Curing Schizophrenia by Program Rewriting in Esterel

Olivier Tardieu
INRIA Sophia Antipolis, France
olivier.tardieu@sophia.inria.fr

Robert de Simone
INRIA Sophia Antipolis, France
robert.de_simone@sophia.inria.fr

Abstract

Synchronous languages such as Esterel can execute a series of statements in a single "instant" of time. If this series spans a loop iteration then it is possible that a computation local to the loop will have several distinct results during that "instant", which is referred to as schizophrenia. This makes the compilation of synchronous languages into more traditional computation models (such as C code or sequential logic) difficult. In a previous work [17], we suggested to deal with schizophrenia through preprocessing in the Esterel language extended with a non-instantaneous jump statement. We now advocate for and experiment with such a program transformation, establishing the correctness, the completeness and the efficiency of our approach.

1. Introduction

In the previous decades the design of hardware circuits benefited greatly from synchronous methods, providing automated tools and techniques for the mapping of RTL descriptions down to optimized, placed-and-routed silicon implementations. This was made possible through the commonality of underlying semantic models (Mealy machines, netlists, sequential logic). Meanwhile the higher-level description models, used in simulation and "golden" specifications phases, often followed different paradigms.

We view the synchronous reactive language Esterel as a high-level structured language which might reconcile the upper-level description purpose with the lower-level strong semantics ties with mathematical models allowing silicon synthesis. It enjoys a rich set of imperative constructs including sequential and parallel compositions, tests, loops, preemption and suspension mechanisms, as well as local declarations of variables and exceptions. Its semantics, while being very expressive, is nevertheless fully synthesisable and also amenable to efficient software generation.

In order to fulfill this program a number of issues have to be solved. In synchronous languages as in synchronous hardware circuits time flows as a succession of discrete instants. In synchronous circuits each gate or wire may only assume one single value at each clock cycle (or at least converge to one unique stable value). It is not the case in synchronous languages, due to two phenomena known as causality cycles and schizophrenia. We focus here on the second issue.

In Esterel, loops may terminate and be reentered within the same instant (clock cycle). Therefore, parts of loop bodies may be executed several times during one instant. We say that they may have a schizophrenic behavior.

Because of schizophrenia, compiling Esterel is difficult. The fact is well established. In particular, the Esterel to circuits translation of Berry [1, 3], the Quartz compiler of Schneider [16], the Esterel to C compilers of Edwards [10] and Potop [14] are linear in the absence of schizophrenia problems, but quadratic in the worst cases of schizophrenia.

Such a complexity is neither specific to circuit generation, nor a particular problem of Esterel. It typically arises from the combination of local declarations with the synchronous paradigm. Local declarations are mandatory for the sake of the programmer, for clear and compositional programming. As a consequence, while hard schizophrenia problems are reported to be rare [3], schizophrenia has to be correctly handled by compilers.

Schizophrenia may be handled by the mean of simple loop unrolling (as the number of instantaneous executions of a statement is statically bounded). This however generates unnecessarily large codes and circuits (exponential worst-case growth).

Complex quadratic schemes have been developed to compile schizophrenic programs more efficiently. They typically require the expansion of Esterel into ad hoc intermediate formats, with fuzzy semantics. The most efficient algorithm we are aware of, implemented in the Esterel to circuits compiler from Ecole des Mines and INRIA [1], has not been successfully adapted to software generation yet. Thus, the need for a simple, formalized, efficient and generic algorithm remains.

In the sequel, we shall describe a new program transformation that rewrites any Esterel program into a non-

0-7803-8509-8/04/$20.00

`nothing`	do nothing
`pause`	await next instant
`signal` S `in` p `end`	declare S in p
`emit` S	emit S
`present` S `then` p `else` q `end`	if S is emitted...
`trap` T `in` p `end`	declare/catch T
`exit` T	raise exception T
p`;` q	p followed by q
`[`p `||` q`]`	p parallel q
`loop` p `end`	repeat p forever

Figure 1. Pure Esterel

schizophrenic semantically equivalent program in the Esterel language extended with a simple non-instantaneous jump statement, which we introduced and formalized in [17]. By selectively applying this transformation to program blocks selected by the static analysis we specified in [18], we obtain a simple and efficient rewriting scheme.

The paper is organized as follows: in Section 2 we briefly describe the Esterel language. We formally define schizophrenia in Section 3. In Section 4, we introduce our preprocessing for a kernel language. We combine it with static analysis in Section 5 and extend it to the full language in Section 6. We discuss our implementation in Section 7.

