PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

Esterel on Hardware [and Discussion]

Gerard Berry, C. A. R. Hoare and W. A. Hunt

Phil. Trans. R. Soc. Lond. A 1992 **339**, 87-104
doi: 10.1098/rsta.1992.0027

Esterel on hardware

By Gérard Berry

Centre de Mathématiques Appliquées, Ecole Nationale Supérieure des Mines de Paris, Sophia-Antipolis, 06565 Valbonne, France

Esterel is a synchronous concurrent programming language for reactive systems (controllers, protocols, man–machine interfaces, etc.). Esterel has an efficient software implementation based on a well-defined mathematical semantics. I 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. The circuit is shown to be semantically equivalent to the source program. The hardware translation is effectively implemented and used on the programmable active memory Perle0 developed by J. Vuillemin and his group at Digital Equipment.

1. Introduction

Esterel (Berry & Cosserat 1984; Berry & Gonthier 1988; Boussinot & de Simone 1991) is a synchronous programming language devoted to *reactive systems*, i.e. to systems that maintain a continuous interaction with their environment by handling and generating 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 (Berry & Gonthier 1991; Murakami & Sethi 1990), man–machine interfaces (Clément & Incerpi 1989), systems drivers, etc.

We present a hardware implementation of the pure synchronization subset of the language that builds a specific circuit for each program. We state the correctness of this implementation with respect to the mathematical semantics of the language. We describe the experiments made so far and the possible uses of the hardware implementation.

(a) *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* (Benveniste & Berry 1991), which states that control transmission, communication, and elementary computation actions take no time. The control structures include sequencing, testing, looping, concurrency, and a powerful exception mechanism which is fully compatible with concurrency. Communication between concurrent statements is performed by instantaneously broadcasting signals.

The perfect synchrony hypothesis is shared by the synchronous data-flow languages Lustre (Caspi *et al.* 1987; Halbwachs *et al.* 1991) and Signal (Le Guernic *et al.* 1991). It makes programming modular and flexible, and it reconciles input–output determinism and concurrency; the intrinsic non-determinism of asynchronous languages such as Occam or Ada often makes reactive programming and debugging needlessly difficult (see Berry 1989).

Esterel is rigorously defined by well-analysed mathematical semantics, given in both denotational and operational styles (Berry & Gonthier 1988; Gonthier 1988).

Phil. Trans. R. Soc. Lond. A (1992) **339**, 87–104
Printed in Great Britain

(b) *Esterel in software*

The standard Esterel compiler is directly based on the mathematical semantics. Sophisticated algorithms compile concurrency away and translate a concurrent reactive program into an equivalent efficient sequential automaton implemented in a conventional language like C.

In addition to the compiler, the Esterel environment includes graphical simulators, symbolic debuggers, and interfaces to automata-based program verification systems (Boudol *et al.* 1990).

(c) *Esterel in hardware*

By its mere conception, Esterel is well adapted to high-level programming of circuit controllers; this logic synthesis task is known to be surprisingly difficult and error-prone with standard techniques such as direct description of finite state machines. The Esterel programming primitives allow the user to write modular programs such that small changes in a specification lead to small changes in a program, a property that is not true of most presently used techniques; a typical example is given in Berry & Gonthier (1991).

When dealing with hardware, we restrict ourselves to the Pure Esterel pure synchronization subset of the language. Pure Esterel programs can be implemented in hardware by feeding standard CAD systems with the compiled automata. However, this indirect implementation makes no use of the source program concurrency structure. This is clumsy in hardware where concurrency is free, unlike in software.

The much better direct hardware implementation we present here is based on Gonthier's semantic analysis of Esterel (Gonthier 1988). 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 one clock cycle, however complex the program is.

The translation has been completely formalized and proved correct with respect to the behavioural semantics of Esterel defined in (Berry & Gonthier 1988). In this inference-rule based semantics, given a program state and an input, the current output and the next state are defined by the unique transition proof that can be established from the semantic rules. The circuit exactly computes that proof.

