Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 203 (2008) 49–64
www.elsevier.com/locate/entcs

Instantaneous Transitions in Esterel

Olivier Tardieu[1,3]

INRIA, Sophia Antipolis

Stephen A. Edwards[2,4]

Columbia University in the City of New York

Abstract

Esterel is an imperative synchronous programming language for the specification of deterministic concurrent reactive systems. While providing the usual control-flow constructs—sequences, loops, conditionals, and exceptions—its lack of a goto instruction makes the programming of arbitrary finite state machines awkward and hinders the design of source-to-source program transformations. We previously introduced to Esterel a non-instantaneous gotopause instruction, which prevents the synchronous execution of code before and code after the transition. Here, we tackle instantaneous transitions. Concurrency demands we assign scopes and priorities to gotos, so we extend Esterel's exception handling mechanism to allow exception handlers in arbitrary locations. We advocate for and formalize the resulting language. We observe that instantaneous gotos complement but do not replace non-instantaneous gotopauses.

Keywords: concurrency, exceptions, SyncCharts, compilation.

1 Introduction

Esterel [3,4,5,6] is a concurrent programming language. Its syntax is imperative and fit for the design of control-oriented reactive systems [10]. Its semantics are synchronous: active threads run in lockstep and communicate via instantly broadcast signals. Like most modern imperative languages, Esterel promotes structured programming. Common programming practice strongly discourages the use of gotos when they are available [8], but Esterel provides none at all.

The lack of goto is not without reason. First, gotos and concurrency do not mix well and Esterel code is hardly ever sequential. Second, loops—simple forms

[1] Email: olivier.tardieu@sophia.inria.fr
[2] Email: sedwards@cs.columbia.edu
[3] Tardieu was at Columbia when this work was performed.
[4] Edwards and his group are supported by the NSF, Intel, Altera, the SRC, and NYSTAR.

doi:10.1016/j.entcs.2008.05.010

of jumps—already cause substantial trouble. To make a long story short, a complex loop unfolding algorithm—reincarnation [3,19]—is a mandatory step in the compilation of Esterel.

Nevertheless, the lack of a goto instruction is a drawback. Many standards explicitly prescribe (unstructured) state machines. For example, the link layer specification of the Serial ATA standard [16] specifies a 31-state machine by listing transitions in a table. To describe such machines, many formalisms, such as SyncCharts [1,2], provide graphical modeling of reactive systems using hierarchical and parallel compositions of finite state machines. While its synchronous semantics match those of Esterel, the translation from SyncCharts to Esterel is awkward and obfuscates the programmer's intent. Transitions are encoded with signaling. Arbitrary state machines can be encoded using one concurrent process per state. But maintaining structural information about exclusive states in the generated code is not easy. In contrast, a goto allows the direct encoding of transitions and the preservation of this information.

Internally, all Esterel compilers use ad hoc intermediate languages (e.g., IC [5] and GRC [14]) that expand Esterel control-flow constructs into jump instructions. This suggests adding gotos to Esterel should not only be feasible but also have a minor impact on code generation. While for code generation, it would be reasonable to translate formalisms such as SyncCharts directly to such internal formats, this would not help users reason about specifications.

Previously, we extended Esterel with a *gotopause* instruction [17]. By design, it ensures that one instant elapses between the execution of the jump instruction and the execution of the code following the target of the jump. Thanks to the definition of well-formed programs, we were able to specify non-instantaneous jumps that are consistent with the principles of deterministic synchronous concurrency. The delay implies their semantics do not involve unfolding, making compilation trivial.

Of course, non-instantaneous jumps are no help for the programming of finite state machines with instantaneous transitions. In this paper, we introduce instantaneous jumps, which we obtain by combining features of loops, exceptions, and non-instantaneous jumps. First, like exceptions, instantaneous jumps have scopes and are prioritized accordingly. In a series of concurrent jumps, all but the highest-priority jump are ignored. Second, as with loops, the semantics of instantaneous jumps rely on unfolding. Finally, the machinery for transferring control to a distant location in the source code already exists in the formal semantics of Esterel thanks to *gotopause*.

We introduce instantaneous jumps by extending the exception handling mechanism of Esterel. Raising an exception normally jumps to the end of the exception scope. Our extension makes it possible to place the exception handler, i.e., the target of the jump, at any point within the scope of the exception. This employs an explicit *catch* instruction, which behaves like a label.

While this "exception handler within a trap" construct may appear strange, simply taking a more traditional goto-and-label approach would come with too many caveats to be any simpler. This paper aims at understanding the interactions

statements	locations	compatible locations
$p, q ::=$ `nothing`	$\emptyset$	$\emptyset$
ℓ`:pause`	$\{\ell\}$	$\emptyset$
`gotopause` ℓ	$\emptyset$	$\emptyset$
p `;` q	$\mathcal{L}_p \cup \mathcal{L}_q$	$\mathcal{C}_p \cup \mathcal{C}_q$
p `||` q	$\mathcal{L}_p \cup \mathcal{L}_q$	$\mathcal{C}_p \cup \mathcal{C}_q \cup (\mathcal{L}_p \times \mathcal{L}_q) \cup (\mathcal{L}_q \times \mathcal{L}_p)$
`[`p`]`	$\mathcal{L}_p$	$\mathcal{C}_p$
`loop` p `end`	$\mathcal{L}_p$	$\mathcal{C}_p$
`signal` S `in` p `end`	$\mathcal{L}_p$	$\mathcal{C}_p$
`emit` S	$\emptyset$	$\emptyset$
`present` S `then` p `else` q `end`	$\mathcal{L}_p \cup \mathcal{L}_q$	$\mathcal{C}_p \cup \mathcal{C}_q$
`suspend` p `when` S	$\mathcal{L}_p$	$\mathcal{C}_p$
`trap` T `in` p `end`	$\mathcal{L}_p$	$\mathcal{C}_p$
`exit` T	$\emptyset$	$\emptyset$

Fig. 1. The syntax of Esterel. Compatible locations.

between concurrency and gotos to provide a formal framework that can be used to add a variety of jump constructs. What if a goto attempts to exit the scope of an exception? What if concurrent gotos target exclusive program states? Our design minimizes the change to the language and its semantics. We only suggest a general, low-level syntax. Additional syntactic sugar is probably necessary.

