A Hardware Implementation of Pure Esterel

G. Berry

Ecole des Mines
Sophia-Antipolis
06565 Valbonne, France

Digital Equipment
Paris Research Laboratory
85, Av. Victor Hugo
92 Rueil Malmaison, France

Abstract

Esterel is a synchronous concurrent programming language dedicated to reactive systems (controllers, protocols, man-machine interfaces, etc.). Esterel has an efficient standard software implementation based on well-defined mathematical semantics. We present a new hardware implementation of the pure synchronization subset of the language. Each program generates a specific circuit that responds to any input in one clock cycle. When the source program satisfies some statically checkable dynamic properties, the circuit is shown to be semantically equivalent to the source program. The hardware translation has been effectively implemented on the programmable active memory Perle0 developed by J. Vuillemin and his group at Digital Equipment.

1 Introduction

Esterel [3, 6, 4, 10] is a synchronous programming language devoted to *reactive systems*, that is to systems that maintain a continuous interaction with their environment by handling hardware or software events. Its software implementation is currently used in industry and education to program software objects such as real-time controllers, communication protocols [5, 23], man-machine interfaces [14], systems drivers, etc. In this paper, we present a hardware implementation of the pure synchronization subset of the language that builds a specific circuit for each program. We prove the correctness of this implementation w.r.t. the mathematical semantics of the language under some conditions to be satisfied by the source program. We describe the experiments made so far and the possible uses of the hardware implementation.

The Perfect Synchrony Hypothesis

Esterel is an imperative concurrent language with very high-level control and event manipulation constructs. It is based on a *perfect synchrony hypothesis* [2], which states that control transmission, communication, and elementary computation actions take no time, or, in other words, that the program is conceptually executed on an infinitely fast machine. The control structures include sequencing, testing, looping, concurrency, and a powerful exception mechanism which is fully compatible with concurrency, unlike in asynchronous concurrent programming languages [1]. The primitive communication device between concurrent statements is *instantaneous broadcasting* of signals.

The perfect synchrony hypothesis is shared by the synchronous data-flow languages LUSTRE [12, 20] and SIGNAL [17, 19]. It makes programming very modular and flexible, and it makes it possible to reconcile input-output determinism and concurrency. This is a great benefit over classical asynchronous languages such as OCCAM or ADA that are inherently non-deterministic, a characteristic that makes reactive programming and debugging needlessly difficult, see [1].

ESTEREL is rigorously defined by well-analyzed mathematical semantics, given in both denotational and operational styles [6, 18].

Esterel in Software

The standard ESTEREL compiler is directly based on one of the mathematical semantics. It uses sophisticated algorithms to translate a concurrent reactive program into an equivalent efficient sequential automaton that can be implemented in any conventional language. Concurrency is compiled away during this process. The resulting automaton can be directly run by actual applications. In addition to the compiler, the ESTEREL environment includes sophisticated tools such as symbolic or graphical simulators and interfaces to automata-based program verification systems such as AUTO [9].

Esterel in Hardware

Since many CAD systems directly support automata-like specifications, ESTEREL programs can be implemented in hardware by first translating them into automata using the standard compiler. However, this indirect translation looses most of the source concurrent structure. This is usually a good idea in software, where run-time concurrency is in fact expensive, but not in hardware, where concurrency is free and should be used as much as possible. Furthermore, there is no simple relation between the source program size and the size of the generated automaton. In the worst case, the automaton size can be exponential in the source program size, and square factors are not rare. Again, this is much more acceptable in software than in hardware.

The direct hardware implementation we present here is conceptually much better; it is based on Gonthier's semantic analysis of ESTEREL [18]. It transforms each program into a digital circuit that exactly reflects the source concurrency and communication structure. The circuit computes the response to any input within exactly one clock cycle, however complex the program is. The translation is purely structural (compositional) and linear in size. However, it is at present limited to the pure synchronization subset of the language, which we call PURE ESTEREL, and it works only under some restrictive conditions to be satisfied by the source program.

The translation is completely formalized and proved correct w.r.t. the mathematical semantics under the above restrictive conditions. Correctness relies on the fact that perfect synchrony does not depart very much from digital circuit synchrony: zero-time is simply replaced by one cycle.

Actual Implementation and Applications

The translation from programs to circuits has been implemented within the existing ESTEREL compiler. We have run very successful experiments using the XILINX^{TM}-based PERLE0 programmable coprocessor developed at DEC Paris Research Laboratory by J. Vuillemin *et. al.* [8, 25].

We are currently investigating two kinds of applications:

- Implementing existing ESTEREL programs in hardware to match high performance constraints. For example, we have directly implemented the kernel of a fast local area network protocol that was developed in Esterel at INRIA [22].
- Programming hardware controllers in ESTEREL. The language turns out to be well-adapted to programming the control part of a circuit, which is known to be difficult and error-prone with usual techniques. We show a toy example in appendix A.

The fact that the language can be implemented either in software or in hardware is useful in two respects: one can use the software programming environment to develop, debug, and verify the programs; one can experiment various trade-offs between hardware and software without changing the source code.

Esterel and Lustre

The LUSTRE synchronous language has also been implemented on hardware at DEC PRL, and the implementations of ESTEREL and LUSTRE are fully compatible[1]. It has to be noted that both languages differ from most existing hardware description languages by the fact that they deal only with *behaviors* and not with hardware objects, and also by the care with which they were mathematically defined and studied. To describe circuits, LUSTRE and ESTEREL are complementary: LUSTRE is well-adapted to data path description, ESTEREL is well-adapted to control automata.

Structure of the Paper

Section 2 presents the pure ESTEREL language and its intuitive semantics. We give enough material for the paper to be self-contained, but not to fully understand the ESTEREL programming style, referring to [4, 6] and to the ESTEREL documentation for these aspects. The mathematical semantics of PURE ESTEREL is given in section 3. Section 4 presents an essential part of the theory of ESTEREL, the coding of states by haltsets. This coding is the root of the hardware translation, whose principle is presented by examples in section 5. The translation is then formalized in section 6 and proved correct in section 7. We discuss the actual implementation on PERLE0 in section 8 and conclude. An appendix gives the example of a simple bus interface and briefly analyzes the adequacy of ESTEREL to program hardware controllers.

[1] A byproduct of our work is a translator from pure ESTEREL into LUSTRE.

2 Pure Esterel

We first present signals and events which are the basic objects manipulated by PURE ESTEREL programs. We then present the kernel language on which the semantics is defined and the full language that includes kernel-definable user-friendly statements.

2.1 Signals and Events

PURE ESTEREL deals with *signals* $\mathtt{S}, \mathtt{S}_1, \ldots$ and with *events* $E, E_1, \ldots$ that are sets of simultaneous signals. A signal that belongs to an event is said to be *present* in that event, otherwise it is said to be *absent*.

The execution of a program associates a sequence of output events with any sequence of input events. The program repeatedly receives an *input event* E_i from its environment and reacts by building an *output event* E'_i. That E_i and E'_i are synchronous is expressed by the fact that any external observer observes a *single* event $E_i \cup E'_i$. This is in particular true of any other program placed in parallel.

The production of an output event from an input event is called a *reaction*. The flow of time being entirely defined by the sequence of reactions, we also call a reaction an *instant*. This give sense to temporal expressions such as "instantaneously" or "immediately", which mean "at the same instant", or "from then on", which means "after the current instant included", or "in the strict future", which means "after the current instant excluded".

We assume that each input event contains a special signal `tick`, which is therefore present at all instants. This addition to the original language of [6] is now supported by the ESTEREL implementation. The `tick` signal is analogous to the constant 1 in circuits or the constant `true` in LUSTRE. When programming digital circuits, it will naturally denote clock ticks.

2.2 Modules

The basic PURE ESTEREL programming unit is the *module*. A module has an *interface*, which specifies its input signals `I`, `I1`, ... and its output signals `O`, `O1`, ..., and a *body*, which is a statement that specifies its behavior[2]. The body can use any number of local signals for internal broadcast communication. To achieve modular programming, a module can instantiate other modules as described later on. Here is a sample module definition:

```
module M:
input I1, I2;
output O1;
statement .
```

2.3 Kernel Statements

The primitive or *kernel* PURE ESTEREL statements are:

[2]There are also inputoutput signals, ignored here for simplicity.

```
nothing
halt
emit S
stat1; stat2
loop stat end
present S then stat1 else stat2 end
do stat watching S
stat1 || stat2
trap T in stat end
exit T
signal S in stat end
```

One can use brackets '`[`' and '`]`' to group statements; by default, '`;`' binds tighter than '`||`'. Both `then` and `else` parts are optional in a `present` statement. If omitted, they are supposed to be `nothing`.

The statements are imperative and manipulate control and signals. Most of them are classical in appearance. The `trap-exit` mechanism is a exception mechanism fully compatible with parallelism. Traps are lexically scoped.

The local signal declaration "`signal S in` *stat* `end`" declares a lexically scoped signal `S` that can be used for internal broadcast communication within *stat*.

2.4 The Intuitive Semantics

The intuitive semantics deals with control transmission between statements and with signal broadcasting. A statement can *start* at some instant and remain *active* until it releases the control at some further instant, either by terminating or by exiting a trap. After termination or exit, a statement becomes inactive. A statement that terminates or exits at the same instant it starts is said to be *instantaneous*. When an active statement does not terminate and exits no trap at an instant, it is said to *halt* at that instant.

The intuitive semantics is defined by structural induction on statements:

- `nothing` terminates instantaneously.
- `halt` never terminates nor exits. It always halts.
- An "`emit S`" statement broadcasts the signal `S` and terminates instantaneously.
- When started, a sequence "$stat_1$`;` $stat_2$" immediately starts $stat_1$ and behaves as it. If and when $stat_1$ terminates, $stat_2$ starts immediately and determines the behavior of the sequence from then on. If and when $stat_1$ exits a trap `T`, so does the whole sequence, $stat_2$ being never started in this case. Notice that $stat_2$ is also never started if $stat_1$ always halts. Notice also that "`emit S1; emit S2`" emits `S1` and `S2` simultaneously and terminates instantly.
- A loop acts as an infinite sequence. When started, "`loop` *stat* `end`" immediately starts its body *stat*. When the body terminates, it is immediately restarted. If the body exits a trap, so does the whole loop. The body of a loop is not allowed to terminate instantaneously when started.

- When a "`present S then` $stat_1$ `else` $stat_2$ `end`" statement starts, it starts immediately $stat_1$ if `S` is present in the current instant and $stat_2$ if `S` is absent. The `present` statement then behaves as the corresponding branch.

- The "`do` *stat* `watching S`" watchdog statement starts immediately its body and behaves as it until the *time guard* `S` occurs.

  - If *stat* terminates or exits a trap strictly before `S` occurs, then the `watching` statement instantaneously terminates or exits the same trap.
  - If, in the strict future of the starting instant, `S` occurs while *stat* is still active, then the `watching` statement terminates instantaneously and kills *stat*, which is not activated in the corresponding instant.

  Notice two boundary problems: the guard becomes active only at the *next* instant following the starting instant; the body is *not* activated when the time guard elapses. As we shall see below, all other possibilities can be derived by combining kernel statements, which would not be true with another choice for `watching`.

- When started, a parallel statement "$stat_1$ `||` $stat_2$" immediately starts $stat_1$ and $stat_2$ in parallel. A parallel terminates instantly if and when both $stat_1$ and $stat_2$ are terminated; they can terminate at different instants, the parallel waiting for the last one to terminate. If, at some instant, one statement exits a trap `T` or both statements exit the same trap `T`, then the parallel exits `T`. If both statements exit distinct traps `T1` and `T2` at the same instant, then the parallel only exits the *outermost* of these traps, the other one being discarded.

- The statement "`trap T in` *stat* `end`" defines a lexically scoped trap `T` within *stat*. When the `trap` statement starts, it starts immediately its body *stat* and behaves as it until termination or exit. If the body terminates, so does the `trap` statement. If the body exits `T`, then the `trap` statement terminates instantaneously. If the body exits an enclosing trap `U`, so does the `trap` statement (traps propagate).

- An "`exit T`" statement instantaneously exits the trap `T`.