(d) *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™-based Perle programmable coprocessor developed at Digital Equipment Paris Research Laboratory by J. Vuillemin *et al.* (Bertin *et al.* 1989; Shand *et al.* 1990).

We are currently investigating two kinds of applications: implementing existing Esterel programs in hardware to match high performance constraints, and programming hardware controllers and control parts of complex circuit in Esterel.

The data-flow oriented synchronous language Lustre has also a hardware implementation based on its mathematical semantics. Esterel and Lustre are complementary and can be used in conjunction for general circuits handling both control and data.

(e) *Structure of the paper*

We present the Pure Esterel language and its intuitive semantics in §2. The hardware translation is explained by examples in §3. In §4, we give hints on the

mathematical aspects and on the correctness proof. We discuss actual implementation and experiments in §5.

The more detailed paper (Berry 1991) presents the formal techniques and correctness proof. The translation presented there works only for a restricted kind of Esterel programs. The restrictions have been removed recently, and a fully complete translation will be presented in a forthcoming paper.

2. Pure Esterel

We first present the basic objects manipulated by Pure Esterel programs: signals and events. We then present the basic language on which the semantics is defined and the full language that includes user-friendly statements definable from basic statements. We give enough material for the paper to be self-contained, but we do not detail fine points nor the Esterel programming style, see the Esterel documentation for such information (Berry & Gonthier 1988; Boussinot & de Simone 1991).

(*a*) *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 *present* in that event, otherwise it is *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 instantaneously building an *output event* E'_i, which is synchronous with E_i in the sense that any external observer observes a *single* event $E_i \cup E'_i$. This is also true of any other program placed in parallel.

The flow of time being entirely defined by the sequence of reactions to input events, we also call a reaction an *instant*. This give meaning 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`. This is a slight addition to the original language of Berry & Gonthier (1988). Being always present, the `tick` signal is analogous to the constant 1 in circuits; when programming digital circuits, it naturally denotes clock ticks.

(*b*) *Modules*

The basic Pure Esterel programming unit is the *module*. A module has an *interface*, which specifies its input and output signals, and a *body*, which is a statement that specifies its behaviour. The body can use any number of local signals for internal broadcast communication. To achieve modular programming, a module can instantiate other modules. Instantiation is done by inline code replacement and will not be described here. Here is a sample module definition:

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

(c) *Basic statements*

The basic statements of Pure Esterel are:

`nothing`
`halt`
`emit S`
stat$_1$; *stat*$_2$
`loop` *stat* `end`
`present S then` *stat*$_1$ `else` *stat*$_2$ `end`
`do` *stat* `watching S`
stat$_1$ || *stat*$_2$
`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.

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

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

(d) *The intuitive semantics*

The intuitive semantics describes control transmission between statements and 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. 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 *wait* at that instant; it will be re-activated at the next instant.

1. `nothing` terminates instantaneously.
2. `halt` never terminates nor exits. It always waits.
3. An '`emit S`' statement broadcasts the signal `S` and terminates instantaneously.
4. When started, a sequence '*stat*$_1$; *stat*$_2$' immediately starts *stat*$_1$ and behaves accordingly. If and when *stat*$_1$ terminates, *stat*$_2$ starts immediately and determines the behaviour of the sequence from then on. If and when *stat*$_1$ exits a trap `T`, so does the whole sequence. Notice that *stat*$_2$ is never started if *stat*$_1$ always waits or exits a trap. Notice also that '`emit S1; emit S2`' emits `S1` and `S2` simultaneously and terminates instantly.
5. 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. To prevent infinite instantaneous loops, the body of a loop is not allowed to terminate instantaneously when started.
6. 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.
7. The '`do` *stat* `watching S`' watchdog statement starts immediately its body *stat* and uses `S` as a time guard for its execution:

   If *stat* terminates or exits a trap strictly before `S` occurs, then the `watching` statement instantaneously terminates or exits the same trap. The time guard has no effect.

If, in the strict future of the starting instant, `S` occurs while *stat* is still active, then the `watching` statement terminates instantaneously, *stat* being not activated at the corresponding instant. In other words, the occurrence of `S` at an instant instantaneously kills *stat* without letting it perform any action at that instant.

