Instantaneous Termination in Pure Esterel

Olivier Tardieu and Robert de Simone

INRIA, Sophia Antipolis, France

Abstract. Esterel is a design language for the representation of embedded systems. Based on the synchronous reactive paradigm, its execution relies on a clear distinction of instants of computation. As a consequence, deciding whether a piece of a program may or may not run instantaneously is central to any compilation scheme, both for correctness and efficiency. In general, this information can be obtained by an exhaustive exploration of all possible execution paths, which is expensive. Most compilers approximate it through algorithmic methods amenable to static analysis. In our contribution, we first formalize the analysis involved in detecting statements that may run instantaneously. Then, we identify statements that may terminate and be instantaneously reentered. This allows us to model precisely these compilers front-end activities with a clear mathematical specification and led us to uncover inefficiencies in the Esterel v5 academic compiler from *Ecole des Mines* and *INRIA*.

1 Introduction

We shall introduce a number of static analysis techniques specific to the synchronous reactive language Esterel [6,7,8,9]. Interestingly, Esterel both allows and requires these static analyses. It allows them because of its structural "process calculus"-flavored syntax, because of its formal operational semantics, and because of its finite-state interpretation, all which Cavour easy application of static analysis methods. It requires them mainly because the synchronous paradigm promotes a crisp division of reactive behaviors between *discrete instants*, and that the definition of instantaneous reactions has to be *well-founded*. Compilers for Esterel not only benefit from static analysis techniques in optimization passes, but they heavily rely on them just to achieve correct compilation!

We give two examples of typical phenomena involved with the notion of discrete reactions, which will be at the heart of our studies:

Instantaneous loops stand for diverging behaviors, which never reach completion inside a reaction, so that the end of the instant never comes. They can be branded as *non-convergent* or *non-Zeno* behaviors. Example 1 of Figure 1 is pattern of program that may exhibit such diverging behavior. If the dots are traversed instantaneously and if the signal I is present then the loop is restarted infinitely many times within an instant. An extra difficulty in the case of Esterel comes from the presence of *parallel* statements as well as fancy *exception* raising/trapping mechanisms, so that a proper static notion of *instantaneous termination* is not all that obvious;

```
      loop                                  loop
        trap T in                             signal S in
          ...;                                  present S then ... end;
(1)       present I then exit T end;   (2)      pause;
          pause                                 emit S
        end                                   end
      end                                   end
```

Fig. 1. Potentially incorrect (1) or schizophrenic (2) loop bodies

Schizophrenic (sub)programs are those compound statements which can be run and exited or terminated *and* reentered *inside* the same reaction [3,5,18]. Such statements can be problematic because the local variables and signals involved may end up having two distinct occurrences in the same reaction, with differing values. In Example 2, the signal S is defined inside the loop. Each iteration refers to a fresh signal S. In a given instant two instances of S cohabit: the first being emitted at the end of the loop, and the second being tested as the loop is reentered. As in general (not here) the multiple values of such a signal have to be computed *simultaneously*, more than one object has to be allocated for this signal. Due to the kind of embedded application code targeted by Esterel this has to be done statically at compile time.

These analyses will serve to formally define and justify front-end parts of the previously developed compiler, which were implemented without such formal information. In some places we even spotted several minor problems in the existing code when (and because of) formalizing these issues.

After a brief presentation of the core Esterel language and its semantic features relevant to our study (Section 2), we turn to the issue of *instantaneous termination* detection by dedicated static analysis computation (Section 3). Then we face the *schizophrenia* problem with similar aims (Section 4). In all cases we provide formal definitions by structural recursion along the syntax for our techniques, and then we informally discuss their correctness and complexity. We conclude with comments on how and where our modelization as static analysis techniques of these specific front-end compiling activities helped gain better insights of them (Section 5).

2 The Pure Esterel Kernel Language

Esterel [6,7,8,9] is an imperative synchronous parallel programming language dedicated to reactive systems [14,17]. Pure Esterel is the fragment of the full Esterel language where data variables and data-handling primitives are discarded. Thus, the information carried by a signal is limited to a *presence/absence* status. While this paper concentrates on Pure Esterel, extending the results to the full Esterel language is straightforward. Moreover, without loss of generality, we focus on the Pure Esterel kernel language as defined by Berry in [5], which retains just enough of the Pure Esterel language syntax to attain its full expressive power.

`nothing`	do nothing (terminate instantaneously)
`pause`	freeze until next instant
`signal` S `in` p `end`	declare signal S in p
`emit` S	emit signal S (i.e. S is present)
`present` S `then` p `else` q `end`	if S is present then do p else do q
`suspend` p `when` S	suspend execution of p if S is present
`loop` p `end`	repeat p forever
p`;` q	do q in sequence with p
`[`p `||` q`]`	do p in parallel with q
`trap` T `in` p `end`	declare and catch exception T in p
`exit` T	throw exception T

Fig. 2. Statements of pure Esterel

2.1 Syntax and Intuitive Semantics

The Pure Esterel kernel language defines three kinds of objects: *statements*, *signals* and *exceptions*. Signals and exceptions are identifiers. Statements are recursively defined as shown by Figure 2. The non-terminals p and q denote statements, S signals and T exceptions. Signals and exceptions are lexically scoped and respectively declared inside statements by the instructions "`signal` *signal* `in` *statement* `end`" and "`trap` *exception* `in` *statement* `end`".

We say a statement is closed with respect to exceptions iff any exception T is bounded by a declaration of T, in other words iff any "`exit T`" occurs inside a "`trap T in` ... `end`" statement. We call these statements *programs*.

We consider free signals in programs as *interface signals*. We call *input signals* the interface signals that are never emitted inside the program. We call *output signals* the rest of them. Moreover, bounded signals are said to be *local signals*.

The disambiguation of the scopes of nested sequential and parallel compositions is obtained by enclosing parallel statements into brackets. Then or else branches of present statements may be omitted. Finally, note that the suspend statement will not be discussed as it introduces no technical difficulty on its own.

2.2 The Synchronous Reactive Paradigm

An Esterel program runs in steps called reactions. Each reaction takes one instant. When the clock first ticks, the execution starts. It may either terminate instantaneously or freeze until next instant (next clock cycle), through *pause* instructions, from where the execution is resumed when the clock ticks again. If so, it may then terminate or freeze again. And so on...