- When started, the statement "`signal S in` *stat* `end`" starts immediately is body *stat* with a fresh signal `S`, overriding the one that may already exist. The statement behaves as its body from then on.

A global *coherence law* relates signal emission and testing:

> *A signal is present at an instant if and only if it is received as input by the environment or emitted by the program itself at that instant.*

Remarks:

Notice that an emission is transient, and that there is an asymmetry between present and absent signals. There is an `emit` statement to set a signal present, but no statement to set it absent: by the coherence law, this is just the default.

Notice also that a loop never terminates by itself; the only way to end it is to kill it by elapsing an enclosing time guard or by explicitly exiting an enclosing trap from within the loop or from a statement placed in parallel with the loop.

Finally, notice that exiting one branch of a parallel terminates instantaneously the corresponding `trap` and therefore kills the whole parallel. All parallel branches are activated at the exit instant. For example, in "`emit S || exit T`", the left branch emits `S` and terminates, the right branch exits `T`, so that the parallel emits `S` and synchronizes both branches by deciding to exit `T`. Therefore, being killed by an exit is less severe than being killed by an enclosing `watching` time guard, which does *not* activate its body when elapsed.

2.5 Examples

The only statement that provokes halting is `halt`. To take a finite but non-zero amount of time, a statement must involve `halt` statements guarded by `watching` statements. The simplest example is "`do halt watching S`" which waits for `S` and terminates: by itself, the body `halt` would halt forever, but the enclosing "`watching S`" guard kills it when `S` occurs, and it makes the whole statement terminate. Hence the statement is guaranteed to "last exactly one `S`" from the time it is started (remembering that an `S` present when the statement starts is not taken into account). Anticipating on the definition of derived statements, we write it as "`await S`".

In the above example, `S` can be any signal, a second as well as a centimeter, a clock tick, or generally any kind of interrupt. Therefore, each signal is seen as defining its own time unit. Nesting temporal statements bearing on different time units is the main characteristic of the ESTEREL style [6, 4]. Here is a program that emits repeatedly `O` every `I` until reception of a signal `STOP`

```
do
  loop
    await I; emit O
  end
watching STOP
```

Here `O` is not emitted when `STOP` occurs, even if `I` is present, since the inner loop is preempted by the external `watching` statement at that instant.

In most event manipulation languages, the basic primitive is `await`, that waits for an event to *start* a computation in sequence. On the contrary, in ESTEREL, the main primitive is `watching`, that waits for an event to *stop* or *preempt* a computation. It is a much more powerful primitive than `await`. In particular, it is easy to derive `await` from `watching`, while the converse is definitely not true.

Remember the boundary problem we mentioned when describing the `watching` statement. To also emit `O` if `I` is present when `STOP` occurs, one uses a `trap`:

```
trap T in
    loop await I; emit O end
  ||
    await STOP; exit T
end
```

This works since when one branch of a parallel exits a trap that encloses the parallel, the other branch is activated in the corresponding instant before being killed. It can perform its "last wills".

The other boundary problem concerns the starting instant. If one wants the guard to be active initially, one writes

present S else do *stat* watching S end

readily abbreviated into the derived statement

do *stat* watching immediate S

The following toy example illustrates the preemption mechanism involved in concurrent exits:

```
trap T1 in
  trap T2 in
    emit S1; exit T1
  ||
    exit T2; emit S2
  end;
  emit S3
end
```

The first parallel branch emits `S1` and exits `T1`. The second parallel branch exits `T2` but does not emit `S2` since an `exit` statement does not terminate. The body of the parallel exits simultaneously `T1` and `T2`; since only the outermost trap matters, `T2` is discarded and `T1` propagates. Hence `S3` is not emitted, and the outermost trap terminates with only `S1` emitted

2.6 Full Esterel

The full language has many useful derived statements. We briefly describe the most important ones. See [6] for the complete list and for the exact expansions into kernel statements.

Temporal Statements

A temporal statement is characterized by the fact that its expansion involves `present`, `watching`, or `halt` kernel statements. We have already seen the simple `await` statement and the `immediate` guard variant. Here are some other useful constructs:

- Boolean expressions on signals can appear in tests or guards, as in "`present S1 and S2`" or "`do` *stat* `watching not S`".

- One can count occurrences of a signal (or boolean expression) within a time guard, as in "`await 3 S`". Occurrence counts are not discussed in this paper but are easy to handle.

- One can add a timeout clause to be executed when a `watching` statement terminates by elapsing its time guard and not when the body terminates by itself:

  do $stat_1$ watching S timeout $stat_2$ end is just an abbreviation for:

```
trap T in
  do stat₁; exit T watching S;
  stat₂
end
```

- The statement "`do` *stat* `upto S`" is just "`do` *stat*`; halt watching S`". Even if the body terminates, the `upto` statement waits for its guard to elapse.

- Deterministic event selection has the form:

```
await
  case S1 do stat₁
  case S2 do stat₂
end
```

  The statement waits simultaneously for `S1` and `S2`. If one of them occurs alone, the control is instantaneously transferred to the corresponding statement. If both signals occur at the same time, the control is transferred to `S1` only. This guarantees determinism.

- There are two temporal loops:

```
loop stat each S
every S do stat end
```

  The first loop starts *stat* at once, and kills and restarts it afresh whenever `S` occurs. The second loop is similar but initially waits for `S` to start *stat*.

- The "`sustain S`" statement emits `S` continuously. It abbreviates

```
loop emit S each tick
```

General Traps

There is a general exception handling mechanism that extends basic traps:

```
trap T1, T2 in
    stat
  handle T1 do stat₁
  handle T2 do stat₂
end
```

When a trap is exited, the corresponding handler is started instantaneously. Here the traps `T1` and `T2` are concurrent. If they are exited simultaneously, both handlers are run in parallel.

Module Instantiation

Modular programming is achieved by the `run` statement, which instantiates a module in place, possibly invoking signal renamings:

```
run M [signal S/I]
```

A `run` statement terminates if and when the copied module body does.

3 The Behavioral Semantics

Several mathematical semantics have been developed for ESTEREL, including a denotational semantics that precisely formalizes the intuitive temporal concepts presented in section 2.3, see [18]. Here we prefer to use the *behavioral semantics* [6] that defines execution reaction by reaction, using Plotkin's Structural Operational Semantics technique (SOS for short). It is shown equivalent to the denotational one in [18].

3.1 Form of the Rules

The behavioral semantics defines transitions of the form $M \xrightarrow[I]{O} M'$ where M is a module, I is an input event, O is the corresponding output event, and M' is a new module that will correctly respond to the next input events. In other words, M' is the new state of M after the reaction to I. The reaction $O_1, O_2, \ldots, O_n, \ldots$ to an input sequence $I_1, I_2, \ldots, I_n, \ldots$ is then defined inductively by chaining elementary reactions:

$$M \xrightarrow[I_1]{O_1} M_1 \xrightarrow[I_2]{O_2} M_2 \ldots M_{n-1} \xrightarrow[I_n]{O_n} M_n \xrightarrow[I_{n+1}]{O_{n+1}} \ldots$$

A behavioral transition $M \xrightarrow[I]{O} M'$ is computed using an auxiliary relation $stat \xrightarrow[E]{E', k} stat'$ defined by structural induction on statements. Here E is the *current event* in which *stat* evolves, E' is the event made of the signals emitted by *stat*, and k is an integer *termination level* that codes the way in which *stat* terminates or exits and is precisely defined below.

The current event E is made of all the signals that are present at the given instant; because of the coherence law, E must contain the set E' of emitted signals, which in turns depends on E. Hence E and E' will be computed as *fixpoints*, the fixpoint equation being located in the local signal rule below.

Let *stat* be the body of M and $stat'$ be the body of M'. The relation between both transition systems is as follows:

$$M \xrightarrow[I]{O} M' \text{ iff } stat \xrightarrow[I \cup O \cup \{\mathtt{tick}\}]{O, k} stat' \text{ for some } k$$

(under the minor restriction that no input signal is internally emitted by *stat*, see [6]).

Termination Levels

The termination level k is 0 if *stat* terminates in the current instant, 1 if *stat* halts in the current instant, and $k+2$ if *stat* exits a trap `T` that is k trap levels above it, i.e. is if the exit must be propagated through $k-1$ traps before reaching its trap. To handle the exit level, it is useful to first decorate the `exit` statements with the corresponding level, as in the following example:

```
trap T in
  exit T^2
||
  trap U in
    exit T^3
  ||
    exit U^2
  end
end
```

Here the first `T` exit and the `U` exit are labeled 2 since there is no intermediate `trap` statement to traverse, while the second `T` exit is labeled 3 since one must traverse the `trap U` statement to reach the `trap T` statement. This way of handling termination is simpler than the one used in [6], but equivalent to it as shown in [18] (see also [16]).

3.2 Inductive Rules

The `nothing` statement terminates instantaneously.

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

The `halt` statements halts and rewrites into itself.

$$\texttt{halt} \xrightarrow[E]{\emptyset, 1} \texttt{halt}$$

An `emit` statement emits its signal and terminates.

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

If the first statement of a sequence terminates, the second statement is started at once; the emitted signals are merged to form the resulting emitted event, according to perfect synchrony.

$$\frac{stat_1 \xrightarrow[E]{E_1', 0} stat_1' \qquad stat_2 \xrightarrow[E]{E_2', k_2} stat_2'}{stat_1 \,;\, stat_2 \xrightarrow[E]{E_1' \cup E_2', k_2} stat_2'}$$

If the first statement of a sequence does not terminate, that is if it halts or exits a trap, the sequence behaves just as the first statement and the second statement is kept unchanged for further reactions.

$$\frac{stat_1 \xrightarrow[E]{E'_1,\, k_1} stat'_1 \qquad k > 0}{stat_1\,\texttt{;}\ stat_2 \xrightarrow[E]{E'_1,\, k_1} stat'_1\,\texttt{;}\ stat_2}$$

A loop instantaneously unfolds itself once. Its body is not allowed to terminate instantaneously.

$$\frac{stat \xrightarrow[E]{E',\, k} stat' \qquad k > 0}{\texttt{loop}\ stat\ \texttt{end} \xrightarrow[E]{E',\, k} stat'\texttt{;}\ \texttt{loop}\ stat\ \texttt{end}}$$

A `present` statement instantaneously selects its `then` branch if the signal tested for is present in the current instant. Otherwise, it instantaneously selects its `else` branch.

$$\frac{\texttt{S} \in E \qquad stat_1 \xrightarrow[E]{E'_1,\, k_1} stat'_1}{\texttt{present S then}\ stat_1\ \texttt{else}\ stat_2\ \texttt{end} \xrightarrow[E]{E'_1,\, k_1} stat'_1}$$

$$\frac{\texttt{S} \notin E \qquad stat_2 \xrightarrow[E]{E'_2,\, k_2} stat'_2}{\texttt{present S then}\ stat_1\ \texttt{else}\ stat_2\ \texttt{end} \xrightarrow[E]{E'_2,\, k_2} stat'_2}$$

A `watching` statement transfers the control to its body and rewrites itself into a `present` statement in order to set the time guard at next instant if the body has halted.

$$\frac{stat \xrightarrow[E]{E',\, k} stat'}{\texttt{do}\ stat\ \texttt{watching S} \xrightarrow[E]{E',\, k} \texttt{present S else do}\ stat'\ \texttt{watching S}}$$

A parallel statements starts its branches instantaneously, merges the emitted signals, and returns the *max* of the termination codes. We leave it to the reader to see that this *max* operation exactly performs the required synchronization in all termination cases.

$$\frac{stat_1 \xrightarrow[E]{E'_1,\, k_1} stat'_1 \qquad stat_2 \xrightarrow[E]{E'_2,\, k_2} stat'_2}{stat_1\ \texttt{||}\ stat_2 \xrightarrow[E]{E'_1 \cup E'_2,\, max(k_1, k_2)} stat'_1\ \texttt{||}\ stat'_2}$$

A `trap` terminates if its body terminates or exits the trap, that is returns termination code 2. If the body halts, so does the trap. If the body exits an enclosing `trap`, then the exit is propagated by subtracting 1 to the exit level.

