[E]{E',k} p' \text{ is abstracted into } p \xrightarrow[\cdot]{\cdot,k} \cdot \text{ which we note } p \hookrightarrow k$$

We define the set of the *potential completion codes of* p: $\Gamma_p = \{k \in I\!N \ s.t.\ p \hookrightarrow k\}$.

Theorem 3. *For any statement* p: $\left\{ k \in I\!N \ s.t.\ \exists E, \exists E', \exists p' : p \xrightarrow[E]{E',k} p' \right\} \subset \Gamma_p$.

Proof. For any completion code k: $\left[\exists E, \exists E', \exists p' : p \xrightarrow[E]{E',k} p' \right] \Rightarrow p \hookrightarrow k$. □

In particular, if $0 \notin \Gamma_b$ then b cannot be instantaneous. Thus, if $0 \notin \Gamma_b$ for each loop body b of a program p then p is safe.

3.4 Algorithm

Figure 5 derives a recursive algorithm for the computation of Γ from the abstract semantics of Figure 4, by collecting the completion codes obtained using all deduction rules that may apply to each Esterel construct. For example, `loop` p `end` $\hookrightarrow k$ iff $p \hookrightarrow k$ with $k \neq 0$, thus $\Gamma_{\texttt{loop } p \texttt{ end}} = \Gamma_p \backslash \{0\}$. Similarly, $k \in \Gamma_{p;\ q}$ iff:

—either $k \in \Gamma_p$ and $k \neq 0$ (sequence-p),
—or $0 \in \Gamma_p$ and $k \in \Gamma_q$ (sequence-q).

In order to ensure that a program p is safe, one has to compute Γ_b for every loop body b of p, and check that none of these sets Γ_b contains zero. This check can be easily embedded within the computation of Γ itself by replacing:

$$\Gamma_{\texttt{loop } p \texttt{ end}} = \Gamma_p \backslash \{0\}$$

with the following definition:

$$\Gamma_{\texttt{loop } p \texttt{ end}} = \text{if } 0 \in \Gamma_p \text{ then } \textit{error} \text{ else } \Gamma_p$$

We shall require the explicit computation of Γ_q in a sequence "p; q" even if $0 \notin \Gamma_p$, so that the computation of Γ_p unconditionally traverses all subterms q of p:

$$\Gamma_{p;\ q} = \text{compute } \Gamma_q; \text{ if } 0 \in \Gamma_p \text{ then } (\Gamma_p \backslash \{0\}) \cup \Gamma_q \text{ else } \Gamma_p$$

THEOREM 4. *If the computation of* Γ_p *completes without error then* p *is safe.*

PROOF. If p contains a loop of body q then "`loop` q `end`" is a subterm of p so that the computation of $\Gamma_{\texttt{loop } q \texttt{ end}}$ completes without error, thus $0 \notin \Gamma_q$, therefore q cannot be instantaneous. As a result, p is safe. □

This algorithm requires a unique exhaustive traversal of the statement. It is linear in the size of the statement provided that the number of levels of nested trap statements (thus the cost of set operations) remains bounded.

From now on, we shall only consider programs for which the computation of Γ completed without error, therefore programs with guaranteed non-instantaneous loop bodies.

4. SCHIZOPHRENIA

As already mentioned in the introduction, it is highly desirable for embedded code targets (either software or hardware) to rely on statically allocated memory space, and even further *Single Static Assignment* properties. As soon as every element of the description assumes only one value at each instant, important synthesis, optimization and mapping techniques become available.

The pure ESTEREL semantics "almost" provides this property. In a loop-free program, each statement is executed at most once. More precisely, the execution of a statement q of the loop-free program p is started or restarted at most once per instant. For example, in "`pause; signal S in pause; pause end; pause`", the execution of the statement "`signal S in ... end`" is started in the second instant of execution of p, restarted in the third and fourth instants.

In particular, local signals have to be allocated or deallocated at most once per instant. In fact, they may be allocated globally, once for all, that is to say statically allocated, provided that statuses are reset between instants. And so on, for all "components" of the program.

Intuitively, every reaction in the execution of a loop-free program p can be completely described with a set of Boolean variables, the size of which is linear in the size of p, and can be computed using at most one assignment to each of these variables per instant. Indeed, the synthesis of a *Digital Sequential Circuit* from a loop-free pure ESTEREL program can be achieved in linear time and space, for example using the so-called "naive" translation scheme of Berry (first translation described in [Berry 1999]) restricted to loop-free programs.

But going from the "dynamic" semantics of pure ESTEREL to such a "static" model of computation is not so easy, once we add (safe) loops to the picture. For instance, the naive translation scheme of Berry is incorrect in general; and the fixed version of the translation (again in [Berry 1999]) is quadratic instead of linear.

The `loop` construct is the only construct that allows to go *back*. In particular, because of loops, and only them, the residual of a reaction defined by the logical behavioral semantics, may be larger than the initial program. Of course, this is not specific to the ESTEREL language. But, as a new iteration of a loop starts in the very instant the previous one finishes, that is to say simultaneously (even if in sequence), loops are responsible for complex behaviors.

This is called *schizophrenia* in ESTEREL literature, a notion which can be grasped through the following examples:

(1) First, statements can be executed several times in a *single* reaction, even if loops are required to be non-instantaneous. Consider the program

```
loop
  present I then pause end; emit O
||
  pause
end
```

in which signal `I` is present in the first instant, and absent in the next. In the second reaction the "`emit O`" instruction is executed first because control had reached the previous `pause` statement in the first instant; then the whole parallel construct terminates instantly, and thus is immediately reentered; this time, as `I` is absent, the "`emit O`" instruction is executed again, so twice in the same reaction.
While this may seem harmless for a statement such as "`emit O`", if we replace it with a "data" statement, for instance "`V:=V+1`", the fact that we have to pay attention to such program patterns becomes obvious.

(2) Second, distinct *instances* of a local signal may coexist in a *single* reaction, if a loop body containing its scope is exited and instantly reentered, as in:

```
loop
  signal S in
    present S then emit O end;
    pause;
    emit S
  end
end
```

From the second instant onward, an old occurrence of `S` (corresponding to the completed iteration of the loop), coexists with a new one (created by the iteration started in the current instant).

Single Static Assignment

Obviously, an encoding of such statements (respectively signals), complying with the *Single Static Assignment* per reaction principle, must be large, that is to say of size at least proportional to the number of possible simultaneous executions (respectively instances).

In general, if a statement is enclosed in n nested loops, it may be executed up to $n + 1$ times [Berry 1999] in an instant. A program of size $O(n)$ (i.e. $O(n)$ loops and $O(n)$ "`emit O`" statements) may produce up to $O(n^2)$ simultaneous "`emit O`", hence the quadratic worst-case complexity of circuit synthesis.

Memory Allocation

The compilation of Esterel loops into less "static" frameworks, as in C code generation for example, may seem easier. In the last example, it is possible to statically allocate a unique instance of `S`, provided that its status is reset each time the scope of `S` is entered, in addition to the usual reset between instants. Thus, in this case, the memory footprint remains linear in the number of local signals.

Let's now consider the program[3]:

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

In this example, while `T` is declared in sequence with `S`, the status of `S` cannot be decided without knowing the status of `A`, thus without deciding the status of `T`. By folding this code using a `loop` statement, we can merge the roles of `S` and `T` and obtain a (different) program where, similarly, the status of the old incarnation of `S` depends on the status of its new incarnation, rather than the other way around:

```
signal A in
  loop
    signal S in
      present S then emit A end;
      pause;
      present A then emit S end
    end
  end
end
```

As a consequence, the memory allocation scheme we described is not correct in general, and simultaneous instances of the same signal cannot share a single statically allocated memory cell. In summary, a single local declaration may require the simultaneous allocation of several memory cells.

Schizophrenic Programs

Berry claims that ESTEREL programs with loops can be linearly translated into *Digital Sequential Circuits* provided that no parallel statement (Example 1) and no local signal declaration (Example 2) is ever left and instantly reentered [Berry 1999]. Therefore, we shall focus on programs with potentially instantly reentered signal declarations and parallel statements[4]. We call them *schizophrenic programs*. We shall formalize these definitions below.

In all cases, schizophrenia can be suppressed by single recursive unrolling of loops (called *reincarnation* in Esterel literature, cf. Section 6). But this has an exponential price in presence of nested loops. So we want to be more perceptive in the code duplication involved. But then the correctness of the method used becomes an issue. This means we need a formal description of schizophrenia, in order to later prove the adequacy of our transformations (i.e. removal of schizophrenia).

[3]The several examples of this section are constructive, that is to say not only logically correct but also correct w.r.t. the constructive semantics of ESTEREL [Berry 1999]. Therefore, the issues discussed here remain the same for constructive programs.