Notice that an occurrence of `S` at the starting instant does not provoke termination and is simply ignored; a variant where an initial `S` does provoke immediate termination can be derived from other basic statements. A variant where the body *stat* is activated a last time when `S` occurs can also be derived. Both variants will be presented below.

8. 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 exists a trap `T` or both statements exit the same trap `T`, then the parallel exits `T`. If both statements simultaneously exit distinct traps `T1` and `T2`, the parallel only exits the *outermost* of these traps, the other one being discarded.

9. 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 accordingly 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).

10. An '`exit T`' statement instantaneously exits the trap `T`.

11. When started, the statement '`signal S in` *stat* `end`' immediately starts the 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.

(i) *Remarks*

An emission is transient, and a `present` test is instantaneous. There is an asymmetry between present and absent signals. The `emit` statement sets a signal present, but no statement sets it absent: by the coherence law, this is just the default.

A loop never terminates by itself; the only way to end it is to kill it by elapse 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.

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`.

(*e*) *Examples*

In a guard, `S` can be any signal, a second as well as a centimetre, a clock tick, or generally any kind of interrupt. Therefore, each signal is seen as defining its own time unit; one can say 'stop in less than 3 m' as well as 'press a button every 2 s'.

The only basic statement that provokes waiting 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 wait forever, but the enclosing '`watching S`' guard kills it and terminates when S occurs. 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 the definition of derived statements, we write it as '`await S`'.

The `watching` primitive that waits for an event to *stop* or *preempt* a computation is much more powerful than the `await` primitive that waits for an event to *start* a computation and has been the basis of most event manipulation languages so far. In particular, we have easily derived `await` from `watching`, while the converse is non-trivial.

Nesting temporal statements based on different time units is the main characteristic of the Esterel style. Here is a toy program that emits repeatedly O every I until reception of a signal STOP:

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

The output 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. This strongly preemptive behaviour tends to be the most useful one. However, there are cases where a weaker preemption is needed. For example, one may want to emit O a last time when STOP occurs; one then uses a `trap` statement:

```
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 will'. Weak preemption is easily derived from strong preemption; the converse derivation would be complex.

In a `watching S` statement, the body is started if S is present at the starting instant, the occurrence of S being ignored. In some cases, one wants S to provoke immediate termination even at the starting instant. For this, 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 branch exits T2 but does

not emit S2 since an exit 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. The exit propagation mechanism is deterministic and very useful in practice.

(*f*) *Full Esterel*

The full language has many user-friendly derived statements. We briefly describe some of them, referring to Berry & Gonthier (1988) for the complete list and the expansions into basic statements.

We have already seen two derived constructs: the simple await statement and the immediate guard variant. All test or guards can involve boolean expressions, as in '`present [S1 and S2]`' or '`do` *stat* `watching [not S]`', as well as occurrence counts, as in '`await 3 S`'.

A timeout clause can be executed when a watching statement terminates by elapsing its time guard before body termination:

`do` $stat_1$ `watching S timeout` $stat_2$ `end`

Several events can be waited for in an await statement:

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

To ensure determinism, only $stat_1$ is started if both S1 and S2 occur simultaneously.

There are two temporal loops: '`loop` *stat* `each S`' and '`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 emit S statement emits S only at the current instant; it is often useful to emit a signal continuously, that is at all instants. For this, one uses the '`sustain S`' statement that simply abbreviates '`loop emit S each tick`'.

Finally, a general exception handling mechanism extends basic traps by providing exception handlers (see Berry & Gonthier (1988) for details).

3. Principle of the hardware implementation

I 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. In the first examples, the translation will be structural. In the general case, some logic duplication is needed to correctly handle loops; I give some hints on how to perform this duplication, but give no details.

(*a*) *A first example*

The first example is the following program that involves no parallel statement:

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

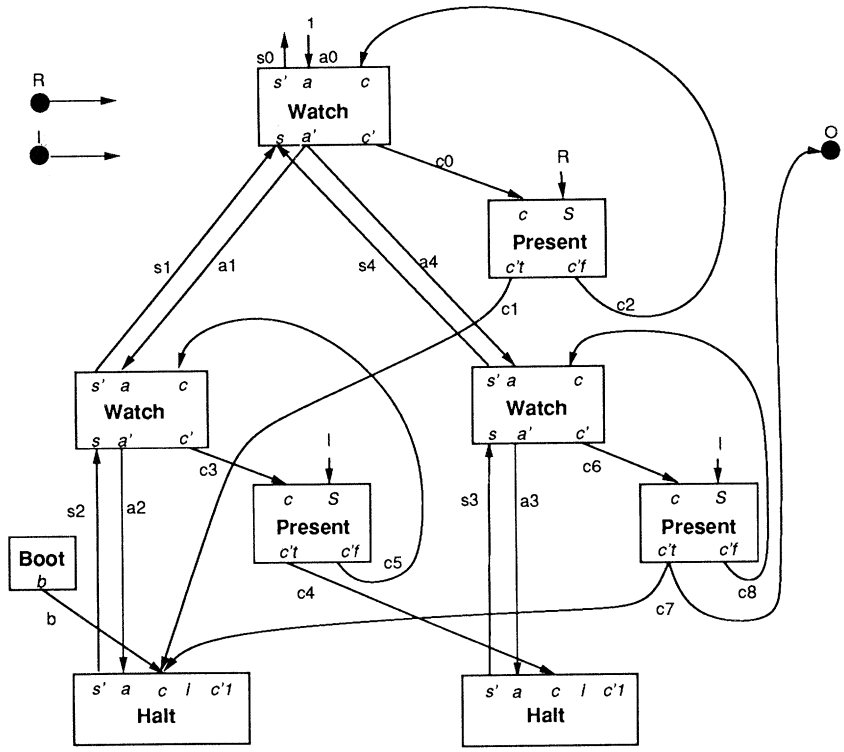

Figure 1. First example.

The behaviour is to emit `O` every two `I`, restarting this behaviour afresh each `R`. The body's expansion into basic statements is:

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

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.

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.

The `Boot` and `Halt` cells each contain one register clocked by the global circuit's clock. The other cells are purely combinational. The `Present` cells 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 cell design and the implementation of signal 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`, `Watch`, and `Present` cells correspond to the first `await` in the original source program, the rightmost ones correspond 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. When the second `await` is active, the wires `s4`, `s3`, and `s0` are set. 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 control wires transfer the control between statements; for example, they are generated by sequences and `present` tests. Furthermore, there is a control wire for each signal; a signal is present if and only if its wire is set.

The cells are defined as follows:

$$\texttt{Boot} \begin{cases} n \coloneqq 1 \\ b = \neg\, n \end{cases}$$

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

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

$$\texttt{Halt}\ s' \coloneqq c \vee (a \wedge s)$$

where the symbol '$\coloneqq$' is used for registers. Registers are supposed to contain initially 0.

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 $c \vee (a \wedge s)$ 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 (this to prevent setting the second `Halt` register in a term such as '`do halt; halt watching S`' when a is set). The definition of `Halt` is only temporary: further pins will be added in §3*b*.

(i) *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 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 through `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 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.

(*b*) *Translating parallel and exceptions*

The most complex operator is of course the parallel one: 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
```

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`'. 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 *c*0, *c*1, and *c*2 record the different termination modes: *c*0 means termination, *c*1 means waiting, and *c*2 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 *c*1 entry of the enclosing `Parallel` cell when such a cell exists.)