$$\frac{\mathit{stat} \xrightarrow[E]{E',k} \mathit{stat}' \qquad k=0 \text{ or } k=2}{\texttt{trap T in } \mathit{stat} \texttt{ end} \xrightarrow[E]{E',0} \texttt{nothing}}$$

$$\frac{\mathit{stat} \xrightarrow[E]{E',k} \mathit{stat}' \qquad (k=1 \text{ and } k'=1) \text{ or } (k>2 \text{ and } k'=k-1)}{\texttt{trap T in } \mathit{stat} \texttt{ end} \xrightarrow[E]{E',k'} \texttt{trap T in } \mathit{stat}' \texttt{ end}}$$

An `exit` statement returns its exit level.

$$\texttt{exit T}^k \xrightarrow[E]{\emptyset,k} \texttt{halt}$$

Finally, the local signal declaration rules wind up the events E and E' according to the coherence law given in section 2.3. Within the body, they impose that a local signal is present in E if and only if it is emitted in E'. A local signal is obviously not propagated outside its declaration.

$$\frac{\mathit{stat} \xrightarrow[E\cup\{\texttt{S}\}]{E'\cup\{\texttt{S}\},k} \mathit{stat}' \qquad \texttt{S} \notin E'}{\texttt{signal S in } \mathit{stat} \texttt{ end} \xrightarrow[E]{E',k} \texttt{signal S in } \mathit{stat}' \texttt{ end}}$$

$$\frac{\mathit{stat} \xrightarrow[E-\{\texttt{S}\}]{E',k} \mathit{stat}' \qquad \texttt{S} \notin E'}{\texttt{signal S in } \mathit{stat} \texttt{ end} \xrightarrow[E]{E',k} \texttt{signal S in } \mathit{stat}' \texttt{ end}}$$

Remarks

The resulting statement stat' is unused and therefore immaterial for any rule returning $k > 1$; it is discarded by the exited `trap`. If a rule returns $k = 0$, then its resulting term is equivalent to `nothing`.

Because of the intrinsic fixpoint character of the local signal rule, our inference system does not yield a straightforward algorithm to compute a transition. Given any input I one must guess the right current event E and use the rules to check that there is a correct transition. Other semantics yield finer analysis and efficient algorithms to compute the reaction; see in particular the *computational semantics* in [6].

3.3 Correct Programs

Not all ESTEREL programs make sense. We say that a module M is *locally correct* if there is only one provable transition $M \xrightarrow[I]{O} M'$ for any input event I. We say that M is *correct* if it is locally correct and if all modules obtained by all possible sequences of provable transitions are locally correct.

Correctness of ESTEREL programs is a difficult issue. It is similar to correctness of digital circuits (absence of races), although much more complex because of the power of the ESTEREL instantaneous `loop` construct. The ESTEREL compiler checks for reasonably general sufficient correctness conditions, see [6]. Here, we just show two examples of (locally) incorrect programs.

The following program has no fixpoint, since `S` should not be emitted if present and emitted if not present. It is analogous to $X = \neg X$ in circuits.

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

The next program has two fixpoints, one of `S1` or `S2` being present in each. It is similar to $X_1 = \neg X_2,\ X_2 = \neg X_1$ in circuits.

```
signal S1, S2 in
  present S1 else emit S2 end
||
  present S2 else emit S1 end
end
```

4 The Haltset Coding of States

We now present an essential concept of the theory of ESTEREL, the unambiguous coding of any state by a set of control points in the original program. Technically, control points are represented by `halt` positions in the kernel expansion of the module body (notice that the expansion of any derived temporal statement generates at most one `halt`). Since ESTEREL is concurrent, a state is given by a *set* of control positions, which we call a haltset. The haltset coding is important in two respects. First, its existence shows the rationality of ESTEREL: only finitely many statements be generated by the rewritings of a given statement. Second, it is the direct basis of the hardware implementation, and it is also heavily used in the software implementation.

The reader might skip this section at first reading and proceed directly with the informal presentation of the hardware translation in section 5. However, an understanding of the material presented here will be necessary to see why the translation is done that way and why it indeed works.

In the sequel, we consider a fixed correct module `M` of expanded body *stat*. For technical reasons, we assume that the body of `M` never terminates, adding a trailing `halt` if necessary. This condition does not change the observable behaviors; of course, adding a trailing `halt` is done after expansion and not in modules copied by M.

Call a *derivative* of *stat* any statement *stat'* that can be reached from *stat* by some sequence of reactions $\xrightarrow[I]{O}$ provable in the behavioral semantics. So far, the derivatives are defined by a rewriting process and bear no obvious structural relation with the source term *stat*. We show that any derivative can be unambiguously coded by a *haltset* H of *stat*, that is by a set of occurrences of `halt` statements in the kernel statement *stat*.

Consider for example the derivatives of "`await S1; await S2; halt`". There are three `halt` statements, the two first ones being respectively generated by the first and the second `await`. Number them $0, 1, 2$. The whole statement itself will be coded by the empty haltset $\emptyset$. The derivative that waits for `S1` is

```
present S1 else
  await S1
end;
await S2;
halt
```

Its haltset will be $\{0\}$, the index of the `halt` generated by the active "`await S1`" statement. The derivative that waits for `S2` is

```
present S2 else
  await S2
end;
halt
```

Its haltset will be $\{1\}$ since the second `await` is active. The final derivative is `halt`, coded by $\{2\}$. Non-singleton haltsets will be constructed by the parallel operator, which will return the union of the haltsets of its branches.

4.1 Haltsets

We number all occurrences of `halt` in *stat* by distinct integers from 0 to n, $n > 0$. Then a haltset H is a subset of $[0..n]$. that satisfies the following *separation* condition: If $stat_1$ and $stat_2$ are the two statements of a sequence or the two branches of a `present` test, then H cannot contain an occurrence of `halt` in $stat_1$ together with an occurrence of `halt` in $stat_2$.

We decorate the behavioral semantics rules by returning a haltset H when executing a numbered term. This haltset will record the places where the term has halted. The rules take the new form $stat \xrightarrow[E]{E', k, H} stat'$. We always return $H = \emptyset$ when $k \neq 1$ and $H \neq \emptyset$ when $k = 1$. Adding haltsets is easy for all rules except the parallel one. Executed `halt` statements are put into the haltset by the rule of `halt` and propagated by the other rules. Since the transformation is fairly obvious, we just list a few rules and leave the other ones to the reader.

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

$$\texttt{halt}^i \xrightarrow[E]{\emptyset, 1, \{i\}} \texttt{halt}^i$$

$$\frac{stat_1 \xrightarrow[E]{E'_1,\,0,\,\emptyset} stat'_1 \qquad stat_2 \xrightarrow[E]{E'_2,\,k_2,\,H_2} stat'_2}{stat_1\ \texttt{;}\ stat_2 \xrightarrow[E]{E'_1 \cup E'_2,\,k_2,\,H_2} stat'_2}$$

$$\frac{stat_1 \xrightarrow[E]{E'_1,\,k_1,\,H_1} stat'_1 \qquad k_1 > 0}{stat_1\ \texttt{;}\ stat_2 \xrightarrow[E]{E'_1,\,k_1,\,H_1} stat'_1\ \texttt{;}\ stat_2}$$

$$\frac{stat \xrightarrow[E]{E',\,k,\,\emptyset} stat' \qquad k = 0 \text{ or } k = 2}{\texttt{trap T in } stat \texttt{ end} \xrightarrow[E]{E',\,0,\,\emptyset} \texttt{nothing}}$$

$$\frac{stat \xrightarrow[E]{E',\,k,\,H} stat' \quad (k = 1 \text{ and } k' = 1) \text{ or } (k > 2 \text{ and } k' = k - 1)}{\texttt{trap T in } stat \texttt{ end} \xrightarrow[E]{E',\,k',\,H} \texttt{trap T in } stat' \texttt{ end}}$$

For a parallel, we return the union of the haltsets returned by the branches unless one of the branches exits a trap, in which case we return an empty haltset. We make an additional technical modification explained later on: when one branch terminates, we rewrite it into `nothing`.

$$\frac{\begin{array}{c} stat_1 \xrightarrow[E]{E'_1,\,k_1,\,H_1} stat'_1 \\ stat_2 \xrightarrow[E]{E'_2,\,k_2,\,H_2} stat'_2 \\ H = \begin{cases} H_1 \cup H_2 & \text{if } max(k_1, k_2) \leq 1 \\ \emptyset & \text{if } max(k_1, k_2) > 1 \end{cases} \\ stat''_i = \begin{cases} stat'_i & \text{if } k_i \neq 0 \\ \texttt{nothing} & \text{if } k_i = 0 \end{cases} \end{array}}{stat_1 \texttt{ || } stat_2 \xrightarrow[E]{E'_1 \cup E'_2,\,max(k_1,k_2),\,H} stat''_1 \texttt{ || } stat''_2}$$

Since a module body is supposed to always halt, its global termination code must be 1. Hence the rules always returns a well-defined haltset H for any immediate derivative. This haltset is easily seen to satisfy the separation condition.

4.2 Recovering derivative from haltsets

We now recover the derivative $stat'$ from $stat$ and H. We proceed in two steps. First we define a labeled term $stat^H$ obtained by labeling the subterms of $stat$ by either $H+$ or $H-$; a subterm is labeled $H+$ if and only if it contains at least one occurrence of `halt` whose number is in H. If we care about the label of $stat^H$ itself, then we write it explicitly, as in $stat^{H+}$. The labels are of course redundant with H, but they make the definitions and proofs much simpler to write.

Then we define a term $\mathcal{R}(stat^H)$ by structural induction on $stat^H$. Subterms labeled by $-$ and `halt` statements are left unchanged.

$$\begin{array}{rcl} \mathcal{R}(stat^{H-}) & = & stat \\ \mathcal{R}(\mathtt{halt}^{i^H}) & = & \mathtt{halt}^i \end{array}$$

`trap` and local signal declaration constructs are handled by trivial structural induction.

$$\begin{array}{rcl} \mathcal{R}(\mathtt{trap\ T\ in}\ stat\ \mathtt{end}^H) & = & \mathtt{trap\ T\ in}\ \mathcal{R}(stat^H)\ \mathtt{end} \\ \mathcal{R}(\mathtt{signal\ S\ in}\ stat\ \mathtt{end}^H) & = & \mathtt{signal\ S\ in}\ \mathcal{R}(stat^H)\ \mathtt{end} \end{array}$$

The only non-trivial cases are:

$$\begin{array}{rcl} \mathcal{R}(stat_1^{H+}\mathtt{;}\ stat_2^{H-}) & = & \mathcal{R}(stat_1^{H+})\mathtt{;}\ stat_2 \\ \mathcal{R}(stat_1^{H-}\mathtt{;}\ stat_2^{H+}) & = & \mathcal{R}(stat_2^{H+}) \\ \mathcal{R}(\mathtt{loop}\ stat_1^{H+}\ \mathtt{end}) & = & \left|\begin{array}{l} \mathcal{R}(stat_1^{H+})\mathtt{;} \\ \mathtt{loop}\ stat_1\ \mathtt{end} \end{array}\right. \\ \mathcal{R}(\mathtt{present\ S\ then}\ stat_1^{H+}\ \mathtt{else}\ stat_2^{H-}\ \mathtt{end}) & = & \mathcal{R}(stat_1^{H+}) \\ \mathcal{R}(\mathtt{present\ S\ then}\ stat_1^{H-}\ \mathtt{else}\ stat_2^{H+}\ \mathtt{end}) & = & \mathcal{R}(stat_2^{H+}) \\ \mathcal{R}(\mathtt{do}\ stat_1^{H+}\ \mathtt{watching\ S}) & = & \left|\begin{array}{l} \mathtt{present\ S\ else} \\ \quad \mathtt{do}\ \mathcal{R}(stat_1^{H+})\ \mathtt{watching\ S} \\ \mathtt{end} \end{array}\right. \\ \mathcal{R}(stat_1^{H+}\ \mathtt{||}\ stat_2^{H+}) & = & \mathcal{R}(stat_1^{H+})\ \mathtt{||}\ \mathcal{R}(stat_2^{H+}) \\ \mathcal{R}(stat_1^{H+}\ \mathtt{||}\ stat_2^{H-}) & = & \mathcal{R}(stat_1^{H+})\ \mathtt{||}\ \mathtt{nothing} \\ \mathcal{R}(stat_1^{H-}\ \mathtt{||}\ stat_2^{H+}) & = & \mathtt{nothing}\ \mathtt{||}\ \mathcal{R}(stat_2^{H+}) \end{array}$$

Notice that these definitions make sense only when the separation condition is satisfied. Notice also why we return `nothing` in the semantics rules when a branch terminates: this simplifies the definition of $\mathcal{R}$.