[4]In [Tardieu 2004b], Chapter 5, we further illustrate the intricacies of such program patterns.

$$\texttt{nothing}^n \xrightarrow[E]{\emptyset, 0, \{n\}} \texttt{nothing} \qquad \text{(nothing)}$$

$$\texttt{pause}^n \xrightarrow[E]{\emptyset, 1, \{n\}} \texttt{nothing}^n \qquad \text{(pause)}$$

$$\texttt{exit}^n\ T_d \xrightarrow[E]{\emptyset, d{+}2, \{n\}} \texttt{nothing} \qquad \text{(exit)}$$

$$\frac{S \in E}{\texttt{emit}^n\ S \xrightarrow[E]{\{S\}, 0, \{n\}} \texttt{nothing}} \qquad \text{(emit)}$$

$$\frac{p \xrightarrow[E]{E', k, M} p' \quad k \neq 0}{\texttt{loop}^n\ p\ \texttt{end} \xrightarrow[E]{E', k, M \uplus \{n\}} \delta^k(p'\texttt{; loop}^n\ p\ \texttt{end})} \qquad \text{(loop)}$$

$$\frac{p \xrightarrow[E]{E', k, M} p' \quad q \xrightarrow[E]{F', l, N} q' \quad m = \max(k, l)}{p\ \texttt{||}^n\ q \xrightarrow[E]{E' \cup F', m, M \uplus N \uplus \{n\}} \delta^m(p'\ \texttt{||}^n\ q')} \qquad \text{(parallel)}$$

$$\frac{S \in E \quad p \xrightarrow[E]{E', k, M} p'}{\texttt{present}^n\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \xrightarrow[E]{E', k, M \uplus \{n\}} p'} \qquad \text{(present+)}$$

$$\frac{S \notin E \quad q \xrightarrow[E]{E', k, M} q'}{\texttt{present}^n\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \xrightarrow[E]{E', k, M \uplus \{n\}} q'} \qquad \text{(present−)}$$

$$\frac{p \xrightarrow[E]{E', k, M} p'}{\texttt{trap}^n\ T\ \texttt{in}\ p\ \texttt{end} \xrightarrow[E]{E', \lfloor k, M \uplus \{n\}} \delta^k(\texttt{trap}^n\ T\ \texttt{in}\ p'\ \texttt{end})} \qquad \text{(trap)}$$

$$\frac{p \xrightarrow[E]{E', k, M} p' \quad k \neq 0}{p\texttt{;}^n\ q \xrightarrow[E]{E', k, M \uplus \{n\}} \delta^k(p'\texttt{;}^n\ q)} \qquad \text{(sequence-p)}$$

$$\frac{p \xrightarrow[E]{E', 0, M} p' \quad q \xrightarrow[E]{F', k, N} q'}{p\texttt{;}^n\ q \xrightarrow[E]{E' \cup F', k, M \uplus N \uplus \{n\}} q'} \qquad \text{(sequence-q)}$$

$$\frac{p \xrightarrow[E \cup \{S\}]{E', k, M} p' \quad S \in E'}{\texttt{signal}^n\ S\ \texttt{in}\ p\ \texttt{end} \xrightarrow[E]{E' \setminus \{S\}, k, M \uplus \{n\}} \delta^k(\texttt{signal}^n\ S\ \texttt{in}\ p'\ \texttt{end})} \qquad \text{(signal+)}$$

$$\frac{p \xrightarrow[E \setminus \{S\}]{E', k, M} p'}{\texttt{signal}^n\ S\ \texttt{in}\ p\ \texttt{end} \xrightarrow[E]{E', k, M \uplus \{n\}} \delta^k(\texttt{signal}^n\ S\ \texttt{in}\ p'\ \texttt{end})} \qquad \text{(signal−)}$$

Fig. 6. Rules with labels

4.1 Formal Characterization

In this section, in order to formally characterize instantly reentered statements and schizophrenic programs, we label all pure ESTEREL constructs with integers:

—$\texttt{signal}^{label}$ ... `in` ... `end`
—... `||`^{label} ...
—etc. for all constructs of pure ESTEREL

Then, in Figure 6, we instrument ESTEREL logical behavioral semantics as follows:

(1) We add an extra component M to the transitions of the logical behavioral semantics of Figure 2 (top right-most position): the *multiset of labels* of a reaction, obtained by collecting the labels of the statements executed during the reaction. For example, if p reacts with the multiset M and q with N then "p `||`^{n} q" produces the multiset $M \uplus N \uplus \{n\}$.
(2) We preserve labels in the rewriting which produces the residual of the reaction, if the execution has to be continued in the next instant (k=1).

Loop unrolling may replicate labels as well as statements. For example,

$$\texttt{loop}^1\ \texttt{pause}^2\ \texttt{end} \xrightarrow[\emptyset]{\emptyset,\,1,\,\{1,2\}} \texttt{nothing}^2\texttt{;}\ \texttt{loop}^1\ \texttt{pause}^2\ \texttt{end} \xrightarrow[\emptyset]{\emptyset,\,1,\,\{1,2,2\}} \ldots$$

Intuitively, if a label of a program initially labeled with pairwise distinct labels is encountered twice in the same instant of execution, this means that corresponding statement of the initial program is left and instantly reentered in this instant, because of some loop, whose unrolling replicated the statement in a previous reaction. In the above example, the initial `pause` statement of label 2 is left and instantly reentered in the second instant of execution, as both the residual $\texttt{nothing}^2$ and the statement $\texttt{pause}^2$ are executed during this instant.

Therefore, we define for a program p initially labeled with pairwise distinct labels:

—*potentially instantly reentered statement:* a statement q of label n of the program p is potentially instantly reentered iff its label n is repeated in the multiset M of one of the reaction of an execution of p:

$$\exists r, \exists E, \exists E', \exists k, \exists M, \exists r' : p \twoheadrightarrow r \wedge r \xrightarrow[E]{E',\,k,\,M} r' \wedge \{n, n\} \subset M$$

—*schizophrenic statement:* a signal declaration or parallel statement is said to be schizophrenic iff it is potentially instantly reentered.
—*schizophrenic program:* the program p is said to be schizophrenic iff it contains a schizophrenic statement.

These definitions do not depend on the labeling, provided that labels are initially pairwise distinct. In the sequel, we shall omit unneeded labels.

Figure 7 deals with our previous example of instantly reentered signal declaration. Initially, the program contains a unique signal declaration, which we here arbitrarily label 5. The deduction tree of the first reaction considers once this signal declaration in its behavior, producing the singleton $\{5\}$. But, as part of the same reaction loop unrolling occurs, and the residual term now contains two declarations with the duplicated label 5. As the program part prior to the loop does instantly terminate,

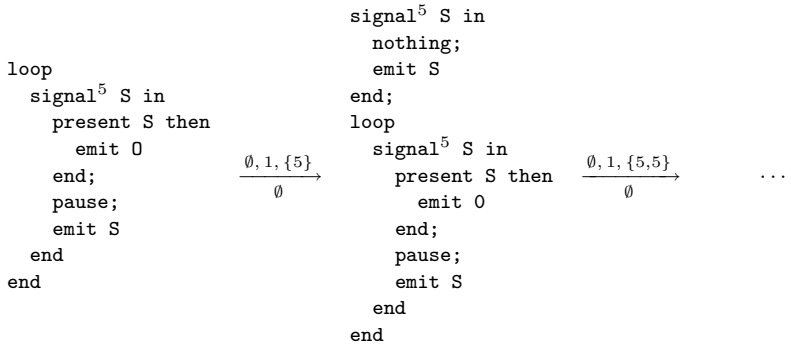

Fig. 7. Schizophrenic Signal Declaration

in the second instant the label 5 is collected twice in the behavior production, leading to the multiset $\{5, 5\}$ adorning the second transition. Thus, this signal declaration is potentially run across twice, so instantly reentered, and the program is schizophrenic.

4.2 Static Analysis

Deciding at compile time whether a program is schizophrenic or not a priori requires to explore all possible execution paths, which is unreasonable for large programs, as mentioned earlier. We need an effective decision procedure amenable to implementation. More precisely, we would like to ensure statically that statements are not schizophrenic, so that a "naive" translation is enough. In this section, we shall build a conservative static analysis for this safety property.

Potential Completion Codes

In Figure 5, we defined function $\Gamma : p \mapsto \Gamma_p$ which provides an overapproximation of the set of possible completion codes of the first reaction of p. Similarly, in Figure 8, we define function $\Omega : p \mapsto \Omega_p$ which overapproximates the set of possible completion codes of chains of reactions starting from p. The computation of Ω_p matches that of Γ_p except for `pause` statements for which $\Gamma_{\texttt{pause}} = \{1\}$ whereas $\Omega_{\texttt{pause}} = \{0, 1\}$.

p	Ω_p
`nothing`	$\{0\}$
`pause`	$\{0, 1\}$
`emit` S	$\{0\}$
`exit` T_d	$\{d + 2\}$
p `||` q	$\{m \in \mathbb{N}\ s.t.\ \exists k \in \Omega_p, \exists l \in \Omega_q, m = \max(k, l)\}$ i.e. $\max(\Omega_p, \Omega_q)$
`loop` p `end`	$\Omega_p \backslash \{0\}$
`present` S `then` p `else` q `end`	$\Omega_p \cup \Omega_q$
`trap` T `in` p `end`	$\{l \in \mathbb{N}\ s.t.\ \exists k \in \Omega_p, l = \downarrow k\}$ i.e. $\downarrow \Omega_p$
p; q	if $0 \in \Omega_p$ then $(\Omega_p \backslash \{0\}) \cup \Omega_q$ else Ω_p
`signal` S `in` p `end`	Ω_p

Fig. 8. Potential Completion Codes of Chains of Reactions

The following result states that Ω_p indeed contains all possible exit levels that may occur in the future (after any number of "normal", non-terminating reactions):

THEOREM 5. *If* $p \xrightarrow[E_1]{E'_1, 1} \ldots \xrightarrow[E_n]{E'_n, k} p_n$ *for some* $n \geq 1$ *then* $k \in \Omega_p$.

PROOF. There are two lemmas:

—$\forall p : \Gamma_p \subset \Omega_p$ by structural induction on p.

—Second, $\forall p, \forall E, \forall E', \forall p' : p \xrightarrow[E]{E', 1} p' \Rightarrow \Omega_{p'} \subset \Omega_p$ by structural induction on p.

Let us for instance suppose $p =$ "`loop` q `end`". If $p \xrightarrow[E]{E', 1} p'$ then by rule (loop), there exists q' such that $q \xrightarrow[E]{E', 1} q'$ and $p' =$ "q'`; loop` q `end`".
$\Omega_p = \Omega_{\texttt{loop } q \texttt{ end}}$. $\Omega_{p'} = \Omega_{q'\texttt{; loop } q \texttt{ end}}$.
$\Omega_p = \Omega_q \backslash \{0\}$ by definition of Ω for loops.
$\Omega_{p'} \subset (\Omega_{q'} \backslash \{0\}) \cup \Omega_{\texttt{loop } q \texttt{ end}}$ by definition of Ω for sequences.
$\Omega_{p'} \subset (\Omega_{q'} \backslash \{0\}) \cup (\Omega_q \backslash \{0\})$ by definition of Ω for loops.
By induction hypothesis, $\Omega_{q'} \subset \Omega_q$, thus $\Omega_{p'} \subset \Omega_p$.

