Approximate Reachability for Dead Code Elimination in Esterel*

Olivier Tardieu and Stephen A. Edwards

Department of Computer Science, Columbia University, New York
{tardieu,sedwards}@cs.columbia.edu

Abstract. Esterel is an imperative synchronous programming language for the design of reactive systems. Esterel* extends Esterel with a non-instantaneous jump instruction (compatible with concurrency, preemption, etc.) so as to enable powerful source-to-source program transformations, amenable to formal verification. In this work, we propose an approximate reachability algorithm for Esterel* and use its output to remove dead code. We prove the correctness of our techniques.

1 Introduction

Esterel [1–3] is a synchronous parallel programming language. Its syntax is imperative, fit for the design of reactive systems where discrete control aspects prevail. Sophisticated controllers may be described using sequential and parallel compositions of behaviors, suspension and preemption mechanisms, conditionals, loops, and synchronization through instantly broadcast signals. Both software [4, 5] and hardware [6, 7] synthesis are supported.

In this paper, we present an efficient, conservative reachability computation for Esterel* and use its output to direct a dead code elimination procedure. Esterel* [8–10] extends the language with a non-instantaneous jump statement that enables many compilation steps to be performed at the source level, making them easier to share across back ends.

We apply our dead code elimination procedure to cleaning up the output of structural transformations in optimizing compilers. Although in theory such transformations can be carefully engineered to avoid introducing dead code, doing so is generally very difficult. Instead, creating and proving a dead code elimination procedure for Esterel* as we do here frees us from this concern, and is a necessary first step toward a verified optimizing compiler. Previous attempts at the mathematical proof of an Esterel compiler [11, 12] have been limited to simple, inefficient compilers.

Because the only known precise reachability algorithms for Esterel or Esterel* are exponential [13–15], we take a cheap, conservative approach that is, however, accurate enough to clean up most machine-generated code.

Source-level reachability analysis for Esterel is both challenging and rewarding. For example, Esterel's concurrent exception-handling mechanism demands any reachability algorithm track the relative priorities of exceptions, but this is fairly easy to do because the needed information is explicit in the program source. The addition of the unstructured jump instruction to Esterel* further complicates an already complex issue.

Surprisingly, not every piece of code in an Esterel program that is never executed can be removed. In fact, the execution of an Esterel program relies on some preliminary probing, called causality analysis. As a result, a piece of code is truly irrelevant if and only if it is not only never executed but also never probed. Because of this distinction, previous work on dead code elimination for concurrent but not synchronous languages is largely irrelevant for Esterel.

Several such causality analyses have been proposed for Esterel by means of various formal semantics. For instance, in the so-called logical causality of the logical semantics [16], "`present S then nothing end`" and "`nothing`" are equivalent (strongly bisimilar). But, in the causality analysis of the constructive semantics [3], they are not, as the test may lead to a deadlock. However, with a maximal causality analysis [17], they would again be equivalent. In general, removing apparently useless code may turn a correct program w.r.t. the logical semantics into an incorrect one (cf. Section 2.3), or an incorrect program w.r.t. the constructive semantics into a correct one (by removing deadlock conditions).

Here, we take a conservative approach and build our analyses from an abstract semantics for Esterel* that safely approximates the logical, constructive, and deterministic semantics [18] of the language by simply ignoring signals. The correctness proofs for the dead code elimination, which we obtain for a particular choice of concrete semantics—a logical semantics—can be easily adapted to the other semantics. This also makes the analysis very efficient.

In Section 2, we describe the Esterel* language and its concrete and abstract semantics. In Section 3 we start with single-step reachability, which answers whether state s can be reached starting from state s_0 in one step of execution of the program p, and then apply a fixed-point iteration to answer whether state s can ever be reached during the execution of p (thus going beyond a purely structural analysis). Finally, in Section 4, we show how to match reachable states to the source code in order to eliminate dead code. We conclude in Section 5.

2 Esterel*

We consider Berry's kernel Esterel language [3] extended with the `gotopause` instruction of Tardieu [8]. We describe its syntax and main semantic features in Section 2.1 and provide a formal semantics for it in Section 2.2, from which we derive an abstract semantics in Section 2.3 that we use for reachability analysis.

2.1 Syntax and Intuitive Semantics

Fig. 1 is the grammar of our language. Non-terminals p and q denote programs, S signal names, ℓ positive integer labels, and d positive integer exit levels.

We denote $\mathcal{L}_p$ the set of labels of the `pause` instructions in p. Our definitions of the "`present`," "`;`," and "`||`" constructs insist these labels are pairwise distinct. In contrast, labels in `gotopause` instructions are unconstrained.

$p, q ::=$	nothing	Do nothing; terminate instantly
	ℓ:pause	Suspend the execution for one instant
	gotopause ℓ	Instantly branch to "ℓ:pause", which suspends the execution for one instant
	signal S in p end	Declare signal S in p and execute p
	emit S	Emit signal S and terminate instantly
	present S then p else q end	Execute p if S is present; q otherwise
	p ; q	Execute p followed by q if/when p terminates
	p || q	Execute p in parallel with q
	loop p end	Repeat p forever
	try p end	Execute p, catching exits in p
	exit d	Exit d enclosing "try ... end" blocks

Fig. 1. Syntax of Esterel*. In statements where both p and q appear, they must contain unique pause labels, i.e., $\mathcal{L}_p \cap \mathcal{L}_q = \emptyset$.

The infix ";" operator binds tighter than "||". Brackets ([and]) may be used to group statements in arbitrary ways. In a present statement, then or else branches may be omitted. For example, "present S then p end" is shorthand for "present S then p else nothing end". Similarly, we may omit the label "ℓ:" of the instruction "ℓ:pause" in a program if there is no matching "gotopause ℓ" in this program.

Instants and reactions. An Esterel* program runs in steps called *reactions* in response to the *ticks* of a *global clock*. Each reaction takes one *instant*. Primitive constructs execute in the same instant except the pause and gotopause instructions. When the clock ticks, a reaction occurs, which computes the *output signals* and the *new state* of the program from the *input signals* and the *current state* of the program. It may either finish the execution instantly or delay part of it until the next instant because it reached at least one pause or gotopause instruction. In this case, the execution is resumed in the next tick of the clock from the locations of the pause instructions reached in the previous instant.