Since they exactly reproduce the (new) behavioral rules right-hand side terms, one easily shows $\mathcal{R}(stat^H) = stat'$ as expected.

We now give the main result: the coding extends from immediate derivatives to general ones. This is not completely obvious since the $\mathcal{R}$ operator can duplicate halts in the `loop` case. The result is as follows:

Theorem: Let *stat* be the body of a correct program. Let H be a haltset in *stat*. Then for any behavioral rewriting of the form

$$\mathcal{R}(stat^H) \xrightarrow[E]{E',1,H'} stat'$$

the haltset H' contains only halts occuring in $stat'$ and one has $stat' = \mathcal{R}(stat^{H'})$.

proof: The proof is by structural induction on *stat* and by case inspection on the rule applied to the whole term $\mathcal{R}(stat^H)$ to yield $stat'$. All cases being similar, we treat the sequence and the loop as examples. We consider a given current event E.

Let first $stat = stat_1;\ stat_2$. There are three cases according to the labeling generated by H.

- If $stat_2$ is labeled by $H+$, then $\mathcal{R}(stat^H) = \mathcal{R}(stat_2^H)$. By correctness and by the hypothesis that *stat* halts, $\mathcal{R}(stat_2^H)$ has a unique rewriting $\mathcal{R}(stat_2^H) \xrightarrow[E]{E',1,H'} stat'$, where H' is a nonempty haltset that only contains halts in $stat_2$. By induction, one has $stat' = \mathcal{R}(stat_2^{H'+})$. Since H' is all in $stat_2$ and nonempty, one has $\mathcal{R}(stat_2^{H'+}) = \mathcal{R}(stat^{H'})$ by definition of $\mathcal{R}()$ and the result follows.

- The two other cases can be grouped into one. They correspond to a term $stat = \mathcal{R}(stat_1^H);\ stat_2$, taking H as given if $stat = \mathcal{R}(stat_1^{H+});\ stat_2$ and $H = \emptyset$ if *stat* itself has label $H-$. By correctness, *stat* has a unique behavior, computed by either the first or the second sequence rule. If the first sequence rule is used, then $stat'$ is generated entirely by $stat_2$ and the results follows as in the first case. If the second sequence rule is used, the termination code of $\mathcal{R}(stat_1^H)$ is 1 since *stat* halts. By induction and by the form of the rule, one has $stat' = \mathcal{R}(stat_1^{H'+});\ stat_2 = \mathcal{R}(stat^{H'})$ for some nonempty H' having all its halts in $stat_1$. The result follows.

Assume now $stat =$ `loop` $stat_1$ `end`. There are two subcases. If *stat* is labeled by $H-$, then $\mathcal{R}(stat^{H-}) =$ `loop` $stat_1$ `end`. The only applicable rule is the loop rule. It asks for computing $stat_1$, which must halt since *stat* does. By induction and by the loop rule, one has $stat \xrightarrow[E]{E',1,H'} \mathcal{R}(stat_1^{H'+});\ stat$ for some H'. The last term is just $\mathcal{R}(stat^{H'})$ as expected. If *stat* is labeled by $H+$, then $\mathcal{R}(stat^{H+}) = \mathcal{R}(stat_1^{H+});\ stat$. If the first term does not terminate, we proceed as in the first loop case. Otherwise, the loop must be unfolded once and we are back again in the first loop case.

Corollary: Let *stat* be a module body. Then any derivative $stat'$ of *stat* is equal to $\mathcal{R}(stat^H)$ for some haltset H, and there are only finitely many derivatives.

proof: by induction on the length of a rewriting sequence $stat \xrightarrow{*} stat'$, since *stat* itself is equal to $\mathcal{R}(stat^{\emptyset})$ and since *stat* always returns $k = 1$. The finiteness property is obvious since there are only finitely many possible haltsets.

5 Principle of the Hardware Implementation

In this section, we show by examples how to translate a Pure Esterel program into a digital circuit that computes the reaction of the program to any input in one clock cycle. The translation is structural: the circuit logical geometry is the same as that of the original program. The translation is directly based on the haltset coding theory of section 4, but we present it in such a way that it can be understood independently of this coding.

We start with a first example involving only `halt` and `watching` statements. Then we show how to handle concurrency and exceptions. Finally, we indicate how to efficiently translate the full language. The formal translation is given in section 6.

5.1 A First Example

Consider the following program:

```
module M:
input I, R;
output O;
loop
  loop
    await I; await I; emit O
  end
each R.
```

After an initialization instant in which `I` is ignored, the behavior is to emit `O` every two `I`, restarting this behavior afresh each `R`. Expanded into kernel statements, the body becomes:

```
loop
  do
    loop
      do
        halt
      watching I;
      do
        halt
      watching I;
      emit O;
    end
  watching R
end
```

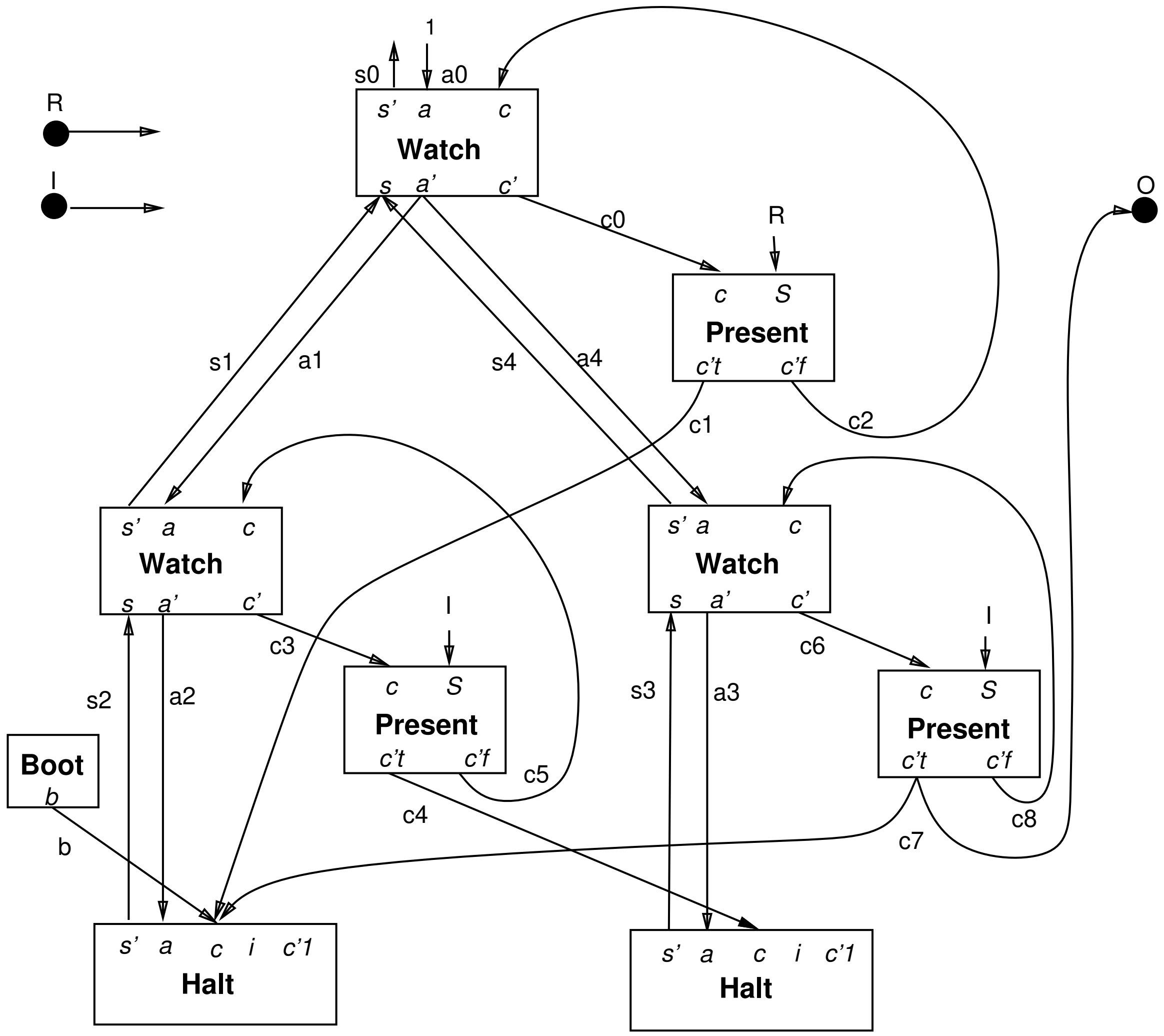

Figure 1: first example

The corresponding circuit is drawn in figure 1. It has two input pins for `I` and `R` and one output pin for `O`. There are four kinds of cells, called `Boot`, `Watch`, `Present`, and `Halt`. Cell output pins are primed.

The `Boot` and `Halt` cells each contain one register, assuming to initially contain value 0 and to be clocked by the global circuit's clock. The other cells are purely combinatorial. The `Present` cell are used for `present` and `watching` source statements, each source "`watching S`" statement being conceptually rewritten into "`watch present S`"; This slight syntactic modification simplifies the cells and makes it easy to implement boolean expressions.

The circuit contains three sorts of wires: the *selection* wires `s0`–`s5`, the *activation* wires `a0`–`a5`, and the *control* wires `c0`–`c8`. The unconnected i and $c'1$ pins of `Halt` cells corresponds to other wires unused here and described later on. Whenever two wires go to the same place, they are implicitly assumed to be combined by an `or` gate.

The selection and activation wires go in reverse directions and form a tree that is called the *skeleton* of the circuit. This tree is determined by the nesting of `halt`, `watching`, and `||`

statements in the source program, following the abstract syntax revealed by the source code indentation. The leftmost `Halt` and `Watch` cells correspond to the first `await`, the rightmost ones to the second `await`.

The selection wires are used to determine which part of the circuit can be active in a given state: in our example, both `await` statements are in mutual exclusion, and only one of them can be active at a time. When the first `await` is active, the wires `s2`, `s1`, and `s0` are set to 1. When the second `await` is active, the wires `s4`, `s3`, and `s0` are set to 1. The sources of the selection wires are the `Halt` cell registers. The upper selection wire `s0` is unconnected here, but we left it there to emphasize the structural character of the translation.

The activation and control wires bear the flow of control. The activation wires handle preemption between `watching` statements. In our example, the outermost `watching` preempts the innermost one: by the semantics of ESTEREL, if `R` is present, the outermost `watching` terminates without letting its body execute. The upper activation wire `a0` is always set.

The cells are defined as follows:

$$\texttt{Boot}\quad \begin{cases} n := 1 \\ b = \neg n \end{cases}$$

$$\texttt{Watch}\quad \begin{cases} s' = s \\ c' = s * a \\ a' = c \end{cases}$$

$$\texttt{Present}\quad \begin{cases} c't = c * S \\ c'f = c * \neg S \end{cases}$$

$$\texttt{Halt}\quad \left\{ s' := c + (a * s) \right.$$

The notation is that of PALASM: '+' is or, '$*$' is and, '$\neg$' is not, an equality is valid at all times, and a register is denoted by ' := '. Registers are supposed to contain initially 0. In the sequel, we say that a wire is *high* or *set* if it has value 1 and *low* or *reset* if it has value 0. We say that a register is *set* if it gets value 1 and *reset* if it gets value 0. Signals are assumed to be present when their wire is set and absent when their wire is reset.

The output signal b of the `Boot` cell is high at first clock tick and remains then low. For a `Halt` cell, the value of the output signal s' is initially low and then that of $c + (a * s)$ delayed one clock cycle. Hence a register is set either if an incoming control wire is set or if the activation wire is set and the register was already set[3]. The definition of `Halt` is only temporary: further pins will be added in section 5.2.