Thanks to Theorem 3, we obtain: $k \in \Gamma_{p_{n-1}} \subset \Omega_{p_{n-1}} \subset \ldots \subset \Omega_p$. □

Contexts

Following standard notation [Barendregt 1981], we say that $C[\ ]$ is the context of the statement q in the program p iff $p = C[q]$. For example, the context of q in $p =$ "`loop pause; ` q `end`" is "`loop pause; [ ] end`". Contexts are statements with a single hole $[\ ]$. They are recursively defined:

—$[\ ]$ is the empty context,
—if $C[\ ]$ is a context and q is a statement then $C[$`present` S `then [ ] else` q `end`$]$ is a context,
—etc. for all constructs of pure ESTEREL.

As usual, $C[x]$ denotes the statement (respectively context) obtained by substituting the hole $[\ ]$ of $C[\ ]$ by the statement (respectively context) x, without any renaming of signals or exceptions.

Risk

Of course the fact that p may terminate or exit and be instantly reentered in $C[p]$ depends primarily on $C[\ ]$, but also on p. For a given context $C[\ ]$ one can identify *risky* completion codes, those that may cause the phenomenon whenever produced by p. For example, if p (non-instantly) terminates with code $k = 0$ or $k = 2$ in either of the following examples, instantaneous reentering is as such:

p is instantly reentered?	$k = 0$	$k = 2$
`loop trap T in` $[p]$`; pause end end`	*no*	*yes*
`trap T in loop` $[p]$ `end end`	*yes*	*no*

We now proceed with the definition of the risk function $C\langle\ \rangle$ associated with a context $C[\ ]$. It provides the exit levels which, if ever produced by the component plugged in the context hole, would cause the context to terminate and restart this

$$
\begin{array}{rcl}
\langle\ \rangle & \stackrel{def}{=} & \emptyset \\
C[\texttt{present}\ S\ \texttt{then}\ \langle\ \rangle\ \texttt{else}\ q\ \texttt{end}] & \stackrel{def}{=} & C\langle\ \rangle \\
C[\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ \langle\ \rangle\ \texttt{end}] & \stackrel{def}{=} & C\langle\ \rangle \\
C[\texttt{loop}\ \langle\ \rangle\ \texttt{end}] & \stackrel{def}{=} & \{0\} \cup C\langle\ \rangle \\
C[\langle\ \rangle\texttt{;}\ q] & \stackrel{def}{=} & \text{if } \Gamma_q \cap C\langle\ \rangle = \emptyset \text{ then } C\langle\ \rangle \backslash \{0\} \text{ else } C\langle\ \rangle \cup \{0\} \\
C[p\texttt{;}\ \langle\ \rangle] & \stackrel{def}{=} & \text{if } 0 \in \Gamma_p \text{ then } C\langle\ \rangle \text{ else } \emptyset \\
C[\texttt{trap}\ T\ \texttt{in}\ \langle\ \rangle\ \texttt{end}] & \stackrel{def}{=} & \{k \in I\!N, \downarrow k \in C\langle\ \rangle\} \\
C[\texttt{signal}\ S\ \texttt{in}\ \langle\ \rangle\ \texttt{end}] & \stackrel{def}{=} & \emptyset \\
C[\langle\ \rangle\ \texttt{||}\ q] & \stackrel{def}{=} & \emptyset \\
C[p\ \texttt{||}\ \langle\ \rangle] & \stackrel{def}{=} & \emptyset
\end{array}
$$

Fig. 9. Risk

component. The definition, provided in Figure 9, is compositional according to language constructs, and thus of course pertain to static analysis techniques. It specifies, for example, that 0 is a risky completion code for any context whose inner construct is a loop: $\forall C[\ ] : 0 \in C[\texttt{loop}\ \langle\ \rangle\ \texttt{end}]$.

The goal here is to achieve the following result: if p is schizophrenic then there exists $C[\ ]$ and q such that $p = C[q]$ and $\Omega_q \cap C\langle\ \rangle \neq \emptyset$ for some $q =$ "`signal` ... `end`" or $q =$ "...`||`...". In other words, if p is schizophrenic then there exists a signal declaration or parallel construct in p whose execution may complete with a risky completion code. Rather than formally establishing this property now, we shall directly prove in Section 6 that an optimized program transformation based on risk analysis successfully cures schizophrenia.

5. GOTOPAUSE

In this section, we extend the ESTEREL language with a new `gotopause` construct, which acts as a non-instantaneous jump instruction compatible with ESTEREL synchronous concurrency. Both `pause` and `gotopause` instructions use integer labels, with the following syntax:

"`gotopause` *label*" and "*label*`:pause`"

These labels should not be confused with those of the previous section. From now on, we shall only consider labels of this new kind.

We want `gotopause` to behave as follows:

—When the control reaches a "`gotopause` *label*" instruction, it stops for the current instant, as if it had reached a regular `pause` instruction.

—At the next instant, execution is resumed out of and past the corresponding pause instruction (the one with identical label), as if this pause was indeed reached in the previous reaction.

For example, the execution of "`gotopause 1;emit S;1:pause`" should not emit `S`.

There are several reasons for this extension [Tardieu 2004a]. For instance, state machines (automata) can be easily encoded with conditional jumps, as illustrated by the example of Figure 10. In the current paper, we are interested in using `gotopause` to efficiently compile schizophrenic programs (cf. Section 6).

```
1:pause;
action 1;
present A then gotopause 2 end;
present B then gotopause 3 end;
gotopause 1;

2:pause;
action 2;
gotopause 1;

3:pause;
action 3;
gotopause 2
```

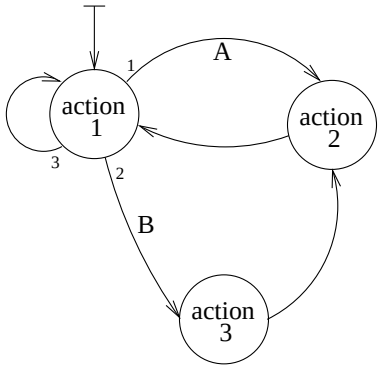

Fig. 10. Automata in ESTEREL⋆

Because of the non-locality of branching, such an extension usually requires some kind of continuation-passing style semantics. In the case of ESTEREL, we first have to reformulate the standard logical behavioral semantics of Section 2 in the form of a state semantics, that we prove equivalent to the former in the sense of bisimulation [Park 1981]. Then, we introduce and formalize `gotopause` and the semantics of the extended language, which we call ESTEREL⋆.

5.1 Labeled Behavioral Semantics

In the sequel, we consider ESTEREL programs with labeled `pause` instructions, unless otherwise stated. The function *unlabel* removes the labels of statements. We do not require labels to be unique yet. We call $\mathcal{L}(p)$ the set of labels of p, for example $\mathcal{L}(\texttt{1:pause; emit S; 2:pause; 1:pause}) = \{1, 2\}$.

In Figure 11, we introduce a labeled behavioral semantics ($\mapsto$) for ESTEREL by replacing the residual p' of the logical behavioral semantics ($\rightarrow$) with a set of labels L:

$$p \xmapsto[E]{E',k} L$$

This set collects the labels of the *active* `pause` instructions of the statement, that is to say those that will retain the control at the end of the reaction. For example, "`present S then 1:pause else 2:pause end`" in the presence of `S` produces the set $\{1\}$. In particular, for the (parallel) rule, the computed set of labels is:

$$\gamma^m(L \cup L') = \begin{cases} L \cup L' & \text{if } m = 1 \\ \emptyset & \text{if } m \neq 1 \end{cases} \quad \text{so that if } p \xmapsto[E]{E',k} L \text{ then } k \neq 1 \Leftrightarrow L = \emptyset$$

THEOREM 6. $\forall (p, E, E', k) : \left[\exists p' : \mathit{unlabel}(p) \xrightarrow[E]{E',k} \mathit{unlabel}(p') \Leftrightarrow \exists L : p \xmapsto[E]{E',k} L \right]$.

PROOF. Possible deductions in these semantics do not depend on the values of residuals ($\rightarrow$) or active labels ($\mapsto$), hence the result by structural induction on p. □

Intuitively, provided that the labeling is non-ambiguous, it should be possible to reconstruct p' from L and p, and precisely relate these two semantics. We formalize the correspondence below, with the definition of states and their expansion.

$$\texttt{nothing} \xmapsto[E]{\emptyset, 0} \emptyset \qquad \text{(nothing)}$$

$$l : \texttt{pause} \xmapsto[E]{\emptyset, 1} \{l\} \qquad \text{(pause)}$$

$$\texttt{exit}\ T_d \xmapsto[E]{\emptyset, d+2} \emptyset \qquad \text{(exit)}$$

$$\frac{S \in E}{\texttt{emit}\ S \xmapsto[E]{\{S\}, 0} \emptyset} \qquad \text{(emit)}$$

$$\frac{p \xmapsto[E]{E', k} L \quad k \neq 0}{\texttt{loop}\ p\ \texttt{end} \xmapsto[E]{E', k} L} \qquad \text{(loop)}$$

$$\frac{p \xmapsto[E]{E', k} L \quad q \xmapsto[E]{F', l} L' \quad m = \max(k, l)}{p\ \texttt{||}\ q \xmapsto[E]{E' \cup F', m} \gamma^m(L \cup L')} \qquad \text{(parallel)}$$

$$\frac{S \in E \quad p \xmapsto[E]{E', k} L}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \xmapsto[E]{E', k} L} \qquad \text{(present+)}$$

$$\frac{S \notin E \quad q \xmapsto[E]{E', k} L}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \xmapsto[E]{E', k} L} \qquad \text{(present−)}$$

$$\frac{p \xmapsto[E]{E', k} L}{\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end} \xmapsto[E]{E', \downarrow k} L} \qquad \text{(trap)}$$

$$\frac{p \xmapsto[E]{E', k} L \quad k \neq 0}{p;\ q \xmapsto[E]{E', k} L} \qquad \text{(sequence-p)}$$

$$\frac{p \xmapsto[E]{E', 0} L_0 \quad q \xmapsto[E]{F', k} L}{p;\ q \xmapsto[E]{E' \cup F', k} L'} \qquad \text{(sequence-q)}$$

$$\frac{p \xmapsto[E \cup \{S\}]{E', k} L \quad S \in E'}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \xmapsto[E]{E' \setminus \{S\}, k} L} \qquad \text{(signal+)}$$

$$\frac{p \xmapsto[E \setminus \{S\}]{E', k} L}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \xmapsto[E]{E', k} L} \qquad \text{(signal−)}$$

Fig. 11. Labeled Behavioral Semantics

5.2 States

We say that a statement is *well labeled* iff the labels of its `pause` instructions are pairwise distinct. From the combination of the well-labeled statement p and the set of labels $L \subset \mathcal{L}(p)$, we build the *state* $\lceil p|L \rfloor$. We say that a `pause` statement of label l is *selected* in $\lceil p|L \rfloor$ if $l \subset L$.