"`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, then emits D and terminates in the third instant. It takes three instants to complete, or in other words, proceeds by three reactions.

Reactions take no logical time. All statements executed during a reaction are considered to be simultaneous. In this first example, there is no notion of B being emitted before C.

Parallelism in Esterel is very different from the asynchronous composition of many concurrent languages or formalisms such as ADA or OCCAM [2,19]: execution propagates in parallel branches in a deterministic synchronous way.

"`[pause; emit A; pause; emit B || emit C; pause; emit D]`" emits the signal C in the first instant of its chain reaction, then emits A and D in the second instant, then emits B and terminates. By definition, "`[`p `||` q`]`" terminates when the last of p and q terminates (in the absence of exceptions).

2.3 Loops and Non-Zenoness Condition

"`loop emit S; pause end`" emits S at each instant and never terminates. This compound statement is the kernel expansion of the instruction "`sustain S`". Note that using exceptions, it will be possible to escape from loops.

Remark the "`pause`" inside the loop. In Esterel, the body of a loop is not allowed to terminate instantaneously when started. It must execute either a pause or an exit statement. This constraint ensures *non-Zenoness* of Esterel programs: as each loop body is traversed at most once at a given instant, the computation of one reaction of a correct program always ends. This is of course expected from a reactive language, which claims to proceed by instantaneous reactions! Section 3 will be devoted to methods that ensure this requirement holds for a given program.

2.4 Exceptions

Exceptions in sequential code behave as structural gotos to the end of the trap. "`trap T in emit A; pause; emit B; exit T; emit C end; emit D`" emits A first, then B and D and terminates. In this case, the statement "`emit C`" is unreachable. Exceptions may also occur in parallel constructions as 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 instant. As expected, "`emit C`" is never reached. The second "`pause`" of the left branch of the parallel is *reached* but not *executed* because it is *preempted* by the simultaneously occurring exception. However, since "`exit T`" does not prevent B to be emitted, we say that exceptions implement *weak* preemption.

Exceptions may also be nested. In such a case, the outermost exception has priority. In the following program, A is not emitted since T has priority on U: "`trap T in trap U in [exit T || exit U] end; emit A end`".

If we consider not only programs but statements in general, we may encounter undeclared exceptions, as in "`trap T in [exit T || exit U] end; exit T`". The left "`exit T`" raises an exception matched within the statement by the enclosing "`trap T in ... end`", while both "`exit U`" and the second "`exit T`" refer to undeclared/unmatched exceptions. We say that by raising such an exception a statement *exits*.

2.5 Signals

In an instant, a signal S is either present or absent. If S is present all executed "`present S`" statements execute their then branch in this instant; if S is absent, they all execute their else branch. A local or output signal S is present iff it is explicitly emitted, absent otherwise. The input signals are provided by the *environment.* Each execution cycle involves the following steps:

– The environment provides the *valuation*[1] of the input signals. The *set of inputs* of a reaction is the set of input signals set present by the environment.
– The reaction occurs.
– The environment observes the resulting valuation of the output signals.

2.6 Logical Correctness and Behavioral Semantics

Since signal emissions can occur inside present statements, it is possible to write incorrect programs such as:

– "`signal S in present S else emit S end end`"
  If S is present it cannot be emitted. On the other hand, if S is not present it is emitted. Both cases are contradictory.
– "`signal S in present S then emit S end end`"
  S may either be present or absent. This program is not deterministic.

A program is said to be *logically correct* iff there exists exactly one possible valuation of its signals for any set of inputs at any stage of the execution. In addition, a valuation of the free signals of a given statement is said to be *admissible* for this statement if it can be extended into a unique valuation of all its signals coherent with the statement semantics. This is formalized by the *logical behavioral semantics* of Esterel, which we briefly sketch in Appendix A.

2.7 Constructive Semantics

There is no efficient algorithm to compute these valuations in general. The program "`signal S in present S then emit S else emit S end end`" is logically correct since S can only be present. Nevertheless, its execution relies on a guess. The signal S has first to be guessed present before it can be emitted.

The *constructive semantics* of Esterel precisely avoid such guesses by restricting the set of correct programs to the so called *constructive programs.* The execution of a "`present S`" statement is blocked until S is known to be present or absent. The signal S is present as soon as one "`emit S`" is certainly executed in the current reaction. It is absent as soon as all "`emit S`" statements are proved to be unreachable in the current reaction, due to effective choices taken so far.

We strongly encourage the reader to refer to [4,5] for further information about these issues and more generally about Esterel formal semantics. In the sequel, we focus on logically correct programs and behavioral semantics as the refinement provided by the constructive semantics is orthogonal to our concerns.

[1] In Pure Esterel, *valuation* is just a shortcut for "present/absent statuses".

3 Instantaneous Termination

The first reaction of a program may either terminate its execution or lead to (at least) one more reaction. Thus, for a given set of inputs, the execution of a program may either be *instantaneous* if it completes in a single reaction or *non-instantaneous* it if does not, that is to say if it lasts for at least two instants. We say that a program *cannot be instantaneous* iff its execution is never instantaneous i.e. is non-instantaneous for any inputs.

We want to extend this definition to statements. For a given admissible valuation of its free signals, the behavior of a statement is deterministic. Its first reaction may either (i) lead to one more reaction or (ii) exit or (iii) terminate its execution (without exiting). Thus, the execution of a statement either:

lasts by taking at least two reactions to complete
exits instantaneously by raising a free exception
terminates instantaneously otherwise

We says that a statement *cannot be instantaneous* iff, for any admissible valuation of its free signals, it does not terminate instantaneously i.e. it either lasts or exits instantaneously. As a consequence, if a statement cannot be instantaneous, its execution cannot start and terminate within a unique reaction of the program it is part of. If a statement p cannot be instantaneous, then q is never reached in the first instant of the execution of "p`;` q". Let's consider a few examples:

– "`exit T`" cannot be instantaneous (as it always exits)
– "`present I then exit T end`" may be instantaneous
– "`present I then exit T else pause end`" cannot be instantaneous
– "`trap T in exit T end`" may be instantaneous (is always instantaneous)

The definition of Esterel specifies that the body of a loop cannot be instantaneous (cf. Section 2.3). In the rest of this section, we discuss methods to ensure that a given statement cannot be instantaneous. First we consider exact analysis in Section 3.1, then we formalize efficient static analyses. Because of exceptions, this is not straightforward. Thus, we start by restricting ourselves to exception-free statements in Section 3.2. We further introduce exceptions in two steps in Sections 3.3 and 3.4. We discuss the current implementation in Section 3.5.