A Sample Execution

At boot time, the `Halt` cell registers contain 0 and the selection wires are all low; the boot control wire `b` is high. Because of the cell equations, all other wires are low. Hence the only effect is to set the leftmost `Halt` register.

On next clock tick, assume that `I` is present and `R` is absent. Then `s2`, `s1`, and `s0` are set by the `Halt` register. Since `a0` is always set, the control flows down by setting `c0` that

[3]The multiplication by s is there to prevent setting the second `Halt` register in a term such as "`do halt; halt watching S`" when a is set.

triggers the test for `R` in the upper `Present` cell. Since `R` is low, the control flows through the $c'f$ pin and sets `c2`, which is connected to the c input pin of the `Watch` cell. This pin is directly connected to the a' output pin, and the control flows though `a1` and `a4` (which are connected with each other and form in fact a single equipotential). Since both `s2` and `a1` are high, the leftmost `Watch` cell sets `c3` and the leftmost `Present` cell sets `c4` since `I` is present. This sets the rightmost `Halt` register. Since `s4` is low, the rightmost `Watch` cell is inactive. Having no incoming control set, the leftmost `Halt` register is reset. This terminates the first "`await I`" statement.

On next clock tick, if `I` is present, the execution is symmetrical: the rightmost `Halt` is reset and the leftmost one is set. The wires set are `s3`, `s4`, `a0`, `c0`, `c2`, `a1` = `a4`, `c6`, and `c7`. Since `c7` is also connected to the output `O`, this output is set. If instead `R` is present, the wires set are `s3`, `s4`, `a0`, `c0`, `c1`, and one is back to the state just after boot. If neither `I` nor `R` are present, then the wires set are `s3`, `s4`, `a0`, `c0`, `c2`, `a1` = `a4`, `c6`, `c8`, and `a3`, and the state is simply restored as expected.

Relation with the Haltset Coding

Intuitively, the relation between our circuit and the haltset coding of derivatives is as follows:

- A state of the circuit is a set of `Halt` cells set to 1. It is therefore exactly a haltset.
- The selection wires just compute the $+$ and $-$ labels of statements, $+$ being represented by a 1 in the selection wire.
- Sending the control to the translation of a subterm $stat_1$ by setting an incoming control wire amounts to execute $stat_1$. For example, setting `b` executes the whole statement, setting `b` or `c1` execute the first `await I`, and setting `c4` executes the second `await I`.
- Sending the control to the translation of a subterm $stat_1$ by setting its incoming activation wire amounts to execute $\mathcal{R}(stat_1^H)$ if $stat_1$ is labeled by $+$ in H, i.e. if the corresponding selection wire is set.

Hence, in a haltset H and an input I, the circuit just mimics the behavioral proof of $\mathcal{R}(stat^H)$ in I. This points will be made very precise in section 7.

Notice that the `Boot` cell is not really necessary since the initial state can also be recognized as the only state where all `Halt` cells have value 0, that is where the wire `s0` is low. We could as well connect the `b` wire to the negation of `s0`. However, it is convenient in practice to add the auxiliary `Boot` cell to reduce the length of wires and the number of logical levels.

5.2 Translating Parallel and Exceptions

The most complex operator is of course the parallel one, since it must synchronize the termination of its branches and propagate exceptions. Consider the following program fragment:

```
trap T in
  await S1
||
  present S2 then exit T end
end
```

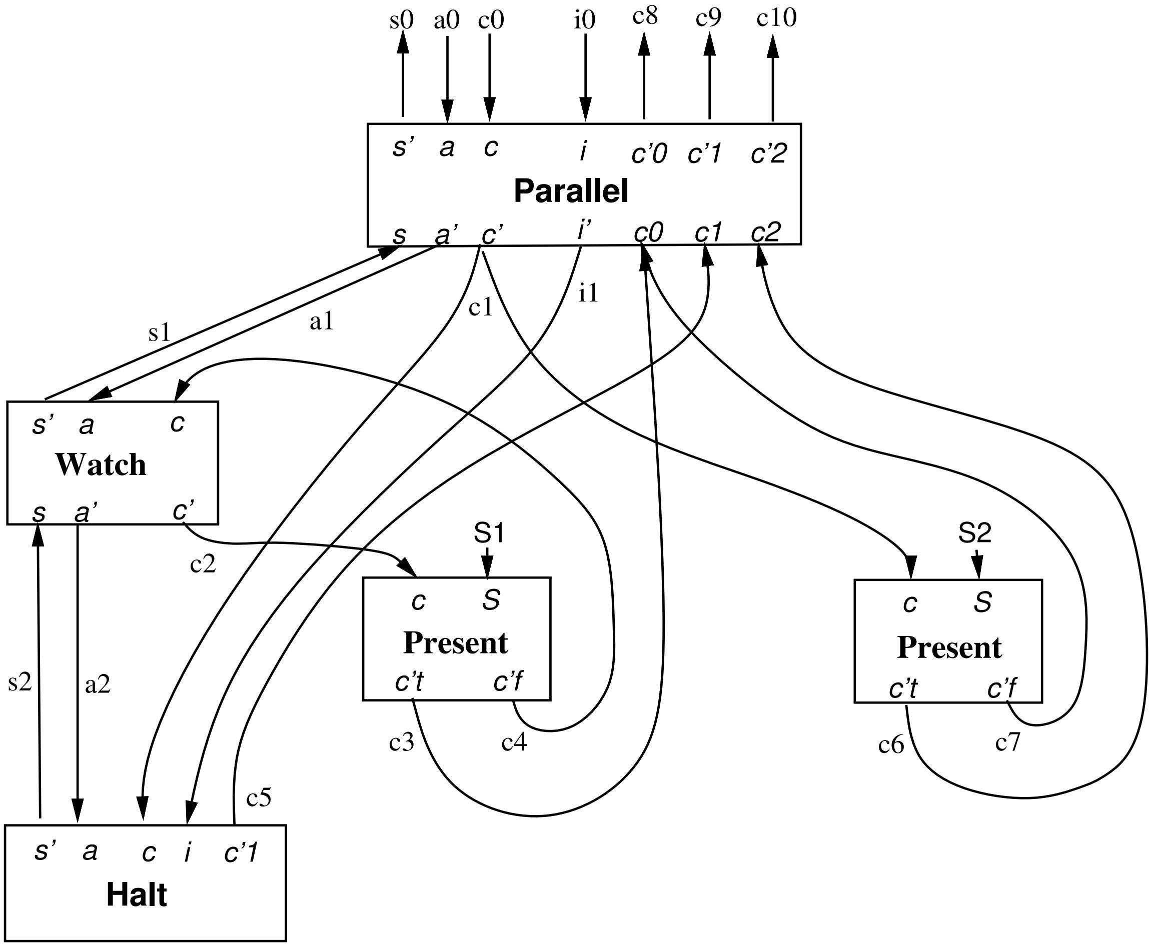

Figure 2: second example

The corresponding circuit fragment is shown in figure 2. The leftmost `Watch`-`Present`-`Halt` cell group is generated by "`await S1`". The rightmost `Present` cell is generated by "`present S2`", (where "`else nothing`" was omitted as usual). The branches are simply put in parallel and synchronized by the `Parallel` cell. The circuit fragment starts when it receives control by setting the `c0` wire.

The `Parallel` cell has two parts: the fork part, which involves the six leftmost pins, and the synchronization part, which involves the eight rightmost ones.

The fork part is simple: selection wires are gathered by an `or` gate and activation and control are dispatched to branches.

The synchronization part is more subtle. The pins $c0$, $c1$, and $c2$ record the different termination modes according to their codes defined in section: $c0$ means termination, $c1$ means halt, and $c2$ means exiting `T`. With each termination pin ci is associated a continuation pin $c'i$. (In fact, $c'1$ is not really a continuation in a usual sense: it is recursively linked to the $c1$ entry of the enclosing `Parallel` cell when such a cell exists.)

As explained in section 3, the synchronization realized by the parallel amounts to compute the *max* of the termination codes of its branches and to only activate the corresponding continuation. It therefore uses a priority queue.

In our example, the left branch can halt, as signaled by setting wire `c5`, or terminate, as signaled by setting wire `c3`. The rightmost branch can terminate or exit `T` as respectively signaled by setting wires `c7` and `c6`. Since exiting `T` or terminating the parallel lead to the same continuation, the continuations wires `c8` and `c10` will reach the same input pin in any global circuit in which our fragment is placed.

When the right branch exits `T`, the leftmost branch must be killed; technically, its `halt` statements must be removed from the current haltset. This is the role of the *inhibition* wire `i1` that sends an inhibition signal to the `halt` register. In an actual execution context, the inhibition signal can also come from an enclosing parallel statement itself killed by some trap exit. It is then received on pin i by the wire `i0`.

The final equations of the `Parallel` and `Halt` cells are:

$$\texttt{Parallel} \quad \left\{ \begin{array}{lcl} s' &=& s \\ a' &=& a \\ c' &=& c \\ c'2 &=& c2 \\ p1 &=& c'2 \\ c'1 &=& c1 * \neg p1 \\ p0 &=& c1 + p1 \\ c'0 &=& c0 * \neg p0 \\ i' &=& i + p1 \end{array} \right.$$

$$\texttt{Halt} \quad \left\{ \begin{array}{lcl} c'1 &=& c + (a * s) \\ s' &:=& (c + (a * s)) * \neg i \end{array} \right.$$

where `p0` and `p1` are local wires used to compute the parallel continuation and inhibition values: if ci is the selected continuation, ci is set and all continuations cj are reset for $j \leq i$, and i' is set if `p1` is.

A Sample Execution

Assume the circuit receives control by `c0` and therefore sets `c1`.

- Assume `S2` is present. Then `c5` is set by the `Halt` cell and `c6` is set by the right `Present` cell. The parallel cell selects the appropriate continuation `c10` and inhibits the halt register by setting `i1`.
- Assume instead `S2` is absent. Then `c5` is set by the `Halt` cell and `c7` is set by the right `Present` cell. The selected continuation is `c9`; it signals halting to an eventual enclosing parallel statement. Since the inhibition wire `i1` is low, the `Halt` cell register is set. The circuit then remains in the same state in further clock cycles as long as the activation wire `a0` remains high and `S1` remains low: the wires set are `s2`, `s1`, `s0`, `a1`, `c2`, `c4`, `a2`, `c5`, and `c9`. If `a0` remains high and `S1` is set, the wires set are `s2`, `s1`, `s0`, `a1`, `c2`, `c3`, and `c8`. The whole construct terminates and the register is reset since

`c1` and `a2` are low. The incoming activation wire `a0` can also become low before `S1` occurs, for example because an enclosing watchdog elapses. Then the `Halt` register is also reset.

General Parallel Cells

In fact, the size of the priority queue in a parallel cell depends on the number of nested traps exited from within its source parallel statement. The number of pins $ci, c'i$ for $i \geq 2$ correspond to the number of enclosing traps. With no trap, there is no such pin. The example explained one level of trap. With two levels of traps, as in

```
trap U in
  trap T in
    ...||...
  end
end
```

there would be a pin $c2$ for `T` and a pin $c3$ for `U`, and so on.

5.3 Summary of the Translation

The translation is done by connecting together cells corresponding to source statements. The cells are the same for all programs, but the parallel cells have a variable continuation arity according to the number of enclosing traps.

The logical *skeleton* of the translation is given by the tree of `Halt`, `Watch`, and `Parallel` cells which mimics the tree of source `halt`, `watching`, and `||` statements. Each edge of the tree is composed of an upward *selection* wire and a downward *activation* wire. Two sets of wires reinforce the skeleton: *control* wires that signal halting and go upwards from `Halt` and `Parallel` cells to `Parallel` cells, and opposite *inhibition* wires that force resetting the `Halt` registers in case of exceptions.

In addition to the above cells, one finds a `Boot` cell used to boot the circuit, and `Present` cells generated by source `present` and `watching` statements. These cells are linked together and to skeleton cells by *control* wires. Each `Present` cells also receives as input a *signal* wire. Signal wires come either from input signal pins or from local signal cells, which are simply `or` gates. Control wires transfer the control from cell to cell. They also emit signals by being connected to output signal pins or to local signal `or` gates. The wiring of control wires is determined by a continuation analysis, see section 6.