We shall use states to represents possible points[5] in the execution of a program. For example, the state $\lceil$`present S then 1:pause else 2:pause end`$|\{1\}\rfloor$ specifies that the execution of "`present S then 1:pause else 2:pause end`" has to be restarted from the `pause` instruction of label 1. In particular, the state $\lceil p|\emptyset \rfloor$ means that the execution of p is over. So, we call it an *inactive state*. Reciprocally, a state with at least one selected `pause` is an *active state*. We also define an extra state, called *initial state*, and noted $\lceil p|* \rfloor$, which tells that the execution of p has not started yet. Our definition of states slightly departs from the original one from [Mignard 1994], as we need to distinguish between *active* locations.

Not all states are useful. For example no execution of "`1:pause; 2:pause`" may generate the state $\lceil$`1:pause; 2:pause`$|\{1;2\}\rfloor$, as no reaction can reach two `pause` statements in sequence. We formalize this idea with the definition of valid states.

We say that q and r are *exclusive* in p, and write $q\#r$, iff there exists three contexts $C[\ ]$, $C_1[\ ]$, and $C_2[\ ]$ such that one of the following holds:

—$p = C[C_1[q]\texttt{;}\ C_2[r]]$
—$p = C[C_1[r]\texttt{;}\ C_2[q]]$
—$p = C[\texttt{present}\ S\ \texttt{then}\ [C_1[q]\ \texttt{else}\ C_2[r]\ \texttt{end}]]$
—$p = C[\texttt{present}\ S\ \texttt{then}\ [C_1[r]\ \texttt{else}\ C_2[q]\ \texttt{end}]]$

For example, in "p`; [`q `||` r`]`", p and q are exclusive, p and r are exclusive, q and r are not exclusive, that is to say *compatible*.

We say that a state of p is *valid* iff it is the initial state $\lceil p|* \rfloor$ of some state $\lceil p|L \rfloor$ such that selected `pause` statements are pairwise compatible. In other words, in a valid state, no two `pause` statements are selected in both parts of a sequence or both branches of a `present` statement

Invalid states are states that cannot be reached in the execution of the program[6].

THEOREM 7. *If p is well labeled and $p \xmapsto[E]{E',k} L'$ then $\lceil p|L' \rfloor$ is valid.*

PROOF. By structural induction on p. For instance, if $p =$ "q`;` r" then

—either $q \xmapsto[E]{E',k} L$ with $k \neq 0$: $\lceil q|L \rfloor$ is valid by induction hypothesis, moreover $L \subset \mathcal{L}(q)$ and $\mathcal{L}(q) \cap \mathcal{L}(r) = \emptyset$, thus $\lceil q\texttt{;}\ r|L \rfloor$ is valid;
—or $q \xmapsto[E]{A,0} L_0$ and $r \xmapsto[E]{B,k} L$ for some A, B, and L_0: $\lceil r|L \rfloor$ is valid by induction hypothesis, hence $\lceil q\texttt{;}\ r|L \rfloor$ is valid. □

[5]We shall use states to represent starting and ending points of reactions, that is to say macro-steps. However, micro-steps within a reaction cannot be represented by such states.
[6]The set of valid states contains the set of reachable states. But there still are unreachable valid states. For example, $\lceil$`1:pause || 2:pause`$|\{1\}\rfloor$ is both valid and unreachable in the execution of "`1:pause || 2:pause`".

initial state :	$\epsilon\lceil p \vert * \rfloor$	$\stackrel{def}{=}$	p	
inactive state :	$\epsilon\lceil p \vert \emptyset \rfloor$	$\stackrel{def}{=}$	`nothing`	
active states :	$\epsilon\lceil l : \texttt{pause} \vert \{l\} \rfloor$	$\stackrel{def}{=}$	`nothing`	
$(L \neq \emptyset)$	$\epsilon\lceil p\texttt{;}\ q \vert L \rfloor$	$\stackrel{def}{=}$	$\epsilon\lceil p \vert L \rfloor\texttt{;}\ q$	if $L \subset \mathcal{L}(p)$
	$\epsilon\lceil p\texttt{;}\ q \vert L \rfloor$	$\stackrel{def}{=}$	$\epsilon\lceil q \vert L \rfloor$	if $L \subset \mathcal{L}(q)$
	$\epsilon\lceil \texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \vert L \rfloor$	$\stackrel{def}{=}$	$\epsilon\lceil p \vert L \rfloor$	if $L \subset \mathcal{L}(p)$
	$\epsilon\lceil \texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \vert L \rfloor$	$\stackrel{def}{=}$	$\epsilon\lceil q \vert L \rfloor$	if $L \subset \mathcal{L}(q)$
	$\epsilon\lceil \texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end} \vert L \rfloor$	$\stackrel{def}{=}$	$\texttt{trap}\ T\ \texttt{in}\ \epsilon\lceil p \vert L \rfloor\ \texttt{end}$	
	$\epsilon\lceil p\ \texttt{||}\ q \vert L \rfloor$	$\stackrel{def}{=}$	$\epsilon\lceil p \vert L \cap \mathcal{L}(p) \rfloor\ \texttt{||}\ \epsilon\lceil q \vert L \cap \mathcal{L}(q) \rfloor$	
	$\epsilon\lceil \texttt{loop}\ p\ \texttt{end} \vert L \rfloor$	$\stackrel{def}{=}$	$\epsilon\lceil p \vert L \rfloor\texttt{;}\ \texttt{loop}\ p\ \texttt{end}$	
	$\epsilon\lceil \texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \vert L \rfloor$	$\stackrel{def}{=}$	$\texttt{signal}\ S\ \texttt{in}\ \epsilon\lceil p \vert L \rfloor\ \texttt{end}$	

Fig. 12. State Expansion

5.3 State Expansion

In Figure 12, we recursively define a *state expansion function* ϵ. It derives a statement from a valid state. Remark that the rule for the empty set L has priority over the other rules. For example, $\epsilon\lceil \texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end} \vert \emptyset \rfloor$ is `nothing` rather than "`trap` T `in nothing end`".

Basically, this ϵ function expands a valid state into a statement of equal "meaning". For example, if a `pause` is selected in the left component p of a sequence "$p\texttt{;}\ q$", then the execution of the sequence has to be continued by the end of this left component, which is exactly the expansion of the state of p, followed by the right component q of the sequence. Therefore, if $L \subset \mathcal{L}(p)$ then $\epsilon\lceil p\texttt{;}\ q \vert L \rfloor = \epsilon\lceil p \vert L \rfloor\texttt{;}\ q$.

The expansion retains labels (in the right-hand side of sequences and loop bodies). We observe that even if $\lceil p \vert L \rfloor$ is a valid state of the well-labeled statement p, the labeled statement $\epsilon\lceil p \vert L \rfloor$ is not necessarily well labeled, as loop unrolling may occur. For example,

$$\epsilon\lceil \texttt{loop 1:pause;2:pause end} \vert \{1\} \rfloor = \texttt{nothing;2:pause;loop 1:pause;2:pause end}$$

This is not an issue, as we shall not build states out of such statements. Moreover, validity is a stable property:

THEOREM 8. *If $\lceil p \vert L \rfloor$ is valid and $\epsilon\lceil p \vert L \rfloor \stackrel{E', k}{\underset{E}{\longmapsto}} L'$ then $\lceil p \vert L' \rfloor$ is valid.*

PROOF. Structural induction on p. Similar to Theorem 7. □

Thanks to this state expansion function, we can now express the fact that the logical and labeled semantics define the same reactions.

THEOREM 9. *If p is well labeled:*

—*If $p \stackrel{E', k}{\underset{E}{\longmapsto}} L$ then $unlabel(p) \stackrel{E', k}{\underset{E}{\longrightarrow}} unlabel(\epsilon\lceil p \vert L \rfloor)$.*

—*If $unlabel(p) \stackrel{E', k}{\underset{E}{\longrightarrow}} p'$ then there exists L s.t. $p \stackrel{E', k}{\underset{E}{\longmapsto}} L$ and $unlabel(\epsilon\lceil p \vert L \rfloor) = p'$.*

PROOF. Structural induction on p. Let us for instance suppose that $p =$ "q `||` r":

—If $p \xmapsto[E]{E', k} L$ then $q \xmapsto[E]{E'_q, k_q} k_q$ and $r \xmapsto[E]{E'_r, k_r} L_r$ for some $E'_q, k_q, k_q, E'_r, k_r, k_r$ such that $E' = E'_q \cup E'_r$, $k = \max(k_q, k_r)$, and $L = L_q \cup L_r$, by definition of rule (parallel) in the labeled behavioral semantics.

By induction hypothesis, $\begin{cases} \mathit{unlabel}(q) \xrightarrow[E]{E'_q, k_q} \mathit{unlabel}(\epsilon\lceil q|L_q\rfloor) \text{ with } L_q \subset \mathcal{L}(q) \\ \mathit{unlabel}(r) \xrightarrow[E]{E'_r, k_r} \mathit{unlabel}(\epsilon\lceil r|L_r\rfloor) \text{ with } L_r \subset \mathcal{L}(r) \end{cases}$

As a consequence, by definition of rule (parallel) in the logical behavioral semantics, $\mathit{unlabel}(q) \mid\mid \mathit{unlabel}(r) \xrightarrow[E]{E', k} \mathit{unlabel}(\epsilon\lceil q|L_q\rfloor) \mid\mid \mathit{unlabel}(\epsilon\lceil r|L_r\rfloor)$. Therefore, $\mathit{unlabel}(p) \xrightarrow[E]{E', k} \mathit{unlabel}(\epsilon\lceil p|L\rfloor)$, by definition of ϵ.