3.1 Exact Analysis

The exact decision procedure is obvious. For a given statement, it consists in computing its first reaction for all possible valuations of its free signals and checking that the execution does not terminate instantaneously in each admissible case.

The number of valuations is finite but exponential in the number of free signals, which can be linear in the size of the statement. In fact, as illustrated by Figure 3, SAT (the problem of boolean satisfiability in propositional logic) can be expressed in terms of instantaneous termination of Pure Esterel programs (by a polynomial reduction). A valuation satisfies the boolean formula iff it makes the

$$(A \vee \neg B \vee C) \wedge (\neg A \vee C \vee \neg D) \wedge (\neg B \vee \neg C \vee D) \text{ is satisfiable}$$
$$\Updownarrow$$

```
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

execution of the corresponding program terminate instantaneously. Reciprocally, there exists no such valuation iff the program cannot be instantaneous.

To the best of our knowledge, this procedure has never been experimented with. Whether it is tractable in practice or not remains an open question. Nevertheless, NP complexity is a strong argument in favor of approximate analysis.

3.2 Static Analysis for Exception-Free Statements

Since we are now interested in conservative analysis, a statement labeled with "may be instantaneous" could well be a statement that "cannot be instantaneous", which we missed because of approximations. On the other hand, a statement labeled with "cannot be instantaneous" truly cannot be.

Figure 4 gives a first set of rules applicable to exception-free statements. For example, in order for the execution of "p`;` q" to terminate instantaneously it has to be true that p is executed instantaneously, instantaneously transferring control to q, which itself has to terminate instantaneously. By quantifying on all admissible valuation, we extract the rule: "p`;` q" cannot be instantaneous as soon as p or q cannot be instantaneous.

`nothing`	may be instantaneous
`pause`	cannot be instantaneous
`signal` S `in` p `end`	cannot be instantaneous if p cannot
`emit` S	may be instantaneous
`present` S `then` p `else` q `end`	cannot be instantaneous if both p and q cannot
`suspend` p `when` S	cannot be instantaneous if p cannot
`loop` p `end`	cannot be instantaneous, *p has to be* non-instantaneous
p`;` q	cannot be instantaneous if p or q cannot
`[`p `||` q`]`	cannot be instantaneous if p or q cannot

Fig. 4. Non-instantaneous exception-free statements

These rules can provably be shown to match formally an abstraction [11] of Pure Esterel semantics. The proof of the last analysis of this section (the more complex) is detailed in Appendix B.

Deriving from the rules a compositional algorithm that proceeds by structural recursion along the syntax of a statement is straightforward: the entries of

Figure 4 are the facts and predicates of a logic program, which can be run by a Prolog like depth-first search algorithm.

This analysis is compositional, in the sense that the denotation of a compound statement is a function of the denotations of its subterms. More precisely in this first framework, the denotation of a statement p is the boolean value $\mathcal{D}_p$ of the predicate "(we know that) p cannot be instantaneous" and the composition functions are boolean conjunctions or disjunctions. In particular, $\mathcal{D}_{p;\ q}$ is equal to $\mathcal{D}_p \vee \mathcal{D}_q$.

However, even if p and q may be instantaneous, this does not imply in general that "p`;` q" may also be. The analysis fails to prove this program cannot be instantaneous: "`present S then pause end; present S else pause end`". As the rules for "p`;` q" and "`[`p `||` q`]`" do not take into account synchronization between p and q, the results may be imprecise (but correct). There is a trade-off between efficiency and precision. The analysis only requires a linear number of constant-time computations, so its complexity is linear in the size of the statement. Dealing with correlated signal statuses and potential synchronization configurations would very quickly reintroduce the complexity of exact analysis.

3.3 Static Analysis for All Statements

In order to extend this analysis to handle exceptions, the calculus detailed in Figure 5 has also to decide whether the body p of a "`trap T in` p `end`" statement may instantaneously raise T or not, since this would lead the statement to terminate instantaneously.

p	$\mathcal{D}_p$	$\mathcal{X}_p$
`nothing`	*false*	$\emptyset$
`pause`	*true*	$\emptyset$
`signal` S `in` p `end`	$\mathcal{D}_p$	$\mathcal{X}_p$
`emit` S	*false*	$\emptyset$
`present` S `then` p `else` q `end`	$\mathcal{D}_p \wedge \mathcal{D}_q$	$\mathcal{X}_p \cup \mathcal{X}_q$
`suspend` p `when` S	$\mathcal{D}_p$	$\mathcal{X}_p$
`loop` p `end`	*true*	$\mathcal{X}_p$
p`;` q	$\mathcal{D}_p \vee \mathcal{D}_q$	$\mathcal{X}_p \cup [\neg\mathcal{D}_p \rightarrow \mathcal{X}_q]$
`[`p `||` q`]`	$\mathcal{D}_p \vee \mathcal{D}_q$	$\mathcal{X}_p \cup \mathcal{X}_q$
`trap` T `in` p `end`	$\mathcal{D}_p \wedge (T \notin \mathcal{X}_p)$	$\mathcal{X}_p \backslash \{T\}$
`exit` T	*true*	$\{T\}$

Fig. 5. Non-instantaneous statements

The denotation of a statement p becomes a pair $(\mathcal{D}_p, \mathcal{X}_p)$ where:

- $\mathcal{D}_p$ remains the predicate "p cannot be instantaneous";
- $\mathcal{X}_p$ is the set of exceptions that p may raise instantaneously.

It is now possible to define $\mathcal{D}_{\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end}}$ as $\mathcal{D}_p \wedge (T \notin \mathcal{X}_p)$. In Figure 5 and thereafter, we use the notation $[\mathcal{P} \rightarrow \mathcal{S}]$ as a shortcut for "if $\mathcal{P}$ then $\mathcal{S}$ else

$\emptyset$". The set of exceptions that "p; q" may raise instantaneously is $\mathcal{X}_p \cup \mathcal{X}_q$ if p may be instantaneous or $\mathcal{X}_p$ only if p cannot be instantaneous, that is to say $\mathcal{X}_p \cup [\neg\mathcal{D}_p \rightarrow \mathcal{X}_q]$.

This new analysis remains linear in the size of the code, if we suppose that the number of levels of nested trap statements never exceeds a fixed bound. We remark that the Esterel v5 compiler has a hard 32 limit.

3.4 Static Analysis Using Completion Codes

In the previous section, we have described a procedure to ensure that a statement cannot be instantaneous. It is approximate but conservative: it may be unable to prove that a statement cannot be instantaneous even if it cannot be, but it never concludes that a statement cannot be instantaneous if it can be.

We achieved linear complexity by (i) providing a compositional algorithm which proceeds by structural recursion along the syntax of the statement and (ii) abstracting away signal statuses so that the denotation of a statement remains a simple object (a boolean plus a set of bounded size).

These two constraints leave hardly any room for improvement. But we can still do better, as we have not yet taken into account trap priorities. Let's consider the following statement:

```
trap T in
  trap U in
    trap V in [exit U || exit V] end; exit T
  end;
  pause
end
```

As we have defined $\mathcal{X}_{[p\ ||\ q]}$ as $\mathcal{X}_p \cup \mathcal{X}_q$, the computation proceeds as follows:

$\mathcal{X}_{\texttt{[exit U || exit V]}} = \{\mathrm{U}, \mathrm{V}\}$
$\mathcal{D}_{\texttt{trap V in [exit U || exit V] end}} = \mathit{false}$
$\mathcal{X}_{\texttt{trap V in [exit U || exit V] end; exit T}} = \{\mathrm{T}, \mathrm{U}\}$
$\mathcal{X}_{\texttt{trap U in trap V in [exit U || exit V] end; exit T end; pause}} = \{\mathrm{T}\}$
$\mathcal{D}_{\texttt{trap T in trap U in trap V in [exit U || exit V] end; exit T end; pause end}} = \mathit{false}$

It concludes that the statement may be instantaneous. However, since U has priority on V, a more precise analysis seems feasible, something like:

$\mathcal{X}_{\texttt{[exit U || exit V]}} = \{\mathrm{U}\}$
$\mathcal{D}_{\texttt{trap V in [exit U || exit V] end}} = \mathit{true}$
$\mathcal{X}_{\texttt{trap V in [exit U || exit V] end; exit T}} = \{\mathrm{U}\}$
$\mathcal{X}_{\texttt{trap U in trap V in [exit U || exit V] end; exit T end; pause}} = \emptyset$
$\mathcal{D}_{\texttt{trap T in trap U in trap V in [exit U || exit V] end; exit T end; pause end}} = \mathit{true}$

In other words, as the analyzer decomposes a compound statement into its parts, it should keep track of the relative priorities of exceptions. Then, this order would be taken into account in the rule for "`[`p `||` q`]`".

Such a calculus is possible. It relies on the idea of completion codes[2] introduced in Esterel by Gonthier [15]. Let's consider a statement s such that no two exceptions of s share the same name (applying alpha-conversion to these names if necessary). There exists a function that associates with each exception T occurring in s a completion code $k_{\mathrm{T}} \in \mathbb{N} \cup \{+\infty\}$ such that:

- $\forall T,\ k_T \geq 2$
- $\forall T,\ k_T = +\infty$ if T is unmatched in s, $k_T \in \mathbb{N}$ otherwise
- $\forall U,\ \forall V,\ U \neq V \Rightarrow k_U \neq k_V$
- $\forall U,\ \forall V,\ \mathit{scope}(V) \subset \mathit{scope}(U)$ (i.e. U has priority over V) $\Rightarrow k_U > k_V$

For example, the set $\{k_{\mathrm{T}} = 8,\ k_{\mathrm{U}} = 3,\ k_{\mathrm{V}} = +\infty\}$ is admissible for the statement "`trap T in trap U in [exit T || exit U] end end; exit V`".

Using these completion codes, it is now possible to compute potential instantaneous behaviors of a subterm p of s with respect to termination as described by Figure 6. The denotation of a statement p is the set $\mathcal{K}_p$ of its potential completion codes, that is to say a set that contains:

- 0 if p may instantaneously terminate
- 1 if p may instantaneously execute a "`pause`" statement
- k_{T} if p may instantaneously raise the exception T local to s
- $+\infty$ if p may instantaneously raise an exception not caught in s

p	$\mathcal{K}_p$
`nothing`	$\{0\}$
`pause`	$\{1\}$
`signal` S `in` p `end`	$\mathcal{K}_p$
`emit` S	$\{0\}$
`present` S `then` p `else` q `end`	$\mathcal{K}_p \cup \mathcal{K}_q$
`suspend` p `when` S	$\mathcal{K}_p$
`loop` p `end`	$\mathcal{K}_p$ (by hypothesis $0 \notin \mathcal{K}_p$)
p`;` q	$(\mathcal{K}_p \backslash \{0\}) \cup [(0 \in \mathcal{K}_p) \rightarrow \mathcal{K}_q]$
`[`p `||` q`]`	$\{\max(k,l) \vert \forall k \in \mathcal{K}_p,\ \forall l \in \mathcal{K}_q\}$
`trap` T `in` p `end`	$(\mathcal{K}_p \backslash \{k_T\}) \cup [(k_T \in \mathcal{K}_p) \rightarrow \{0\}]$
`exit` T	$\{k_T\}$

Fig. 6. Potential completion codes

This analysis is essentially equivalent to the previous one. As expected, the rule for "`exit T`" encodes the level of priority of T. Remark the rule associated with "`[`p `||` q`]`". If p may have completion code k and q may have completion code l then "`[`p `||` q`]`" may admit completion code $\max(k,l)$ as illustrated by Figure 7. With completion codes, we have not only encoded trap priorities

[2] For the sake of simplicity, we refrain from introducing here the classical de Bruijn encoding [12] of completion codes and describe a similar but less efficient encoding.

`[` p `||` q `]`	k	l	$\max(k,l)$
`[ nothing || nothing ]`	0	0	0
`[ nothing || pause ]`	0	1	1
`[ pause || pause ]`	1	1	1
`[ nothing || exit` T `]`	0	k_T	k_T
`[ pause || exit` T `]`	1	k_T	k_T
`[ exit` T `|| exit` U `]`	k_T	k_U	$\max(k_T, k_U)$

Fig. 7. The Max formula

making the last line of this table possible, but also precedence relations between nothing, pause and exit statements, so that the "max" formula always works.

The analysis is conservative. Its complexity is linear in the size of the statement under the hypotheses of (i) bounded depth of nested traps and (ii) usage of de Bruijn [12,15] completion codes. The subset $\mathcal{K}_s$ of $\{0,\ 1,\ +\infty\}$ computed for the statement s itself contains the set of completion codes that may be observed while running s. Going back to the initial problem, we conclude:

– s cannot be instantaneous if $0 \notin \mathcal{K}_s$;
– s must be instantaneous if $\mathcal{K}_s = \{0\}$.

3.5 Comparison with Current Implementation

In the Esterel v5 compiler, the analysis of instantaneous termination occurs twice, applied first to an internal representation of the kernel language structure, then to the circuit representation (as part of the cyclicity analysis). While the initial rejection of potentially instantaneous loop bodies relies on the formalism of the last section, the second analysis (more precisely the current combination of the translation into circuits with the cyclicity analysis) is less precise. For example, "`loop [... || pause]; emit S; pause; present S then ... end end`" passes the first analysis but not the second one!