5.4 Optimization

The reader may find that our circuits contain lots of wires and of logical levels, even for simple programs. In fact, this is because they are obtained by a strucural translation process and there is much room for automatic optimization. Many wires are simply connected with each other. Many generated logical functions are readily grouped by logic optimizers. Constant folding can also be used: for instance, the top activation wire is always set; using this fact, one can statically simplify many gates.

Therefore, our circuits should not be directly implemented; they should instead be given as input to logic optimizers. We presently use optimizers based on Binary Decision Dags (or

BDD's), see [11, 15, 24]. They drastically reduces the actual size of circuits. They can also dicover redundancies between registers and suppress some of them [7].

Altogether, we believe that we can obtain final circuits that are as good as carefully hand-designed ones. Because of the power and efficiency of BDD-based optimization techniques, we think there is no need to search for a more sophisticated translation process.

5.5 The Translation is Sometimes Incorrect

Our translation does not translate correctly all programs. There are difficulties with local signals and with loops over parallel statements.

First, we have allocated a single wire for a local signal. But even within a single reaction, an Esterel signal can have several independent avatars. Consider a statement of the form

```
loop
  signal S in stat end
end
```

When the body terminates, it is restarted at the same instant with a *fresh* signal `S`. This is made obvious by unfolding the body to get

```
loop
  signal S in stat end;
  signal S in stat end
end
```

which is semantically equal and where there are clearly two distinct signals.

In our circuits, a signal wire has only one state at a time: we cannot implement general local signals. We must require all local signals to be declared at toplevel in the module body. This is not too big a restriction in practice.

The second incorrectness is more subtle. The translation of the statement

```
loop
  await S
end
```

is correct, but the translation of the equivalent statement

```
loop
  await S
||
  nothing
end
```

is not since it involves an unstable combinatorial loop through the parallel synchronizer: when `S` occurs, the parallel terminates and the loop makes it halt at the same time on `await S`. But halting justs inhibits the termination that should provoke it, hence the combinatorial loop. Unfolding the body would solve the problem; it still builds a combinatorial loop, but this time a safe one.

The Esterel software checks for sufficient conditions for translation correctness. We are presently investigating a more powerful translation that will correctly translate all Esterel programs. It will be reported in another paper.

6 The Formal Translation to Hardware

We define the translation formally and prove its correctness in absence of bad loops over parallels. As explained in section 5, we assume all local signal declarations to be at toplevel in the module body.

6.1 Circuits

We consider a circuit to be given by a set of *input wires*, a set of *output wires*, a set of *local wires* and a set of *wire definitions* that define output and local wires. There are two kinds of wire definitions:

- An *implication* definition $w \Leftarrow exp$ expresses a partial definition, read as "connect exp to w". There can be several implications per wire.
- A *register* definition $w := exp$ defines a wire to be initially 0 and then the value of exp at previous clock cycle. There can be only one register definition per wire.

Given a circuit C and a wire w, the set of implications $w \Leftarrow exp_i$ in C defines w as $w = \bigvee_i exp_i$. Hence the right-hand-sides of implications are connected to an `or` gate. If a wire w has no definition, it is considered to have an empty set of implication definitions, and therefore to be defined by $w = 0$. To stress the fact that a wire has a single implication definition in a circuit, we can write this definition using '=' instead of '$\Leftarrow$'.

Given any register state and any input, the semantics of a circuit is classically defined as a unique fixpoint of the equations, and a circuit is correct if a unique fixpoint always exist in any (reachable) state. We assume this to be well-known.

6.2 The Translation Environment

The formal translation is done by natural semantics inference rules [21]. The sequents have the form $\rho \vdash stat \rightarrow C$, where ρ is a wire environment, $stat$ is an Esterel statement, and C is the resulting circuit.

As in natural semantics or in PROLOG, allocation of new wires is implicit and done when encountering free variables. To make things clear, we shall comment each rule and explicitly tell which are the newly allocated wires.

The environment ρ is made of several wires, whose functions have been explained in section 5. It contains the following fields

- An incoming control wire c.
- A selection wire s.
- An activation wire a.
- An inhibition wire i.
- A vector of continuation wires $\vec{c}$. The wire $\vec{c}^{\,0}$ corresponds to termination, the wire $\vec{c}^{\,1}$ corresponds to halting, the wire $\vec{c}^{\,k+2}$ corresponds to exit $k+2$, that is to exiting k trap levels.

- A set of signal wires S, one for each input, output, or local signal S. For simplicity, we assume that all local signals have distinct names; then all local signal wires can be preallocated.

We use the classical dot notation to get environment components: for instance, $\rho.c$ denotes the control wire of ρ. Given an environment ρ, we shall often need to consider another environment ρ' that differs from ρ by the value of one field, say by changing $\rho.c$ into c'. We then write $\rho' = \rho[c \leftarrow c']$. The notation extends naturally when changing several fields.

To translate a module, we allocate a boot control wire `b` and a register `n` of equations $\mathtt{b} = \neg\mathtt{n}$, $\mathtt{n} := 1$ as in section 5, a dumy selection wire `s`, two dummy wires `c0` and `c1` for the (unused) continuations, a dummy inhibition wire `i`, and one wire `S` per signal, declaring respectively input and output signals as inputs and outputs to the circuit. We translate the module body in the environment

$$\rho_0 = (\mathtt{b}, \mathtt{s}, 1, \mathtt{i}, (\mathtt{c0}, \mathtt{c1}), \vec{\mathtt{S}})$$

6.3 The Translation Rules

The cells of section 5 were useful for an intuitive explanation, but in rules it is simpler to produce directly equations.

For a `nothing` statement, we connect the incoming control to the termination continuation wire.

$$\rho \vdash \mathtt{nothing} \longrightarrow \rho.\vec{c}^{\,0} \Leftarrow \rho.c$$

For a `halt` statement, we connect the incoming control to the halt continuation wire, to signal halting to an enclosing parallel statement. We allocate a new selection wire s' defined as a register with input as explained in section 5. We connect it to the environment selection wire $\rho.s$.

$$\rho \vdash \mathtt{halt} \longrightarrow \left| \begin{array}{l} \rho.\vec{c}^{\,1} \Leftarrow \rho.c + (\rho.a * \rho.s) \\ \rho.s \Leftarrow s' \\ s' := (\rho.c + (\rho.a * \rho.s)) * \neg\rho.i \end{array} \right.$$

For an `emit` statement, we connect the incoming control to the termination wire and to the signal wire.

$$\rho \vdash \mathtt{emit\ S} \longrightarrow \left| \begin{array}{l} \rho.\vec{c}^{\,0} \Leftarrow \rho.c \\ \rho.\mathtt{S} \Leftarrow \rho.c \end{array} \right.$$

For a sequence, we allocate a new wire c' for control transmission. We translate the first statement with c' as termination and the second statement with c' as incoming control.

$$\frac{\begin{array}{c} \rho[\vec{c}^{\,0} \leftarrow c'] \vdash stat_1 \longrightarrow C_1 \\ \rho[c \leftarrow c'] \vdash stat_2 \longrightarrow C_2 \end{array}}{\rho \vdash stat_1 ; stat_2 \longrightarrow \left| \begin{array}{l} C_1 \\ C_2 \end{array} \right.}$$

For a loop, we allocate a new wire c' to handle looping and we connect the incoming control to it. We translate the body with c' both as incoming control and as outgoing continuation.

$$\frac{\rho[c \leftarrow c', \vec{c}^{\,0} \leftarrow c'] \vdash stat \longrightarrow C}{\rho \vdash \texttt{loop}\ stat\ \texttt{end} \longrightarrow \left| \begin{array}{l} c' \Leftarrow \rho.c \\ C \end{array} \right.}$$

For a `present` statement, we allocate two new control wires c_1 and c_2; then c_1 is set when the incoming control is present and the signal is present, while c_2 is set when the incoming control is present and the signal is absent. We translate the branches with c_1 and c_2 as respective incoming controls.

$$\frac{\begin{array}{c} \rho[c \leftarrow c_1] \vdash stat_1 \longrightarrow C_1 \\ \rho[c \leftarrow c_2] \vdash stat_2 \longrightarrow C_2 \end{array}}{\rho \vdash \texttt{present S then}\ stat_1\ \texttt{else}\ stat_2\ \texttt{end} \longrightarrow \left| \begin{array}{l} c_1 = \rho.c * \rho.\mathsf{S} \\ c_2 = \rho.c * \neg\rho.\mathsf{S} \\ C_1 \\ C_2 \end{array} \right.}$$

For a `watching` statement, we allocate a new selection wire s' and connect it to $\rho.s$, and we allocate a new activation wire a'. The outgoing activation wire a' is set if if s' and $\rho.a$ are set and the signal is absent. The outgoing termination wire $\rho.\vec{c}^{\,0}$ is set if s' and $\rho.a$ are set and the signal is present.

$$\frac{\rho[s \leftarrow s', a \leftarrow a'] \vdash stat \longrightarrow C}{\rho \vdash \texttt{do}\ stat\ \texttt{watching S} \longrightarrow \left| \begin{array}{l} \rho.s \Leftarrow s' \\ a' = \rho.a * \rho.s * \neg\rho.\mathsf{S} \\ \rho.\vec{c}^{\,0} \Leftarrow \rho.a * \rho.s * \rho.\mathsf{S} \\ C \end{array} \right.}$$

The parallel rule is of course the most complex one. It follows exactly the intuitive explanation given in section 5. We allocate a selection wire s' connected to $\rho.s$, an inhibition wire i', a continuation vector $\vec{c'}$ of the same length k as $\rho.\vec{c}$, and a priority vector $\vec{p}$ of length $k-1$. We recursively translate the body with the new selection, inhibition, and continuation wires. Then we establish the priority queue to compute the outgoing continuations and the new inhibition wire i'.

$$\frac{\begin{array}{c} k = |\rho.\vec{c}| \\ \rho[s \leftarrow s', i \leftarrow i', \vec{c} \leftarrow \vec{c'}] \vdash stat_1 \longrightarrow C_1 \\ \rho[s \leftarrow s', i \leftarrow i', \vec{c} \leftarrow \vec{c'}] \vdash stat_2 \longrightarrow C_2 \end{array}}{\rho \vdash stat_1 \,||\, stat_2 \longrightarrow \left| \begin{array}{l} \rho.s \Leftarrow s' \\ \rho.\vec{c}^{\,k-1} \Leftarrow \vec{c'}^{\,k-1} \\ \vec{p}^{\,k-2} = \vec{c'}^{\,k-1} \\ \rho.\vec{c}^{\,k-2} \Leftarrow \vec{c'}^{\,k-2} * \neg\vec{p}^{\,k-2} \\ \vec{p}^{\,k-3} = \vec{c'}^{\,k-2} + \vec{p}^{\,k-2} \\ \ldots \\ \vec{p}^{\,0} = \vec{c'}^{\,1} + \vec{p}^{\,1} \\ \rho.\vec{c}^{\,0} \Leftarrow \vec{c'}^{\,0} * \neg\vec{p}^{\,0} \\ i' = \begin{cases} \rho.i & \text{if } k \leq 3 \\ \rho.i + \vec{p}^{\,2} & \text{if } k > 3 \end{cases} \\ C_1 \\ C_2 \end{array} \right.}$$

For a `trap`, we shift by 1 all wires in $\rho.\vec{c}$ after position 2 and we insert the termination continuation $\rho.\vec{c}^{\,0}$ at exit position 2. The vector notations are obvious.

$$\frac{\rho[\vec{c} \leftarrow (\rho.\vec{c}^{\,0}, \rho.\vec{c}^{\,1}, \rho.\vec{c}^{\,0}) \bullet \rho.\vec{c}^{\,2..}] \vdash stat \longrightarrow C}{\rho \vdash \texttt{trap T in } stat \texttt{ end} \longrightarrow C}$$

For an `exit`, we connect the incoming control to the appropriate continuation.

$$\rho \vdash \texttt{exit T}^k \longrightarrow \rho.\vec{c}^{\,k} \Leftarrow \rho.c$$

For a local signal declaration, we simply translate the body since the signals have been pre-allocated.

$$\frac{\rho \vdash stat \longrightarrow C}{\rho \vdash \texttt{signal S in } stat \texttt{ end} \longrightarrow C}$$

7 Correctness of the translation

We first explain roughly the proof idea as if the translation was always correct. Consider the body *stat* of a correct module placed in the initial environment where the local signal wires have been cut. Then there are two separate wires for each local signal, one for input and one for output. Consider a signal environment E and a haltset H. There exists a unique behavior $stat_1 \xrightarrow[E]{E',1,H'} stat_1'$ with $stat_1' = \mathcal{R}(stat_1^{H'})$, and a unique behavior $\mathcal{R}(stat_1^{H}) \xrightarrow[E]{E',1,H''} stat_1''$ with $stat_1'' = \mathcal{R}(stat_1^{H''})$; uniqueness is obvious since there are no local signal declarations in $stat_1$.

The circuit fragment $C(stat_1)$ obtained by translating $stat_1$ has two incoming control wires c and a. Then setting c realizes the first behavior, while setting the activation wire

a realizes the second behavior. Furthermore, because of loops, c and a can be both set. Then the circuit sums up both behaviors with no interference between them. The proof goes simply by structural induction.

Once this is shown, close the local signal wires. Then, for the module body *stat*, for any state H and real input event I, there exists a unique local event L and a unique output event O such that

$$\mathcal{R}(stat^{H}) \xrightarrow[I \cup L \cup O \cup \{\texttt{tick}\}]{O,1,H'} \mathcal{R}(stat^{H'})$$

But closing the local signal wires in the circuit has exactly the same coherence effect as in the semantics: a signal is there if and only if it is emitted. Since the circuit can do nothing but mimic the behavioral semantics and since there is only one fixpoint in the semantics by the correctness hypothesis, there is only one fixpoint in the circuit and it is the required one[4].

Therefore, one can view the circuit a a *folding of all possible behavioral semantics proof trees* of a program and of its residuals in all possible environments. What the electrons do is to select the right prooftree in one clock cycle given a residual and an input.

The only problem with the above proof argument is that sending control to a parallel by both c and a does *not* sum up the behaviors: one of the continuations can be discarded by the other one. Here, we shall simply prove that the circuit works fine under the assumption that the problem can never appear dynamically[5]. This leads to the following condition:

Condition NSP: a correct program is said to be *NSP* (Non Schizophrenic for Parallels) if for any haltset H and for any event E, no parallel subterm $stat = stat_1 \mid\mid stat_2$ that contains a halt in H is evaluated in the behavioral semantics proof of the reaction of the module body under E both under the form *stat* and under the form $\mathcal{R}(stat^{H+})$.

This is certainly a strange and non-structural condition, but its main advantage is to be amazingly trivial to check in the Esterel software compiling process. We have put an appropriate specific option in the Esterel compiler to report its failure.

Theorem: For any correct NSP Esterel module M, the circuit $C(M)$ has exactly the same input-output behavior as M.

proof: The proof goes just as sketched, but we must inductively ensure that no parallel receives c and a together.

We first study the circuit reactions when the local signal wires are opened. We consider a given haltset H and a given input event E. Let P be the proof of $\mathcal{R}(stat^{H}) \xrightarrow[E]{E',1,H'} \mathcal{R}(stat^{H'})$.

Given a subterm $stat_1$ of *stat*, define the type of $stat_1$ in P as follows: $stat_1$ is of type *null* if it does not appear in P, of type c if it appears in P only under the form $stat_1$, of type a if it appears in P only in the form $\mathcal{R}(stat^{H+})$, and of type ca if it appears in both forms.

[4] We talk here of abstract circuits, or equivalently we assume that concrete circuits do always find the unique fixpoint when it exists.

[5] The right solution would be to use *two* synchronizers, one for c and one for a, and to duplicate some of the logic of the body to signal termination to the appropriate synchronizer; in fact, one must use more than two synchronizer in the general case to properly handle parallel statement nesting; this will be the subjet of a forthcoming paper.

For the circuit $C(stat_1)$ generated by $stat_1$, we say that we send the control *null* if we set neither c nor a, the control c if we set c and not a, the control a if we set a and not c while s is set, and the control ca if we set both c and a while s is set.

We show the following properties on any subterm $stat_1$ by structural induction:

(a) If $stat_1$ receives the control as indicated by its type in P, then it will itself send the control to all its subterms as indicated by their type in P.

(b) Under control *null*, $C(stat_1)$ sets no continuation, no signal, and no halt.

(c) If $stat_1$ is of type c and $stat_1 \xrightarrow[E]{E'_c,\,k_c,\,H_c} stat'_1$, then, under control c, $C(stat_1)$ emits E'_c, sets the sole continuation $\vec{c}^{\,k_c}$, and sets exactly the halts in H_c iff its incoming inhibition wire i has value 0.

(d) If $stat_1$ is of type a and $\mathcal{R}(stat_1^{H+}) \xrightarrow[E]{E'_a,\,k_a,\,H_a} stat'_1$, then, under control a, $C(stat_1)$ emits E'_a, sets the sole continuation $\vec{c}^{k_a}$, and sets exactly the halts in H_c iff its incoming inhibition wire i has value 0.

(e) If $stat_1$ is of type ca, then, under control ca, $C(stat_1)$ realizes the union of the behaviors of case (c) and (d).

First notice some general facts. The s wire is set for $stat_1$ iff $stat_1^{H+}$. Hence only statements that contain halts in H will receive both a and s. By construction, any circuit $C(stat_1)$ does nothing under control *null* and sets no halt when its incoming inhibition wire i is set; otherwise, its sets its halts normally. Also, since all statements merge their emitted signals by or gates, the signal behavior will always be the expected one.

The statements `nothing`, `emit S`, and `exit T` are always of type *null* or c and they exhibit the (c) behavior under c. A `halt` can be of any type, but it always sets $\vec{c}^{\,1}$ and its register if $i = 0$ as required under control c, a, or ca.

Consider a sequence $stat_1$; $stat_2$ of type c. Then $stat_1$ is itself of type c, and the induction tells that $C(stat_1)$ behaves just as $stat_1$ under c. If $stat_1$ terminates, then $stat_2$ is of type c since the first sequence rule must be applied in the proof (it cannot be of type ca, otherwise the sequence itself would be of that type). But $C(stat_1)$ sets $\vec{c}^{\,0}$ that starts $stat_2$ under control c by the sequence wiring. The induction shows (c). If $stat_1$ does not terminate, then $stat_2$ is of type *null* and $C(stat_2)$ receives no control and does nothing; hence the sequence behaves just as $stat_1$, which shows (c). Condition (a) also follows from this case analysis.

The proof of (d) and (a) is similar for a sequence of type a, analyzing separately the cases $stat_1^{H+}$ and $stat_2^{H+}$.

Consider finally a sequence of type ca. First assume $stat_1^{H+}$. Then $stat_1$ itself is of type ca, and the induction applies to it. Furthermore, $stat_2$ is started under c iff $stat_1$ terminates under c, a, or both. But giving twice the control to $stat_2$ is just the same as giving it once, since incoming control wires are gathered by an `or` gate, and (e) follows. Next assume $stat_2^{H+}$. Then $stat_1$ is of type c, while $stat_2$ is of type a if $stat_1$ does not terminate, making (e) obvious, and of type ca if $stat_1$ terminates; in the latter case, (e) is established by induction on $stat_2$. The case analysis is finished for the sequence, and it also shows (a) in all cases.

The other operators are handled in the same way. For a parallel, one is never in case (d) by the NSP hypothesis, and one remembers that the i wire is set in case of exit to kill the haltsets of the subterms.

Finally, as explained before, the circuit is forced to compute the same fixpoint as the behavioral semantics when closing the local signal wires. To finish the proof, just notice that the module body *stat* receives c at the first instant from the boot wire and a at the next instants from the selection wire that is plugged back as the activation wire.

8 Implementation

8.1 Actual Implementation on Perle0

We have experimented our hardware implementation on the PERLE0 board developed at DEC PRL [8]. It consists of a set of 25 synchronous XILINX programmable logic cell arrays placed on a board and piloted by a SUN^{TM} workstation.

The translation is performed by the *strldg* processor (ESTERELtodigital), which is integrated in the standard ESTEREL compiler[6]. The generated logical circuit is printed out in PERLE0 format and translated into XILINX native format by the PERLE0 software (we could as well produce portable formats such as PALASM). The logical circuit is then given to optimizers and the optimized result is fed into an automatic placer-router, without any pre-placing indication. This gives a XILINX circuit specification. Using this environment, the turnover is on the order of 15 minutes from source program to running circuit for a medium-size program.

On PERLE0, we provide a symbolic debugging and exact speed measure environment, with interactive symbolic input and output from within Lisp or C. The speed measure reports at which maximal clock speed a circuit correctly handles a benchmark. In practice, the speed is 30 to 75 nanoseconds for a small program (30 ns for the circuit presented in the appendix), and 75 to 100 nanoseconds for a medium size program that still fits into a single chip (about 2-4 pages of source ESTEREL code), this on a 3020 XILINX chip.

In debug or speed-measure mode, the ESTEREL program is implemented on a single chip and other chips are devoted to bus and debug interfaces. The applications we have handled so far are man-machine interfaces, real-size local area network controllers [22], and various circuit controllers including those used in the PERLE0 board itself to communicate with the bus and with the tested program.

8.2 Simulation and Correctness Proofs

ESTEREL and LUSTRE are themselves able to describe digital hardware. The `strldg` processor is also able to unparse the circuit in ESTEREL or LUSTRE. There are two main uses:

- After compiling the ESTEREL version of the circuit, we can use the full ESTEREL programming environment to perform simulations, analysis, and optimizations.
- Once the circuit behavior automaton is generated by ESTEREL compiler, we can use the AUTO verification system [9] to automatically check for equivalence between the source code and the circuit automata. This may seem unnecessary since the translation

[6]In fact, most of the skeleton and continuation analysis is already done by the standard compiler first pass.

has been mathematically proved correct, but software is software and double-checks are always useful. Furthermore, the translation can work properly even if the sufficient correctness conditions are not met. If AUTO reports equivalence, the circuit is perfectly usable even if it works by chance!

Of course, using the ESTEREL standard compiler for such a circuit unparsing analysis makes sense only if the circuit has a reasonable number of states, say 50 to 500, which is usually the case for controllers.

9 Conclusion

Although ESTEREL was not at all designed as a harware description language, the work presented here shows it well-suited to very high-level verified hardware generation. The hardware implementation is directly based of the formal semantics. The electrons circulating in the wires perform the computation of the proof tree associated with a program and an input within a single clock cycle. The circuit itself can be viewed as a folding of all possible semantical proof trees into a graph structure.

The translation we have presented is not general since programs are assumed to obey a sufficient NSP condition; we are now in the process of releasing a full correct translation of ESTEREL into circuits, based on extensions of the same ideas.

We investigate three main kinds of applications: implementing existing ESTEREL programs on hardware to improve their performance, using ESTEREL to directly program hardware controllers, and using ESTEREL to build reference controllers to which actual hand-tailored controllers can be automatically proved equivalent. Our present experiments are very promising and leave place for sophisticated optimization.

To our knowledge, the closest related works are the hardware implementation of LUSTRE and SML [13]. The LUSTRE and ESTEREL implementations are developed in parallel and are fully compatible. Compared to SML, ESTEREL is much more elaborate as a programming language, having in particular watchdogs, exceptions, and instantaneous broadcast. Our implementation is direct and does not use a translation to automata, although such a translation is also available. LUSTRE, SML, and ESTEREL all give access to temporal logic or process calculi based verifiers. We need more experience to compare the relative qualities of the languages and of their verification tools.

Acknowledgements: This work was motivated by discussions with Jean Vuillemin and Patrice Bertin from DEC Paris Research Laboratory. It owes much to the work of Georges Gonthier on the semantics of Esterel. The actual implementation on Perle0 was done at DEC PRL under the supervision of Patrice Bertin, who provided unvaluable help. The experiments with BDD optimizers were conducted by Olivier Coudert and Jean-Christophe Madre (BULL CRG), as well as by Hervé Touati (DEC PRL).