The program "emit A ; pause ; emit B ; emit C ; pause ; emit D" emits signal A in the first instant of its execution, then emits B and C in the second instant, finally emits D and terminates in the third instant. It takes three instants to complete, i.e., proceeds by three reactions. To the environment, signals B and C appear *simultaneously* since their emissions occur in the same instant.

Synchronous concurrency and preemption. One reaction of the parallel composition "p || q" is made of exactly one reaction of each non-terminated branch, until all branches terminate.

In sequential code, the "exit d" instruction jumps to the end of d enclosing "try ... end" blocks. When "exit d" executes in a group of parallel branches, it also terminates all the other branches. In Fig. 2a, A and D are emitted in the first instant, then B, E, and G in the second and final one. Neither C nor F is emitted. However, "exit 1" in the first branch does not prevent E from being emitted in the second one. This is *weak preemption.*

```
try
  emit A ; pause ; emit B ; exit 1 ; emit C
||
  emit D ; pause ; emit E ; pause ; emit F
end ;
emit G
```

(a)

```
try
  try
    try
      exit 1 || exit 2
    end ; emit A
  end ; emit B
end
```

(b)

Fig. 2. Concurrency and Preemption: Two Examples

When groups of statements running in parallel execute multiple `exit` instructions, priority is given to the branch with the highest exit level, i.e., the exit level of a group of parallel branches is the maximum executed level. Thus, the program in Fig. 2b only emits `B`.

Loops. The program "`loop emit S ; pause end`" emits `S` in each instant and never terminates. Combining `loop`, `try`, and `exit` constructs can produce loops that terminate after a finite number of iterations. Loop bodies may not be instantaneous [19]. For instance, "`loop emit S end`" is illegal since it suggests an unbounded amount of work in a reaction.

Signals. The instruction "`signal` S `in` p `end`" declares the *local* signal S in p. The free signals of a program are said to be *interface* signals for this program.

In an instant, a signal S is *emitted* if and only if one or more "`emit` S" statements are executed that instant. The *status* of S is either *present* or *absent*. A local signal is present iff it is emitted. An interface signal is present iff it is emitted or provided by the *environment*. If S is present in an instant then all "`present` S `then` p `else` q `end`" statements executed in this instant execute their then branches in that instant, otherwise they all execute their else branches.

The presence of a signal is therefore not persistent. For example, in program "`signal S in emit S ; pause ; present S then emit O end end`," signal `S` is emitted and thus present in the first instant of execution only, therefore `O` is not emitted by this statement, as `S` is absent at the time of the test.

Interface signals can interact with local signals. For example, in

```
signal S in
  present S then emit O end  ||  present I then emit S end
end,
```

signal `I` is present iff it is provided by the environment. Signal `S` is emitted, thus present, iff `I` is present. Signal `O` is emitted iff `S` is present. As a result, `O` is present iff `I` or `O` is provided by the environment. Only `I` and `O` may be observed by the environment as `S` cannot escape its scope of definition.

The instantaneous broadcast and instantaneous feedback implied by the *signal coherence law*, i.e., a local signal is present iff emitted, raise correctness issues. Broadly, there are three issues, as illustrated by the following examples:

1. In "`signal S in present S then emit S end end`," `S` could be absent or present. Such *non-deterministic* programs are illegal.

2. In "`signal S in present S else emit S end end`," `S` can neither be absent nor present. This program is illegal because it is *non-reactive*: it has no possible behavior.
3. Finally, signals can be self-reinforcing. Signal `S` can only be present in the program "`signal S in present S then emit S else emit S end end`" for instance. This program is said to be *logically correct*, being both reactive and deterministic. However, it is not *causal* (for the constructive causality of the constructive semantics), since the status of `S` has to be guessed before being confirmed. In summary, this program is legal w.r.t. a logical semantics, but not w.r.t. the constructive semantics.

Strengthening the signal coherence law to precisely define the semantics of such intricate examples is the central concern in choosing a causality analysis for Esterel. In this work, as mentioned before, we do not want to commit ourselves to a particular set of choices. As a result, we shall not only formalize a concrete semantics for the language (making such choices), but also an abstract semantics that conservatively approximates many possible concrete semantics.

Jumps. The `gotopause` instruction permits jumps in Esterel. The execution of the code following the target `pause` starts exactly one instant after the code preceding the `gotopause` instruction terminates. It makes it possible to specify state machines in a natural way and allows loops to be expressed differently. For instance, "`1:pause ; emit S ; gotopause 1`" emits `S` in each instant of execution starting from the second one. In fact, "`loop emit S ; pause end`" can be expanded into "`try exit 1 ; 1:pause end ; emit S ; gotopause 1`" using the pattern "`try exit 1 ; ... end`" to avoid a startup delay.

Just as "`exit` d" in "`exit` d `|| pause`" has priority over the `pause` instruction, it preempts the jump in "`exit` d `|| gotopause` ℓ".

We make no assumptions about the label in a `gotopause` instruction. A "`gotopause` ℓ" instruction may have no target if the program contains no corresponding "ℓ`:pause`" instruction. In such a case, the completion of the execution is delayed by one instant, but nothing takes place in its last instant. For example, the execution of "`gotopause 1; emit S`" takes two instants; `S` is not emitted in the second instant of execution.

Concurrent `gotopause` instructions may target arbitrary `pause` locations. In "`[gotopause 1 || gotopause 2] ; [1:pause ; ... || 2:pause ; ...]`" for instance, this is fine: in the second instant, the execution is resumed from the locations of the two `pause` instructions in parallel. In contrast, in the program "`[gotopause 1 || gotopause 2] ; 1:pause ; ... ; 2:pause ; ...`," resuming the execution from two locations in a sequence does not make sense. We say that this last program is not *well formed*, which we formalized earlier [10].

In this paper however, we have no need for such a correctness criterion. We decide that, in the last example and in similar cases, the execution will non-deterministically restart from either `pause` location. Importantly, giving such a peculiar semantics to non-well-formed programs (which are illegal anyway) does not make the analysis of well-formed program more costly or less precise, but simplifies the formalism.