—If $\mathit{unlabel}(p) \xrightarrow[E]{E', k} p'$ then there exist E'_q, k_q, q', L_q such that $\mathit{unlabel}(q) \xrightarrow[E]{E'_q, k_q} q'$ and $q \xmapsto[E]{E'_q, k_q} L_q$ and $\mathit{unlabel}(\epsilon\lceil q|L_q\rfloor) = q'$ and similarly for r, with $p \xmapsto[E]{E', k} L_q \cup L_r$. By definition of ϵ, $\mathit{unlabel}(\epsilon\lceil p|L_q \cup L_r\rfloor) = q' \mid\mid r' = p'$.

And so on for all constructs of pure ESTEREL. □

This proves that p' can be obtained from $\lceil p|L\rfloor$ and vice versa. The result of a reaction is equivalently characterized by either the residual p' or the set of active labels L we have just introduced. This is the key that enables the definition of a state behavioral semantics for ESTEREL.

5.4 State Behavioral Semantics

We define a state behavioral semantics ($\circ\!\!\rightarrow$) for ESTEREL as follows:

$$\lceil p|L\rfloor \circ\!\xrightarrow[E]{E', k} \lceil p|L'\rfloor \text{ iff } \epsilon\lceil p|L\rfloor \xmapsto[E]{E', k} L'$$

One reaction of the well-labeled statement p in the valid state $\lceil p|L\rfloor$ produces the valid state $\lceil p|L'\rfloor$ iff L' is the set of active labels computed by the labeled semantics for the statement $\epsilon\lceil p|L\rfloor$.

THEOREM 10. *The logical and state semantics are equivalent, in the sense of bisimulation [Park 1981].*

PROOF. We construct a candidate bisimulation "$\sim$" between valid states and non-labeled statements (i.e. original ESTEREL statements) as follows:

$$\text{for all valid state } s,\ s \sim \mathit{unlabel}(\epsilon(s))$$

By construction of the relation (for details see [Tardieu 2004b], pages 99–100),

—For each valid state s there exists a statement $p = \mathit{unlabel}(\epsilon(s))$ such that $s \sim p$.

—For each statement p there exists a valid state $s = \lceil p|*\rfloor$ such that $s \sim p$.

—If $s \sim p$ and $s \circ\!\xrightarrow[E]{E', k} s'$ then there exists p' such that $p \xrightarrow[E]{E', k} p'$ and $s' \sim p'$.

—If $s \sim p$ and $p \xrightarrow[E]{E', k} p'$ then there exists s' such that $s \circ\!\xrightarrow[E]{E', k} s'$ and $s' \sim p'$.

As a consequence, this relation is a bisimulation between the two semantics. □

In particular, this confirms that pure ESTEREL programs are *finite state*.

5.5 Labeled Behavioral Semantics of ESTEREL⋆

The "`gotopause` *label*" instruction can be introduced in the labeled behavioral semantics using a simple rule:

$$\texttt{gotopause}\ l \xmapsto[E]{\emptyset,\,1} \{l\} \qquad \text{(gotopause)}$$

It specifies, that "`gotopause` l" activates the label l. For example,

$$\lceil \texttt{gotopause 1; emit S; 1:pause} | * \rfloor \circ\!\xrightarrow[E]{\emptyset,\,1} \lceil \texttt{gotopause 1; emit S; 1:pause} | \{1\} \rfloor$$

For simplicity, we suppose that, in a program, there is no `gotopause` without target, that is to say `pause` instruction of equal label. We do not suppose however that `gotopause` statements have pairwise distinct labels. Simultaneous jumps to the same target make sense.

5.6 Well-formedness

Arbitrary simultaneous jumps are not always harmless, however. In fact, validity is no longer preserved by the extended semantics:

$$\lceil \texttt{[gotopause 1||2:pause];1:pause} | * \rfloor \circ\!\xrightarrow[\emptyset]{\emptyset,\,1} \lceil \texttt{[gotopause 1||2:pause];1:pause} | \{1,2\} \rfloor$$

The expansion of $\lceil$ `[gotopause 1||2:pause];1:pause` $|\{1,2\}\rfloor$ is undefined as two `pause` instructions are selected in sequence. Such a state does not make sense, as the activity status of locations contradict the structural shape of the program. Therefore, the initial state $\lceil$ `[gotopause 1||2:pause];1:pause` $|*\rfloor$ cannot be considered to be correct, and the program "`[gotopause 1||2:pause];1:pause`" has to be rejected. This is dealt with through a proper definition (and compile time analysis) of well-formedness, which ensures that `gotopause` is compatible with ESTEREL concurrency, as well as other language constructs[7].

If "k`:pause`" and "l`:pause`" are exclusive, i.e. cannot be executed simultaneously, then we shall ensure that they are never activated simultaneously.

Formally, we say that a well-labeled program p is *well formed* iff:

$$\forall k, \forall l : k\texttt{:pause}\ \#\ l\texttt{:pause} \Rightarrow \begin{cases} \texttt{gotopause}\ k\ \#\ \texttt{gotopause}\ l \\ \texttt{gotopause}\ k\ \#\ l\texttt{:pause} \\ \qquad k\texttt{:pause}\ \#\ \texttt{gotopause}\ l \end{cases}$$

Thanks to well-formedness, we can now recover the stability of validity as follows:

THEOREM 11. *If p is well formed, $\lceil p|L\rfloor$ valid, and $\epsilon\lceil p|L\rfloor \xmapsto[E]{E',\,k} L'$ then $\lceil p|L'\rfloor$ is valid.*

PROOF. Similar to Theorem 8. A reaction of $\epsilon\lceil p|L\rfloor$ producing the set of active labels L' may reach at most one element of any pair of exclusive `pause`/`gotopause` instructions. Thus, by definition of well-formedness, the `pause` statements selected in $\lceil p|L'\rfloor$ are exclusive. □

[7]In principle, checking that no invalid state can be reached in the execution of an ESTEREL⋆ program requires a complete state space exploration. The aim of well-formedness is to avoid such a prohibitive computation. We provide a simple conservative soundness criterion, amenable to efficient implementation, which nevertheless allows for the program patterns we are interested in.

Well-formedness is a static semantic condition. It can be checked easily while building the abstract syntax tree of a program. Compatible/exclusive `pause` instructions are already identified for optimization purposes.

The `gotopause` construct is compatible with concurrency in the sense that, for instance, well-formedness let us compose any set of well-formed programs in parallel.

Of course, every well-labeled ESTEREL program is a well-formed ESTEREL* program. In the sequel, we shall only generate well-formed ESTEREL* programs.

5.7 State Behavioral Semantics of ESTEREL*

We define a state behavioral semantics ($\circ\!\!\rightarrow$) for ESTEREL* as follows:

$$\lceil p|L\rfloor \circ\!\!\xrightarrow[E]{E',k} \lceil p|L'\rfloor \text{ iff } \epsilon\lceil p|L\rfloor \xmapsto[E]{E',k} L'$$

One reaction of the well-formed statement p in the valid state $\lceil p|L\rfloor$ produces the valid state $\lceil p|L'\rfloor$ iff L' is the set of active labels computed by the labeled semantics (including the (gotopause) rule) for the statement $\epsilon\lceil p|L\rfloor$.

The state semantics of ESTEREL* restricted to ESTEREL programs is exactly the state semantics of Section 5.4, which we have shown to be equivalent to the initial logical behavioral semantics. Therefore, ESTEREL* is truly an extension of the original pure ESTEREL language.

6. REWRITING LOOPS

The programming style advocated by ESTEREL (local declarations plus local concurrency plus imperative loops) naturally leads to schizophrenic specifications [Clement and Incerpi 1989]. So, compilers cannot afford to ignore or reject such program patterns, and let the user deal with schizophrenia all by himself.

The idea of automatically rewriting schizophrenic programs into equivalent non-schizophrenic programs is thus a natural one. It has been proposed by Mignard [Mignard 1994]. His method consists in recursively *dup*licating loop bodies:

$$\begin{aligned} dup(\texttt{nothing}) &\stackrel{def}{=} \texttt{nothing} \\ dup(\texttt{signal } S \texttt{ in } p \texttt{ end}) &\stackrel{def}{=} \texttt{signal } S \texttt{ in } dup(p) \texttt{ end} \\ &\ldots \\ dup(\texttt{loop } p \texttt{ end}) &\stackrel{def}{=} \texttt{loop } dup(p)\texttt{; } dup(p) \texttt{ end} \end{aligned}$$

This program transformation is called *reincarnation* as it explicitly distributes the several *incarnations* (simultaneous instances) of a single statement into several distinct "bodies". In other words, one statement having several incarnations is changed into several statements, each of them having a unique incarnation.

First, if p is safe, it can be shown that p and $dup(p)$ behave the same in all contexts. Technically, they are strongly bisimilar with respect to ESTEREL logical semantics[8]. In particular, if p cannot terminate instantly then $dup(p)$ cannot either. As a consequence, each loop body of a rewritten safe program consists of a sequence of two identical non-instantaneous blocks. Neither block can be instantly left and reentered. Therefore, $dup(p)$ is not schizophrenic.

[8]Due to the form of ESTEREL semantics, strong bisimulation relations are congruence relations.

$$
\begin{array}{rcl}
\mathit{surf}(\texttt{nothing}) & \stackrel{def}{=} & \texttt{nothing} \\
\mathit{surf}(\texttt{exit } T) & \stackrel{def}{=} & \texttt{exit } T \\
\mathit{surf}(\texttt{emit } S) & \stackrel{def}{=} & \texttt{emit } S \\
\mathit{surf}(p \texttt{ || } q) & \stackrel{def}{=} & \mathit{surf}(p) \texttt{ || } \mathit{surf}(q) \\
\mathit{surf}(\texttt{present } S \texttt{ then } p \texttt{ else } q \texttt{ end}) & \stackrel{def}{=} & \texttt{present } S \texttt{ then } \mathit{surf}(p) \texttt{ else } \mathit{surf}(q) \texttt{ end} \\
\mathit{surf}(\texttt{trap } T \texttt{ in } p \texttt{ end}) & \stackrel{def}{=} & \texttt{trap } T \texttt{ in } \mathit{surf}(p) \texttt{ end} \\
\mathit{surf}(\texttt{signal } S \texttt{ in } p \texttt{ end}) & \stackrel{def}{=} & \texttt{signal } S \texttt{ in } \mathit{surf}(p) \texttt{ end} \\
\hline
\mathit{surf}(\texttt{loop } p \texttt{ end}) & \stackrel{def}{=} & \mathit{surf}(p) \\
\mathit{surf}(p\texttt{; } q) & \stackrel{def}{=} & \text{if } 0 \in \Gamma_p \text{ then } \mathit{surf}(p)\texttt{; } \mathit{surf}(q) \text{ else } \mathit{surf}(p) \\
\mathit{surf}(\mathit{label}\texttt{:pause}) & \stackrel{def}{=} & \texttt{gotopause } \mathit{label}
\end{array}
$$

Fig. 13. Surface of Statements

This would once and for all take care of schizophrenic programs if the transformation was efficient enough. It is not the case: $dup(p)$ may be exponentially larger than p because of nested unfoldings (cf. example in Section 6.3).