We characterized the patterns of programs that expose this behavior of the compiler and identified the changes required in the translation into circuits to avoid it.

4 Schizophrenia

In Section 3, we have formalized an algorithm to check if a Pure Esterel statement cannot be instantaneous. In this section, we shall consider a second, rather similar problem. Given a statement q in a program p, is it possible for the statement q to terminate or exit and be restarted within the same reaction? For example, if p is "`loop` q `end`", when q terminates (if it eventually does), then q is instantaneously restarted by the enclosing loop. On the other hand, if p is "`loop pause;` q `end`" then q cannot terminate or exit and be restarted instantaneously, thanks to the "`pause`" statement.

As usual in the Esterel terminology, we say that q is *schizophrenic* if the answer is yes [3,5,18]. The point is: we do not like signal and parallel statements to be schizophrenic! A schizophrenic signal may carry two different values within a single reaction (cf. Section 1). Similarly, a schizophrenic parallel requires two synchronizations. Obviously, without some kind of unfolding, both are incompatible with single-static-assignment frameworks such as Digital Sequential Circuits[3].

Having introduced the notion of schizophrenic contexts in Section 4.1, we discuss static analyses again in two steps: first considering exception-free contexts in Section 4.2, then getting rid of the restriction in Section 4.3. We relate our formalization to the current implementation in Section 4.4.

4.1 Contexts

From now on, if q is a statement of the program p, we call *context of q in p* and note $\mathcal{C}[\ ]$ the rest of p, that is to say, p where q has been replaced by a hole $[\ ]$. In the last example, the context of q in p is $\mathcal{C}[\ ] \equiv$ "`loop pause; [ ] end`".

Contexts are recursively defined:

- $[\ ]$ is the empty context;
- if $\mathcal{C}[\ ]$ is a context then $\mathcal{C}[$`present` S `then [ ] else` q `end`$]$ is a context...

As usual [1], $\mathcal{C}[x]$ denotes the statement (respectively context) obtained by substituting the hole $[\ ]$ of $\mathcal{C}[\ ]$ by the statement (respectively context) x. We say that $\mathcal{C}[\ ]$ is a *valid* context for the statement p, if $\mathcal{C}[p]$ is not only a statement but also a correct program. In the sequel, we shall consider only such compositions.

The fact that p is schizophrenic in $\mathcal{C}[\ ]$ depends on both p and $\mathcal{C}[\ ]$. If $\mathcal{C}[\ ]$ is "`signal S in loop [ ]; present S then pause end end end`" and p is "`pause`" then p is schizophrenic in this context. On the other hand, if p is "`pause; emit S`" then p is not schizophrenic since the then branch of the present statement is taken. We say that $\mathcal{C}[\ ]$ is *schizophrenic* if and only if there exists a p such that p is schizophrenic in $\mathcal{C}[\ ]$.

4.2 Static Analysis for Exception-Free Contexts

This first case is quite simple. Obviously in order for a statement to be instantaneously restarted, it has to appear enclosed in a loop. It may be enclosed in many nested loops, however since loops are infinite only the innermost loop has to be taken into account. Then, if in the body of this loop, this statement is in sequence somehow with a statement which cannot be instantaneous, it is not schizophrenic. Otherwise, it probably is.

This is exactly the reasoning steps we implement in Figure 8. In a manner similar to Figure 4, we describe sufficient conditions for a context to be non-schizophrenic. These conditions provide the rules of a conservative static analysis of contexts.

[3] In fact, unfolding in time (i.e. using a memory cell twice within a reaction) is not correct in general. Thus, the same result holds even in the absence of the single-static-assignment constraint, as in C code generation for example.

`[ ]`	is not schizophrenic
$\mathcal{C}$`[signal` S `in [ ] end]`	is not schizophrenic if $\mathcal{C}$[ ] is not
$\mathcal{C}$`[present` S `then [ ] else` q `end]`	is not schizophrenic if $\mathcal{C}$[ ] is not
$\mathcal{C}$`[present` S `then` p `else [ ] end]`	is not schizophrenic if $\mathcal{C}$[ ] is not
$\mathcal{C}$`[suspend [ ] when` S`]`	is not schizophrenic if $\mathcal{C}$[ ] is not
$\mathcal{C}$`[loop [ ] end]`	is schizophrenic
$\mathcal{C}$`[[ ];` q`]`	is not schizo. if $\mathcal{C}$[ ] is not or q cannot be inst.
$\mathcal{C}$`[`p`; [ ]]`	is not schizo. if $\mathcal{C}$[ ] is not or p cannot be inst.
$\mathcal{C}$`[[ ] ||` q`]`	is not schizophrenic if $\mathcal{C}$[ ] is not
$\mathcal{C}$`[`p `|| [ ]]`	is not schizophrenic if $\mathcal{C}$[ ] is not

Fig. 8. Non-schizophrenic exception-free contexts

4.3 Static Analysis for All Contexts

In order to handle all contexts, we associate with a context $\mathcal{C}[\ ]$ a set of completion codes $\mathcal{S}_{\mathcal{C}[\ ]}$ such that: for all p, if p does not admit any completion code in $\mathcal{S}_{\mathcal{C}[\ ]}$ then p is not schizophrenic in $\mathcal{C}[\ ]$. For example, if $\mathcal{C}[\ ]$ is the context "`loop trap T in loop trap U in [ ]; pause end end end; pause end`", p is schizophrenic in $\mathcal{C}[\ ]$ iff if it may raise exception U. Thus $\{k_\mathtt{U}\}$ is an admissible value for $\mathcal{S}_{\mathcal{C}[\ ]}$. Note that a larger set (less precise), such as $\{0, k_\mathtt{U}\}$, would also be.

Figure 9 describes the computation of $\mathcal{S}_{\mathcal{C}[\ ]}$ we propose[4]. In summary:

- $\mathcal{C}[\ ]$ is proven to be non-schizophrenic iff $\mathcal{S}_{\mathcal{C}[\ ]}$ is empty;