In particular, we show instantaneous gotos do not generalize non-instantaneous gotos but complement them: *gotopause* instructions are not simply instantaneous jumps plus delays.

We describe the syntax and semantics of the Esterel language and the *gotopause* instruction in Section 2. We introduce and formalize the *catch* instruction in Section 3. Through an example, we illustrate the encoding of state machines with instantaneous transitions. We also discuss loop elimination as an instance of a source-to-source program transformation relying on the new construct. We discuss related work in Section 4 and conclude in Section 5.

2 Esterel and *gotopause*

Without loss of generality, we focus on the pure Esterel language augmented with a *gotopause* instruction. The full language is obtained from its pure fragment by adding data-centric constructs irrelevant to our discussion.

2.1 Syntax and Intuitive Semantics

We describe the grammar of our kernel language in Fig. 1, Col. 1. The non-terminals p and q denote statements, S signal identifiers, T exception identifiers, and ℓ integer labels. The infix "`;`" operator binds tighter than "`||`."

In Cols. 2 and 3, we recursively define the locations $\mathcal{L}_p$ and the compatible locations $\mathcal{C}_p$ of the statement p. The locations of p are the labels of the *pause* instructions

in p. They must be pairwise distinct. Formally, in statements when both p and q occur, the sets $\mathcal{L}_p$ and $\mathcal{L}_q$ must be disjoint. For example, `1:pause ; 1:pause` is illegal. We discuss compatible locations later.

The execution of an Esterel program, i.e., a statement, consists of a possibly infinite sequence of atomic execution steps called reactions. Each reaction is said to last for one instant. *Pause* instructions represent reaction boundaries, i.e., the progress of time.

- `nothing` does nothing; terminates instantly, that is to say a statement immediately after this instruction is run instantly.
- ℓ`:pause` suspends the execution for one instant. The statement immediately after this instruction, if any, is run in the next instant of execution.
- `gotopause` ℓ suspends the execution for one instant. The statement immediately after the *pause* instruction with label ℓ is run in the next instant of execution.
- p `;` q executes p instantly followed by q if/when p terminates; instantly terminates if/when q terminates. If the execution of p raises an exception then it is instantly propagated upward and q is not run. If the execution of q raises an exception then it is instantly propagated upward.
- p `||` q executes p in parallel with q synchronously: one reaction of p `||` q consists of one reaction of p and one reaction of q until p or q terminates. If p terminates first then q continues running and p `||` q instantly terminates when q does (and vice versa). If p and q raise exceptions in the same instant, the exception with higher priority is instantly propagated upward. If p only raises an exception then q is allowed to complete its current reaction before this exception is instantly propagated upward. Even if incomplete, the execution of q is not resumed in the next instant (and vice versa).
- `[`p`]` runs p. This allows sequences of parallel statements, e.g., `[`p `||` q`];[`r `||` s`]`.
- `loop` p `end` repeats p forever unless p raises an exception, which is instantly propagated upward. Two iterations of the loop may not complete in the same instant. E.g., `loop nothing end` is illegal. This constraint ensures that atomic execution steps (reactions) can be computed with statically bounded resources [18].
- `signal` S `in` p `end` declares the local signal S in p and executes p. Signals are lexically scoped. Signal declarations are not mandatory. Undeclared signals occurring in *emit* and *present* constructs are considered global.
- `emit` S emits signal S and terminates instantly. Global signals may be emitted by the environment in addition to the program itself.
- `present` S `then` p `else` q `end` executes p if S is emitted in this instant (by the program or the environment if global), and executes q otherwise. If the execution of the chosen branch requires more than one instant, it is continued in the next instants independently from the status of S in these instants.
- `suspend` p `when` S instantly starts executing p and ignores the status of S. However, if the execution of p does not complete instantly, it is only allowed to

run in later instants in which S is not emitted (otherwise, it is suspended).

- `trap` T `in` p `end` declares exception T in p and executes p. Exceptions are lexically scoped. If p terminates or raises exception T then `trap` T `in` p `end` terminates instantly. If p raises a different exception it is propagated upward. In case of nested exception declarations, the outermost declaration has the highest priority.
- `exit` T raises exception T. We define *depth*(`exit` T) as the number of *trap* constructs enclosing the *exit* and enclosed in the declaration of T.

For example,

```
trap T
  emit A ; 1:pause ; emit B ; exit T ; emit C
||
  emit D ; 2:pause ; emit E ; emit F ; 3:pause ; emit G
end ; emit H
```

emits signals `A` and `D` in its first reaction, then `B`, `E`, `F`, and `H` in its second and final reaction. Neither `C` nor `G` is emitted. Here, the depth of `exit T` is 0.

Locations represent possible suspension points for the execution between two reactions. In previous example, after the first reaction, the execution is suspended at locations 1 and 2.

In Fig. 1, Col. 3, we define compatible locations. Two locations ℓ and ℓ' are compatible in p, i.e., $(\ell, \ell') \in \mathcal{C}_p$, iff these locations belong to concurrent branches of p. By construction, in the usual Esterel language (no *gotopause*), only compatible locations may be reached simultaneously. If L_0 is a set of pairwise compatible locations of the program p, we write p/L_0 for the program p suspended at locations L_0. We say p/L_0 is a state of the program p.

In Esterel with *gotopause*, several *gotopause* instructions may be executed concurrently. Their target locations must exist and be pairwise compatible [19]:

- `[ gotopause 1 || gotopause 2 ] ; [ 1:pause || 2:pause ]` is fine.
- `gotopause 1 ; 2:pause` is illegal because the *gotopause* instruction lacks a target *pause* instruction.
- `[ gotopause 1 || gotopause 2 ] ; 1:pause ; 2:pause` is illegal because the target *pause* instructions of the jump are not compatible.