6.1 Replacing `loop` with `gotopause`

In the sequel, we shall always consider *safe* and *well-labeled* Esterel programs (pairwise distinct labels). Thanks to `gotopause`, we can partially *unfold* loop bodies as follows:

$$
\begin{array}{rcl}
\mathit{unfold}(\texttt{nothing}) & \stackrel{def}{=} & \texttt{nothing} \\
\mathit{unfold}(\texttt{signal } S \texttt{ in } p \texttt{ end}) & \stackrel{def}{=} & \texttt{signal } S \texttt{ in } \mathit{unfold}(p) \texttt{ end} \\
& \ldots & \\
\mathit{unfold}(\texttt{loop } p \texttt{ end}) & \stackrel{def}{=} & \mathit{unfold}(p)\texttt{; } \mathit{surf}(p)
\end{array}
$$

Intuitively, we shall only duplicate the *surface* of p, that is to say the part of p which is instantly reachable, and jump *back* from the surface to the "regular" copy of p, using `gotopause` statements. Again, this unfolding has to be recursive because of nested loops.

Function *surf*, for surface, is recursively defined in Figure 13. The three non-elementary rules are boxed:

—A loop cannot be taken instantly, so it may be removed from the surface.

—If p in "p`;` q" cannot be instantaneous then q cannot be reached instantly.

—`pause` statements are changed into `gotopause` statements, thus effectively replacing the initial `loop` instruction by a bunch of non-instantaneous jumps.

For example,

$$
\begin{array}{rcl}
\mathit{surf}(\texttt{emit S;1:pause}) & = & \texttt{emit S;gotopause 1} \\
\mathit{unfold}(\texttt{loop emit S;1:pause end}) & = & \texttt{emit S;1:pause;emit S;gotopause 1}
\end{array}
$$

Theorem 12. *If p is well labeled then "p; $\mathit{surf}(p)$" is well formed.*

Proof. Let k and l be two labels of `pause` or `gotopause` instructions occurring in compatible locations in "p; $\mathit{surf}(p)$". Since p and $\mathit{surf}(p)$ are composed in sequence, k and l must both occur in p or both in $\mathit{surf}(p)$ in compatible locations.

In the second case, by definition of *surf*, there exists q and r such that k occurs in $surf(q)$ and l occurs in $surf(r)$ and "q `||` r" is a subterm of p. Therefore, k occurs in q and l occurs in r, so that k and l occurs in p in compatible locations. In summary, if k and l occur in "p`;` $surf(p)$" in compatible locations, then they occur in p in compatible locations. Since p contains no `gotopause` instructions, the "k : `pause`" and "l : `pause`" instructions of p are compatible. So "p`;` $surf(p)$" is well formed. □

THEOREM 13. *If p is non-instantaneous then "p; $surf(p)$" and "*`loop` p `end`*" behave the same in all contexts, that is to say are strongly bisimilar.*

PROOF. There are several lemmas:

—$\forall q$ safe, $\forall E, \forall E', \forall k, \forall L : q \xrightarrow[E]{E', k} L \Leftrightarrow surf(q) \xrightarrow[E]{E', k} L$ by induction on q.

—$\forall L : \lceil p\texttt{;}\ surf(p) | L \rfloor$ is valid iff $\lceil$`loop` p `end`$|L\rfloor$ is valid.

—$\forall L : \lceil$`loop` p `end`$|L\rfloor$ and $\lceil p\texttt{;}\ surf(p)|L\rfloor$ are strongly bisimilar (if valid).

—$\lceil$`loop` p `end`$|*\rfloor$ and $\lceil p\texttt{;}\ surf(p)|*\rfloor$ are strongly bisimilar,

Their proofs are given in [Tardieu 2004b]. □

THEOREM 14. *If p is well labeled then $unfold(p)$ is well formed. Moreover, if p is safe then p and $unfold(p)$ behave the same.*

PROOF. By induction on nested loops, using the previous two theorems. □

In summary, we have successfully replaced `loop` statements with `gotopause` statements, with very limited code replication, as we discard non-instantly reachable pieces of code in the generated copy of the statement. Of course, the resulting program no longer contains potentially instantaneous loops or potentially schizophrenic statements, since no statement may be left and instantly reentered without loops.

THEOREM 15. *If p is safe then $unfold(p)$ is not schizophrenic.*

6.2 Alternate Rewriting

Even if the previous transformation is the obvious one, it is not the one we work with. Rather than putting $surf(p)$ in sequence *after* the regular copy of p, we shall put it in sequence *before* this regular copy. We recursively define function $unfold'$ as follows, *surf* being unchanged:

$$\begin{aligned} unfold'(\texttt{nothing}) &\stackrel{def}{=} \texttt{nothing} \\ unfold'(\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end}) &\stackrel{def}{=} \texttt{signal}\ S\ \texttt{in}\ unfold'(p)\ \texttt{end} \\ &\cdots \\ unfold'(\texttt{loop}\ p\ \texttt{end}) &\stackrel{def}{=} \texttt{loop}\ surf(p)\texttt{;}\ unfold'(p)\ \texttt{end} \end{aligned}$$

We can no longer get rid of the `loop` statement; but this new transformation is again provably correct. First, it preserves the behavior of programs:

THEOREM 16. *If p is well labeled then "$surf(p)$; p" is well formed.*

THEOREM 17. *If p is non-instantaneous then p and "$surf(p)$`;` p" behave the same in all contexts.*

THEOREM 18. *If p is well labeled then $unfold'(p)$ is well formed. Moreover, if p is safe then p and $unfold'(p)$ behave the same.*

PROOF. Similar to the proofs of Theorems 12, 13, and 14. □

Second, this new transformation cures schizophrenia as well.

THEOREM 19. *If p is safe then $unfold'(p)$ is not schizophrenic.*

PROOF. By structural induction on p. Let us consider the case: $p =$ `loop` q `end`. $\epsilon(\lceil$`loop` $surf(q)$`;` $unfold'(q)$ `end`$|L\rfloor) = \epsilon(\lceil unfold'(q)|L\rfloor)$`;` `loop` $surf(q)$`;` $unfold'(q)$ `end` Since $surf(p)$ cannot be instantaneous, any reaction of the above statement can involve at most one copy of $unfold'(p)$ (left occurrence) and $surf(p)$. By induction hypothesis, $unfold'(q)$ is not schizophrenic, thus no two copies of the same signal declaration or parallel constructs of $unfold'(q)$ in the expansion $\epsilon(\lceil unfold'(q)|L\rfloor)$ can be reached in one reaction. Therefore, no two copies of the same signal declaration or parallel constructs of "`loop` $surf(q)$`;` $unfold'(q)$ `end`" can be reached in a single reaction of its expansion "$\epsilon(\lceil unfold'(q)|L\rfloor)$`;` `loop` $surf(q)$`;` $unfold'(q)$ `end`". As a result, p is not schizophrenic. □

While it may seem worse than the previous one, this new rewriting scheme is a much better starting point for improvements because of Theorem 17. In comparison with Theorem 13, it provides an unfolded equivalent for all non-instantaneous statements rather than loops only. In the current definition of *unfold′*, we systematically replace loop bodies p, by their unfolding "$surf(p)$`;` $unfold'(p)$". In the sequel, we shall consider doing such substitutions in a much more selective fashion.

6.3 Algorithm

At this point, we can define a first original reincarnation algorithm that rewrites any safe ESTEREL program p into a non-schizophrenic equivalent ESTEREL* program $unfold'(\hat{p})$:

—We first label the `pause` statements of the program p with pairwise distinct labels. We denote the result with $\hat{p}$.

—We then compute the image of $\hat{p}$ by function *unfold′*.

The rewriting of p into $surf(\hat{p})$ is linear, and $unfold'(\hat{p})$ is at most quadratically larger than p. For example,

- `loop [`p `||` `loop` q `end] end`

$\overset{dup}{\Longrightarrow}$

```
loop
  [dup(p) || loop dup(q);dup(q) end];
  [dup(p) || loop dup(q);dup(q) end];
end
```

$\overset{unfold'}{\Longrightarrow}$

```
loop
  [surf(p) || surf(q)];
  [unfold'(p) || loop surf(q);unfold'(q) end]
end
```

- `loop [`p `||` `loop [`p `||` `loop` p `end] end] end`

$\overset{dup}{\Longrightarrow}$ $2 + 4 + 8 = 14$ times p *(exponential growth)*

$\overset{unfold'}{\Longrightarrow}$ $2 + 3 + 4 = 9$ times p *(quadratic growth)*

p	$unfold^\star(p, C\langle\,\rangle)$
`nothing`	`nothing`
label : `pause`	*label* : `pause`
`exit` T	`exit` T
`emit` S	`emit` S
`present` S `then` p `else` q `end`	`present` S `then` $unfold^\star(p, C[\texttt{present}\ S\ \texttt{then}\ \langle\,\rangle\ \texttt{else}\ q\ \texttt{end}])$ `else` $unfold^\star(q, C[\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ \langle\,\rangle\ \texttt{end}])$ `end`
`loop` p `end`	`loop` $unfold^\star(p, C[\texttt{loop}\ \langle\,\rangle\ \texttt{end}])$ `end`
`trap` T `in` p `end`	`trap` T `in` $unfold^\star(p, C[\texttt{trap}\ T\ \texttt{in}\ \langle\,\rangle\ \texttt{end}])$ `end`
p `;` q	$unfold^\star(p, C[\langle\,\rangle\ \texttt{;}\ q])\ \texttt{;}\ unfold^\star(q, C[p\ \texttt{;}\ \langle\,\rangle])$
`signal` S `in` p `end`	if $C\langle\,\rangle \cap \Omega_p = \emptyset$ then `signal` S `in` $unfold^\star(p, \emptyset)$ `end` else `signal` S `in` $surf(p)$ `end;` $skip$(`signal` S `in` $unfold^\star(p, \emptyset)$) `end`
p `||` q	if $C\langle\,\rangle \cap \Omega_{p \,||\, q} = \emptyset$ then $unfold^\star(p, \emptyset)$ `||` $unfold^\star(q, \emptyset)$ else `[`$surf(p)$ `||` $surf(q)$`];` $skip$(`[`$unfold^\star(p, \emptyset)$ `||` $unfold^\star(q, \emptyset)$`]`

Fig. 14. Improved Rewriting using Static Analysis

But in these examples, the parallel statements cannot terminate, a fortiori be instantly restarted by the loops. Hence, their unfolding is useless. In general, less unfolding is possible, if we take into account our static analysis of schizophrenia.