- p is proven to be non-schizophrenic in $\mathcal{C}[\ ]$ iff p does not admit a completion code in $\mathcal{S}_{\mathcal{C}[\ ]}$.

$$\begin{aligned}
\mathcal{S}_{[\ ]} &\equiv \emptyset \\
\mathcal{S}_{\mathcal{C}[\mathtt{signal}\ S\ \mathtt{in}\ [\ ]\ \mathtt{end}]} &\equiv \mathcal{S}_{\mathcal{C}[\ ]} \\
\mathcal{S}_{\mathcal{C}[\mathtt{present}\ S\ \mathtt{then}\ [\ ]\ \mathtt{else}\ q\ \mathtt{end}]} &\equiv \mathcal{S}_{\mathcal{C}[\ ]} \\
\mathcal{S}_{\mathcal{C}[\mathtt{present}\ S\ \mathtt{then}\ p\ \mathtt{else}\ [\ ]\ \mathtt{end}]} &\equiv \mathcal{S}_{\mathcal{C}[\ ]} \\
\mathcal{S}_{\mathcal{C}[\mathtt{suspend}\ [\ ]\ \mathtt{when}\ S]} &\equiv \mathcal{S}_{\mathcal{C}[\ ]} \\
\mathcal{S}_{\mathcal{C}[\mathtt{loop}\ [\ ]\ \mathtt{end}]} &\equiv \{0\} \cup \mathcal{S}_{\mathcal{C}[\ ]} \\
\mathcal{S}_{\mathcal{C}[[\ ];\ q]} &\equiv (\mathcal{S}_{\mathcal{C}[\ ]} \backslash \{0\}) \cup [(\mathcal{K}_q \cap \mathcal{S}_{\mathcal{C}[\ ]} \neq \emptyset) \rightarrow \{0\}] \\
\mathcal{S}_{\mathcal{C}[p;\ [\ ]]} &\equiv [(0 \in \mathcal{K}_p) \rightarrow \mathcal{S}_{\mathcal{C}[\ ]}] \\
\mathcal{S}_{\mathcal{C}[\mathtt{trap}\ T\ \mathtt{in}\ [\ ]\ \mathtt{end}]} &\equiv \mathcal{S}_{\mathcal{C}[\ ]} \cup [(0 \in \mathcal{S}_{\mathcal{C}[\ ]}) \rightarrow \{k_T\}]
\end{aligned}$$

Fig. 9. Potentially schizophrenic completion codes

Let's focus on the last four rules:

- If p terminates in $\mathcal{C}$`[loop` p `end]` then it is instantaneously restarted by the inner loop. Consequently, $\{0\} \in \mathcal{S}_{\mathcal{C}[\mathtt{loop}\ [\ ]\ \mathtt{end}]}$. Moreover, if p raises exception T, it traverses "`loop` p `end`" and reaches the context $\mathcal{C}[\ ]$. Thus, $(\mathcal{S}_{\mathcal{C}[\ ]} \backslash \{0\}) \subset \mathcal{S}_{\mathcal{C}[\mathtt{loop}\ [\ ]\ \mathtt{end}]}$. As a consequence, $\mathcal{S}_{\mathcal{C}[\mathtt{loop}\ [\ ]\ \mathtt{end}]} \equiv \{0\} \cup \mathcal{S}_{\mathcal{C}[\ ]}$.

[4] We omitted the entries corresponding to parallel contexts. In practice, there is no need for such rules. A correct but weak extension of this formalism to parallel contexts may be obtained via the rules: $\mathcal{S}_{\mathcal{C}[[\ ]\ ||\ q]} \equiv \mathcal{S}_{\mathcal{C}[p\ ||\ [\ ]]} \equiv [(\mathcal{S}_{\mathcal{C}[\ ]} \neq \emptyset) \rightarrow \mathbb{N}]$.

- As in the previous case, if p may raise exception T, it may traverse "`[`p`;` q`]`" and reach the context $\mathcal{C}[\ ]$. Thus, $(\mathcal{S}_{\mathcal{C}[\ ]}\backslash\{0\}) \subset \mathcal{S}_{\mathcal{C}[[\ ];\ q]}$. In addition, if q may instantaneously produce a completion code in $\mathcal{S}_{\mathcal{C}[\ ]}$ i.e. if $\mathcal{K}_q \cap \mathcal{S}_{\mathcal{C}[\ ]} \neq \emptyset$ then, if p eventually terminates, "`[`p`;` q`]`" may be instantaneously restarted. Thus, in this case $\{0\} \in \mathcal{S}_{\mathcal{C}[[\ ];\ q]}$.
- If p cannot be instantaneous then q in $\mathcal{C}[p\texttt{;}\ q]$ cannot be instantaneously restarted. On the other hand, if p may, then $\mathcal{S}_{\mathcal{C}[p;\ [\ ]]}$ is equal to $\mathcal{S}_{\mathcal{C}[\ ]}$.
- If p raises exception U ($U \neq T$) or terminates, then "`trap` T `in` p `end`" does the same, so $\mathcal{S}_{\mathcal{C}[\ ]} \subset \mathcal{S}_{\mathcal{C}[\texttt{trap}\ T\ \texttt{in}\ [\ ]\ \texttt{end}]}$. Note that $k_T \notin \mathcal{S}_{\mathcal{C}[\ ]}$ since we have supposed that no two exceptions share the same name. Moreover, if $0 \in \mathcal{S}_{\mathcal{C}[\ ]}$ then $k_T \in \mathcal{S}_{\mathcal{C}[\texttt{trap}\ T\ \texttt{in}\ [\ ]\ \texttt{end}]}$, since by raising exception T, p makes "`trap` T `in` p `end`" terminate.

4.4 Comparison with Current Implementation

The Esterel v5 implementation of the detection of schizophrenic contexts cannot be directly compared with this last analysis. It is in some cases more precise, less in others. Nevertheless, the analysis implemented and the one we presented have common weaknesses, that our formalization helped to identify.

For example, in "`loop signal S in pause; ... end end`", the declaration of the signal S occurs in a schizophrenic context, thus it triggers unfolding routines. However, because of the pause statement, the status of S is not used when entering the loop. As a consequence, unfolding is not necessary.

We remarked that both analyses could be refined to embody a static liveness analysis of signals. This has been implemented into the Esterel v5 compiler.

5 Conclusion and Future Work

We have formalized several important property checks on Esterel in the form of static analysis methods. Namely, analyses are used to establish when:

- a statement cannot terminate instantaneously,
- a statement cannot terminate or exit and be instantaneously reentered.

The correctness of these analyses is induced by the fact that they can be shown to be abstractions of the "official" Pure Esterel *logical behavioral* semantics. Their complexity is quasi-linear in practice in the program size.

This work was mostly motivated by a revisitation of the Esterel v5 academic compiler, from *Ecole des Mines* and *INRIA*. Front-end processors are heavily relying on algorithms implementing those property checks, but so far without a formal specification. These checks are needed to enforce programs to be free of instantaneous loops, and to contain no schizophrenic subcomponents (in the first case faulty programs are discarded as incorrect, in the second case they are unfolded to secure proper separation).

Static analysis cannot be deactivated: it is required to generate correct code. Thus, the need for precise formalization and correctness proof is far greater, in