The synchronization realized by the parallel statement was defined in the intuitive semantics; it amounts to computing the *max* of the termination modes of the branches, activating only the corresponding continuation: waiting preempts

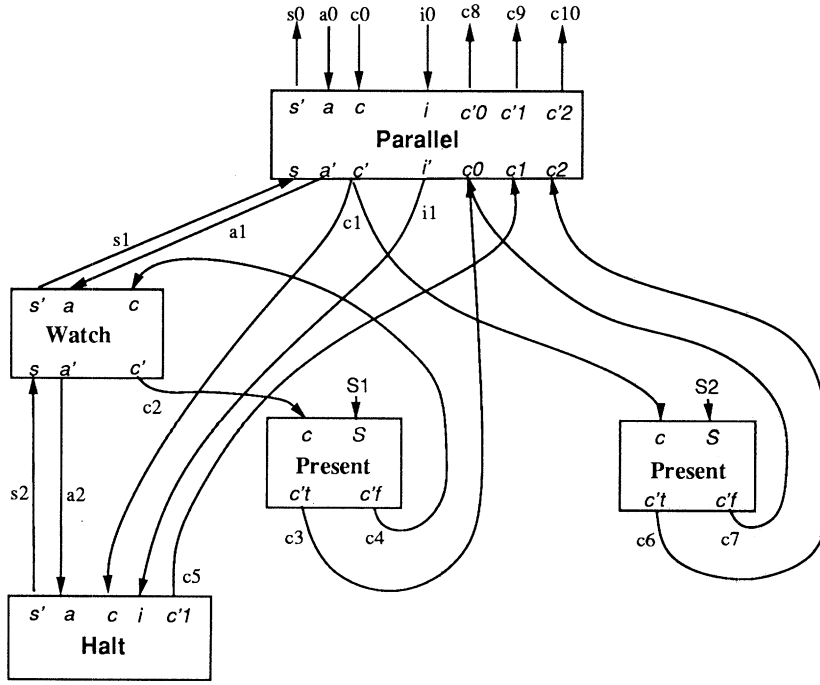

Figure 2. Second example.

termination, trap exit preempts waiting, and outermost traps preempt innermost ones. Therefore the synchronization device is simply a priority queue.

In our example, the left branch can wait, as signalled by setting wire `c5`, or terminate, as signalled by setting wire `c3`. The rightmost branch can terminate or exit `T` as respectively signalled by setting wires `c7` and `c6`. Since exiting `T` or terminating the parallel lead to the same continuation, the continuation 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. 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:

$$\texttt{Parallel}\begin{cases} s' = s \\ a' = a \\ c' = c \\ c'2 = c2 \\ p1 = c'2 \\ c'1 = c1 \wedge \neg\, p1 \\ p0 = c1 \vee p1 \\ c'0 = c0 \wedge \neg\, p0 \\ i' = i \vee p1 \end{cases}$$

$$\texttt{Halt}\begin{cases} c'1 = c \vee (a \wedge s) \\ s' = (c' \vee (a \wedge s)) \wedge \neg i, \end{cases}$$

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 $j \leqslant i$, and i' is set if p1 is.

(i) *A sample execution*

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

1. 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.

2. 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 waiting 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 become low before S1 occurs, for example because an enclosing watchdog elapses. Then the Halt register is also reset.

(ii) *General parallel cells*

In general, 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 $i \geqslant 2$ 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.

(c) *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 three 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 waiting 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 cell also receives as input a *signal* wire. Signal wires come either from input signal pins or from

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

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 (Berry 1991).

(*d*) *Loops and logic duplication*

The simple translation above does not translate correctly all programs, because it does not handle properly the subtlety of the `loop` construct. The difficulties appear in loops containing local signal declarations or parallel statements.

First, it is not possible to allocate a single wire for a local signal: even within a single reaction, an Esterel signal can have several independent incarnations. 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 equivalent and where there are clearly two distinct signals that can be independently present or absent at the instant where control flows between the two occurrences of *stat*. In the actual translation process, we detect this fact and duplicate some of the loop body's logic.

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

```
loop
  await S
end
```

is correct, but the simple translation of the equivalent statement

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

would be incorrect since it would involve an unstable electrical loop through the parallel synchronizer: when `S` occurs, the parallel terminates, the loop restarts it immediately and the parallel waits on `await S`. But waiting justs inhibits termination, hence the combinational loop: the synchronizer is asked to perform two distinct synchronizations at a time. Here again, logic duplication is necessary and cleverly performed by the actual translation.

In all cases, logic duplication concerns only control wires and synchronizers. The registers and skeleton are kept untouched. In the worst (artificial) case, logic duplication can square the size of the logic; in practice, one only observes small linear expansion factors, and optimization efficiently reduces the logic, see §5.