6.4 Improved Rewriting

Schizophrenia arises from the nesting of signal declarations or parallel statements within loops (Section 4). Instead of *systematically* unfolding *whole* loop bodies, we could (*i*) expand signal declarations or parallel statements only, and (*ii*) condition expansion on the result of our static analysis. These two ideas lead the definition of $unfold^\star$ in Figure 14, *surf* remaining unchanged.

Function $unfold^\star$ is now context-dependent. It takes two arguments: the statement p to rewrite and the (initially empty) set $C\langle\,\rangle$ of risky completion codes for the current context. The recursive computation of this second argument exactly matches that of the *risk* function (cf. Figure 9). As announced, loops no longer replicate code on their own. Moreover, signal declarations and parallel statements are only expanded if potentially schizophrenic ($C\langle\,\rangle \cap \Omega_p \neq \emptyset$).

Up to now only non-instantaneous statements (loop bodies) where unfolded. With this new program transformation, potentially instantaneous blocks (signal declarations and parallel statements) may be unfolded. Therefore, the unfolding step:

$$p \longrightarrow surf(p)\,;\ p$$

is replaced in this last program transformation by:

$$p \longrightarrow surf(p)\,;\ skip(p)$$

where function *skip* can be defined for instance as the following:

$$skip(p) = \texttt{trap}\ T\ \texttt{in exit}\ T\texttt{;}\ p\ \texttt{end} \quad \text{(where } T \text{ is a fresh exception name)}$$

Intuitively, function *skip* let us "skip" over its argument, so that even if $surf(p)$ in "$surf(p)$; $skip(p)$" terminates instantly then p and "$surf(p)$; $skip(p)$" behave the same. In other words, p in $skip(p)$ can only be reached using `gotopause` instructions.

THEOREM 20. *If p is well labeled then "$surf(p)$;$skip(p)$" is well formed.*

THEOREM 21. *For all p, p and "$surf(p)$;$skip(p)$" behave the same in all contexts.*

THEOREM 22. *If p is well labeled then $unfold^\star(p, \emptyset)$ is well formed. Moreover, if p is safe then p and $unfold^\star(p, \emptyset)$ behave the same.*

PROOF. Similar to the proofs of Theorems 12, 13, and 14. □

THEOREM 23. *If p is safe then $unfold^\star(p, \emptyset)$ is not schizophrenic.*

PROOF. This complex proof is sketched in [Tardieu 2004b], pages 127–129. □

6.5 Improved Algorithm

In summary, the reincarnation algorithm we propose for an ESTEREL program p consists of traversing p a first time to compute Γ and Ω and label p with pairwise distinct labels, producing $\hat{p}$, then computing $unfold^\star(\hat{p}, \emptyset)$. For example,

```
loop                                    loop
  signal S in                             signal S in
    present S then emit O end;              present S then emit O end;
    pause;                                  1:gotopause;
    emit S                      unfold*     end;
  end;                          ⟹          signal S in
  present I then emit O end                   present S then emit O end;
end                                           1:pause;
                                              emit S;
                                            end;
                                            present I then emit O end
                                          end
```

Further code size reduction, such as the removal of the unreachable test (in *italic*) can be achieved via standard dead code elimination techniques[9].

Again, $unfold^\star(\emptyset, \hat{p})$ may be quadratically larger than p in the worst case. But this last algorithm is in practice quasi-linear, as we shall measure in Section 8. In particular, in the absence of potentially schizophrenic statements, there is no unfolding at all.

[9]In our current ESTEREL* compiler (see Section 8), we achieve dead code elimination at the circuit level by running the exact same constant propagation algorithm that was used prior to the introduction of `gotopause` in the source language. In other words, no new dead code elimination heuristics is required in order to compile efficiently ESTEREL* programs.

7. REINCARNATION IN FULL ESTEREL

Full ESTEREL adds to pure ESTEREL the ability to manipulate data of various kinds: private variables, shared values, counters, registers, etc. The good news is that data do not lead to more schizophrenia problems. Therefore, extending our characterization and static analysis to full ESTEREL is straightforward. The bad news is it breaks our rewriting scheme. Let's consider a program where a variable is declared within a schizophrenic signal scope (`var` statement):

```
loop                                          loop
  signal S in                                   signal S in
    var V in ...; pause;... end  ⟹*  end;       var V in ...; gotopause 1;... end
  end                                           signal S in
end                                               var V in ...; 1:pause;... end
                                                end
                                              end
```

Unlike signal statuses, variables retain their values between instants. Thus, duplicating the declaration of `V` and jumping from one declaration to the other one, changes the semantics of the program.

There are two obvious fixes. First, we may move declarations of data (but not initializations!) up to the loop, using alpha-renaming when needed. In our example, we obtain:

```
loop
  var V in
    signal S in ...; 1:gotopause;... end;
    signal S in ...; 1:pause;... end
  end
end
```

But this requires additional syntax for counters for instance. Moreover, much scoping information is lost in this process.

Second, we may introduce *static aliasing* in ESTEREL⋆, expressing that the two distinct declarations of `V` in the naive rewriting in fact correspond to a unique memory cell. For instance, we can index variables before expansion, thus planning memory allocation in advance:

```
loop
  signal S in
    var V@1 in ...; gotopause 1;... end
  end;
  signal S in
    var V@1 in ...; 1:pause;... end
  end
end
```

We have chosen the latter solution in our implementation. This technique can be applied to all data defined in full ESTEREL. It solves the only problem raised by the extension of our techniques from the pure subset of ESTEREL to the full language.

program	number of kernel statements			description
	source	algorithm (6.3)	algorithm (6.5)	
global	10286	566585	16867	avionics man-machine interface
cabine	7644	67680	8020	avionics cockpit interface
atds100	890	1372	990	video generator
ww	432	833	439	wristwatch
tcint	403	725	418	turbochannel bus
P18	28	86	58	a lot of schizophrenia
abro	14	18	14	no schizophrenia

Table I. Comparison

8. IMPLEMENTATION

We have designed an algorithm that translates any ESTEREL program into a non-schizophrenic equivalent ESTEREL* program, via a rewriting of schizophrenic signal declarations and parallel statements.

Using this algorithm, we have implemented a prototype compiler for full ESTEREL into *Digital Sequential Circuits*, generating `sc6` files[10]. The compiler code consists of about 5000 lines of OCaml, structured as follows:

(1) parsing and macro expansion,
(2) link (i.e. source-level inlining of submodules),
(3) static analysis and reincarnation,
(4) compilation (of non-schizophrenic programs),
(5) a bit of boolean optimization (for `sc6` compliance).

Relevant to our discussion are Steps 3 and 4 and their relationship. Step 3 rewrites linked macro-expanded ESTEREL source code using the algorithms we have described in the previous section. Step 4 essentially implements the naive circuit synthesis of Berry [Berry 1999], in which we incorporate `gotopause` and data.

Compiling `gotopause` is straightforward:

—In our circuit generation, we allocate as usual one bit-register per `pause` statement. But in addition to the regular connection of one wire to the input pin of this register required by the `pause` statement itself, we connect (through an `or` gate) one extra wire per `gotopause` statement with corresponding label.
—ESTEREL compilers are typically based on internal representations of programs as graphs in which `gotopause` is easily encoded.

We have conducted some early experiments, summarized in Table I. We count the number of statements (after macro expansion) in programs of various kinds and sizes (from [Berry 1999] and [Potop-Butucaru 2002]), before and after reincarnation, using both the algorithms of Sections 6.3 and 6.5. In the absence of static analysis, the expansion ratio is unacceptable. With static analysis however, it remains low in practice[11].

[10]The `sc6` file format defines a normalized circuit representation, which can in turn be converted into C programs by existing tools [INRIA et al. 2000].
[11]For the "global" example for instance, we could achieve an even tighter expansion by implementing dead code elimination at the source level (instead of constant propagation in Step 5, i.e. circuit level). "P18" is designed to trigger as much expansion as possible for its size.

We remark also that the circuit size reduction performed by the standard boolean optimizations of ESTEREL compilers is typically linear in all cases. In particular, such optimizations cannot undo the excessive unfolding of unoptimized reincarnation algorithms. Static analysis at the source level is mandatory for the production of small circuits.

What makes our compiler architecture really attractive in our view is the combination of the following properties:

—Step 4 is completely independent from Step 3. In other words, the compilation phase does not need to know anything about the static analysis/reincarnation phase. They can be implemented independently.

—Step 3 output being an ESTEREL* program, is still essentially an ESTEREL program, as ESTEREL* preserves the syntax and the semantics of ESTEREL. Instead of having to cope with schizophrenia, a programmer (respectively algorithm) just has to understand (respectively accept) simple, fully formalized `gotopause` statements.

—Even with this complete separation, the generated code is good. Quadratic worst-case complexity is standard [Berry 1999; INRIA et al. 2000; Schneider and Wenz 2001; Potop-Butucaru 2002; Edwards et al. 2004]. Thanks to static analysis, our algorithm is quasi-linear in practice. In particular, it is just as effective[12] as the reference compiler for ESTEREL [INRIA et al. 2000] which internally uses a static analysis of equal power, but much less formalized, and much more complex.