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

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

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

$$\frac{\ell \in L_0}{\ell\texttt{:pause}/L_0 \xrightarrow[E]{\emptyset,\,0} \emptyset} \qquad \text{(resume)}$$

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

$$\texttt{exit}\ d \xrightarrow[E]{\emptyset,\,d+1} \emptyset \qquad \text{(exit)}$$

$$\frac{p\backslash X \xrightarrow[E]{O,\,k} L}{\texttt{try}\ p\ \texttt{end}\backslash X \xrightarrow[E]{O,\,\downarrow k} L} \qquad \text{(try)}$$

$$\frac{p\backslash X \xrightarrow[E]{O,\,k} L \qquad k \neq 0}{\texttt{loop}\ p\ \texttt{end}\backslash X \xrightarrow[E]{O,\,k} L} \qquad \text{(no-loop)}$$

$$\frac{p\backslash X \xrightarrow[E]{O,\,0} \emptyset \qquad q \xrightarrow[E]{O',\,k} L}{p\ \texttt{;}\ q\backslash X \xrightarrow[E]{O \cup O',\,k} L} \qquad \text{(seq)}$$

$$\frac{p\backslash X \xrightarrow[E]{O,\,k} L \qquad k \neq 0}{p\ \texttt{;}\ q\backslash X \xrightarrow[E]{O,\,k} L} \qquad \text{(seq-left)}$$

$$\frac{q/L_0 \xrightarrow[E]{O,\,k} L}{p\ \texttt{;}\ q/L_0 \xrightarrow[E]{O,\,k} L} \qquad \text{(seq-right)}$$

$$\frac{p\backslash X \xrightarrow[E \cup \{S\}]{O,\,k} L \qquad S \in O}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end}\backslash X \xrightarrow[E]{O\backslash\{S\},\,k} L} \qquad \text{(signal+)}$$

$$\frac{p\backslash X \xrightarrow[E\backslash\{S\}]{O,\,k} L}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end}\backslash X \xrightarrow[E]{O,\,k} L} \qquad \text{(signal−)}$$

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

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

$$\frac{p/L_0 \xrightarrow[E]{O,\,k} L}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end}/L_0 \xrightarrow[E]{O,\,k} L} \qquad \text{(present-left)}$$

$$\frac{q/L_0 \xrightarrow[E]{O,\,k} L}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end}/L_0 \xrightarrow[E]{O,\,k} L} \qquad \text{(present-right)}$$

$$\frac{p/L_0 \xrightarrow[E]{O,\,0} \emptyset \qquad p \xrightarrow[E]{O',\,k} L \qquad k \neq 0}{\texttt{loop}\ p\ \texttt{end}/L_0 \xrightarrow[E]{O \cup O',\,k} L} \qquad \text{(loop)}$$

$$\frac{p/L_0 \xrightarrow[E]{O,\,k} L \qquad L_0 \cap \mathcal{L}_q = \emptyset}{p\ \texttt{||}\ q/L_0 \xrightarrow[E]{O,\,k} L} \qquad \text{(par-left)}$$

$$\frac{q/L_0 \xrightarrow[E]{O,\,k} L \qquad L_0 \cap \mathcal{L}_p = \emptyset}{p\ \texttt{||}\ q/L_0 \xrightarrow[E]{O,\,k} L} \qquad \text{(par-right)}$$

$$\frac{p\backslash X \xrightarrow[E]{O,\,k} L \qquad q\backslash X \xrightarrow[E]{O',\,l} L' \qquad m = \max(k, l)}{p\ \texttt{||}\ q\backslash X \xrightarrow[E]{O \cup O',\,m} \begin{cases} L \cup L' & \text{if } m = 1 \\ \emptyset & \text{if } m \neq 1 \end{cases}} \qquad \text{(par)}$$

Fig. 3. Logical State Semantics

2.2 Logical State Semantics

In Fig. 3, we specify the logical state semantics of Esterel* as a set of facts and deduction rules in a structural operational style [20]. Reactions of a program p are expressed via two kinds of labeled transitions:

$p \xrightarrow[E]{O,\,k} L$	for the first instant of execution
$p/L_0 \xrightarrow[E]{O,\,k} L$	for subsequent instants of execution, L_0 being the set of `pause` locations (labels) the execution is resumed from.

corresponding to two classes of program *states* [7]: a unique *initial state*, simply written p; and many *intermediate states*, written p/L_0 for $L_0 \subseteq \mathbb{N}$. We denote $\mathcal{S}_p$ the set of all states of p and $p\backslash X$ a state of p of either class.

The set O lists the interface signals emitted by the reaction. The set E lists the interface signals assumed present at the time of the reaction. The set $L \subseteq \mathcal{L}(p)$ lists the labels of the `pause` and `gotopause` instructions *reached* by the reaction. The *completion code* $k \in \mathbb{N}$ encodes the status of the execution:

- $k = 0$ if the execution completes normally. L is empty.
- $k = 1$ if the reaction does not complete the execution of p. L is not empty.
- $k = 2, 3, \ldots$ if the execution terminates because of an `exit` instruction and $k - 1$ enclosing "`try` ... `end`" blocks must be exited. L is empty.

Technically, it is easier not to require L_0 in p/L_0 to be a subset of $\mathcal{L}(p)$. Nevertheless, we can identify the states p/L_0 and p/L_1 if $L_0 \cap \mathcal{L}_p = L_1 \cap \mathcal{L}_p$, thus only consider $2^{|\mathcal{L}_p|}$ intermediate states for the program p, thanks to:

Lemma 1. *$p/L_0 \xrightarrow[E]{O,k} L$ iff $p/(L_0 \cap \mathcal{L}_p) \xrightarrow[E]{O,k} L$.*

Proof. This and the following lemmas are established by induction on the structure of a program, or the structure of a proof tree of a reaction. For lack of space, we shall not include the proofs in the paper.

Rule (exit) defines the completion code of "`exit` d" as $d + 1$. In rule (try), if k is the completion code of p, then the completion code of "`try` p `end`" is:

$$\downarrow\!k = \begin{cases} 0 & \text{if } k = 0 \text{ or } k = 2 \quad \text{(normal termination or caught exception)} \\ 1 & \text{if } k = 1 \quad \text{(non-terminated execution)} \\ k - 1 & \text{if } k > 2 \quad \text{(uncaught exception)} \end{cases}$$

Rule (no-loop) applies when the control does not reach the end of the loop; otherwise rule (loop) does.

Rule (par) applies when a parallel statement is first reached—its execution starts—or restarted from both branches. Rules (par-left) and (par-right) apply when the parallel statement is resumed from one branch only, that is to say when the execution of the other branch has already completed.

As announced, if a state points to several locations in a sequence, the execution is non-deterministically resumed from one of them. For instance, respectively by rule (seq) and rule (seq-right),

`1:pause ; 2:pause`$/\{1, 2\} \xrightarrow[\emptyset]{\emptyset, 1} \{2\}$ and `1:pause ; 2:pause`$/\{1, 2\} \xrightarrow[\emptyset]{\emptyset, 0} \emptyset$.