4. Correctness of the translation

The translation into a circuit is itself formally defined by a set of structural equations. See Berry (1991) for the subcase where no logic duplication is necessary; the general translation is similar and will be published in a forthcoming paper.

So far, I have presented the generated circuits and their behaviour on a purely intuitive ground. The correctness of the translation relies on the fact that one can also view circuits as mathematical structures deeply related to the proof theory of the Esterel semantics. I briefly describe the behavioural semantics, the haltset coding of states, and the correctness proof arguments.

(a) *The behavioural semantics*

The *behavioural semantics* (Berry & Gonthier 1988) defines the reaction of a program to an input event using Plotkin's Structural Operational Semantics technique (Plotkin 1981). It 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 to an input sequence is then defined inductively by chaining elementary reactions:

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

To compute a transition $M \xrightarrow[I]{O} M'$, we use an auxiliary relation $stat \xrightarrow[E]{E', k} stat'$

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, as explained in §3. The auxiliary relation is defined by a set of inference rules (Berry & Gonthier 1988; Berry 1991).

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 turn depends on E. Hence E and E' will be computed as *fixpoints*. What makes the semantics non-trivial is the non-monotonic character of the fixpoint computation, implied by the possibility of testing both for presence and absence of signals.

Not all Esterel programs have well-defined semantics; some exhibit causality defects analogous to unstable electrical loops in circuits (see Berry & Gonthier 1988; Berry 1991). For example, the program:

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

has no meaning since S should be present if and only if it is absent. It is similar to a circuit equation of the form $x = \neg x$ that has no solution. The precise study of the existence of (causal) solutions in both Esterel and circuits is delicate and well outside

the scope of this paper. In practice, we restrict ourselves to programs that generate loop-free circuits, which obviously have deterministic behaviours; this property is statically checked by the Esterel compiler. Our experience shows that the class of Esterel programs that yield such circuits covers most practical cases.

(*b*) *The haltset coding of states*

The technical basis of the hardware translation is the *haltset* coding of states. 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 behavioural semantics. The derivatives bear an important structural relation with the source term *stat*: any derivative can be unambiguously coded by a *haltset* H of *stat*, that is by a set of occurrences of `halt` statements in the expansion of *stat* into basic statements. Each `halt` represents a position on which the control is currently waiting, and a haltset acts as a distributed program counter.

Given a haltset H in *stat*, one defines a new statement $\mathscr{R}(stat^H)$ that represents the state of *stat* when the control is on H. See the equations in Berry (1991).

(*c*) *The correctness proof*

The key correctness argument is that the circuit acts as a *fixpoint solver* and as a (fast) *theorem prover* for the semantics. The circuit itself can be viewed as a *folding of all possible semantical proof trees of all possible derivatives into a graph structure*. Given a state of the circuit – i.e. a derivative – and an input, the wires set represent exactly the right proof tree.

Technically, given any derivative *stat′* of *stat* and any input event I, the behavioural semantics determines a reaction

$$stat' \xrightarrow[I]{O} stat''.$$

Let H' and H'' be the haltsets of *stat′* and *stat″*. Then, one proves that having the registers of H' set in the circuit exactly amounts to perform a reaction of $stat' = \mathscr{R}(stat^{H'})$, that the set of `Halt` registers set after the circuit tick is precisely H'', and that the outputs are those of O.

To prove this, one shows inductively that for any statement *stat* and haltset H the selection wires precisely implement the $\mathscr{R}()$ function, that setting the input control wire `c` amounts to execute *stat*, and that setting the activation wire `a` amounts to execute $\mathscr{R}(stat^H)$. Since both the circuit and the behavioural semantics have unique solutions, these solutions coincide and the circuit indeed compute the required semantics. See Berry (1991) for the detailed proof.

5. Optimization and implementation

The reader may find that our circuits contain lots of wires and of logical levels, even for simple programs. In fact, there is much room for automatic optimization. Many wires are simply connected with each other. Constant folding simplifies many gates. Many generated logical functions are readily grouped by logic optimizers.