While former compiler architectures have already exposed some of these benefits, ours is the first one to gather them all. In particular, in order to add a native fast C backend to our compiler, we shall reuse the frontend made of the full first three steps of our current compilation chain.

9. CONCLUSION

Synchronous languages like ESTEREL have simple and solid mathematical foundations, that make formal reasoning tractable. In particular, logic synthesis or code generation techniques can be seen as chains of semantic-preserving transformations amenable to certification by formal proof.

In this work, we provide a formal description of one of the key steps of the compilation process of ESTEREL: the treatment of loops. Even if its relevance might seem limited to very specific ESTEREL paradigms in the first place, we feel that it establishes a clear link between two distinct representation levels, ubiquitous in the embedded world: a *dynamic* one where *resources* (time, space, etc.) are allocated on demand, and a *static* one where *resources* are fixed, and *behaviors* are no longer allowed to request more.

In practice, we move from one representation level where loops have to be dynamically unfolded *instantly* in the execution of the program, to another one where unfolding has been completed statically, so that dynamic unfolding is no longer necessary. In particular, behaviors that might require infinite unfolding to react to new inputs, i.e. request an infinite amount of time, are discarded as incorrect.

In ESTEREL vocabulary, this means we have:

[12]Both the circuits produced and the duration of the synthesis are essentially the same.

—rejected potentially instantaneous loops, using static analysis techniques;
—unfolded potentially schizophrenic loops, using both program rewriting techniques and static analysis to optimize the rewriting.

Our transformation is:

—very efficient: to the best of our knowledge, no better algorithm is available (formalized or as part of an implementation).
—provably correct: the skeleton of the proof is sketched in the paper.

Instantaneous loop issues have been discussed in the context of many synchronous formalisms or languages, schizophrenia problems in a few, for instance in QUARTZ [Schneider and Wenz 2001]. Similar attempts at the verification of the compiling process (including schizophrenia) have been reported in [Schneider 2000; Schneider et al. 2004]. These works focus on the embedding of a proof for a complete but pessimistic circuit synthesis in the HOL theorem prover, while, in the current paper, we concentrate on one part of the translation, but with efficiency in mind. We have already started discussing the integration of our results into their system.

Our transformation is essentially a source to source transformation, based on a new ESTEREL non-instantaneous jump primitive: `gotopause`. Therefore, this *preprocessing* can be used for logic synthesis as well as fast C code generation. Moreover, we believe the output of this preprocessor is quite valuable, as for instance, debugging a non-schizophrenic program, which is to say a program in some kind of *Single Static Assignment* form, is much nicer than working on the original program.

Thanks to `gotopause`, compiling ESTEREL* and compiling *to* ESTEREL* are respectively easier than compiling ESTEREL and compiling *to* ESTEREL, because of (*i*) the efficient preprocessing of loops in ESTEREL*, and (*ii*) the efficient encoding of state machines in ESTEREL*.

A prototype compiler embodying our current results has been realized.

REFERENCES

ANDRÉ, C. 1996a. Representation and analysis of reactive behaviors: A synchronous approach. In *Proceedings of the IMACS/IEEE SMC Multiconference on Computational Engineering in Systems Applications*. IEEE-SMC, Lille, France, 19–29.

ANDRÉ, C. 1996b. SyncCharts: a visual representation of reactive behaviors. Tech. Rep. RR 95–52, rev. RR 96–56, I3S, Sophia-Antipolis, France.

BARENDREGT, H. P. 1981. *The Lambda Calculus, Its Syntax and Semantics*. Studies in Logics and the Foundations of Mathematics, vol. 103. North-Holland, Amsterdam, The Netherlands.

BENVENISTE, A., CASPI, P., EDWARDS, S., HALBWACHS, N., LE GUERNIC, P., AND DE SIMONE, R. 2003. The synchronous languages twelve years later. *Embedded Systems, Proceedings of the IEEE, Special issue 91,* 1, 64–83.

BERRY, G. 1999. The constructive semantics of pure Esterel. http://www-sop.inria.fr/esterel.org/.

BERRY, G. 2000a. The Esterel language primer v5_91. http://www-sop.inria.fr/esterel.org/.

BERRY, G. 2000b. The foundations of Esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, Cambridge, MA, 425–454.

BERRY, G. AND COSSERAT, L. 1984. The synchronous programming language Esterel and its mathematical semantics. In *Seminar on Concurrency*. Lecture Notes in Computer Science, vol. 197. Springer, Pittsburg, PA, 389–448.

Berry, G. and Gonthier, G. 1992. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming 19,* 2, 87–152.

Boussinot, F. and de Simone, R. 1991. The Esterel language. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue 79,* 9, 1293–1304.

Brayton, R. K., Hachtel, G. D., Sangiovanni-Vincentelli, A., Somenzi, F., Aziz, A., Cheng, S. T., Edwards, S., Khatri, S., Kukimoto, Y., Pardo, A., Qadeer, S., Ranjan, R. K., Sarwary, S., Shiple, T. R., Swamy, G., and Villa, T. 1996. VIS: a system for verification and synthesis. In *Proceedings of the Eighth International Conference on Computer Aided Verification*. Lecture Notes in Computer Science, vol. 1102. Springer, New Brunswick, NJ, 428–432.

Clement, D. and Incerpi, J. 1989. Specifying the behavior of graphical objects using Esterel. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development*. Lecture Notes in Computer Science, vol. 352. Springer, Barcelona, Spain, 111–125.

Closse, E., Poize, M., Pulou, J., Venier, P., and Weil, D. 2002. Saxo-RT: Interpreting Esterel semantics on a sequential execution structure. In *Proceedings of the Synchronous Languages, Applications, and Programming Workshop*. Electronic Notes in Theoretical Computer Science, vol. 65. Elsevier, Grenoble, France.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programmings Languages*. ACM Press, Los Angeles, CA, 238–252.

Edwards, S. A. 2000. *Languages for Digital Embedded Systems*. Kluwer Academic Publishers, Norwell, MA.

Edwards, S. A., Kapadia, V., and Halas, M. 2004. Compiling Esterel into static discrete-event code. In *Proceedings of the Synchronous Languages, Applications, and Programming Workshop*. Electronic Notes in Theoretical Computer Science. Elsevier, Barcelona, Spain.

Gonthier, G. 1988. Sémantique et modèles d'exécution des langages réactifs synchrones: application à Esterel. Ph.D. thesis, Université d'Orsay.

Halbwachs, N. 1992. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, Norwell, MA.

Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. 1991. The synchronous dataflow programming language Lustre. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue 79,* 9, 1305–1320.

Hardin, R., Har'El, Z., and Kurshan, R. 1996. COSPAN. In *Proceedings of the Eighth International Conference on Computer Aided Verification*. Lecture Notes in Computer Science, vol. 1102. Springer, New Brunswick, NJ.

INRIA, ENSMP, and ARMINES. 2000. The Esterel v5_92 Compiler. http://www-sop.inria.fr/esterel.org/.

Malik, S. 1993. Analysis of cyclic combinational circuits. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*. IEEE, Santa Clara, CA, 618 – 625.

Maraninchi, F. 1991. The Argos language: Graphical representation of automata and description of reactive systems. In *Proceedings of the IEEE Workshop on Visual Languages*. IEEE, Kobe, Japan.

McMillan, K. 1993. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA.

Mignard, F. 1994. Compilation du langage Esterel en systèmes d'équations booléen-nes. Ph.D. thesis, Ecole des Mines de Paris.

Namjoshi, K. S. and Kurshan, R. P. 1999. Efficient analysis of cyclic definitions. In *Proceedings of the 11th International Conference on Computer Aided Verification*. Lecture Notes in Computer Science, vol. 1633. Springer, Trento, Italy, 394–405.

Park, D. 1981. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*. Lecture Notes in Computer Science, vol. 104. Springer, Karlsruhe, Germany, 167 – 183.

Plotkin, G. 1981. A structural approach to operational semantics. Report DAIMI FN-19, Aarhus University, Denmark.

Potop-Butucaru, D. 2002. Optimizations for faster execution of Esterel programs. Ph.D. thesis, Ecole des Mines de Paris.

Schneider, K. 2000. A verified hardware synthesis of Esterel programs. In *Proceedings of the IFIP WG10.3/WG10.4/WG10.5 International Workshop on Distributed and Parallel Embedded Systems: Architecture and Design of Distributed Embedded Systems*. Kluwer, B.V., Deventer, The Netherlands, 205–214.

Schneider, K., Brandt, J., and Schüle, T. 2004. A verified compiler for synchronous programs with local declarations. In *Proceedings of the Synchronous Languages, Applications, and Programming Workshop*. Electronic Notes in Theoretical Computer Science. Elsevier, Barcelona, Spain.

Schneider, K. and Wenz, M. 2001. A new method for compiling schizophrenic synchronous programs. In *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM Press, New York, NY, 49–58.

Shiple, T., Berry, G., and Touati, H. 1996. Constructive analysis of cyclic circuits. In *Proceedings of the International Design and Testing Conference*. IEEE, Paris, France, 328–333.

Tardieu, O. 2004a. Goto and concurrency: Introducing safe jumps in Esterel. In *Proceedings of the Synchronous Languages, Applications, and Programming Workshop*. Electronic Notes in Theoretical Computer Science. Elsevier, Barcelona, Spain.

Tardieu, O. 2004b. Loops in Esterel: from operational semantics to formally specified compilers. Ph.D. thesis, Ecole des Mines de Paris. http://olivier.tardieu.free.fr/papers/these.pdf.

Tardieu, O. and de Simone, R. 2003. Instantaneous termination in pure Esterel. In *Proceedings of the 10th International Symposium on Static Analysis*. Lecture Notes in Computer Science, vol. 2694. Springer, San Diego, CA, 91–108.

Tardieu, O. and de Simone, R. 2004. Curing schizophrenia by program rewriting in Esterel. In *Proceedings of the Second ACM-IEEE International Conference on Formal Methods and Models for Codesign*. IEEE, San Diego, CA.

Received April 2004; revised January 2005; accepted January 2005