Rule (present) applies when a "`present` S" statement is reached with S present. Rule (absent) applies instead if S is absent. Rules (present-left) and (present-right) specify how the execution of the `present` statement is resumed (non-deterministically from either branch if both $L_0 \cap \mathcal{L}_p \neq \emptyset$ and $L_0 \cap \mathcal{L}_q \neq \emptyset$).

Rules (emit), (signal+), and (signal−) enforce the signal coherence law. We shall not discuss it further as we now abstract signals in this formal semantics.

2.3 Abstract Semantics

In "`signal S in present S then emit S else emit S end end`," `S` must be present. Therefore, "`else emit S`" is never executed. However, removing the else branch changes the behavior of the program, since `S` may be both absent and present in "`signal S in present S then emit S end end`". Because of signals, never executed code is not necessarily dead. Therefore, we choose to

$$\texttt{nothing} \xrightarrow{0} \emptyset \qquad \text{(nothing)}$$

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

$$\ell\texttt{:pause} \xrightarrow{1} \{l\} \qquad \text{(pause)}$$

$$\frac{\ell \in L_0}{\ell\texttt{:pause}/L_0 \xrightarrow{0} \emptyset} \qquad \text{(resume)}$$

$$\texttt{gotopause}\ \ell \xrightarrow{1} \{l\} \qquad \text{(goto)}$$

$$\texttt{exit}\ d \xrightarrow{d+1} \emptyset \qquad \text{(exit)}$$

$$\frac{p\backslash X \xrightarrow{k} L}{\texttt{try}\ p\ \texttt{end}\backslash X \xrightarrow{\downarrow k} L} \qquad \text{(try)}$$

$$\frac{p\backslash X \xrightarrow{0} \emptyset \quad q \xrightarrow{k} L}{p\ \texttt{;}\ q\backslash X \xrightarrow{k} L} \qquad \text{(seq)}$$

$$\frac{p\backslash X \xrightarrow{k} L \quad k \neq 0}{p\ \texttt{;}\ q\backslash X \xrightarrow{k} L} \qquad \text{(seq-left)}$$

$$\frac{q/L_0 \xrightarrow{k} L}{p\ \texttt{;}\ q/L_0 \xrightarrow{k} L} \qquad \text{(seq-right)}$$

$$\frac{p\backslash X \xrightarrow{k} L}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end}\backslash X \xrightarrow{k} L} \qquad \text{(signal)}$$

$$\frac{p\backslash X \xrightarrow{k} L}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end}\backslash X \xrightarrow{k} L} \qquad \text{(then)}$$

$$\frac{q\backslash X \xrightarrow{k} L}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end}\backslash X \xrightarrow{k} L} \qquad \text{(else)}$$

$$\frac{p/L_0 \xrightarrow{0} \emptyset \quad p \xrightarrow{k} L \quad k \neq 0}{\texttt{loop}\ p\ \texttt{end}/L_0 \xrightarrow{k} L} \qquad \text{(loop)}$$

$$\frac{p\backslash X \xrightarrow{k} L \quad k \neq 0}{\texttt{loop}\ p\ \texttt{end}\backslash X \xrightarrow{k} L} \qquad \text{(no-loop)}$$

$$\frac{p\backslash X \xrightarrow{k} L \quad q\backslash X \xrightarrow{l} L' \quad m = \max(k,l)}{p\ \texttt{||}\ q\backslash X \xrightarrow{m} \begin{cases} L \cup L' & \text{if } m = 1 \\ \emptyset & \text{if } m \neq 1 \end{cases}} \qquad \text{(par)}$$

$$\frac{p/L_0 \xrightarrow{k} L \quad L_0 \cap \mathcal{L}_q = \emptyset}{p\ \texttt{||}\ q/L_0 \xrightarrow{k} L} \qquad \text{(par-left)}$$

$$\frac{q/L_0 \xrightarrow{k} L \quad L_0 \cap \mathcal{L}_p = \emptyset}{p\ \texttt{||}\ q/L_0 \xrightarrow{k} L} \qquad \text{(par-right)}$$

Fig. 4. Abstract Semantics

first abstract signals in the concrete semantics, then define dead code w.r.t. the resulting abstract semantics.

The abstract semantics is easily derived from the logical state semantics by making abstraction of signals. Its rules are gathered in Fig. 4. Rules (present) and (present-left) are merged into a unique (then) rule, (absent) and (present-right) into (else), and (signal+) and (signal−) into (signal).

It safely approximates the concrete semantics, i.e., preserves its reactions:

Lemma 2. *If* $p\backslash X \xrightarrow[E]{O,k} L$ *then* $p\backslash X \xrightarrow{k} L$.

Importantly, the abstract semantics also safely approximates other semantics such as the constructive semantics of Berry [3], dedicated to hardware synthesis.

3 Reachability Analysis

We say that a state p/L is *reachable in the execution of* p iff it may result of a chain of reactions starting from p. Exact reachability analysis is in general intractable, even w.r.t. the abstract semantics, as the number of states of a program may be exponential in its size. As a result, we choose to approximate the reachable states by means of reachable labels.

In “`present S then 1:pause else 2:pause || 3:pause end`” for example, we shall aim at computing the set of reachable labels $R = \{1,2,3\}$ rather than the more precise set of reachable states $S = \{p/\{1\}, p/\{2,3\}, p/\emptyset\}$. Of course, we want R to contain all the labels in S, i.e., $\bigsqcup S = \bigcup_{p/L_i \in S}\{L_i\}$, while

$$
\begin{array}{rl|rl}
s(\texttt{nothing}) = & \{0\} & s(\texttt{emit } S) = & \{0\} \\
s(\texttt{gotopause } \ell) = & \{1_\ell\} & s(\ell\texttt{:pause}) = & \{1_\ell\} \\
s(\texttt{signal } S \texttt{ in } p \texttt{ end}) = & s(p) & s(\texttt{try } p \texttt{ end}) = & {\downarrow} s(p) \\
s(\texttt{present } S \texttt{ then } p \texttt{ else } q \texttt{ end}) = & s(p) \cup s(q) & s(\texttt{exit } d) = & \{d{+}1\} \\
s(\texttt{loop } p \texttt{ end}) = & s(p) \setminus \{0\} & s(p \texttt{ || } q) = & \max(s(p), s(q)) \\
\end{array}
$$

$$
s(p \texttt{ ; } q) = \begin{cases} s(p) & \text{if } 0 \notin s(p) \\ (s(p) \setminus \{0\}) \cup s(q) & \text{if } 0 \in s(p) \end{cases}
$$

Fig. 5. Instantaneous Reachability from the Initial State

being as small as possible. In other words, we shall only consider sets of states of the form $\{p/L\}_{L \subseteq R}$ for $R \subseteq \mathbb{N}$.

While this choice leads to a less precise reachability analysis, it especially makes sense in the context of dead code elimination. In any case, the input of the dead code elimination algorithm (cf. Section 4) will be the list of alive `pause` instructions, that is to say R rather than S.

In Section 3.1, we first consider instantaneous reachability, i.e., reachability through a single reaction. We conclude for chains of reactions in Section 3.2.

3.1 Instantaneous Reachability

By definition,

– the intermediate state p/L is *instantly reachable from* $p\backslash X$ iff $\exists k : p\backslash X \xrightarrow{k} L$;
– the label ℓ is *instantly reachable from* $p\backslash X$ iff $\exists k, \exists L : p\backslash X \xrightarrow{k} L \wedge \ell \in L$.

We write $p\backslash X \Rightarrow p/L$ in the first case, $p\backslash X \Rightarrow \ell$ in the second. Importantly,

Lemma 3. *If* $p/L_0 \Rightarrow \ell$ *then* $\exists \ell_0 \in L_0 : p/\{\ell_0\} \Rightarrow \ell$.

Lemma 4. *If* $L_0 \subseteq R$ *and* $p/L_0 \Rightarrow p/L$ *then* $L \subseteq \bigcup_{\ell_0 \in R} \{\ell \in \mathcal{L}_p : p/\{\ell_0\} \Rightarrow \ell\}$.

Therefore, it makes sense to approximate a family $S = (p/L_i)_{i \in I}$ of intermediate states by the set of labels that appear in these states $R = \bigsqcup S = \bigcup_{i \in I} \{L_i\}$:

– $\bigsqcup \{p/L \in \mathcal{S}_p : p \Rightarrow p/L\} = \{\ell \in \mathcal{L}_p : p \Rightarrow \ell\}$
– $\bigsqcup \left(\bigcup_{p/L_0 \in S} \{p/L \in \mathcal{S}_p : p/L_0 \Rightarrow p/L\} \right) \subseteq \bigcup_{\ell_0 \in \bigsqcup S} \{\ell \in \mathcal{L}_p : p/\{\ell_0\} \Rightarrow \ell\}$

The set of labels instantly reachable from a family S of intermediate states can be safely approximated by the set of labels instantly reachable from the family of states $S' = (p/\{\ell_0\})_{\ell_0 \in \bigsqcup S}$, with the following trade-off:

– Because we replace the family S with up to $2^{|\mathcal{L}(p)|}$ states, by the family S' of at most $|\mathcal{L}(p)|$ states, the cost of the computation improves exponentially.
– Loss of precision may occur if $S \neq S'$. For instance, if p is the program "`try 1:pause ; exit 1 || 2:pause ; 3:pause end`," then the only instantly reachable state is $p/\{1, 2\}$. Because we approximate this state with the set of states $\{p/\emptyset, p/\{1\}, p/\{2\}, p/\{1, 2\}\}$, we end up computing that label 3 may be reachable in the execution of p (being instantly reachable from $p/\{2\}$), although it cannot be reached from $p/\{1, 2\}$.

$$
\begin{array}{rl|rl}
d_R(\texttt{nothing}) = & \emptyset & d_R(\texttt{emit } S) = & \emptyset \\
d_R(\texttt{gotopause } \ell) = & \emptyset & d_R(\ell\texttt{:pause}) = & \begin{cases} \emptyset & \text{if } \ell \notin R \\ \{0\} & \text{if } \ell \in R \end{cases} \\
d_R(\texttt{exit } d) = & \emptyset & & \\
d_R(\texttt{signal } S \texttt{ in } p \texttt{ end}) = & d_R(p) & d_R(\texttt{try } p \texttt{ end}) = & \downarrow d_R(p) \\
d_R(\texttt{present } S \texttt{ then } p \texttt{ else } q \texttt{ end}) = & d_R(p) \cup d_R(q) & d_R(p \texttt{ || } q) = & d_R(p) \cup d_R(q) \\
\hline
\end{array}
$$

$$
d_R(\texttt{loop } p \texttt{ end}) = \begin{cases} d_R(p) & \text{if } 0 \notin d_R(p) \\ (d_R(p) \cup s(p)) \setminus \{0\} & \text{if } 0 \in d_R(p) \end{cases}
$$

$$
d_R(p \texttt{ ; } q) = \begin{cases} d_R(p) \cup d_R(q) & \text{if } 0 \notin d_R(p) \\ (d_R(p) \setminus \{0\}) \cup s(q) \cup d_R(q) & \text{if } 0 \in d_R(p) \end{cases}
$$

Fig. 6. Instantaneous Reachability from Intermediate States

Qualified completion codes. Denoting by "1_ℓ" a completion code 1 due to a `pause` or `gotopause` of label ℓ, we obtain the set of *qualified completion codes* $\mathcal{K} = \{0, 1_0, 1_1, \ldots, 2, 3, \ldots\}$. Formally, $\mathcal{K}$ is $(\mathbb{N} \setminus \{1\}) \cup \bigcup_{\ell \in \mathbb{N}} \{1_\ell\}$. Thanks to $\mathcal{K}$, we shall compute feasible completion codes and reachable labels simultaneously.

To recover regular completion codes from qualified completion codes, we define the projection $k \mapsto \hat{k}$ from $\mathcal{K}$ to $\mathbb{N}$ so that $\forall k \in \mathbb{N} \setminus \{1\} : \hat{k} = k$ and $\forall \ell \in \mathbb{N} : \hat{1_\ell} = 1$.

We equip $\mathcal{K}$ with the preorder "$\leq$" such that $k \leq l$ in $\mathcal{K}$ iff $\hat{k} \leq \hat{l}$ in $\mathbb{N}$. If $K, K' \subseteq \mathcal{K}$ then $\max(K, K')$ is $\{k \in K : \exists k' \in K', k' \leq k\} \cup \{k' \in K' : \exists k \in K, k \leq k'\}$. For instance, $\max(\{0, 1_4, 4, 6\}, \{1_1, 1_3, 3, 4\}) = \{1_1, 1_3, 1_4, 3, 4, 6\}$.

Finally, for $\ell \in \mathbb{N}$, we define $\downarrow 1_\ell = 1_\ell$.

We now compute the labels instantly reachable from p, then the labels instantly reachable from the set of states $\{p/L\}_{L \subseteq R}$ given the set of labels $R \subseteq \mathbb{N}$.

Initial state. Previously [19], we formalized a static analysis that computes the possible completion codes of reactions of the program p for a similar abstract semantics. We now extend this analysis so as to deal with `gotopause` instructions and obtain the labels instantly reachable from p at the same time.

Let $\mathcal{E}$ be the set of all Esterel* programs and $\mathcal{P}(\mathcal{K})$ be the powerset of $\mathcal{K}$. In Fig. 5, we specify the analysis function $s : \mathcal{E} \to \mathcal{P}(\mathcal{K})$ that overapproximates the set of qualified completion codes reachable from the initial state. It is easily derived from the rules of the abstract semantics for initial states. For instance, $s($`present S then exit 7 end || 3:pause || gotopause 4`$) = \{1_3, 1_4, 8\}$.

Lemma 5. *$\exists L \subseteq \mathcal{L}_p : p \xrightarrow{k} L$ iff $\hat{k} \in s(p)$. Moreover, $p \Rightarrow \ell$ iff $1_\ell \in s(p)$.*

Hence, the set of labels instantly reachable from p is:

Lemma 6. $\bigsqcup\{p/L \in \mathcal{S}_p : p \Rightarrow p/L\} = \{\ell \in \mathcal{L}_p : 1_\ell \in s(p)\}$.

Intermediate states. For $R \subseteq \mathbb{N}$, we define $d_R : \mathcal{E} \to \mathcal{P}(\mathcal{K})$ in Fig. 6 by now considering the rules applicable to intermediate states. Intuitively, $d_R(p)$ is meant to be the set of possible qualified completion codes of reactions of $p/\{\ell_0\}$ for $\ell_0 \in R$. If p is "`1:pause ; 2:pause || 3:pause ; 4:pause ; exit 6`" for instance, then $d_{\{1\}}(p) = \{1_2\}$, $d_{\{3,4\}}(p) = \{1_4, 7\}$, $d_{\{1,2,3,4\}}(p) = \{0, 1_2, 1_4, 7\}$.

Lemma 7. *$\exists L \subseteq \mathcal{L}_p : p/\{\ell_0\} \xrightarrow{k} L$ iff $\hat{k} \in d_{\{\ell_0\}}(p)$. Moreover, $p/\{\ell_0\} \Rightarrow \ell$ iff $1_\ell \in d_{\{\ell_0\}}(p)$.*

Lemma 8. *For all R, R', $p : d_{R \cup R'}(p) = d_R(p) \cup d_{R'}(p)$.*

Hence, the set of labels instantly reachable from the states $\{p/L_0\}_{L_0 \subseteq R}$ is:

Lemma 9. $\bigsqcup \left(\bigcup_{L_0 \subseteq R} \{p/L \in \mathcal{S}_p : p/L_0 \Rightarrow p/L\} \right) = \{\ell \in \mathcal{L}_p : 1_\ell \in d_R(p)\}$.

In Lemmas 6 and 9, we obtain equalities. Thus, this reachability analysis is exact w.r.t. the abstract semantics and for set of states of the form $\{p/L\}_{L \subseteq R}$.

3.2 Fixed Point

We define $\Delta_p : L_0 \mapsto \{\ell \in \mathcal{L}_p : 1_\ell \in s(p) \cup d_{L_0}(p)\}$. This function is monotonic over the complete lattice $\mathcal{P}(\mathcal{L}_p)$, therefore [21] it has a least fixed point $\mathcal{R}_p$. We already know that if $p \Rightarrow L$ then $L \subseteq \Delta_p(\emptyset)$ and if $p/L_0 \Rightarrow L$ then $L \subseteq \Delta_p(L_0)$. Therefore, $\mathcal{R}_p$ overapproximates the labels reachable in the execution of p, thus the reachable states.

Theorem 1. *If $p \Rightarrow L_0$, $p/L_0 \Rightarrow L_1$, ..., $p/L_{n-1} \Rightarrow L_n$ then $L_n \subseteq \mathcal{R}_p$.*

4 Dead Code Elimination

Checking program equivalence at the abstract level does not make sense since, for instance, "`nothing`" and "`emit S`" behave the same in the abstract semantics. For $R \subseteq \mathbb{N}$, we say that the programs p and q are:

- *initially equivalent* iff $\forall E, O, k, L : p \xrightarrow[E]{O,k} L \Leftrightarrow q \xrightarrow[E]{O,k} L$, and
- *R-equivalent* iff $\forall L_0 \subseteq R, \forall E, O, k, L : p/L_0 \xrightarrow[E]{O,k} L \Leftrightarrow q/L_0 \xrightarrow[E]{O,k} L$.

As with the definition of instantaneous reachability, these definitions give us the ability to derive program equivalence (a property of executions) from instantaneous equivalence properties (properties of reactions).

Lemma 10. *If p and q are initially equivalent and $\mathcal{R}_p$-equivalent then they are strongly bisimilar, that is to say behave the same in all contexts [22].*

Defining dead (unreachable) code can be managed at the abstract level. In the previous section, we defined instantly reachable labels, i.e., instantly reachable `pause` instructions. In fact, thanks to the structural definitions of s and d_R, we can extend the idea of instantaneous reachability to blocks of code:

- q in p is not *instantly reachable*, iff the computation of $s(p)$ does not involve the computation of $s(q)$.
- q in p is not *R-reachable*, iff $d_R(q)$ is empty and the computation of $d_R(p)$ does not involve the computation of $s(q)$.

Intuitively, q is not instantly reachable in p iff it is in sequence after a block of code r that cannot terminate instantly, that is to say $0 \notin s(r)$. Moreover, q is not R-reachable in p iff q does not contain any `pause` instruction with a label in R and q is not in sequence after a block of code that can terminate instantly if restarted from some `pause` instruction with a label in R.

In the sequel, we shall simplify programs so as to eliminate unreachable code while preserving program equivalence. Although our equivalence proofs depend on the semantics for which we define program equivalence, since our transformations only involve unreachable code, which we define at the abstract level, we claim that similar equivalence results hold for other concrete semantics, provided they share the same abstraction.

Simplifying "`try` p `|| exit 1 ;` q `end`" so as to preserve initial equivalence and R-equivalence for some set R requires to simplify p in the same way. Importantly, in Esterel*, q cannot be simply discarded because it can be reached through `gotopause` instructions. However, because of "`exit 1`," preserving R-equivalence for q is good enough. In other words, if p is "`emit S ; ...`" then "`emit S`" must be preserved. But, if q is, then "`emit S`" can be discarded.

To start with, we define the functions $\{sd_R\}_{R\subseteq\mathbb{N}}$ that gather the possible completion codes of the behaviors of the program p we want to reproduce:

$$\begin{array}{rl} sd_R : \mathbb{B}\times\mathcal{E} & \rightarrow \mathcal{K} \\ (0,p) & \mapsto d_R(p) \\ (1,p) & \mapsto d_R(p) \cup s(p) \end{array}$$

The Boolean b is 0 if only the behaviors of intermediates states (with labels in R) are relevant; b is 1 if, in addition, the initial behaviors of p are relevant.

In Fig. 7, we formalize code elimination as a family of R-equivalence-preserving functions $\{r_R : \mathbb{B} \times \mathcal{E} \rightarrow \mathcal{E}\}_{R\subseteq\mathbb{N}}$, which also preserve initial equivalence, if called with a Boolean parameter equal to 1. These functions are designed to delete as much code as possible while preserving the required equivalences.

As with the definition of d_R before, the structural definition of r_R makes sure that r_R is recursively applied to each block of the initial program, since any statement, whatever its position, can potentially be reached through jumps in Esterel*. However, not all statements can be reached "from the left," that is to say are in sequence after a statement that may terminate. The value of the Boolean parameter b in recursive calls is computed accordingly.

The simplifications implemented by r_R are the following:

- All `emit`, `exit`, and `gotopause` instructions are preserved iff left-reachable.
- A `pause` instruction is preserved if its label is in R or if it is left-reachable.
- The "`try` ... `end`" construct in "`try` p `end`" is preserved if the corresponding exception may occur. If removed, exception levels have to be adjusted accordingly: $\searrow p$ is obtained by decrementing all exit levels in p greater than the number of enclosing "`try` ... `end`" constructs in p, while replacing all `exit` instructions targeting the removed construct by `nothing`. For instance, $r_R(1,$`try exit 1 || exit 2 end`$) = r_R(1,$`nothing || exit 1`$)$.

$$r_R(b,\texttt{nothing}) = \texttt{nothing}$$

$$r_R(b,\texttt{emit } S) = \begin{cases} \texttt{nothing} & \text{if } b=0 \\ \texttt{emit } S & \text{if } b=1 \end{cases}$$

$$r_R(b,\ell\texttt{:pause}) = \begin{cases} \texttt{nothing} & \text{if } (\ell \notin R) \wedge (b=0) \\ \ell\texttt{:pause} & \text{if } (\ell \in R) \vee (b=1) \end{cases}$$

$$r_R(b,\texttt{gotopause } \ell) = \begin{cases} \texttt{nothing} & \text{if } b=0 \\ \texttt{gotopause } \ell & \text{if } b=1 \end{cases}$$

$$r_R(b,\texttt{exit } d) = \begin{cases} \texttt{nothing} & \text{if } b=0 \\ \texttt{exit } d & \text{if } b=1 \end{cases}$$

$$r_R(b,\texttt{try } p \texttt{ end}) = \begin{cases} r_R(b,\searrow p) & \text{if } 2 \notin sd_R(b,p) \\ \texttt{try } r_R(b,p) \texttt{ end} & \text{if } 2 \in sd_R(b,p) \end{cases}$$

$$r_R(b,p \texttt{ ; } q) = \begin{cases} r_R(b,p)\boxed{;}\, r_R(0,q) & \text{if } 0 \notin sd_R(b,p) \\ r_R(b,p)\boxed{;}\, r_R(1,q) & \text{if } 0 \in sd_R(b,p) \end{cases}$$

$$r_R(b,\texttt{signal } S \texttt{ in } p \texttt{ end}) = \begin{cases} r_R(b,p) & \text{if } S \text{ does not occur in } r_R(b,p) \\ \texttt{signal } S \texttt{ in } r_R(b,p) \texttt{ end} & \text{if } S \text{ occurs in } r_R(b,p) \end{cases}$$

$$\begin{array}{r} r_R(b,\texttt{present } S \texttt{ then } p \\ \texttt{else } q \texttt{ end}) \end{array} = \begin{cases} r_R(0,p \texttt{ ; try exit 1 ; } \nearrow q \texttt{ end}) & \text{if } b=0 \\ \texttt{present } S \texttt{ then } r_R(1,p) \texttt{ else } r_R(1,q) \texttt{ end} & \text{if } b=1 \end{cases}$$

$$r_R(b,\texttt{loop } p \texttt{ end}) = \begin{cases} r_R(b,p) & \text{if } 0 \notin sd_R(b,p) \\ \texttt{loop } r_R(1,p) \texttt{ end} & \text{if } 0 \in sd_R(b,p) \end{cases}$$

$$r_R(b,p \texttt{ || } q) = r_R(b,p)\ \boxed{||}\ r_R(b,q)$$

Fig. 7. Dead Code Elimination

- Depending on the possible termination of p in "p `;` q," q is rewritten so as to preserve initial equivalence or not. Moreover, if p or q end up being `nothing` after simplification, it can be discarded as well as the "`;`" operator. We define: $p\boxed{;}$ `nothing` $= p$, `nothing`$\boxed{;}\ q = q$, $p\boxed{;}q = p;\ q$ otherwise.
- Signal declarations are deleted if possible.
- A left-reachable "`present` S `then` p `else` q `end`" test is recursively simplified. A non-left-reachable test is first replaced by "p `; try exit 1 ;` $\nearrow q$ `end`" to remove the test itself while preserving the branches, then simplified. Exception levels in q have to be adjusted: $\nearrow q$ is obtained by incrementing exit levels greater than the number of enclosing "`try` ... `end`" constructs in q. For example, $r_R(0,$`present S then 1:pause ; emit 0 else 2:pause ; exit 3 end`$)$ is equal to $r_R(0,$`1:pause ; emit 0; try exit 1; 2:pause ; exit 4 end`$)$.
- A `loop` construct is deleted if its body never terminates. If it may terminate then initial equivalence must be preserved for the body, whatever b.
- The branches of a parallel are recursively rewritten. The parallel itself may be deleted if a branch reduces to `nothing`, hence the $\boxed{||}$ operator.

We establish initial equivalence and R-equivalence by structural induction:

Lemma 11. *Whatever R, $p \xrightarrow[E]{O,k} L \Leftrightarrow r_R(1,p) \xrightarrow[E]{O,k} L$.*

Lemma 12. *Whatever b, if $L_0 \subseteq R$ then $p/L_0 \xrightarrow[E]{O,k} L \Leftrightarrow r_R(b,p)/L_0 \xrightarrow[E]{O,k} L$.*

```
loop
  try
    present I then
      1:pause;
      exit 1
    end;
    2:pause;
    emit O
  end
end
```

$\xRightarrow{\text{loop expansion}}$

```
try
  exit 1;
  try
    present I then
      1:pause ; exit 1
    end;
    2:pause ; emit O
  end
end;
try
  present I then
    gotopause 1 ; exit 1
  end;
  gotopause 2 ; emit O
end
```

$\xRightarrow{\text{dead code elimination}}$

```
try
  exit 1;
  try
    1:pause;
    exit 1;
    2:pause;
    emit O
  end
end;
present I then
  gotopause 1
end;
gotopause 2
```

Fig. 8. Example

Theorem 2. *p and $r_{\mathcal{R}_p}(1, p)$ behave the same (are strongly bisimilar).*

This dead code elimination procedure is far from complete because of the approximations involved, yet it is already powerful, in particular w.r.t. machine-generated code. For instance, applying it after loop expansion [8] produces compact code, as illustrated in Fig. 8. While the expansion makes two copies of the loop body, the final code only contains one copy of "`present I`" and "`emit O`".

$$\texttt{loop}\ p\ \texttt{end} \xRightarrow{\text{loop expansion}} \texttt{try exit}\ 1\ \texttt{;}\ \nearrow p\ \texttt{end}\ \texttt{;}\ p[\ell\texttt{:pause} \mapsto \texttt{gotopause}\ \ell]$$

5 Conclusions

We have specified a static reachability analysis and an algorithm for dead code elimination in Esterel* programs. While we designed the reachability analysis with dead code elimination in mind, it can be used independently.

By abstracting signals from the analysis, we ensure it is applicable for many different semantics of the language. In addition, it makes it possible to safely dismantle states into their elementary components (locations of `pause` instructions), thus dramatically cutting the cost of the required fixed-point computation. Our analysis is exact and optimal with respect to this approximation.

In fact, apart from a single fixed-point computation, both the analysis and the transformation are obtained from structural traversals of the program source, resulting in simple correctness proofs.

We would like to extend this work in two directions. First, adopting either the logical or the constructive semantics of Esterel*, we shall consider synchronizations through signals, in addition to synchronizations through exceptions that we already take into account. Second, we plan to formalize and check our correctness proofs using a theorem prover, paving the way for embedding this work in a certified compiler for Esterel*.

References

1. Berry, G., Gonthier, G.: The Esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming **19** (1992) 87–152
2. Boussinot, F., de Simone, R.: The Esterel language. Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue **79** (1991) 1293–1304
3. Berry, G.: The constructive semantics of pure Esterel, draft version 3. http://www-sop.inria.fr/esterel.org/ (1999)
4. Closse, E., Poize, M., Pulou, J., Vernier, P., Weil, D.: Saxo-rt: Interpreting Esterel semantic on a sequential execution structure. In: SLAP'02. Volume 65 of Electronic Notes in Theoretical Computer Science., Elsevier (2002)
5. Edwards, S.A., Kapadia, V., Halas, M.: Compiling Esterel into static discrete-event code. In: SLAP'04. Electronic Notes in Theoretical Computer Science, Elsevier (2004)
6. Berry, G.: Esterel on hardware. Philosophical Transactions of the Royal Society of London, Series A **19(2)** (1992) 87–152
7. Mignard, F.: Compilation du langage Esterel en systèmes d'équations booléennes. PhD thesis, Ecole des Mines de Paris (1994)
8. Tardieu, O.: Goto and concurrency: Introducing safe jumps in Esterel. In: SLAP'04. Electronic Notes in Theoretical Computer Science, Elsevier (2004)
9. Tardieu, O., de Simone, R.: Curing schizophrenia by program rewriting in Esterel. In: MEMOCODE'04. (2004)
10. Tardieu, O.: Loops in Esterel: from operational semantics to formally specified compilers. PhD thesis, Ecole des Mines de Paris (2004)
11. Schneider, K.: A verified hardware synthesis of Esterel programs. In: DIPES'00. (2001) 205–214
12. Schneider, K., Brandt, J., Schüele, T.: A verified compiler for synchronous programs with local declarations. In: SLAP'04. Electronic Notes in Theoretical Computer Science, Elsevier (2004)
13. Malik, S.: Analysis of cyclic combinational circuits. In: ICCAD'93. (1993) 618–625
14. Shiple, T., Berry, G., Touati, H.: Constructive analysis of cyclic circuits. In: Proc. International Design and Testing Conf (ITDC), Paris. (1996)
15. Namjoshi, K.S., Kurshan, R.P.: Efficient analysis of cyclic definitions. In: CAV'99. (1999) 394–405
16. Berry, G.: The semantics of pure Esterel. In Broy, M., ed.: Program Design Calculi. Volume 118 of Series F: Computer and System Sciences., NATO ASI Series (1993) 361–409
17. Schneider, K., Brandt, J., Schüele, T., Tuerk, T.: Maximal causality analysis. In: ACSD'05. (2005)
18. Tardieu, O.: A deterministic logical semantics for Esterel. In: SOS Workshop'04. Electronic Notes in Theoretical Computer Science, Elsevier (2004)
19. Tardieu, O., de Simone, R.: Instantaneous termination in pure Esterel. In: SAS'03. Volume 2694 of Lecture Notes in Computer Science., Springer (2003) 91–108
20. Plotkin, G.: A structural approach to operational semantics. Report DAIMI FN-19, Aarhus University, Denmark (1981)
21. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics **5** (1955) 285–309
22. Park, D.: Concurrency and automata on infinite sequences. In: 5th GI Conference. Volume 104 of Lecture Notes in Computer Science., Springer (1981)