2.2 Formal Semantics

We denote by $p\backslash X$ either the program p itself—the program p in its initial state—or the program p in some state p/L_0. Reactions of a program p are expressed via labeled transitions of the form:

$$p\backslash X \xrightarrow[I]{O,k} L$$

- $p\backslash X$ is the state from which the reaction starts;

- I is the set of signals emitted by the environment; [5]
- O is the set of signals emitted by the program;
- k is the completion code of the reaction:
  - $k = 0$ if the execution terminates instantly,
  - $k = 1$ if part of the execution is delayed due to *pause*(s) or *gotopause*(s),
  - $k \geq 2$ if an exception is reported; and
- p/L is state reached by the reaction. By construction, $L \neq \emptyset$ iff $k = 1$.

In Fig. 2, we specify the semantics of Esterel with *gotopause* as a set of facts and deduction rules in a structural operational style [13]. All but the two rules marked (∗) will be preserved unchanged in the specification of Esterel plus *gotopause* plus *catch* in Section 3.4.

Consider the rule

$$\frac{p\backslash X \xrightarrow[I\cup O']{O,0} \emptyset \qquad q \xrightarrow[I\cup O]{O',k} L}{p \;;\; q\backslash X \xrightarrow[I]{O\cup O',k} L}.$$

It specifies that $p \;;\; q$ when started (resp. restarted in state $p \;;\; q/L_0$) may react to inputs I with outputs $O \cup O'$, completion code k, and reaches the state $p \;;\; q/L$ if

- p when started (resp. restarted in state p/L_0) reacts to inputs $I \cup O'$ by terminating instantly with outputs O; and
- q when started reacts to inputs $I \cup O$ with outputs O', completion code k, and reaches the state q/L.

Because of the *synchrony hypothesis*, not only are the outputs O of p inputs of q, but reciprocally the outputs O' of q are inputs of p.

2.3 Instantaneous Loops and Reincarnation

Using the extended exception handling mechanism we propose, one can implement loops without the *loop* construct. We focus here on understanding the properties of loops, which our language extension must preserve.

The formal semantics of the *loop* construct consists of two rules so that

- `loop` p `end` $\xrightarrow[I]{O,k} L$ iff $p \xrightarrow[I]{O,k} L \wedge k \neq 0$ and
- `loop` p/L_0 `end` $\xrightarrow[I]{O,k} L$ iff $\begin{cases} \text{either } p/L_0 \xrightarrow[I]{O,k} L \wedge k \neq 0 \\ \text{or } p/L_0 \xrightarrow[I\cup B]{A,0} \emptyset \wedge p \xrightarrow[I\cup A]{B,k} L \wedge k \neq 0 \wedge O = A \cup B \end{cases}$.

When `loop` p `end` starts executing, it starts executing its body p, which may either suspend its execution (because of *pause* or *gotopause* instructions) or raise an exception; but p cannot terminate instantly. When the loop is restarted from the state L_0, it restarts its body. Now, if p terminates instantly, a new iteration—a

[5] This differs from the usual presentations of the language semantics, where present signals are considered instead ($E = I \cup O$). We choose such a presentation here because we find it more intuitive. This choice has no impact on the language extension we propose.

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

$$\texttt{emit}\ S \xrightarrow[I]{\{S\},0} \emptyset$$

$$\texttt{exit}\ T \xrightarrow[I]{\emptyset,\ depth(\texttt{exit}\ T)+2} \emptyset$$

$$\frac{p\backslash X \xrightarrow[I\cup O']{O,0} \emptyset \quad q \xrightarrow[I\cup O]{O',k} L}{p\ ;\ q\backslash X \xrightarrow[I]{O\cup O',k} L}$$

$$\frac{p\backslash X \xrightarrow[I]{O,k} L \quad k\neq 0}{p\ ;\ q\backslash X \xrightarrow[I]{O,k} L}$$

$$\frac{q/L_0 \xrightarrow[I]{O,k} L}{p\ ;\ q/L_0 \xrightarrow[I]{O,k} L}$$

$$\frac{p \xrightarrow[I]{O,k} L}{\texttt{suspend}\ p\ \texttt{when}\ S \xrightarrow[I]{O,k} L}$$

$$\frac{S\notin I\cup O \quad p/L_0 \xrightarrow[I]{O,k} L}{\texttt{suspend}\ p\ \texttt{when}\ S/L_0 \xrightarrow[I]{O,k} L}$$

$$\frac{p\backslash X \xrightarrow[I]{O,k} L \quad k\neq 0}{\texttt{loop}\ p\ \texttt{end}\backslash X \xrightarrow[I]{O,k} L}$$

$$\frac{p/L_0 \xrightarrow[I]{O,k} L \quad L_0\cap\mathcal{L}_q=\emptyset}{p\ \texttt{||}\ q/L_0 \xrightarrow[I]{O,k} L}$$

$$\frac{q/L_0 \xrightarrow[I]{O,k} L \quad L_0\cap\mathcal{L}_p=\emptyset}{p\ \texttt{||}\ q/L_0 \xrightarrow[I]{O,k} L}$$

$$\forall k,k' : k\sqcup k' = \max(k,k')$$
$$\begin{cases}\downarrow 0 = \downarrow 2 = 0\\ \downarrow 1 = 1\\ \downarrow n = n-1 \quad \forall n>2\end{cases}$$

$$\ell\texttt{:pause} \xrightarrow[I]{\emptyset,1} \{\ell\}$$

$$\texttt{gotopause}\ \ell \xrightarrow[I]{\emptyset,1} \{\ell\}$$

$$\frac{\ell\in L_0}{\ell\texttt{:pause}/L_0 \xrightarrow[I]{\emptyset,0} \emptyset}$$

$$\frac{p\backslash X \xrightarrow[I\backslash\{S\}]{O,k} L}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end}\backslash X \xrightarrow[I]{O\backslash\{S\},k} L}$$

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

$$\frac{S\notin I\cup O \quad q \xrightarrow[I]{O,k} L}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \xrightarrow[I]{O,k} L}$$

$$\frac{p/L_0 \xrightarrow[I]{O,k} L}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end}/L_0 \xrightarrow[I]{O,k} L}$$

$$\frac{q/L_0 \xrightarrow[I]{O,k} L}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end}/L_0 \xrightarrow[I]{O,k} L}$$

$$\frac{p/L_0 \xrightarrow[I\cup O']{O,0} \emptyset \quad p \xrightarrow[I\cup O]{O',k} L \quad k\neq 0}{\texttt{loop}\ p\ \texttt{end}/L_0 \xrightarrow[I]{O\cup O',k} L}$$

$$\frac{p \xrightarrow[I\cup O']{O,k} L \quad q \xrightarrow[I\cup O]{O',k'} L'}{p\ \texttt{||}\ q \xrightarrow[I]{O\cup O',k\sqcup k'} \begin{cases} L\cup L' & \text{if } k\sqcup k'=1\\ \emptyset & \text{if } k\sqcup k'\neq 1\end{cases}}$$

$$\frac{p/L_0\cap\mathcal{L}_p \xrightarrow[I\cup O']{O,k} L \quad q/L_0\cap\mathcal{L}_q \xrightarrow[I\cup O]{O',k'} L'}{p\ \texttt{||}\ q/L_0 \xrightarrow[I]{O\cup O',k\sqcup k'} \begin{cases} L\cup L' & \text{if } k\sqcup k'=1\\ \emptyset & \text{if } k\sqcup k'\neq 1\end{cases}}$$

$$\frac{S\in I \quad L_0\neq\emptyset}{\texttt{suspend}\ p\ \texttt{when}\ S/L_0 \xrightarrow[I]{\emptyset,1} L_0} \quad (*)$$

$$\frac{p\backslash X \xrightarrow[I]{O,k} L}{\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end}\backslash X \xrightarrow[I]{O,\downarrow k} L} \quad (*)$$

Fig. 2. The semantics of Esterel with *gotopause*.

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

(a)

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

(b)

Fig. 3. Loops and reincarnation.

new execution of p—is instantly started. Again, this iteration cannot terminate instantly.

First, observe that a program such as `loop nothing end` admits no possible execution: it deadlocks. In the sequel, we introduce similar safeguards to the semantics of exceptions that choose deadlocks over instantly diverging behaviors.

Second, *loop* and *signal* constructs do not commute. In Fig. 3, program (a) emits signal `A` from the second instant onwards. In contrast, program (b) never emits `A` because, in each reaction, the test applies to a fresh signal `S` distinct from the emitted signal `S`. We say signal `S` is reincarnated because of the loop. In the sequel, we implement comparable interaction rules for *signal* and *trap* scopes so loops built from *trap-exit-catch* constructs behave in the same way.

3 Introducing *catch* in Esterel

We now extend Esterel with a new *catch* instruction. The syntax becomes

$$p, q ::= \texttt{nothing} \mid \ell\texttt{:pause} \mid \ldots \mid \texttt{exit}\ T \mid \texttt{catch}\ T$$

with the constraint that there can be at most one `catch` T statement in the scope of each `trap` T `in` ... `end` construct under the usual scoping rules. For instance, if there are two nested declarations for the same exception identifier T, then there can be at most one `catch` T statement inside the inner declaration plus at most one `catch` T statement between the declarations.

If there is no such *catch* instruction, we always implicitly add one at the end of the *trap* body:

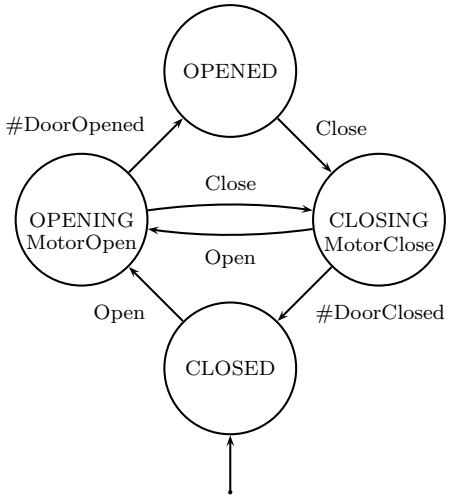
$$\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end} \quad \rightarrow \quad \texttt{trap}\ T\ \texttt{in}\ p\ \texttt{;}\ \texttt{catch}\ T\ \texttt{end}$$

Hence, in the sequel, we assume there is exactly one `catch` T statement for each declaration of T.

The *catch* instruction grabs control instantly when the corresponding exception occurs. Intuitively, *exit* is like a *goto* with *catch* as its label.

3.1 Example

In Fig. 4, we demonstrate the encoding of a state machine for an elevator door using *catch*. It has four states: OPENING, OPENED, CLOSING, and CLOSED the ini-



(a)

```
trap OPENING in trap OPENED in
trap CLOSING in trap CLOSED in

  catch CLOSED;
    present Open then pause; exit OPENING end;
    pause; exit CLOSED;

  catch OPENING;
    present DoorOpened then exit OPENED end;
    emit MotorOpen;
    present Close then pause; exit CLOSING end;
    pause; exit OPENING;

  catch OPENED;
    present Close then pause; exit CLOSING end;
    pause; exit OPENED;

  catch CLOSING;
    present DoorClosed then exit CLOSED end;
    emit MotorClose;
    present Open then pause; exit OPENING end;
    pause; exit CLOSING;

end end end end
```

(b)

Fig. 4. Encoding an arbitrary state machine with *trap-exit-catch*. (a) A state machine for an elevator door. DoorOpened and DoorClosed are sensors that indicate the door's position; Open and Close initiate or override commands; and MotorOpen and MotorClose control the motor. (b) Coding this using the *catch* instruction.

tial state of the machine. The input signals Open and Close convey user commands. The input signals DoorOpened and DoorClosed indicate the door's position. The output signals MotorOpen and MotorClose control the motor. Control signals must be sustained over a period of time for the door to fully open or fully close.

In this design, the DoorOpened and DoorClosed sensor signals must be taken into account instantly—as specified with #—so that the motor is shut down without delay. Moreover, we want instantaneous transitions to take priority over non-instantaneous transitions.

This design is implemented as follows. One exception is declared for each state. Exception priorities are irrelevant here because we never raise two exceptions simultaneously. State entry points are specified with *catch* constructs. Instantaneous transitions are encoded by *exit* constructs. Non-instantaneous transitions are delayed by *pause* instructions. Alternatively, *gotopause* instructions could be used for non-instantaneous transitions here.

3.2 Catch in Sequential Code

The *exit-catch* construct mimics the *goto-label* construct of C. For example,

```
trap T in exit T ; emit A ; catch T ; emit B end
```

only emits `B`. In general, the semantics of *exit* is that the body of its enclosing *trap* is terminated and restarted at the *catch*. In particular, the *catch* instruction may occur to the left of the corresponding *exit*(s). For instance,

```
trap T in emit A ; catch T ; emit B ; 1:pause ; exit T end
```

behaves just like

```
signal S in
  trap T in
    emit S;
    exit T;
    catch T;
    present S then
      emit A        % runs
    end
  end
end
```

(a)

```
trap T in
  signal S in
    emit S;
    exit T;
    catch T;
    present S then
      emit A        % does not run
    end
  end
end
```

(b)

Fig. 5. The effect of scopes.

```
emit A ; loop emit B ; 1:pause end
```

Incidentally, this means that `catch T`, when run immediately after `emit A`, does nothing and terminates instantly.

In general, the expansion of loops

$$\texttt{loop}\ p\ \texttt{end} \quad \rightarrow \quad \texttt{trap}\ T\ \texttt{in exit}\ T\ \texttt{;}\ \texttt{catch}\ T\ \texttt{;}\ p\ \texttt{;}\ \texttt{exit}\ T\ \texttt{end}$$

is semantics-preserving provided T is a fresh exception identifier. In particular, p cannot terminate instantly in this context. We prove the correctness of the expansion and motivate the first *exit* in Section 3.5.

Since the semantics of *exit* is that the body of its enclosing *trap* is terminated and restarted at the *catch*, the signals local to the *trap* body are reincarnated as the control jumps from *exit* to *catch*. In Fig. 5, program (a), signal `A` is emitted because the *signal* statement is not restarted. In contrast, in program (b), signal `S` is reincarnated because the *exit* statement causes the body of the *trap*, which includes the *signal* scope, to be terminated and restarted. Thus, a second, fresh incarnation of signal `S` appears and signal `A` is not emitted here.

3.3 Catch and Concurrency

Several *exit*s may execute concurrently, as illustrated in Fig. 6. When program (a) runs, `exit T1` and `exit T2` both execute. However, because exception `T1` takes precedence over `T2`, only `catch T1` is relevant: control resumes from there, and `A` is emitted instantly. Signal `B` is not emitted because control is only transferred to the first parallel branch; the second parallel branch is treated as having terminated.

In contrast, in program (b), the two *gotopause* statements are equally relevant, jumping to both branches of the second parallel, meaning that both `A` and `B` are emitted in the second instant.

Furthermore, we observe that program (c) is legal whereas program (d) is not. In program (c), two *exit* statements execute instantly, but again only the outermost exception affects control, so only `B` is emitted. However, concurrent *gotopause* statements that send control into a sequence—incompatible locations—are illegal. Priorities eliminate this potential problem with *exit* statements.

```
trap T1 in
  trap T2 in
    [
      exit T1
    ||
      exit T2
    ];
    [
      catch T1;
      emit A % runs
    ||
      catch T2;
      emit B % doesn't
    ]
  end
end
```

(a)

```
[
  gotopause 1
||
  gotopause 2
];
[
  1:pause;
  emit A % runs
||
  2:pause;
  emit B % runs
]
```

(b)

```
% OK
trap T1 in
  trap T2 in
    [
      exit T1
    ||
      exit T2
    ];
    catch T2;
    emit A
    catch T1;
    emit B
  end
end
```

(c)

```
% Erroneous
[
  gotopause 1
||
  gotopause 2
];
2:pause;
emit A;
1:pause;
emit B
```

(d)

Fig. 6. The difference between *gotopause* and *trap-exit-catch*.

Since *gotopause*(s) and *exit*(s) implement dual approaches to concurrency, *gotopause* instructions do not reduce to *trap-exit-catch* constructs plus delays. On the one hand, *trap-exit-catch* constructs cannot replace *gotopause* instructions when several targets must be reached concurrently and the scopes of the concurrent jumps intersect.[6] On the other hand, *gotopause* instructions cannot encode the instantaneous transitions of SyncCharts specifications. As a result, we believe it makes sense to retain both constructs.

3.4 Formal Semantics

Previously, we defined the states of a program p as pairs p/L_0 where L_0 is a set of compatible locations of p and also the initial state of p, which we identified with p. To express the semantics of the *catch* instruction, we now introduce exception states: for each statement in the scope of a `trap` T `in` ... `end` construct and containing a `catch` T statement, we associate the exception state $p/\#T$. In other words, we extend the locations of p to contain not only the locations of its *pause* instructions but also the locations of its *catch* instructions. Moreover, we consider these new locations to be first pairwise incompatible and second incompatible with *pause* locations. Now, the set L_0 in p/L_0 is either a potentially empty set of compatible *pause* labels of p or the single location $\#T$ of some `catch` T statement in p.

The formal semantics of Fig. 2 consists of twenty-four rules. To extend Esterel with the *catch* instruction, we preserve the first twenty-two rules, discard the two

[6] The scope of a non-instantaneous jump is the least program piece that contains both the source *gotopause* and target *pause* instructions of the jump. The scopes of concurrently executed jumps are typically pairwise disjoint when using *gotopause* to encode SyncCharts non-instantaneous transitions thanks to SyncCharts restrictions on inter-level transitions. In contrast, these scopes are typically not disjoint when using *gotopause* to cure schizophrenia [19].

$$\texttt{catch}\ T \xrightarrow[I]{\emptyset,0} \emptyset$$

$$\frac{X \neq \#T \quad p\backslash X \xrightarrow[I]{O,k} L \quad k \neq 2}{\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end}\backslash X \xrightarrow[I]{O,\downarrow k} L}$$

$$\frac{X \neq \#T \quad p\backslash X \xrightarrow[I\cup O']{O,2} \emptyset \quad p/\#T \xrightarrow[I\cup O]{O',k} L \quad k \neq 2}{\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end}\backslash X \xrightarrow[I]{O\cup O',\downarrow k} L}$$

$$\texttt{catch}\ T/\#T \xrightarrow[I]{\emptyset,0} \emptyset$$

$$\frac{S \in I \quad L_0 \neq \emptyset \quad L_0 \neq \#T}{\texttt{suspend}\ p\ \texttt{when}\ S/L_0 \xrightarrow[I]{\emptyset,1} L_0}$$

$$\frac{S \in I \quad p/\#T \xrightarrow[I]{O,k} L}{\texttt{suspend}\ p\ \texttt{when}\ S/\#T \xrightarrow[I]{O,k} L}$$

Fig. 7. The semantics of *catch*.

rules marked (∗), and add the six rules in Fig. 7 for *catch*, *trap*, and *suspend*:

- `catch` T does nothing and terminates instantly when started or restarted from location $\#T$.
- `trap` T `in` p `end` behaves like p if exception T is never raised. If T is raised then the *trap* construct instantly restarts p at location $\#T$. This execution cannot instantly raise T again ($k \neq 2$). Both rules for the *trap* construct carefully avoid capturing another exception with same identifier T by using the test $X \neq \#T$, which is shorthand for "if $p\backslash X$ is of the form p/L_0 then $L_0 \neq \#T$."
- `suspend` p `when` S when requested to restart from some location $\#T$, does so ignoring the status of signal S. Because the semantics of the *trap* construct consists in exiting and restarting its body if the exception occurs, inner *suspend* statements are considered to be in their first instant of execution when restarted. Thus, as usual, we want to ignore the status of S in the first instant.

By construction, the final state of any reaction cannot be an exception state: exception states are both generated and evaluated within the instant.

The `trap` T `in` p `end` statement, by preventing exception T from occurring twice instantly in p, effectively forbids instantaneous loops. Because the *trap* instruction starts a fresh incarnation of p when the exception occurs, reincarnation of signals local to p takes place as expected.

3.5 Correctness Results

We first check the correctness of our language extension by proving that the extended semantics matches the initial semantics for a program without *catch* instructions. We then prove the loop expansion of Section 3.2.

In this section, we denote by $\circ\!\!\rightarrow$ the reactions defined by the initial semantics and by $\rightarrow$ the reactions defined by the extended semantics.

Since we decided earlier to deal with absent *catch* instructions by inserting them at the end of their respective *trap* blocks, we consider the statements:

- initial language: p and $P = \texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end}$, and
- extended language: q and $Q = \texttt{trap}\ T\ \texttt{in}\ q\ \texttt{;}\ \texttt{catch}\ T\ \texttt{end}$.

We prove that P and Q are equivalent if p and q are.

Lemma 3.1 *If* $\forall X, \forall I, \forall O, \forall k : p\backslash X \circ\!\xrightarrow[I]{O,k} L \Leftrightarrow q\backslash X \xrightarrow[I]{O,k} L$ *then:*
$\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end}\backslash X \circ\!\xrightarrow[I]{O,k} L \Leftrightarrow \texttt{trap}\ T\ \texttt{in}\ q\ ;\ \texttt{catch}\ T\ \texttt{end}\backslash X \xrightarrow[I]{O,k} L.$

Proof. $\forall T', \forall X \neq \#T', \forall I, \forall O, \forall k$: let $\hat{\kappa}$ be k if $k \leq 1$ or $k+1$ otherwise.
First, $\texttt{trap}\ T\ \texttt{in}\ q\ ;\ \texttt{catch}\ T\ \texttt{end}\backslash \#T'$ deadlocks for all T' since q does.

Second, $\texttt{trap}\ T\ \texttt{in}\ q\ ;\ \texttt{catch}\ T\ \texttt{end}\backslash X \xrightarrow[I]{O,k} L$

iff $\begin{cases} \text{either} \quad q\ ;\ \texttt{catch}\ T\backslash X \xrightarrow[I]{O,\hat{\kappa}} L \\ \text{or} \quad O = A \cup B \wedge q\ ;\ \texttt{catch}\ T\backslash X \xrightarrow[I\cup B]{A,2} \emptyset \wedge q\ ;\ \texttt{catch}\ T/\#T \xrightarrow[I\cup A]{B,\hat{\kappa}} L \end{cases}$

iff $q\backslash X \xrightarrow[I]{O,\hat{\kappa}} L$ or $q\backslash X \xrightarrow[I]{O,2} \emptyset \wedge \hat{\kappa} = 0$

iff $p\backslash X \circ\!\xrightarrow[I]{O,\hat{\kappa}} L$ or $p\backslash X \circ\!\xrightarrow[I]{O,2} \emptyset \wedge \hat{\kappa} = 0$

iff $\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end}\backslash X \circ\!\xrightarrow[I]{O,k} L.$ □

Theorem 3.2 *If* p *contains no catch instruction then the initial and extended semantics define the same reactions for all states of* p.

Proof. By induction on the number of nested exception declarations in p. □

We now return to the loop expansion of Section 3.2. Comparing the semantics of the *loop* and *trap* constructs, we observe that the *loop* body is never permitted to terminate instantly, neither in its first nor in subsequent iterations. The *trap* body however may instantly raise the exception. The rules only forbid the exception to occur again when restarting the body from the *catch* location. Therefore, to ensure a correct expansion of *loop*s into *trap-exit-catch* constructs in Section 3.2, we insert a second *exit* at the beginning of the *trap* body in addition to the obvious one at the end of the body.

For simplicity,[7] we establish

Theorem 3.3 *If* T *is a fresh identifier then these statements are equivalent:*

- $\texttt{trap}\ T\ \texttt{in loop}\ p\ \texttt{end end},$
- $\texttt{trap}\ T\ \texttt{in exit}\ T\ ;\ \texttt{catch}\ T\ ;\ p\ ;\ \texttt{exit}\ T\ \texttt{end}.$

Proof. $\forall L_0 \neq \#T, \forall I, \forall O, \forall k$: let $\hat{\kappa}$ be k if $k \leq 1$ or $k+1$ otherwise.
First, $\texttt{trap}\ T\ \texttt{in exit}\ T\ ;\ \texttt{catch}\ T\ ;\ p\ ;\ \texttt{exit}\ T\ \texttt{end}/L_0 \xrightarrow[I]{O,k} L$

[7] The enclosing *trap* construct in the first statement ensures exception depths are identical in the two statements. Hence, there is no need to micromanage depths in the proof.

$$\text{iff} \begin{cases} \text{either} \quad \texttt{exit}\ T\ \texttt{;}\ \texttt{catch}\ T\ \texttt{;}\ p\ \texttt{;}\ \texttt{exit}\ T/L_0 \xrightarrow[I]{O,\hat{\kappa}} L \\ \text{or} \quad O = A \cup B \wedge \begin{cases} \texttt{exit}\ T\ \texttt{;}\ \texttt{catch}\ T\ \texttt{;}\ p\ \texttt{;}\ \texttt{exit}\ T/L_0 \xrightarrow[I\cup B]{A,2} \emptyset \\ \texttt{exit}\ T\ \texttt{;}\ \texttt{catch}\ T\ \texttt{;}\ p\ \texttt{;}\ \texttt{exit}\ T/\#T \xrightarrow[I\cup A]{B,\hat{\kappa}} L \end{cases} \end{cases}$$

$$\text{iff} \begin{cases} \text{either} \quad p\ \texttt{;}\ \texttt{exit}\ T/L_0 \xrightarrow[I]{O,\hat{\kappa}} L \\ \text{or} \quad O = A \cup B \wedge p\ \texttt{;}\ \texttt{exit}\ T/L_0 \xrightarrow[I\cup B]{A,2} \emptyset \wedge p\ \texttt{;}\ \texttt{exit}\ T \xrightarrow[I\cup A]{B,\hat{\kappa}} L \end{cases}$$

$$\text{iff} \begin{cases} \text{either} \quad p/L_0 \xrightarrow[I]{O,\hat{\kappa}} L \wedge \hat{\kappa} \neq 0 \\ \text{or} \quad O = A \cup B \wedge p/L_0 \xrightarrow[I\cup B]{A,0} L \wedge p \xrightarrow[I\cup A]{B,\hat{\kappa}} L \wedge \hat{\kappa} \neq 0 \end{cases}$$

iff $\texttt{loop}\ p\ \texttt{end}/L_0 \xrightarrow[I]{O,\hat{\kappa}} L$, thus iff $\texttt{trap}\ T\ \texttt{in}\ \texttt{loop}\ p\ \texttt{end}\ \texttt{end}/L_0 \xrightarrow[I]{O,k} L$.

Second, $\texttt{trap}\ T\ \texttt{in}\ \texttt{exit}\ T\ \texttt{;}\ \texttt{catch}\ T\ \texttt{;}\ p\ \texttt{;}\ \texttt{exit}\ T\ \texttt{end} \xrightarrow[I]{O,k} L$

$$\text{iff} \begin{cases} \text{either} \quad \texttt{exit}\ T\ \texttt{;}\ \texttt{catch}\ T\ \texttt{;}\ p\ \texttt{;}\ \texttt{exit}\ T \xrightarrow[I]{O,\hat{\kappa}} L \quad (\textit{impossible}) \\ \text{or} \quad O = A \cup B \wedge \begin{cases} \texttt{exit}\ T\ \texttt{;}\ \texttt{catch}\ T\ \texttt{;}\ p\ \texttt{;}\ \texttt{exit}\ T \xrightarrow[I\cup B]{A,2} \emptyset \\ \texttt{exit}\ T\ \texttt{;}\ \texttt{catch}\ T\ \texttt{;}\ p\ \texttt{;}\ \texttt{exit}\ T/\#T \xrightarrow[I\cup A]{B,\hat{\kappa}} L \end{cases} \end{cases}$$

iff $p\ \texttt{;}\ \texttt{exit}\ T \xrightarrow[I]{O,\hat{\kappa}} L$, thus iff $p \xrightarrow[I]{O,\hat{\kappa}} L \wedge \hat{\kappa} \neq 0$

iff $\texttt{loop}\ p\ \texttt{end} \xrightarrow[I]{O,\hat{\kappa}} L$, thus iff $\texttt{trap}\ T\ \texttt{in}\ \texttt{loop}\ p\ \texttt{end}\ \texttt{end} \xrightarrow[I]{O,k} L$.

Finally, both statements deadlock if required to start from location $\#T$. □

4 Related Work

The origin of this paper was the usual connection between transitions in finite state machine and gotos in imperative languages. A transition from state A to state B is nothing but a jump from block A to the beginning of block B, where blocks A and B implement the behaviors in states A and B.

While graphical design formalisms à la StateCharts [9,20] permit arbitrary, unstructured state machines, Esterel makes it awkward because of its lack of goto.

The goto-like constructs we formalize here follow directly from SyncCharts [1,2], a StateCharts-like graphical modeling language with well-defined synchronous semantics à la Esterel. But our constructs are more expressive than the collection of transitions available in SyncCharts. In particular, the trap-catch-exit construct makes it possible to exit and reenter several layers of nested macrostates at once. While SyncCharts drawings abide by rigid nesting rules and drastically restrict inter-level transitions, we allow them whenever possible.

Coding arbitrary state machines is even harder in pure dataflow synchronous languages because the programmer is responsible for specifying all sequential behavior. To address this, researchers have proposed language extensions such as mode au-

tomata [12] in Argos [11] and more recently in Lucid Synchrone [7]. Faithful to the languages they extend, these proposals restrict transitions to avoid complex causal dependencies and schizophrenia. We do not. In particular, we allow arbitrarily (finitely) many transitions to be taken in one instant.

While we want to ease the encoding of graphical synchronous specifications into textual Esterel programs, others have attempted the converse: automatically synthesizing graphical specifications from textual Esterel programs [15]. We hope to eventually combine the two to provide a user-friendly way of switching between graphical and textual representations of a specification.

5 Conclusions

We extend the *trap-exit* construct of Esterel with a new *catch* instruction that allows exception handlers to appear anywhere in the body of the *trap*. One can think of the *exit* instruction as a goto to the location of the corresponding *catch* instruction.

Simultaneous *exit*s result in a single jump to the highest-priority handler. Thus, our *trap-exit-catch* construct supplements but does not supplant the existing *gotopause* instruction for concurrent non-instantaneous jumps. We believe both must coexist in the language. Only *gotopause* can decouple the structure of program states from that of the source code while the *catch* instruction makes it possible to specify finite state machines with instantaneous transitions. In particular, it greatly simplifies the translation of SyncCharts into Esterel.

Although we did not address causality, especially constructive causality [3], we think there is no issue. The semantics of the new construct is obtained by combining existing pieces: loops for reincarnation, exceptions for priorities, and non-instantaneous jumps for locations. Synchronous digital circuit synthesis for the extended language, thus constructive semantics, should be similarly derived. For the same reason, implementing the new construct should be straightforward.

References

[1] André, C., *SyncCharts: A visual representation of reactive behaviors*, RR 95–52, I3S (1995). URL http://www.i3s.unice.fr/sports/SyncCharts/TR95-52.ps.gz

[2] André, C., *Representation and analysis of reactive behaviors: A synchronous approach*, in: *Proceedings of Computational Engineering in Systems Applications (CESA)*, Lille, France, 1996, pp. 19–29.

[3] Berry, G., *The constructive semantics of pure Esterel* (1999), draft book. URL http://www.esterel-technologies.com/files/book.zip

[4] Berry, G. and L. Cosserat, *The ESTEREL synchronous programming language and its mathematical semantics*, in: S. D. Brooks, A. W. Roscoe and G. Winskel, editors, *Seminar on Concurrency*, Lecture Notes in Computer Science **197**, Springer-Verlag, Heidelberg, Germany, 1984 pp. 389–448.

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

[6] Boussinot, F. and R. de Simone, *The ESTEREL language*, Proceedings of the IEEE **79** (1991), pp. 1293–1304.

[7] Colaço, J.-L., B. Pagano and M. Pouzet, *A conservative extension of synchronous data-flow with state machines*, in: *Proceedings of the International Conference on Embedded Software (Emsoft)*, Jersey City, New Jersey, 2005, pp. 173–182.

[8] Dijkstra, E. W., *Letters to the editor: go to statement considered harmful*, Commun. ACM **11** (1968), pp. 147–148.

[9] Harel, D., *Statecharts: A visual formalism for complex systems*, Science of Computer Programming **8** (1987), pp. 231–274.

[10] Harel, D. and A. Pnueli, "On the Development of Reactive Systems," NATO ASI Series. Series F, Computer and Systems Sciences **13**, Springer-Verlag, 1985 pp. 477–498.

[11] Maraninchi, F., *The Argos language: Graphical representation of automata and description of reactive systems*, in: *Proceedings of the IEEE Workshop on Visual Languages*, Kobe, Japan, 1991.
URL ftp://ftp.imag.fr/pub/SPECTRE/ARGONAUTE/ArgosIEEEVisual.ps.gz

[12] Maraninchi, F. and Y. Rémond, *Mode-automata: About modes and states for reactive systems*, in: *Proceedings of the European Symposium on Programming (ESOP)* (1998).

[13] Plotkin, G. D., *A structural approach to operational semantics*, Technical Report DAIMI FN-19, Aarhus University, Åarhus, Denmark (1981).
URL http://www.dcs.ed.ac.uk/home/gdp/

[14] Potop-Butucaru, D., *Optimizations for faster execution of Esterel programs*, in: *Proceedings of the 1st International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, Mont St. Michel, France, 2003, pp. 227–236.

[15] Prochnow, S., C. Traulsen and R. von Hanxleden, *Synthesizing safe state machines from Esterel*, in: *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Ottawa, Canada, 2006, p. FIXME.

[16] Serial ATA Workgroup, "Serial ATA: High Speed Serialized AT Attachment," (2001), www.serialata.org.

[17] Tardieu, O., *Goto and concurrency: Introducing safe jumps in Esterel*, in: *Proceedings of Synchronous Languages, Applications, and Programming (SLAP)*, Electronic Notes in Theoretical Computer Science (2004).

[18] Tardieu, O. and R. de Simone, *Instantaneous termination in pure Esterel*, in: *Proceedings of the 10th Annual Static Analysis Symposium*, Lecture Notes in Computer Science **2694**, San Diego, California, 2003, pp. 91–108.

[19] Tardieu, O. and R. de Simone, *Curing schizophrenia by program rewriting in Esterel*, in: *Proceedings of the 2nd International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, San Diego, California, 2004.

[20] von der Beeck, M., *A comparison of Statecharts variants*, in: *Formal Techniques in Real-Time and Fault-Tolerant Systems: Third International Symposium Proceedings*, Lecture Notes in Computer Science **863** (1994).