2. Esterel

Esterel [3, 4, 5, 7, 8] is a concurrent programming language dedicated to reactive systems [9, 12]. It was born in the eighties [6], and evolved since. In this work, we consider the Esterel v5 dialect of Esterel [4], accepted by the last academic compilers available from Ecole des Mines and INRIA [1] or Columbia University [10].

Pure Esterel is the fragment of Esterel where data variables and data handling primitives are discarded. We shall first concentrate on Pure Esterel (Sections 2.1 to 5).

Full Esterel nevertheless introduces an additional difficulty: data are persistent. The program transformation we initially define on Pure Esterel programs has to be further refined to handle data. We shall return to Full Esterel and address persistence in Section 6.

2.1. Syntax and Intuitive Semantics

An Esterel program runs in steps called *reactions* in response to the *ticks* of a *global clock*. Each reaction takes one *instant*. Primitive statements execute in zero-time except for one `pause` instruction. When the clock ticks, a reaction occurs. It may either finish the execution instantly or delay (part of) it till the next tick, because of `pause` instructions.

Without loss of generality, we focus on the kernel of Pure Esterel as defined by Berry in [3], which retains just enough of the Pure Esterel language syntax to attain its full expressive power. In addition, we do not consider the `suspend` statement in the sequel. It raises no particular problem.

Figure 1 describes the grammar of this language, as well as the intuitive behavior of its constructs. The non-terminals p and q denote *statements* (i.e. *programs*), S *signals* and T *exceptions*. Signals and exceptions are identifiers lexically scoped and respectively declared within statements by `signal` and `trap` instructions.

2.2. Examples

Program 1 of Figure 2 emits A in the first instant of its execution, then emits B and C in the second instant, then emits D and terminates in the third instant. It takes three instants to complete (three reactions).

Program 2 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. Execution propagates in parallel branches in a deterministic synchronous way: one reaction of the parallel composition is made of exactly one reaction of each branch, until the termination of all branches.

The `exit` statement behaves as a goto to the end of the `trap` block. An exception occurring in a parallel statement causes it to terminate instantly. In Program 3, A and D are emitted in the first instant, then B and E in the second and final instant. As expected "`emit C`" is never reached. Note that "`exit T`" does not prevent B to be emitted: exceptions implement *weak preemption*. Exception declarations may be nested. The outermost exception has priority.

"`loop emit S; pause end`" emits S at each instant. It never terminates. Finite loops may be obtained by combining `loop` and `trap` statements. Loop bodies may not be *instantaneous*. They must either retain the control for at least one instant or raise an exception. For example, "`loop emit S end`" is not a correct program. Such a pattern would prevent the reaction to reach completion. Therefore, it is forbidden (cf. Section 2.4).

Program 4 does not emit O, as S is not emitted at the time of the test. In an instant, a signal S is either *present* or *absent*. A signal is present iff explicitly emitted (or set by the *environment* in the case of unbound signals). In particular, the *status* of a signal does not depend on its status at the previous instant.

In Program 5, O is emitted *because* of S, but not *after* S. Both are present for the duration of the reaction.

2.3. Formal Semantics

The *logical semantics* of Esterel [3] defines the reactions of a program p via a labeled transition system:

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

1. `emit A; pause; emit B; emit C; pause; emit D`
2. `pause; emit A; pause; emit B || emit C; pause; emit D`
3. `trap T in [emit A; pause; emit B; pause; emit C || emit D; pause; exit T] end; emit E`
4. `signal S in emit S; pause; present S then emit O end end`
5. `signal S in [present S then emit O end || emit S] end`

Figure 2. Examples

The integer k is the *completion code* of the reaction, the program p' is its *residual*:

- 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:
  - $k = 0$ if the execution completes normally,
  - $k \geq 2$ if some exception[1] escapes from p.

The sets E of *present signals* and E' of *emitted signals* encode the I/Os of the reaction. The set E regroups the signals emitted by p, thus $E' \subset E$, and by the environment. In other words, p reacts to *inputs* I with *outputs* O iff:

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

An execution of p is a potentially infinite chain of reactions, such that all completion codes but the last one are equal to 1. For example, in the finite case:

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

Figure 3 expresses the logical semantics of Esterel as a set of deduction rules in a structural operational style. The extra labels (top right-most position, and labels on signal and parallel statements) will be discussed in Section 3.3.

For example "`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:

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

2.4. Logical Correctness and Causality

A program is said to be *logically correct* iff there exists exactly one deduction tree defining its reaction at any stage of any execution (that is to say after any number of reactions and for any sequence of inputs). This is not always so:

- `loop nothing end` is incorrect as no rule applies. Instantaneous loop bodies ($k = 0$) are forbidden.
- `signal S in present S else emit S end end` is incorrect since no deduction tree may be built for the empty set of inputs (for example). On the one hand, if we suppose S absent for the duration of the reaction then it is emitted. On the other hand, if we suppose S present then it has no emitter.
- `signal S in present S then emit S end end` is incorrect since two trees may be built for the empty set of inputs. Intuitively, there is no way to tell whether S is absent or present.

From now on, we shall only consider logically correct programs. By definition, they have *reactive* (deadlock free) and *deterministic* executions.

While logical correctness ensures that valuations of signals are unique, it 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:

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

The logical semantics of Esterel can be refined into various semantics formalizing causality, such as the *constructive semantics* [3]. Causality and causality analysis have been extensively discussed. In the sequel, we shall consider schizophrenia in the framework of the logical semantics as we do not want to depend on causality analysis.

3. Schizophrenia

In Esterel, it may be possible for a statement to terminate or exit and be restarted during the same reaction, that is to say *instantly restarted* or *instantly reentered*.

3.1. Schizophrenic Signal Declarations

In the following example, the signal O is never emitted, as the test always considers a fresh signal S, which is not emitted at the time of the test.

[1] Exceptions are numbered ($T \mapsto k_T$) according to priorities [3, 11].

$$\texttt{nothing} \xrightarrow[E]{\emptyset,0,\emptyset} \texttt{nothing}$$

$$\texttt{pause} \xrightarrow[E]{\emptyset,1,\emptyset} \texttt{nothing}$$

$$\texttt{exit}\ T \xrightarrow[E]{\emptyset,k_T,\emptyset} \texttt{nothing}$$

$$\texttt{emit}\ S \xrightarrow[E]{\{S\},0,\emptyset} \texttt{nothing}$$

$$\frac{p \xrightarrow[E]{E',k,L} p' \quad k \neq 0}{\texttt{loop}\ p\ \texttt{end} \xrightarrow[E]{E',k,L} p';\ \texttt{loop}\ p\ \texttt{end}}$$

$$\frac{p \xrightarrow[E]{E',k,L} p' \quad q \xrightarrow[E]{F',l,L'} q' \quad m = \max(k,l)}{[p\ ||\ q]^n \xrightarrow[E]{E'\cup F',m,L\uplus L'\uplus\{n\}} [p'\ ||\ q']^n}$$

$$\frac{S \in E \quad p \xrightarrow[E]{E',k,L} p'}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \xrightarrow[E]{E',k,L} p'}$$

$$\frac{p \xrightarrow[E]{E',k,L} p' \quad k = k_T}{\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end} \xrightarrow[E]{E',0,L} \texttt{nothing}}$$

$$\frac{S \notin E \quad q \xrightarrow[E]{F',l,L} q'}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \xrightarrow[E]{F',l,L} q'}$$

$$\frac{p \xrightarrow[E]{E',k,L} p' \quad k \neq k_T}{\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end} \xrightarrow[E]{E',k,L} \texttt{trap}\ T\ \texttt{in}\ p'\ \texttt{end}}$$

$$\frac{p \xrightarrow[E]{E',0,L} p' \quad q \xrightarrow[E]{F',l,L'} q'}{p;\ q \xrightarrow[E]{E'\cup F',l,L\uplus L'} q'}$$

$$\frac{p \xrightarrow[E\cup\{S\}]{E',k,L} p' \quad S \in E'}{\texttt{signal}^n\ S\ \texttt{in}\ p\ \texttt{end} \xrightarrow[E]{E'\setminus\{S\},k,L\uplus\{n\}} \texttt{signal}^n\ S\ \texttt{in}\ p'\ \texttt{end}}$$

$$\frac{p \xrightarrow[E]{E',k,L} p' \quad k \neq 0}{p;\ q \xrightarrow[E]{E',k,L} p';\ q}$$

$$\frac{p \xrightarrow[E\setminus\{S\}]{E',k,L} p' \quad S \notin E'}{\texttt{signal}^n\ S\ \texttt{in}\ p\ \texttt{end} \xrightarrow[E]{E',k,L\uplus\{n\}} \texttt{signal}^n\ S\ \texttt{in}\ p'\ \texttt{end}}$$

Figure 3. Formal Semantics

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

Starting from the second instant of execution, each reaction involves two instances or *incarnations* of the signal S: an *old* incarnation inherited from previous instant (as `pause` is in the scope of the declaration of S) and a *new* incarnation since this scope is left and instantly reentered.

Gates and wires in synchronous circuits may only assume one single value at each clock cycle. Thus, compiling synchronous languages with instantly reentered local declarations – including Esterel – to circuits requires to carefully distinguish between these incarnations.

Compiling Esterel to C may seem easier. We could translate Esterel local signal declarations into C local variable declarations. In such a scheme, the *old* incarnation of S would be computed first, then the *new* one. While this makes sense for the current example, it fails in general.

In the following example, while T is declared in sequence with S, the status of S cannot be decided without knowing the status of A, thus without looking *forward* at the "`signal T in ... end`" block.

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

By folding this code using a `loop` statement, we can merge the roles of S and T and obtain a program where the the value of the *old* incarnation of S depends on the value 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
```

In summary, compiling instantly reentered signal declarations, that is to say *schizophrenic signal declarations*, to software code or synchronous hardware is difficult[2].

[2]On the other hand, a simple translation of local signals into C *global* variables is fine, as far as no local signal declaration is instantly reentered.

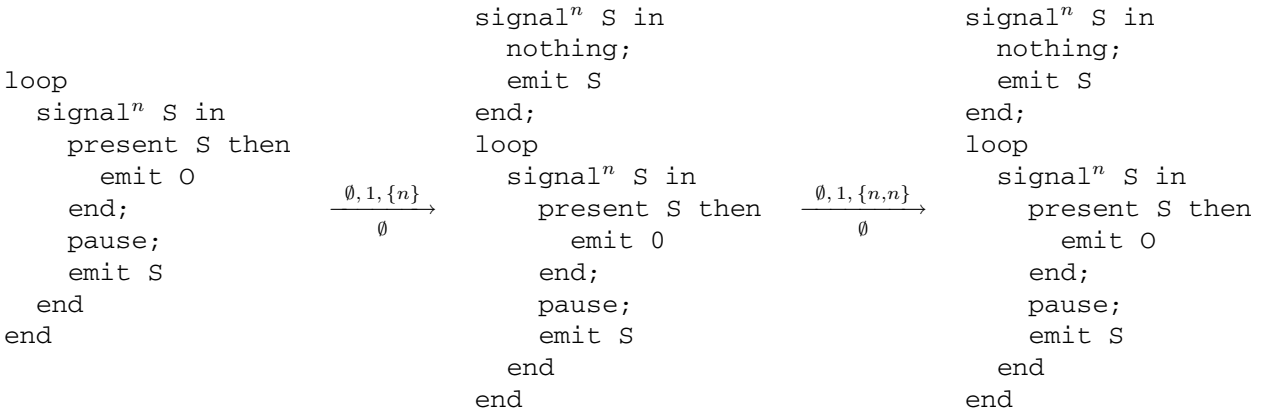

Figure 4. Schizophrenia

3.2. Schizophrenic Parallel Statements

Parallel statements may also terminate and be instantly reentered, leading to complex behaviors. In the following example, if input I is first present then absent, variable V is incremented twice in the second instant of execution:

```
loop
  present I then pause end; V:=V+1
||
  pause
end
```

For reasons similar to those above (in Pure and Full Esterel), the behaviors of the several instances of an instantly reentered parallel statement may be interdependent. Therefore, we shall also identify instantly reentered parallel statements, which we call *schizophrenic parallel statements*.

3.3. Formal Characterization

Instantly reentered statements generates intricate interactions between they distinct incarnations. This point is often discussed [3, 13, 14, 15, 16, 18]. Beyond examples of so-called schizophrenic behaviors, a few informal characterizations of schizophrenia in Esterel have been described (schizophrenic signals, programs...), or could be derived from these discussions.

For example, schizophrenic programs are sometimes defined as those programs that are not correctly compiled by some *naive* compilation scheme [3]. Schizophrenic signal declarations may be defined as those which do not commute with enclosing loops [13]. *Potentially* schizophrenic programs may also be defined by the mean of some static analysis [18], etc. But to the best of our knowledge no simple formal definition of schizophrenia has been proposed yet. We shall provide one here.

In Figure 3, we have instrumented Esterel logical semantics in order to formally characterize schizophrenic programs as follows:

- We label signal declarations and parallel statements:
  - `signal`^{label} ... `in` ... `end`
  - `[`... `||` ...`]`^{label}
- We add an extra component L to the labeled transition system (top right-most position in Figure 3): the *multiset of labels* of a reaction, obtained by collecting the labels of the statements executed during the reaction.
- We preserve labels in the rewriting which produces the residual of the reaction.

We define:

- **Schizophrenic reaction:** a reaction is said to have a schizophrenic behavior iff its multiset of labels contains a repeated label.
- **Schizophrenic program:** a logically correct program is said to be schizophrenic iff, being initially labeled with pairwise distinct labels, there exists an execution involving a schizophrenic reaction. Remark that this definition does not depend on the initial labeling.

Figure 4 illustrates the process on our very first example of schizophrenic program (instantly reentered signal declaration). Initially, the program contains a unique signal declaration, which we label n. The first reaction involves this signal declaration, thus the multiset $\{n\}$. Because loop unrolling occurs, the residual now contains two declarations with label n. In the second instant and thereafter, the label n is encountered twice, leading to the multiset $\{n, n\}$.

4. Reincarnation

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

4.1. Naive Reincarnation

Rewriting schizophrenic programs into equivalent non-schizophrenic programs has been proposed in [13]. The method consists of recursively *dup*licating loop bodies:

$$\texttt{nothing} \overset{dup}{\Longrightarrow} \texttt{nothing}$$
$$\texttt{signal } S \texttt{ in } p \texttt{ end} \overset{dup}{\Longrightarrow} \texttt{signal } S \texttt{ in } dup(p) \texttt{ end}$$
$$\ldots$$
$$\texttt{loop } p \texttt{ end} \overset{dup}{\Longrightarrow} \texttt{loop } dup(p)\texttt{; } dup(p) \texttt{ end}$$

This program transformation is called *reincarnation* as it explicitly distributes the several *incarnations* 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, it can be shown that p and $dup(p)$ behave the same in all contexts (technically, they are strongly bisimilar [2, 17] with respect to Esterel logical semantics).

In particular, if p cannot terminate instantly then $dup(p)$ cannot either. As a consequence, each loop body of a rewritten logically correct 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.

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 4.4).

4.2. Esterel^{sp}

In order to get rid of schizophrenia by program rewriting, we propose to extend the Esterel language. We first add pairwise distinct labels[3] to `pause` statements, which we denote "$label$`: pause`". We then introduce in the language the instruction "`gotopause` $label$". It behaves as a goto to the `pause` statement with corresponding $label$. For example, S is not emitted by the reaction:

`gotopause 1; emit S; 1: pause; emit T`

$$\xrightarrow[\emptyset]{\emptyset,\,1} \texttt{nothing; emit T}$$

[3]These labels are not related to those we used to define schizophrenia.

We have formalized the semantics of the extended language, which we call Esterel^{sp} in [17]. Restrictions have to be imposed on the relative locations of `pause` and `gotopause` statements. Intuitively, concurrent jumps to sequential program blocks have to be forbidden. Otherwise, `gotopause` behaves just as expected. In particular, `gotopause` cannot contribute to instantaneous loops, causality problems or schizophrenia because these jumps have non-instantaneous effects only, the target always being a `pause` statement.

For lack of space and need, we do not go into the details of the semantics of Esterel^{sp} here. We shall introduce `gotopause` statements in Esterel programs in a controlled way, which ensures the correctness of the resulting Esterel^{sp} programs.

4.3. Reincarnation with Esterel^{sp}

In the sequel, we consider Esterel^{sp} programs, that is to say we suppose `pause` statements are always labeled with pairwise distinct labels. Thanks to `gotopause`, we shall now partially replicate loop bodies and define dup^{sp} by replacing the rule:

$$\texttt{loop } p \texttt{ end} \overset{dup}{\Longrightarrow} \texttt{loop } dup(p)\texttt{; } dup(p) \texttt{ end}$$

by rule:

$$\texttt{loop } p \texttt{ end} \overset{dup^{sp}}{\Longrightarrow} \texttt{loop } surf(p)\texttt{; } dup^{sp}(p) \texttt{ end}$$

Intuitively, we shall only replicate the *surface* of each loop body, that is to say the part of the loop body which is instantly reachable. Function *surf* is recursively defined:

$$\texttt{loop } p \texttt{ end} \overset{surf}{\Longrightarrow} surf(p)$$
$$p\texttt{; } q \overset{surf}{\Longrightarrow} \begin{array}{l}\text{if p may instantly terminate}\\ \quad\text{then } surf(p)\texttt{; } surf(q)\\ \quad\text{else } surf(p)\end{array}$$
$$label\texttt{: pause} \overset{surf}{\Longrightarrow} \texttt{gotopause } label$$

Omitted rules correspond to simple recursive calls. A loop body cannot be instantaneous, thus the first rule. If p in $p\,;q$ cannot be instantaneous then q cannot be reached instantly, thus the second rule.

At the end of the first instant of execution of a rewritten loop, the control goes from $surf(p)$ to the "regular" copy of p through `gotopause` statements (last rule). Therefore, the execution will be restarted the "right" state. In other words, provided that p is non-instantaneous, $surf(p)\,;p$ and p are equivalent (strongly bisimilar). By recursion, for all logically correct program p, the programs p and $dup^{sp}(p)$ behave the same.

Moreover, $surf(p)$ cannot terminate instantly if p cannot. Therefore, if p is logically correct then $dup^{sp}(p)$ is not schizophrenic.

p	Γ_p	Ω_p
`pause`	$\{1\}$	$\{0,1\}$
`loop` p `end`	if 0 in Γ_p then **error** else Γ_p	$\Omega_p \backslash \{0\}$
`nothing`	$\{0\}$	$\{0\}$
`signal` S `in` p `end`	Γ_p	Ω_p
`emit` S	$\{0\}$	$\{0\}$
`present` S `then` p `else` q `end`	$\Gamma_p \cup \Gamma_q$	$\Omega_p \cup \Omega_q$
p`;` q	if $0 \in \Gamma_p$ then $(\Gamma_p \backslash \{0\}) \cup \Gamma_q$ else Γ_p	if $0 \in \Omega_p$ then $(\Omega_p \backslash \{0\}) \cup \Omega_q$ else Ω_p
`[`p `||` q`]`	$\{\max(x,y), \forall x \in \Gamma_p, \forall y \in \Gamma_q\}$	$\{\max(x,y), \forall x \in \Omega_p, \forall y \in \Omega_q\}$
`trap` T `in` p `end`	Γ_p with substitution of k_T by 0	Ω_p with substitution of k_T by 0
`exit` T	$\{k_T\}$	$\{k_T\}$

Figure 5. Static Analysis of Completion Codes of Reactions (Γ) and Executions (Ω)

4.4. Algorithm

We obtain a reincarnation algorithm that rewrites any program p into a non-schizophrenic equivalent Esterel^{sp} program $dup^{sp}(\hat{p})$, by first labeling the `pause` statements of the program p with pairwise distinct labels – we denote the result with $\hat{p}$ – then computing the image of $\hat{p}$ by function dup^{sp}. Since $surf(\hat{p})$ does not contain loops, the rewriting of p into $surf(\hat{p})$ is linear, and $dup^{sp}(\hat{p})$ 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`

$\overset{dup^{sp}}{\Longrightarrow}$
`loop`
`[`$surf(p)$ `||` $surf(q)$`]`;
`[`$dup^{sp}(p)$ `||` `loop` $surf(q)$`;` $dup^{sp}(q)$ `end]`
`end`

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

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

$\overset{dup^{sp}}{\Longrightarrow} 2+3+4 = 9$ times p *(quadratic growth)*

5. Efficient Reincarnation

In the example of last section, the parallel statement cannot terminate, a fortiori be instantly restarted by the enclosing loop. Therefore, there is no schizophrenia here, and no need for program rewriting. In general, less expansion is possible, provided we do some program analysis.

5.1. Static Analysis

In [18], we formalized the analysis of schizophrenia implemented in the Esterel compiler from Ecole des Mines and INRIA [1]. It relies on the computation of *potential completion codes*, which we recall in Figure 5. We define by structural induction:

- The over approximation Γ_p of the completion codes the program p may produce in its first reaction[4]. If $p \xrightarrow[E]{E',k} p'$ then $k \in \Gamma_p$.
- The over approximation Ω_p of the completion codes the program p may produce at any stage of its execution. If $p \xrightarrow[E_1]{E'_1,1} \ldots \xrightarrow[E_n]{E'_n,k} p_n$ then $k \in \Omega_p$.

The fact that p may terminate or exit and be instantly restarted in some context depends on both p and the context. For example, if the execution of p non-instantly terminates with completion code k,

p is instantly restarted?	k=0	k=k_T
`loop trap T in` p`;pause end end`	*no*	*yes*
`trap T in loop` p `end end`	*yes*	*no*

We say that k_T in the first example, and 0 in the second one are *unsafe completion codes*. In [18] again, we expressed the computation of an over approximation of the set of unsafe completion codes of each statement of a program (more exactly of the context of each statement of the program). We do not recall it here as we shall embed this computation within our reincarnation algorithm (recursive computation of K in Figure 6).

If $K_{s/p}$ is the over approximation of the set of unsafe completion codes for the statement s within the program p, then s may potentially terminate and be instantly reentered in p if $K_{s/p} \cap \Omega_s$ is not empty. On the other hand, if $K_{s/p} \cap \Omega_s = \emptyset$ then s is not schizophrenic in p.

[4] If a loop may be instantaneous, the computation produces an **error**.

p	$dup^{sp}(K,p)$
`nothing`	`nothing`
label `: pause`	*label* `: pause`
`gotopause` *label*	`gotopause` *label*
`exit` T	`exit` T
`emit` S	`emit` S
`present` S `then` p `else` q `end`	`present` S `then` $dup^{sp}(K,p)$ `else` $dup^{sp}(K,q)$ `end`
`loop` p `end`	`loop` $dup^{sp}(K \cup \{0\},p)$ `end`
`trap` T `in` p `end`	`trap` T `in` $dup^{sp}(K[0 \to k_T],p)$ `end`
p`;` q	$dup^{sp}((\text{if } K \cap \Gamma_q = \emptyset \text{ then } K\backslash\{0\} \text{ else } K \cup \{0\}),p)$`;` $dup^{sp}((\text{if } 0 \in \Gamma_p \text{ then } K \text{ else } \emptyset),q)$
`signal` S `in` p `end`	if $K \cap \Omega_p = \emptyset$ then `signal` S `in` $dup^{sp}(\emptyset,p)$ `end` else `signal` S `in` $surf(p)$ `end;` `signal` S `in` $dup^{sp}(\emptyset,p)$ `end`
`[`p `||` q`]`	if $K \cap \Omega_{\texttt{[}p \texttt{ \|\| } q\texttt{]}} = \emptyset$ then `[`$dup^{sp}(\emptyset,p)$ `||` $dup^{sp}(\emptyset,q)$`]` else `[`$surf(p)$ `||` $surf(q)$`];` `[`$dup^{sp}(\emptyset,p)$ `||` $dup^{sp}(\emptyset,q)$`]`

Figure 6. Reincarnation with Static Analysis

5.2. Combining Analysis and Reincarnation

Schizophrenia arises from the nesting of signal declarations or parallel statements within loops. Instead of *systematically* unrolling *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 to a new definition of dup^{sp} in Figure 6, *surf* remaining unchanged.

Function dup^{sp} is now context-dependent. It takes two arguments: the statement p to rewrite and the (initially empty) set K of potentially unsafe completion codes for p.

As announced, loops no longer replicate code on their own. Moreover, signal declarations and parallel statements are now selectively expansed if potentially schizophrenic ($K \cap \Omega_{\ldots} \neq \emptyset$).

5.3. Algorithm

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 $dup^{sp}(\emptyset,\hat{p})$. For example,

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

is rewritten into:

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

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

Again, $dup^{sp}(\emptyset,\hat{p})$ 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 7, and in the absence of potentially schizophrenic statements, there is no expansion at all[5].

6. Reincarnation in Full Esterel

Full Esterel adds to Pure Esterel the ability to manipulate data of various kinds. 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.

[5] In particular, the function $p \mapsto dup^{sp}(\emptyset,p)$ is idempotent.

Let's consider an example where a variable is declared within a schizophrenic signal scope (with a `var` statement):

```
loop
  signal S in
    var V in ...; pause; ... end
  end
end
```

is rewritten into:

```
loop
  signal S in
    var V in ...; gotopause 1; ... end
  end;
  signal S in
    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 in the abstract syntax tree, using alpha-renaming when needed. In our example, we obtain:

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

Second, we may introduce *static aliasing* in Esterel^{sp}, expressing that the two distinct declarations of V in the naive rewriting in fact correspond to a unique memory cell. We may index variables before expansion for example, 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.

7. Implementation

We have designed an algorithm that translates any Esterel program into a non-schizophrenic equivalent Esterel^{sp} program, by reshaping potentially 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[6]. 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 algorithm we described in previous section. Step 4 essentially implements the naive Esterel to Circuits translation (for non-schizophrenic programs) of Berry [3], in which we incorporate `gotopause` and data.

Compiling `gotopause` is straightforward:

- In our circuit generator, 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.
- In general, Esterel compilers are based on internal representations of programs as graphs, that are expressive enough to directly support the addition of `gotopause`.

We have conducted some early experiments, summarized in Table 1. We count the number of statements (after macro expansion) in programs of various kinds and sizes (from [3] and [14]), before and after reincarnation, using both the algorithms of Sections 4.4 and 5.3. In the absence of static analysis, the expansion ratio is unacceptable. With static analysis however, it remains low in practice[7].

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[8].

[6] Supported by Ecole des Mines and INRIA, the `sc6` format defines a normalized circuit representation, which can in turn be converted into C programs by existing tools [1].

[7] We expect to achieve a tighter expansion with dead code elimination in "global" (not implemented yet). The "p18" program is designed to trigger as much expansion as possible for its size.

[8] Of course, a shared parser is a good idea.

	no reincarnation	algorithm (4.4)	algorithm (5.3)	description
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	multiple reincarnation
abro	14	18	14	tiny synchronization protocol

Table 1. Numbers of Kernel Statements

- Step 3 output being an Esterel^{sp} program, is still essentially an Esterel program, as Esterel^{sp} preserves the syntax and the semantics of Esterel. Instead of having to cope with schizophrenia, one 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 [1, 10, 3, 14, 16]. Thanks to static analysis, our algorithm is quasi-linear in practice. In particular, it is just as effective[9] as the compiler from Ecole des Mines and INRIA [1] which internally uses a static analysis of equal power.

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.

8. Conclusion

While semantics usually define loops via loop unrolling, i.e. code replication, efficient implementations replace them by iterated traversal of a single loop body. In the case of synchronous languages however the trick does not work because of *schizophrenia* problems.

In this paper, we thoroughly study schizophrenia in Esterel, starting from definition up to implementation. Our key contributions are: (i) a simple characterization of schizophrenia based on execution traces, (ii) a program transformation that completely gets rid of schizophrenia at the expense of the introduction of a non-instantaneous goto in Esterel. Our preprocessing is generic and efficient. It can be used as a starting point for the production of efficient software code as well as efficient synchronous hardware. It is also fully formalized, thus amenable to certified code generation.

[9] Both the circuits produced and the complexity (duration) of the compilation are essentially the same.

References

[1] The Esterel v5_92 Compiler. http://www-sop.inria.fr/esterel.org/.

[2] A. Arnold and I. Castellani. An algebraic characterization of observational equivalence. *Theoretical Computer Science*, 156:289–299, 1996.

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

[4] G. Berry. The Esterel language primer v5_91. http://www-sop.inria.fr/esterel.org/, 2000.

[5] G. Berry. The foundations of Esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[6] G. Berry and L. Cosserat. The synchronous programming language ESTEREL and its mathematical semantics. *LNCS*, 197, 1984.

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

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

[9] S. A. Edwards. *Languages for Digital Embedded Systems*. Kluwer, 2000.

[10] S. A. Edwards, V. Kapadia, and M. Halas. Compiling Esterel into Static Discrete-Event Code. In *SLAP*, 2004.

[11] G. Gonthier. *Sémantique et modèles d'exécution des langages réactifs synchrones: application à Esterel*. PhD thesis, Université d'Orsay, 1988.

[12] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer, 1993.

[13] F. Mignard. *Compilation du langage Esterel en systèmes d'équations booléennes*. PhD thesis, Ecole des Mines de Paris, 1994.

[14] D. Potop-Butucaru. *Optimizations for Faster Execution of Esterel Programs*. PhD thesis, Ecole des Mines de Paris, 2002.

[15] K. Schneider. A verified hardware synthesis for Esterel. In *DIPES*, pages 205–214, 2000.

[16] K. Schneider and M. Wenz. A new method for compiling schizophrenic synchronous programs. In *CASES*, pages 49–58, 2001.

[17] O. Tardieu. Goto and concurrency: introducing safe jumps in Esterel. In *SLAP*, 2004.

[18] O. Tardieu and R. de Simone. Instantaneous termination in pure Esterel. In *SAS*, pages 91–108, 2003.