```
loop
  [
    present S then pause; ... end;     loop
    present S else pause; ... end  -->   present S then pause; ... end;
  ||                                     present S else pause; ... end
    pause                              end
  ]
end
```

Fig. 10. A simple program transformation

our view, than in the case of "-O3" kind of static analysis. We see our work as a form of formal compiler specification, which can be referred to as a guideline for correctness.

As the Esterel v5 algorithmic approach was only informally specified, we were able to spot several minor mistakes in the actual implementation, as well as to introduce more aggressive optimization techniques for specific program patterns, saving unnecessary unfolding.

Since the methods we described and more generally the techniques in use only provide approximate (but conservative) results, some Esterel programs are rejected by current compilers [10,13,21] while they are provably correct. This is rather unusual! Of course, this is well known and comes from other sources as well, mainly an incomplete causality analysis [21]. But as a consequence these compilers have a weak support for program transformation. Even the simple rewriting illustrated by Figure 10 produces a program that is rejected by compilers, as the static analysis of the rewritten loop body fails.

According to the authors of the Esterel v5 compiler, the semantic program equivalence relation is not sufficient when it comes to compiling programs. Because of the static analyses involved, the behavior of the compiler (both rejection and optimization) is sometimes unstable and may change a lot from one program to an equivalent program. In particular, and in opposition to what suggests the documentation of the language, the expansion of high-level constructions into low-level primitives has to be done very carefully to avoid unexpected issues related to those techniques.

In the future we plan to investigate the feasibility of more powerful analyses, hopefully having a more intuitive and stable behavior, starting from the exact analysis of Section 3.1. In addition, we would like to consider the more generic problem of distance analysis in Esterel. In this paper we considered the question: "will there be a pause executed between these two points of the program?" Now, we would like to know how many pause statements there are. Combined with classical delay analysis (i.e. analysis of counters) [16], we believe this would lead to powerful verification tools.

References

1. H. P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.
2. G. Berry. Real-time programming: General purpose or special-purpose languages. In G. Ritter, editor, *Information Processing 89*, pages 11–17. Elsevier Science Publishers B.V. (North Holland), 1989.
3. G. Berry. Esterel on hardware. *Philosophical Transactions of the Royal Society of London, Series A*, 19(2):87–152, 1992.
4. G. Berry. The semantics of pure Esterel. In M. Broy, editor, *Program Design Calculi*, volume 118 of *Series F: Computer and System Sciences*, pages 361–409. NATO ASI Series, 1993.
5. G. Berry. The constructive semantics of pure Esterel. Draft version 3. `http://www-sop.inria.fr/meije/`, July 1999.
6. G. Berry. The Esterel v5 language primer. `http://www-sop.inria.fr/meije/`, July 2000.
7. G. Berry. The foundations of Esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.
8. G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
9. F. Boussinot and R. de Simone. The Esterel language. *Another Look at Real Time Programming, Proceedings of the IEEE*, 79:1293–1304, 1991.
10. E. Closse, M. Poize, J. Pulou, P. Vernier, and D. Weil. Saxo-rt: Interpreting Esterel semantic on a sequential execution structure. *Electronic Notes in Theoretical Computer Science*, 65, 2002.
11. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252, 1977.
12. N. G. de Bruijn. Lambda calculus notation with nameless dummies. a tool for automatic formula manipulation with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.
13. S.A. Edwards. Compiling Esterel into sequential code. In *Proceedings CODES'99*, Rome, Italy, May 1999.
14. S.A. Edwards. *Languages for Digital Embedded Systems*. Kluwer, 2000.
15. G. Gonthier. *Sémantique et modèles d'exécution des langages réactifs synchrones: application à Esterel*. Thèse d'informatique, Université d'Orsay, Paris, France, March 1988.
16. N. Halbwachs. Delay analysis in synchronous programs. In *Computer Aided Verification*, pages 333–346, 1993.
17. N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer, 1993.
18. F. Mignard. *Compilation du langage Esterel en systèmes d'équations booléennes*. Thèse d'informatique, Ecole des Mines de Paris, October 1994.
19. R. Milner. *Communication and Concurrency*. Series in Computer Science. Prentice Hall, 1989.
20. G. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Aahrus University, 1981.
21. H. Toma. *Analyse constructive et optimisation séquentielle des circuits générés à partir du langage synchrone réactif Esterel*. Thèse d'informatique, Ecole des Mines de Paris, September 1997.

A Logical Behavioral Semantics of Esterel

Reactions are defined in a structural operational style [20] by a statement transition relation of the form:

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

Here, E' lists the free signals of p emitted by p under the hypothesis that E is the set of present signals, because of (i) the statement p itself and (ii) its context/environment. The rules of the semantics enforce E' to be a subset of E. k is the completion code of the reaction in the sense of Section 3.4. If $k \neq 0$ then p' represents the new state reached by p after the reaction, that is to say the residual statement that has to be executed in the next reaction.

$$(1)\quad \texttt{nothing} \xrightarrow[E]{\emptyset,\,0} \texttt{nothing}$$

$$\frac{S \in E \qquad p \xrightarrow[E]{E',\,k} p'}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \xrightarrow[E]{E',\,k} p'}\quad(7)$$

$$(2)\quad \texttt{pause} \xrightarrow[E]{\emptyset,\,1} \texttt{nothing}$$

$$\frac{S \notin E \qquad q \xrightarrow[E]{F',\,l} q'}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \xrightarrow[E]{F',\,l} q'}\quad(8)$$

$$(3)\quad \texttt{exit}\ T \xrightarrow[E]{\emptyset,\,k_T} \texttt{nothing}$$

$$\frac{p \xrightarrow[E]{E',\,k} p' \qquad k = 0 \text{ or } k = k_T}{\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end} \xrightarrow[E]{E',\,0} \texttt{nothing}}\quad(9)$$

$$(4)\ \frac{S \in E}{\texttt{emit}\ S \xrightarrow[E]{\{S\},\,0} \texttt{nothing}}$$

$$\frac{p \xrightarrow[E]{E',\,k} p' \qquad k > 0 \text{ and } k \neq k_T}{\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end} \xrightarrow[E]{E',\,k} \texttt{trap}\ T\ \texttt{in}\ p'\ \texttt{end}}\quad(10)$$

$$(5)\ \frac{p \xrightarrow[E]{E',\,k} p' \qquad k \neq 0}{p;\ q \xrightarrow[E]{E',\,k} p';\ q}$$

$$\frac{p \xrightarrow[E]{E',\,k} p' \qquad q \xrightarrow[E]{F',\,l} q}{[p||q] \xrightarrow[E]{E'\cup F',\,\max(k,l)} [p'||q']}\quad(11)$$

$$(6)\ \frac{p \xrightarrow[E]{E',\,0} p' \qquad q \xrightarrow[E]{F',\,l} q}{p;\ q \xrightarrow[E]{E'\cup F',\,l} q'}$$

$$\frac{p \xrightarrow[E]{E',\,k} p' \qquad k \neq 0}{\texttt{loop}\ p\ \texttt{end} \xrightarrow[E]{E',\,k} p';\ \texttt{loop}\ p\ \texttt{end}}\quad(12)$$

$$(13)\ \frac{p \xrightarrow[E\cup\{S\}]{E',\,k} p' \qquad S \in E'}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \xrightarrow[E]{E'\backslash\{S\},\,k} \texttt{signal}\ S\ \texttt{in}\ p'\ \texttt{end}}$$

$$(14)\ \frac{p \xrightarrow[E\backslash\{S\}]{E',\,k} p' \qquad S \notin E'}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \xrightarrow[E]{E',\,k} \texttt{signal}\ S\ \texttt{in}\ p'\ \texttt{end}}$$

Fig. 11. Behavioral semantics

Figure 11 sketches the semantics as a set of deduction rules. For simplicity, we omit the rules defining the suspend statement.

A valuation E is *admissible* for the statement p iff there exists a unique proof tree that establishes a fact of the form $p \xrightarrow[E]{E',k} p'$.

The rules 13 and 14 introduce potential non-determinism in the system. As announced in Section 2.7, in order to avoid *guesses*, more powerful semantic tools are required, that is to say the *constructive semantics of Esterel*.

Remark the side condition $k \neq 0$ in the rule 12. It corresponds to rejecting instantaneous loop bodies. Note the systematic unrolling. It takes care of schizophrenia.

For detailed explanations of these rules please refer to [5].

B Proof of the Analysis of Section 3.4

The rules of Figure 6 are derived from the rules of Figure 11 via the abstraction:

$$p \xrightarrow[E]{E',k} p' \text{ is abstracted into } p \xrightarrow[\cdot]{\cdot,k} \cdot \text{ (that we note } p \hookrightarrow k \text{ in the sequel).}$$

It consists in forgetting E, E' and p' in the rules. The set $\mathcal{K}_p$ introduced in Section 3.4 precisely gathers all completion codes that can be derived for the statement p in the abstract proof domain. Figure 12 lists the abstract deduction rules corresponding to the concrete rules of Appendix A. The rules of Figure 6 are obtained by regrouping the abstract rules corresponding to the same statement (i.e. rules 5 and 6, rules 7 and 8, rules 9 and 10, rules 13 and 14).

$$(1) \quad \texttt{nothing} \hookrightarrow 0 \qquad\qquad \frac{p \hookrightarrow k}{\texttt{present } S \texttt{ then } p \texttt{ else } q \texttt{ end} \hookrightarrow k} \quad (7)$$

$$(2) \quad \texttt{pause} \hookrightarrow 1 \qquad\qquad \frac{q \hookrightarrow l}{\texttt{present } S \texttt{ then } p \texttt{ else } q \texttt{ end} \hookrightarrow l} \quad (8)$$

$$(3) \quad \texttt{exit } T \hookrightarrow k_T \qquad\qquad \frac{p \hookrightarrow k \quad k = 0 \text{ or } k = k_T}{\texttt{trap } T \texttt{ in } p \texttt{ end} \hookrightarrow 0} \quad (9)$$

$$(4) \quad \texttt{emit } S \hookrightarrow 0 \qquad\qquad \frac{p \hookrightarrow k \quad k > 0 \text{ and } k \neq k_T}{\texttt{trap } T \texttt{ in } p \texttt{ end} \hookrightarrow k} \quad (10)$$

$$(5) \quad \frac{p \hookrightarrow k \quad k \neq 0}{p;\ q \hookrightarrow k} \qquad\qquad \frac{p \hookrightarrow k \quad q \hookrightarrow l}{[p||q] \hookrightarrow \max(k,l)} \quad (11)$$

$$(6) \quad \frac{p \hookrightarrow 0 \quad q \hookrightarrow l}{p;\ q \hookrightarrow l} \qquad\qquad \frac{p \hookrightarrow k \quad k \neq 0}{\texttt{loop } p \texttt{ end} \hookrightarrow k} \quad (12)$$

$$(13) \frac{p \hookrightarrow k}{\texttt{signal } S \texttt{ in } p \texttt{ end} \hookrightarrow k} \qquad\qquad \frac{p \hookrightarrow k}{\texttt{signal } S \texttt{ in } p \texttt{ end} \hookrightarrow k} \quad (14)$$

Fig. 12. Abstract deduction rules