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 diagrams (or BDDs) (see Brayton *et al.* 1990; Coudert & Madre 1990; Savoj

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

et al. 1991). They drastically reduce the actual size of circuits. They can also discover redundancies between registers and suppress some of them.

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.

We have experimented with our hardware implementation on the Perle board developed at Digital Equipment Paris Research Laboratory (Bertin *et al.* 1989). It consists of a set of 25 synchronous Xilinx programmable logic cell arrays placed on a board and controlled by a conventional workstation.

The translation is performed by a specific processor integrated in the standard Esterel compiler. The generated logical circuit is fed into BDD-based optimizers and finally into the Perle CAD system. Using this environment, the turnover is of the order of 15 min from source program to running circuit for a medium-size program. The execution environment provides a symbolic debugger from actual circuit states to source code and a dynamic exact speed measurement from benchmarks.

The applications we have handled so far are man–machine interfaces, real-size local area network controllers (Mejia 1989), and various circuit controllers including those used in the Perle board itself to communicate with the workstation bus.

Program correctness proofs can be performed using either the verification tools of the Esterel or Lustre environments or any circuit verification system.

6. Conclusion

Although Esterel was not at all designed as a hardware description language, the work presented here shows it is well suited to very high-level verified hardware generation. The Esterel style is well adapted to hardware controller programming. The hardware generation is directly based on the formal semantics and is proved correct. After logic optimization, the generated circuits are of excellent size and speed quality.

To our knowledge, the closest related works are the hardware implementation of Lustre and SML (Clarke *et al.* 1991). The Lustre and Esterel implementations are developed in parallel and are fully compatible. To our belief, compared with 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. We need more experience to compare the relative qualities of the languages and of their verification tools.

This work was motivated by discussions with Jean Vuillemin and Patrice Bertin from Digital Equipment Paris Research Laboratory (PRL), done partly in this laboratory, and co-financed by INRIA. It owes much to the work of Georges Gonthier on the semantics of Esterel. The actual implementation on Perle0 was done at PRL under the supervision of Patrice Bertin, who provided invaluable help. The experiments with BDD optimizers were conducted by Olivier Coudert and Jean-Christophe Madre at BULL and by Hervé Touati at PRL.

References

Benveniste, A. & Berry, G. 1991 The synchronous approach to reactive and real-time systems. In *Another look at real time programming Proc. IEEE* **79**, 1270–1282.

Berry, G. 1989 *Real-time programming: general purpose or special-purpose languages, information processing 1989* (ed. G. X. Ritter), pp. 11–17. Elsevier.

Berry, G. 1991 A hardware implementation of pure Esterel. *Proceedings International Workshop on Formal Methods in VLSI*. Springer-Verlag LNCS.

Berry, G. & Cosserat, L. 1984 The synchronous programming languages Esterel and its mathematical semantics. In *Seminar on concurrency* (ed. S. Brookes & G. Winskel), pp. 389–448. Springer-Verlag Lecture Notes in Computer Science 197.

Berry, G. & Gonthier, G. 1988 The Esterel synchronous programming language: design, semantics, implementation. INRIA Res. Rep. 842. *Sci. Comp. Prog.* (In the press.)

Berry, G. & Gonthier, G. 1991 Incremental development of an HDLC entity in Esterel. *Comp. Networks* **22**, 35–49.

Berry, G., Couronné, P. & Gonthier, G. 1988 Synchronous programming of reactive systems: an introduction to Esterel. *Programming of future generation computers* (ed. M. Nivat & K. Fuchi), pp. 35–55. Elsevier.

Berthet, C., Coudert, O. & Madre, J. C. 1990 New ideas on symbolic manipulations of finite state machines. *Proceedings of International Conference on Computer Design* (*ICCD*).

Bertin, P., Roncin, D. & Vuillemin, J. 1989 Introduction to programmable active memories. In *Systolic array processors* (ed. J. McCanny, J. McWhirter & E. Swarzlander), pp. 301–309. Prentice-Hall.