References

[1] G. Berry. Real-time programming: General purpose or special-purpose languages. In *IFIP World Computer Congress*, 1989.

[2] G. Berry and A. Benveniste. The synchronous approach to reactive and real-time systems. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, Sept. 1991.

[3] G. Berry and L. Cosserat. The synchronous programming languages Esterel and its mathematical semantics. In S. Brookes and G. Winskel, editors, *Seminar on Concurrency*, pages 389–448. Springer Verlag Lecture Notes in Computer Science 197, 1984.

[4] G. Berry, P. Couronné, and G. Gonthier. Synchronous programming of reactive systems: an introduction to Esterel. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 35–55. Elsevier Science Publisher B.V. (North Holland), 1988. INRIA Report 647.

[5] G. Berry and G. Gonthier. Incremental development of an HDLC protocol in esterel. In *Proc. Ninth International Symposium on Protocol Specification, Testing, and Verification*. North-Holland, 1989.

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

[7] C. Berthet, O. Coudert, and J.-C. Madre. New ideas on symbolic manipulations of finite state machines. In *Proc. of International Conference on Computer Design (ICCD), Cambridge, USA*, 1990.

[8] P. Bertin, D. Roncin, and J. Vuillemin. Introduction to programmable active memories. In J. McCanny, J.McWhirter, and E. Swartzlander, editors, *Systolic Array Processors*, pages 301–309. Prentice-Hall, 1989.

[9] G. Boudol, V. Roy, R. de Simone, and D. Vergamini. Process calculi, from theory to practice: Verification tools. In *Automatic Verification Methods for Finite State Systems, LNCS 407*, pages 1–10. Springer-Verlag, 1990.

[10] F. Boussinot and R. de Simone. The esterel language. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, Sept. 1991.

[11] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli. Multilevel Logic Synthesis. *Proceedings of the IEEE*, 78(2):264–300, February 1990.

[12] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. Lustre: A declarative language for programming synchronous systems. In *Proc. 14th. Annual ACM Symposium on Principles of Programming Languages*, 1987.

[13] E. M. Clarke, D.E. Long, and K. L. McMillan. A language for compositional specification and verification of finite state hardware controllers. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, Sept. 1991.

[14] D. Clément and J. Incerpi. Programming the behavior of graphical objects using Esterel. In *TAPSOFT '89, Springer-Verlag LNCS 352*, 1989.

[15] O. Coudert and J.-C. Madre. A unified framework for the formal verification of sequential circuits. In *Proc. of International Conference on Computer Aided Design (ICCAD), Santa Clara, USA*, 1990.

[16] G. Cousineau. An algebraic definition for control structures. *Theoretical Computer Science*, 12:175–192, 1980.

[17] T. Gauthier, P. Le Guernic, and L. Besnard. Signal, a declarative language for synchronous programming of real-time systems. In *Proc. 3rd. Conf. on Functional Programming Languages and Computer Architecture, Springer-Verlag LNCS 274*, 1987.

[18] G. Gonthier. Sémantique et modèles d'exécution des langages réactifs synchrones; application à Esterel. Thèse d'informatique, Université d'Orsay, 1988.

[19] P. Le Guernic, M. Le Borgne, T. Gauthier, and C. Le Maire. Programming real time applications with signal. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, Sept. 1991.

[20] N. Halbwachs, P. Caspi, and D. Pilaud. The synchronous dataflow program ming language lustre. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, Sept. 1991.

[21] G. Kahn. Natural semantics. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 237–258. Elsevier Science Publisher B.V. (North Holland), 1988.

[22] M. C. Mejia Olvera. Contribution à la conception d'un réseau local temps réel pour la robotique. Thèse de docteur-ingénieur, Université de Rennes, 1989.

[23] G. Murakami and Ravi Sethi. Terminal call processing in esterel. Research Report 150, AT&T Bell Laboratories, 1990.

[24] H. Savoj, H. Touati, and R. K. Brayton. The Use of Image Computation Techniques in Extracting Local Don't Cares and Network Optimization. In *to appear in Proceedings of IEEE International Conference on Computer-Aided Design*, November 1991.

[25] M. Shand, P. Bertin, and J. Vuillemin. Hardware speedups in long integer multiplication. In *Proc. 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, Island of Crete, Greece*, pages 138–145, 1990.

Appendix — A Simple Bus Interface Example

As a toy application example, we program an interface module between a bus and a hardware application. This interface is a slight simplification of the one effectively used in the PERLE0 board to run actual ESTEREL programs hardware translations. Although the program is very small, we use submodules to illustrate modular programming.

The Interface informal Specification

The interface repeatedly waits for input from the bus, tells the application to store the corresponding data word, triggers a computation, and tells the application to send back the output data word to the bus when the computation is terminated and the bus is ready for output.

The interface receives two signals from the bus, `BUS_WRITE` for input and `BUS_READ` for output. It acknowledges both input and output by sending back `BUS_ACK`.

Data words are received or emitted directly by the application. To control data input, the interface tells the application to connect its input buffers to the data bus by setting a signal `OPEN_INPUT`. This signal is maintained until the arrival of `BUS_WRITE` included. After one clock cycle, the interface sends `BUS_ACK` and starts the computation by sending a signal `GO` to the application. When the computation is terminated, the application sends back a signal `FINISHED`. The output data is then ready in the application output buffers. The interface tells the application to connect its output buffers to the bus by sending a signal `OPEN_OUTPUT`. This can be done only when the computation is finished and when the bus has sent `BUS_READ`. After waiting a clock cycle for the data to be effectively present on the bus, the interface sends `BUS_ACK`.

In addition, we assume that the bus can send at any time a `RESET` signal telling the interface to reset itself to its initial state.

The Interface Esterel Program

The interface module is written as follows:

```
module Interface :
input BUS_READ, BUS_WRITE, RESET;     % from bus
output BUS_ACK;                       % to bus
output OPEN_INPUT, OPEN_OUTPUT, GO;   % to application
input FINISHED;                       % from application

loop
   loop
      run Input;
      run ComputeAndOutput
   end
each RESET.
```

Notice that the `RESET` signal is completely factored out and effectively resets the interface independently of its current internal state.
The `Input` submodule is written as follows:

```
module Input :
input BUS_WRITE;   % from bus
output BUS_ACK;    % to bus
output OPEN_INPUT; % to application
trap INPUT in
   sustain OPEN_INPUT
||
   await BUS_WRITE do exit INPUT end
end;
await tick;
emit BUS_ACK.
```

Here we use a `trap` construct to ensure that `OPEN_INPUT` is emitted when `BUS_WRITE` is received. One could write as well:

```
do
   sustain OPEN_INPUT
watching BUS_WRITE;
emit OPEN_INPUT;
```

By the semantics of the `watching` construct, the statement "`sustain OPEN_INPUT`" is not executed when `BUS_WRITE` occurs. This is why `OPEN_INPUT` must be explicitly emitted at that instant.

The `ComputeAndOutput` module is written as follows:

```
module ComputeAndOutput :
input BUS_READ;          % from bus
output BUS_ACK;          % to bus
output GO, OPEN_OUTPUT;  % to application
input FINISHED;          % from application
[
   await BUS_READ
||
   emit GO;
   await FINISHED;
];
emit OPEN_OUTPUT;
await tick;
emit BUS_ACK.
```

Notice how the parallel statements realizes the synchronization: it terminates exactly when the computation is finished and the bus ready to read.

Once optimized, placed, and routed, the circuit uses up 9 cells on on a XILINX 3020 circuit. There are 5 registers and 11 logical functions with a total of 35 inputs.

The Advantages of Esterel

The automaton generated by the ESTEREL compiler is pictured in figure 3. Notice the diamond generated by the parallel statement that appears in `ComputeAndOutput`. Notice also the reset arrows that go from any state into state 1: they are all generated by the

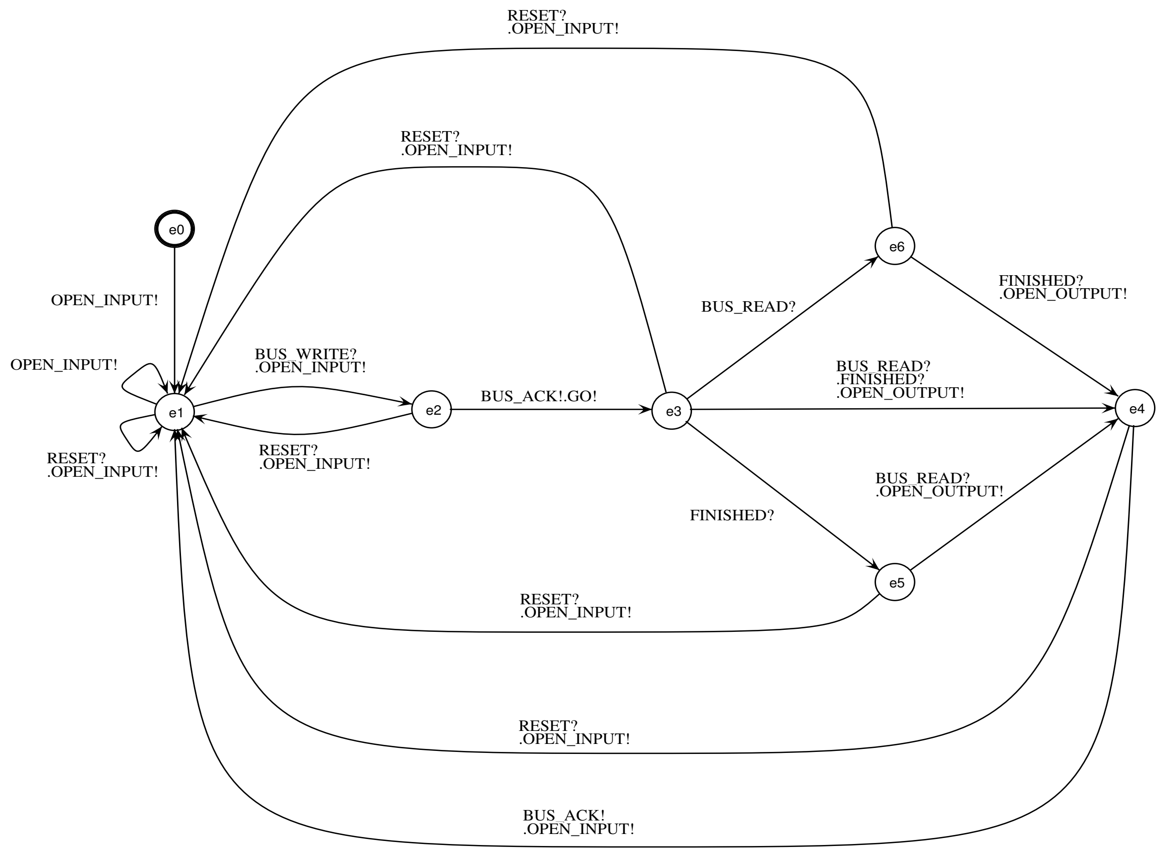

Figure 3: The Interface Automaton

single "`loop ... each RESET`" statement. Of course, such a small automaton can be easily designed by hand. The advantage of ESTEREL programming really appears for more complex controllers. The modularity of the language, its built-in concurrency, and the power of its control structures allow the user to build controllers by assembling individually simple modules into bigger ones. For example, to perform speed benchmarks on PERLE0, we use a variant of the bus interface that inputs two data words and performs computation and output twice in a row. To obtain this interface, one just changes the `Interface` module body into (roughly):

```
run Input [signal OPEN_INPUT_1 / OPEN_INPUT];
run Input [signal OPEN_INPUT_2 / OPEN_INPUT];
run ComputeAndOutput [signal OPEN_OUTPUT_1 / OPEN_OUTPUT,
                                 GO_1 / GO];
run ComputeAndOutput [signal OPEN_OUTPUT_2 / OPEN_OUTPUT,
                                 GO_2 / GO]
```

Usually, a relatively simple change to a specification involves a simple and local change to an ESTEREL program. This is definitely not true of finite automata, which are highly unstable w.r.t. specification changes. We strongly believe that programming controllers in ESTEREL is one order of magnitude simpler that designing finite state machines by hand.