Boudol, G., Roy, V., de Simone, R. & Vergamini, D. 1990 Process calculi, from theory to practice: verification tools. In *Automatic verification methods for finite state systems*, pp. 1–10. Springer-Verlag LNCS 407.

Boussinot, F. & de Simone, R. 1991 The Esterel language. In *Another look at real time programming. Proc. IEEE* **79**, 1293–1304.

Brayton, R. K., Hatchel, G. D. & Sangiovanni-Vincentelli, A. L. 1990 Multilevel logic synthesis. *Proc. IEEE* **78**, 264–300.

Caspi, P., Halbwachs, N., Pilaud, D. & Plaice, J. 1987 Lustre: a declarative language for programming synchronous systems. *Proceedings 14th Annual ACM Symposium on Principles of Programming Languages*, pp. 178–188.

Clarke, E., Lond, D. E. & McMillan, K. L. 1991 A language for compositional specification and verification of finite state hardware controllers. In *Another look at real time programming. Proc. IEEE* **79**, 1283–1292.

Clément, D. & Incerpi, J. 1989 Programming the behavior of graphical objects using Esterel. *Proceedings TAPSOFT '89*, pp. 111–126. Springer-Verlag LNCS 352.

Coudert, O. & Madre, J. C. 1990 A unified framework for the formal verification of sequential circuits. *Proceedings of International Conference on Computer Aided Design* (*ICCAD*), pp. 126–129.

Cousineau, G. 1980 An algebraic definition for control structures. *Theor. Comp. Sci.* **12**, 175–192.

Gonthier, G. 1988 Sémantique et modèles d'exécution des langages réactifs synchrones; application à Esterel. Thèse d'Informatique, Université d'Orsay, France.

Halbwachs, N., Caspi, P., Raymond, P. & Pilaud, D. 1991 The synchronous dataflow programming language Lustre. In *Another look at real time programming. Proc. IEEE* **79**, 1305–1320.

Kahn, G. 1988 Natural semantics. In *Programming of future generation computers* (ed. M. Nivat & K. Fuchi), pp. 237–258. Elsevier.

Le Guernic, T., Gautier, T., Le Borgne, M. & Le Maire, C. 1991 Programming real time applications with signal. In *Another look at real time programming. Proc. IEEE* **79**, 1321–1336.

Mejia Olvera, M. C. 1989 Contribution à la conception d'un réseau local temps réel pour la robotique. Thèse de Docteur-Ingénieur, Université de Rennes, France.

Murakami, G. & Sethi, R. 1990 Terminal call processing in Esterel. AT & T Bell Laboratories Res. Rep. no. 150.

Plotkin, G. 1981 A Structural approach to operational semantics. Tech. Rep. DAIMI FN-19, University of Aarhus, Denmark.

Savoj, H., Touati, H. & Brayton, R. K. 1991 The use of image computation techniques in extracting local don't cares and network optimization. *Proceedings of IEEE International Conference on Computer-Aided Design* (*ICCAD*), pp. 514–517.

Shand, M., Bertin, P. & Vuillemin, J. 1990 Hardware speedups in long integer multiplication. *Proceedings 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 138–145.

Discussion

C. A. R. Hoare (*University of Oxford, U.K.*). Does Dr Berry use the Xilinx automatic placement and routing software?

G. Berry. Yes. The output of the compiler is simply an unplaced and unrouted netlist. The compiler provides no meaningful placement information. However, any other more efficient placement and routing tool can be used instead.

W. A. Hunt (*Computational Logic, Inc., Texas, U.S.A.*). What is the formal connection between Lustre and Esterel?

G. Berry. Both languages use exactly the same perfect synchrony concept and are semantically compatible. Lustre is actually one of the target languages of the Esterel-to-circuits compiler. Lustre and its variant Pollux that incorporates static arrays are fully adequate to data-path description, while Esterel is much better for controllers.

For the full Esterel language that also embodies computation actions, the situation is slightly more complex. We know how to freely mix Esterel and Lustre programs, although this has not been done yet. However, the existing Esterel and Lustre compilers produce the same object code.

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES