Electronic Notes in Theoretical Computer Science 65 No. 5 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume65.html 15 pages

SAXO-RT: Interpreting ESTEREL Semantic on a Sequential Execution Structure

Etienne Closse, Michel Poize, Jacques Pulou [1]
Patrick Venier, Daniel Weil [1]

France Telecom R&D, 28 chemin du Vieux Chêne, 38243 Meylan cedex, France

Abstract

The SAXO-RT compiler implements a original method for compiling the concurrent synchronous language ESTEREL into sequential C code. The method is optimized for embedded systems with very tight memory and real-time constraints and shows significant performance improvement on industrial size examples. Source code is sliced into small code sequences called control points, statically scheduled so as to be compatible with ESTEREL semantic. Speed reaction is optimized without increasing code size, by executing at each reaction only active code sequences. In this paper, we present in detail how ESTEREL semantic is interpreted on our execution structure.

1 Introduction

Real-time applications which are embedded in systems like planes, satellite or GSM mobile phones, are doubly critical: first because the security of the system depends on them (a failure can lead to system crash) and second, they usually have very tight energy and memory constraints. At first sight, designing these systems with a Finite State Automata approach can seem appropriate, but it does not fit with increase of system complexity. Complex embedded systems need to be designed as modular and composable (or concurrent) elementary components. Thus, synchronous languages appear to be a natural choice for specifying and implementing critical parts of these systems: they offer a high-level interface for describing finite state automata and are therefore well suited to automatic proving techniques, even on industrial size systems.

In this paper, we consider *software* aspects of these systems. Data-flow oriented synchronous languages like LUSTRE [6] are now widely used in many control-command systems. An application written in LUSTRE consists in a

[1] Email:firstname.lastname@rd.francetelecom.com

set of concurrent equations on signals that are triggered by usually periodic clocks. Lustre compilation techniques are now mature and produce very efficient codes which are embedded in critical systems like Airbus planes and nuclear plants. But the equational style of Lustre doesn't fit very well with applications like alarm systems or telecommunications protocols. A better choice for the developer is a control oriented synchronous language like Esterel [2]. Esterel mostly distinguish from a data-flow synchronous language by an imperative programming style (instead of equational) and by the ability of describing hierarchical automata.

Unfortunately, if much effort has been made to compile Esterel on hardware targets [8], traditional Esterel compiling methods are not efficient for software targets. Complexity analysis of Esterel classical compiling methods shows that for most industrial applications, global automaton compilation [7] has exponential code and data size while boolean equations compilation [8] produces a code with a too slow reaction time.

Therefore, a good approach is to cut the code into small statically scheduled sequences which are executed only when needed, and avoiding as much as possible any code duplication. This interest of this kind of approach is confirmed by the work done in [14] on generation of sequential code from parallel code: linearization is based on a control flow graph extracted from the input parallel language. In our case, the difficulties consist in finding the right control flow graph which catches the semantic of Esterel and then sequentializing this graph and mapping it on a efficient execution structure. So far, two approaches have been proposed:

- The approach proposed by S. Edwards [9] [10] constructs the control flow graph from the execution tree represented by Esterel intermediate code (IC) and targets a hierarchical execution machine which reproduces the source code parallel and branching structure.
- Our approach whose principle was first published in French [11] almost concurrently with [9], published in English and tested with industrial size examples in [12]. The main difference is that the semantic of Esterel is interpreted directly into a control flow graph. Sequential code sequences are then extracted from this graph, statically scheduled and mapped on a execution structure which is mostly flat instead of being hierarchical.

The drawbacks presented in [12] required our compiler to be completely redesigned with a finer scheduling algorithm which was shortly exposed in [13].

This paper presents in a detail manner how Esterel semantic is interpreted on our execution structure (§2). Each Esterel kernel instructions is first transformed into an *event graph* (§3 and §4. This graph is transformed into a sequential control flow graph (§5 and §6) and into sequential code (§7). Performance comparisons are then presented in §9.

2 Interpreting Esterel Semantic on a Sequential Execution Structure

2.1 Esterel *semantic*

Esterel constructive semantic relies on four basic rules [4]:

Reactivity: given a set of inputs, there is always at least one set of possible outputs.

Determinism: given a set of inputs, there is never more than one set of possible outputs.

Signal coherence law (also called signal atomicity): a signal S can be present if and only if the signal S is emitted during the same synchronous reaction.

Constructivity: no speculative computation is performed on signal status: in other words, causality analysis is monotonic (and must respect the sequence operator ";").

The execution of an Esterel program on the sequential execution structure presented below requires an interpretation of Esterel constructive semantic which are summarized by the following rules:

R1: Set or reset of a signal must always *precede* any test of this signal (signal coherence law and constructivity).

R2: The termination test of a parallel construct must always occur *after* each parallel branch has terminated.

R3: The test guarding a strong preemption (`abort` and `suspend`) is always evaluated *before* the execution of the instruction body.

R4: Terminating a `trap` construct by an `exit` nested in a parallel construct must occur *after* the other parallel branches have met a pause or terminated.

2.2 Execution Structure

The Esterel syntactic graph is transformed into a *Event Graph* (EG). At the difference with Program Dependance Graphs that are used in [14] or with the Concurrent Control Flow Graph (CCFG) of [9], an Event Graph not only represents dependencies between nodes but also represents action of nodes on other nodes. In this paper, the only possible actions are switching on or off a node for the next or current synchronous reaction and testing the state of a node.

In order to produce a code executable on a sequential machine, the event graph is flatten down into an execution structure that we call Sequential Control Flow Graph (SCFG). The construction of the EG (cf. §3) is determined by the characteristics of this execution structure (fig.2):

- the SCFG nodes is a *totally ordered subset* of the EG.

- at each synchronous reaction, only a subset of *active* nodes is executed, with the total order defined by the SCFG.
- to each node are associated two Booleans $S_{current}(n)$ and $S_{next}(n)$ which are respectively true if and only if n is scheduled for the current reaction (respectively for the next reaction).
- a node n can change $S_{current}(n')$ if and only if $n < n'$, where "$<$" is the total order defined by the SCFG.
- a node n can change the status $S_{next}(n')$ of any node n'.
- a node n can test the status $S_{current}(n')$ only if $n' < n$
- a node n can test the status $S_{next}(n')$ of any node n'

So as to produce efficient code, the SCFG is partitioned into a totally ordered set of *Control Points* which defines our execution structure (§6):

> *a synchronous reaction consists in scrolling the current status of each control point and executing only those that are active in a static order which respects the total order defined above.*

3 Event Graph Characteristics

3.1 Nodes

We restrict in this paper to the Esterel kernel defined in [4] (`p` and `q` are any syntaxically correct Esterel terms):

`nothing`	`p;q`
`pause`	`p‖q`
`signal S in p end`	`loop p end`
`emit S`	`trap T in p end`
`present S then p else q end`	`suspend p when S`

The EG of an Esterel program is constructed with 7 sets of nodes $N_{nothing}$, N_{signal}, N_{set}, N_{reset}, N_{assign}, N_{exit}, N_{test}:

- N_{signal} contains nodes representing Esterel signals. We note N^S_{Signal}, the set of nodes representing signal S. Signal nodes are used to interpret signal coherence law and to link signals emission and test (§4.3, §4.4 and §4.5).
- N_{set} and N_{reset} represent nodes emitting or resetting signals (§4.4 and §4.3).
- N_{assign} represent nodes assigning value to variables, e.g. counters that are used in the parallel construct (cf. §4.9).
- N_{exit} contains nodes raising a exception, i.e. corresponding to the `exit` Esterel instruction. The nodes raising a given exception T (instruction `exit T`) are represented by the subset N^T_{exit} (§4.6).
- N_{test} groups all the nodes that test a signal status, a variable or evaluate some boolean expression on the activity of other nodes. Two set of out-

going links, true and false links, are used to represent actions which must be executed when the boolean expression evaluates to true or false (§4.5, §4.7, §4.9 and §4.10).
- $N_{nothing}$ groups all other nodes. As we consider here only Esterel kernel, no action is associated to these nodes.

3.2 Links

EG links represent not only dependancy between nodes but also action from nodes on other nodes. We consider three types of links between EG nodes, action links L_{Action}, test links L_{Test} and order links L_{Order}. We note $n \xrightarrow{\lambda} n'$ if nodes n and n' are linked by a link of type λ.

Action links represent actions of a node on another one and are identified by two kind of flags: an action flag $\alpha \in \{On, Off, PauseOn, PauseOff, Suspend\}$ and a choice flag $\sigma \in \{\diamond, +, -\}$. We note $n \xrightarrow{\sigma\alpha} n'$, an action link between n and n' with action α and choice σ.

Action flags have the following semantic:
- $n \xrightarrow{\sigma On} n' \Leftrightarrow n$ activates n' for the current synchronous instant
- $n \xrightarrow{\sigma Off} n' \Leftrightarrow n$ deactivates n' for the current synchronous instant
- $n \xrightarrow{\sigma PauseOn} n' \Leftrightarrow n$ activates n' for the next synchronous instant
- $n \xrightarrow{\sigma PauseOff} n' \Leftrightarrow n$ deactivates n' for the next synchronous instant
- $n \xrightarrow{\sigma Suspend} n' \Leftrightarrow n$ suspends n', i.e. if n' is active for the current instant, deactivates it for the current instant and activates it for the next instant

Choice flags have the following semantic:
- $n \xrightarrow{\diamond\alpha} n' \Leftrightarrow n$ executes α on n'
- $n \xrightarrow{+\alpha} n'$ and $n \in N_{test} \Leftrightarrow n$ executes α on n' if the expression tested by n is *true*
- $n \xrightarrow{-\alpha} n'$ and $n \in N_{test} \Leftrightarrow n$ executes α on n' if the expression tested by n is *false*

Order links represent scheduling constraints between nodes. $n \xrightarrow{\ll} n'$ expresses the fact that during a given reaction, n *must* be always executed *before* n'. They are used to translate the 4 rules defined in §2.1.

Note that L_{On}, L_{Off}, $L_{Suspend}$ links are implicit order links.

Test links can only leave test nodes. The condition evaluated by a test node is either a *signal status* or *a boolean expression on the test links leaving this node*. There are two types of test links, $L_{On?}$ and $L_{PauseOn?}$:
- $n \xrightarrow{On?} n'$: n tests if n' *has been* executed during the current reaction. Therefore, it implies that $n' \xrightarrow{\ll} n$. $L_{On?}$ links are used for testing that no exception of upper level has been raised in a trap or in a parallel construct (§4.9 and 4.7).
- $n \xrightarrow{PauseOn?} n'$: n test if n' is scheduled for the next reaction.

4 Event Graph Construction

The EG is constructed recursively from the inside to the outside. For this purpose, we define an application $\mathcal{G}$ which associates to any syntactically correct Esterel term p a subgraph $\mathcal{G}_p$ of the EG. Each subgraph is builded from the composition of its own subgraphs.

A EG subgraph $\mathcal{G}_p$ is characterized by:

- a head node $Head(\mathcal{G}_p)$
- a tail node $Tail(\mathcal{G}_p)$
- a set of exit nodes raising exception declared *outside* p:

$$Exit(\mathcal{G}_p) = \mathcal{G}_p \cap \bigcup \{N^T_{exit} / T \notin Exception(p)\}$$

  where $Exception(p)$ represents the set of exceptions that are declared inside p
- a set of pause nodes, i.e. the nodes of $\mathcal{G}_p$ which can be activated for the next instant: $Pause(\mathcal{G}_p) = \mathcal{G}_p \cap \{n / n' \xrightarrow{\sigma PauseOn} n, \sigma \in \{\diamond, +, -\}\}$

We describe now the subgraph $\mathcal{G}$ associated to each Esterel kernel statement.

4.1 nothing

A `nothing` statement does nothing, its subgraph is trivial and contains a single node:

$$Head(\mathcal{G}_{nothing}) = Tail(\mathcal{G}_{nothing}) \in N_{nothing}$$
$$Exit(\mathcal{G}_{nothing}) = Pause(\mathcal{G}_{nothing}) = \emptyset$$

4.2 pause

The graph of a `pause` statement only consists of a node which activates another node for the next instant:

$$Head(\mathcal{G}_{pause}) \in N_{nothing} \xrightarrow{\diamond PauseOn} Tail(\mathcal{G}_{pause}) \in N_{nothing}$$

$$Exit(\mathcal{G}_{pause}) = \emptyset \qquad Pause(\mathcal{G}_{pause}) = \{Tail(\mathcal{G}_{pause})\}$$

4.3 signal S in p end

The `signal` statement declares a new signal S which is reset before entering p:

$$Head(\mathcal{G}_{signal}) \in N^S_{Reset} \xrightarrow{\diamond On} Head(\mathcal{G}_p)$$

To be compatible with Esterel signal coherence law (rule **R1** in §2), this reset must be done before any test on this signal. This is expressed by a scheduling constraint with a signal node n_S associated with each signal declaration:

$$Head(\mathcal{G}_{signal}) \xrightarrow{\ll} n_S \in N^S_{signal}$$

The `signal` construct terminates as soon as p terminates:

$$Tail(\mathcal{G}_p) \xrightarrow{\diamond On} Tail(\mathcal{G}_{signal}) \in N_{nothing}$$

Pause and exit nodes of $\mathcal{G}_{signal}$ are those of $\mathcal{G}_p$:

$$Exit(\mathcal{G}_{signal}) = Exit(\mathcal{G}_p) \qquad Pause(\mathcal{G}_{signal}) = Pause(\mathcal{G}_p)$$

4.4 emit S

The graph of the `emit` construct is trivial:

$$Head(\mathcal{G}_{emit}) = Tail(\mathcal{G}_{emit}) \in N_{Set}$$

The only constraint is that the emission of S occurs before any test on this signal (rule **R1**):

$$\forall n_S \in N^S_{signal}, Head(\mathcal{G}_{emit}) \xrightarrow{\ll} n_S$$

No pauses, no exceptions are raised: $Pause(\mathcal{G}_{emit}) = Exit(\mathcal{G}_{emit}) = \emptyset$

4.5 present S then p else q end

The head of the subgraph is a node which tests S status and must therefore be scheduled after any emission or reset of S (**R1**):

$$\forall n_S \in N^S_{signal}, n_S \xrightarrow{\ll} Head(\mathcal{G}_{present}) \in N_{test}$$

If S is present, p starts, else q starts:

$$Head(\mathcal{G}_{present}) \xrightarrow{+On} Head(\mathcal{G}_p) \qquad Head(\mathcal{G}_{present}) \xrightarrow{-On} Head(\mathcal{G}_q)$$

The `present` terminates as soon as p or q terminates:

$$Tail(\mathcal{G}_p) \xrightarrow{\diamond On} Tail(\mathcal{G}_{present}) \qquad Tail(\mathcal{G}_q) \xrightarrow{\diamond On} Tail(\mathcal{G}_{present})$$

Pause and exit nodes of $\mathcal{G}_{present}$ are the union of those of p and q:

$$Exit(\mathcal{G}_{present}) = Exit(\mathcal{G}_p) \cup Exit(\mathcal{G}_q)$$

$$Pause(\mathcal{G}_{present}) = Pause(\mathcal{G}_p) \cup Pause(\mathcal{G}_q)$$

4.6 exit T

In Esterel, the `exit` construct behaves as a branching statement to the end of the `trap` construct. The head node is initially not connected to anything, it is connected later to the tail of the `trap` construct it belongs to (§4.7):

$$Head(\mathcal{G}_{exit}) = n \in N^T_{exit} \qquad Tail(\mathcal{G}_{exit}) = \emptyset$$

To each `exit` node n is associated a integer $Level(n)$ corresponding to the number of trap declarations that must be traversed before reaching that of T (cf. trap completion codes in [3]).

$Head(\mathcal{G}_{exit})$ is the only exit node and the subgraph is instantaneous:

$$Exit(\mathcal{G}_{exit}) = Head(\mathcal{G}_{exit}) \qquad Pause(\mathcal{G}_{exit}) = \emptyset$$

4.7 trap T in p end

The body p of the `trap` starts immediately and terminates when p terminates normally:

$$Head(\mathcal{G}_p) \xrightarrow{\diamond On} Head(\mathcal{G}_p) \qquad Tail(\mathcal{G}_p) \xrightarrow{\diamond On} Tail(\mathcal{G}_{trap})$$

But p must also terminate when an exception T is raised in p and each exit node $n \in N^T_{exit}$ is linked to a branching node :

$$\forall n \in N^T_{exit}, n \xrightarrow{\diamond On} Tail(\mathcal{G}_{trap}) \in N_{nothing}$$

The exit nodes visible outside are those of p that do not raise the exception T and no pauses are added:

$$Exit(\mathcal{G}_{trap}) = Exit(\mathcal{G}_p) \setminus N^T_{exit} \qquad Pause(\mathcal{G}_{trap}) = Pause(\mathcal{G}_p)$$

4.8 p ; q

The sequence construct is simple. p starts immediately and q starts as soon as p terminates. The construct terminates when q terminates:

$$Head(\mathcal{G}_{sequence}) \xrightarrow{\diamond On} Head(\mathcal{G}_p) \qquad Tail(\mathcal{G}_p) \xrightarrow{\diamond On} Head(\mathcal{G}_q)$$

$$Tail(\mathcal{G}_q) \xrightarrow{\diamond On} Tail(\mathcal{G}_{sequence})$$

Pause and exit nodes are the union of those of p and q:

$$Exit(\mathcal{G}_{sequence}) = Exit(\mathcal{G}_p) \cup Exit(\mathcal{G}_q)$$

$$Pause(\mathcal{G}_{sequence}) = Pause(\mathcal{G}_p) \cup Pause(\mathcal{G}_q)$$

4.9 p || q

With the signal coherence law, the semantic of the parallel is the heart of Esterel semantic. Its semantic is simple : each branches starts immediately and the parallel construct terminates *normally* when each branch has terminated or *abnormally* when an exception declared outside the parallel construct is raised in one of the branches.

The head of a parallel subgraph consists of a reset node which activates immediately each parallel branches. This reset node initializes a counter to the number of parallel branches :

$$Head(\mathcal{G}_{parallel}) = n_{reset} \in N_{assign} \xrightarrow{\diamond On} Head(\mathcal{G}_p) \qquad n_{reset} \xrightarrow{\diamond On} Head(\mathcal{G}_q)$$

Normal termination is constructed by a link from the tail of each branch to a node $n_{decr} \in N_{assign}$ which decrements the counter that has been initialized by $Head(\mathcal{G}_{parallel})$. These nodes are connected to $n_{join} \in N_{test}$ node which tests if the value of this counter is zero, which means that all branches have terminated normally (i.e. not by raising an exception):

$$Tail(\mathcal{G}_p) \xrightarrow{\diamond On} n^1_{decr} \xrightarrow{\diamond On} n_{join} \qquad Tail(\mathcal{G}_q) \xrightarrow{\diamond On} n^2_{decr} \xrightarrow{\diamond On} n_{join}$$

$$n_{join} \xrightarrow{+On} Tail(\mathcal{G}_{parallel})$$

Abnormal termination occurs when an `exit` raises an exception in one of the parallel branches. If no exit of upper level is raised in the other parallel branch (cf. §4.6):

$$n_1 \in Exit(\mathcal{G}_p), n_2 \in Exit(\mathcal{G}_q), Level(n_1) < Level(n_2) \Rightarrow n_1 \xrightarrow{On?} n_2$$

all pause nodes in the other parallel branch must deactivated for next instant, before branching to its corresponding `trap` statement (**R4**) [2] :

$$n_{exit} \in Exit(\mathcal{G}_p) \text{ and } n' \xrightarrow{\sigma PauseOn} n \in Pause(\mathcal{G}_q) \Rightarrow n_{exit} \xrightarrow{-PauseOff} n \text{ and } n' \xrightarrow{\ll} n_{exit}$$

Exit and pause nodes of $\mathcal{G}_{parallel}$ are the union of those of $\mathcal{G}_p$ and $\mathcal{G}_q$:

$$Exit(\mathcal{G}_{parallel}) = Exit(\mathcal{G}_p) \cup Exit(\mathcal{G}_q)$$

$$Pause(\mathcal{G}_{parallel}) = Pause(\mathcal{G}_p) \cup Pause(\mathcal{G}_q)$$

4.10 suspend p when S

The `suspend` construct is rather simple. p starts immediately while the test of S starts only at the next reaction:

$$Head(\mathcal{G}_{suspend}) \xrightarrow{\diamond On} Head(\mathcal{G}_p) \qquad Head(\mathcal{G}_{suspend}) \xrightarrow{\diamond PauseOn} n_{test} \in N_{test}$$

If S is present, the execution of p is suspended and scheduled for the next reaction:

$$n \in Pause(\mathcal{G}_p) \Rightarrow n_{test} \xrightarrow{+Suspend} n$$

[2] Note also that any PauseOn action must occur before any PauseOff action

Note that this implies $n_{test} \xrightarrow{\ll} n$ (rule **R3**). The node n_{test} activates itself for the next reaction:

$$n_{test} \xrightarrow{\diamond PauseOn} n_{test}$$

And if p terminates, the `suspend` construct terminates and the node n_{test} is turned off:

$$Tail(\mathcal{G}_p) \xrightarrow{\diamond On} Tail(\mathcal{G}_{suspend}) \xrightarrow{\diamond PauseOff} n_{test}$$

The exit nodes are those of p but the node n_{test} is a pause node:

$$Exit(\mathcal{G}_{suspend}) = Exit(\mathcal{G}_p) \quad Pause(\mathcal{G}_{suspend}) = Pause(\mathcal{G}_p) \cup \{n_{test}\}$$

4.11 loop p end

The `loop` statement p starts immediately and never terminates:

$$Head(\mathcal{G}_{loop}) = Head(\mathcal{G}_p) \qquad Tail(\mathcal{G}_{loop}) = \emptyset$$

But the difficulty of constructing $\mathcal{G}_{loop}$ is that $Tail(\mathcal{G}_p)$ must not be connected directly to $Head(\mathcal{G}_p)$ in order to cope with signal reincarnation and to eliminate conflicts between loop surface (nodes that can be reached immediately from $Head(\mathcal{G}_p)$) and loop depth (nodes that are separated from $Head(\mathcal{G}_p)$ by at least one $L_{PauseOn}$ link)[3]. A fonction $SurfaceCopy(p)$ duplicates all the nodes that can be reached from $Head(\mathcal{G}_p)$ by following only L_{On} links:

$$Tail(\mathcal{G}_p) \xrightarrow{\diamond On} Head(\mathcal{G}_{(SurfaceCopy(p))})$$

Exits nodes are possibly augmented with exit nodes from the loop surface and no pause nodes are added:

$$Exit(\mathcal{G}_{loop}) = Exit(\mathcal{G}_p) \cup Exit(\mathcal{G}_{SurfaceCopy(p)}) \quad Pause(\mathcal{G}_{loop}) = Pause(\mathcal{G}_p)$$

5 Compatibility Analysis

The operations described above create too many links and unnecessary scheduling constraints between nodes that are never active during the same reaction. These kinds of nodes are called *exclusive* nodes. A pair of nodes belonging each to a true and false branches of the same test node or a pair of nodes that are in sequence but separated by a $L_{PauseOn}$ link are exclusive. A couple of nodes belonging to two different branches of a parallel instruction are not exclusive but are said to be compatible.

This algorithm removes unnecessary L_{order} links and allows the compiler to accept cyclic programs like:

[3] This problem is also known as schizophrenia, cf. [5]

```
  pause; emit TIC;
||
  every TIC do
    pause; emit TAC
  end
||
  every TAC do
    pause; emit TIC;
  end
```

Fig. 1. Cyclic Program

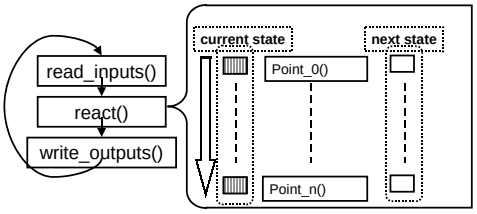

Fig. 2. Execution Structure

```
present A then emit B end;
pause;
present B then emit A end;
```

Actually, this analysis rejects some programs that are constructive like the example shown on fig.1 where no static scheduling is possible without duplicating some code: accepting this kind of programs would require exploring in some way the whole state space and proving that for each state, there exists a correct constructive reaction [4]. This fortunately concerns only very few programs and most industrial examples do not belong to this category.

The class of programs accepted by our compiler is a superset of acyclic ESTEREL programs, in the sense of [3].

6 Scheduling and Control Point Generation

The EG must be transformed into a Sequential Control Flow Graph (SCFG) respecting the properties defined in §2.2. A partial order relation "$<$" between the EG nodes is defined by:

$$n < n' \Leftrightarrow \text{ there exists a path going from } n \text{ to } n' \text{ containing only } L_{On} \text{ and } L_{order} \text{ links}$$

We first search for a total order compatible with"$<$". If a cycle between two nodes is found, i.e. $n < n'$ and $n' < n$, the program is rejected as non-causal. If no cycle is found, there exists a total order between EG Nodes and the EG can be transformed into a SCFG.

In order to construct efficient code, the EG is sliced into a set of *control points* $\mathbf{P}$. A control point can be seen as a thread of nodes that can always be executed with the same order. Actually, $\mathbf{P}$ is a partition of the EG which is compatible with the order relation "$<$" and we note $\widetilde{n}$ the equivalence class of node n. This partition has the following properties:

- each pause node starts a control point:

$$p_1 \neq p_2 \text{ and } \exists (n_1, n_2)/n_1 \xrightarrow{\sigma PauseOn} p_1 \text{ and } n_2 \xrightarrow{\sigma PauseOn} p_2 \Rightarrow \widetilde{p_1} \neq \widetilde{p_2}$$

- we note $n \xrightarrow{seq} n'$ if there exists a "sequential" path between n and n'

[4] the causality analysis of the ESTEREL V5 compiler explores this state space

composed of only L_{On} links. If two nodes belongs to the same control point, they have a commun father:

$$\widetilde{n_1} = \widetilde{n_2} \Rightarrow \exists n / n \xrightarrow{seq} n_1 \text{ and } n \xrightarrow{seq} n_2$$

- If two distinct control points have a sequential path to a commun node, then this node is the start of a new control point (control point splitting):

$$\widetilde{n_1} \neq \widetilde{n_2} \text{ and } n_1 \xrightarrow{\sigma On} n_3 \text{ and } n_2 \xrightarrow{\sigma On} n_3 \Rightarrow \widetilde{n_3} \neq \widetilde{n_1} \text{ and } \widetilde{n_3} \neq \widetilde{n_2}$$

- **P** is compatible with " $<''$. In other words, we construct a total order relation " $\ll''$ on **P** such that $n < n' \Rightarrow \widetilde{n} \ll \widetilde{n'}$

Control points must be as big as possible in order to minimize unnecessary tests in the embedded code: **P** can be seen as a *coarsest* partition of EG respecting the conditions described above.

Once **P** is constructed, it can be considered as a set of totally ordered control points $\mathbf{P}_i$, each point $\mathbf{P}_i$ being composed itsel of totally ordered nodes which can be directly mapped on our execution structure (fig.2). Yet, it does not correspond to any executable sequential code.

7 Code Generation

The last compiling step consists in transforming the SCFG composed of the subgraph $\mathbf{P}_i$ into sequential code, avoiding any code duplication and using a minimum number of guards. This step is not trivial but similar to the classical problem of linearizing parallel code which has been very well studied by J. Ferrante in [14] and probably close to the transformation of a PDG into a SCFG described in [10].

An original algorithm has been developed to translate our SCFG into a well structured and human-readable C-code (fig.4):

- After the EG has been partitioned in control points, $L_{PauseOn}$, L_{Off}, $L_{PauseOff}$ and $L_{Suspend}$ links can have as target only heads of control points and are transformed into action nodes modifying the status of these control points. Bit vectors are used so as to merge action nodes together.
- L_{On} are transformed either into an action of a node on another control point if source and target nodes do not belong to the same control point. If source and target nodes belong to the same control point, the nodes are just executed sequentially.
- Test nodes are transformed into `if...then...else...` constructs. As the code that must be put in the `then` and `else` branches is not necessarily found in sequence but can be interleaved with other nodes, boolean guards must sometime be inserted. The difficulty consists in generating a code with a minimum number of guards [14]. No `goto` are generated so that our compiler is also able to produce code for languages that do not have a `goto` construct like Java.

```
input S,I;
output O;
signal A,R in
   every S do
     await I;
     weak abort
       sustain R
     when immediate A;
     emit O;
   ||
     loop
       pause;pause;
       present R then emit A end
     end loop
   end every
end signal
```

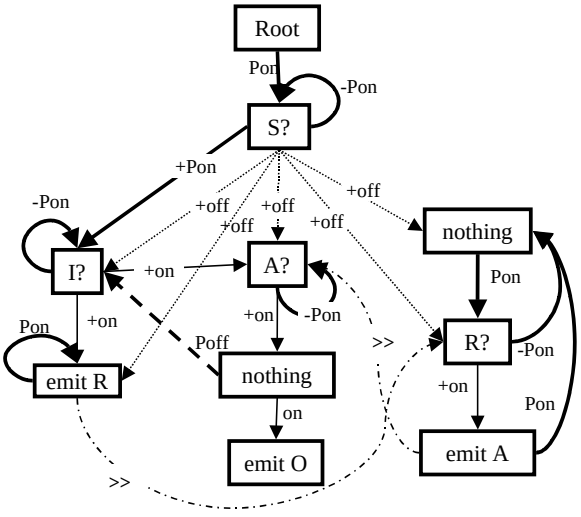

Fig. 3. a small example and its EG

8 Example

So as to help comparison with [10], we have chosen the same example (fig.3) that is is used in [10]. The behavior of this example is explained quite easily by the EG.

At the first instant, local signals A and R are reset and the `every` construct is activated for next instant. Then at each instant, if signal S is present, the whole body of the construct is turned off (the $L_{PauseOff}$ links) and the two parallel branches are activated for the next instant. The `weak abort` construct can be recognized by the node testing signal A and turning off the `sustain` construct (the node emitting signal R). The node testing R (the `present` statement) activates both the node emitting A and the `pause` starting the `loop` statement.

The sequentialization of the EG produces a C-code composed of 6 control points (fig.4): point #1 contains nodes `Root`, `Reset A` and `Reset R`, point #2 contains node `S?`, etc.

```
if (IsOn(#0))
   PauseOn(#1);
if (IsOn(#1)) {
   if (S) {
     Off( #2 #3 #4 #5 #6);
     PauseOn( #2 #5 );
   }
}
if (IsOn(#2)) {
   if (I) {
     On( #3 #6 );
     Off( #2 );
   }
}
```

```
if (IsOn(#3))
   EMIT(R);
if (IsOn(#4)) {
   if (R) {
     EMIT(A);
   }
   PauseOn( #5 );
}
if (IsOn(#5))
   PauseOn( #4 );
```

```
if (IsOn(#6)) {
   if (A) {
     EMIT(O);
     PauseOff( #3 );
     Off( #6 );
   }
}
Current[] &= 0x01001010;
Current[] |= Next[];
Next[] = 0;
Signal[] = 0;
```

Fig. 4. C-code produced by Saxo-rt compiler

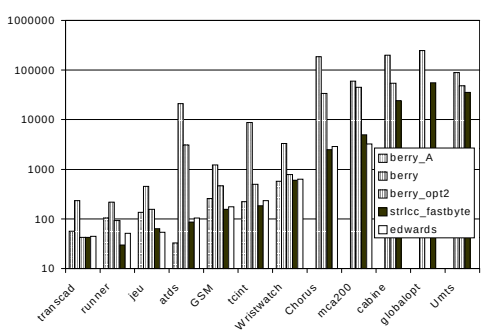

Fig. 5. Reaction times

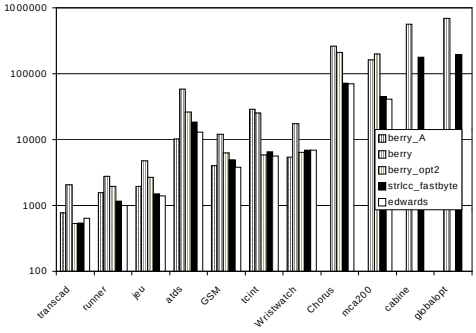

Fig. 6. Code+data sizes

9 Results and Comparative Performance Analysis

Experiments have been run to compare the performance of our compiler Saxo-rt to that from the V3 automata compiler, the V5 boolean-equations compiler [3] and S. Edwards's EC compiler [10]. Except for the largest examples for which it does not terminate, code produced by the boolean equations compiler has been optimized by the *blifopt* tool based on the SIS logic optimizer developed by Berkeley.

We used more or less the same benchmark that was used in [10]. Results obtained on 11 benchmarks are sorted from small to bigger industrial size examples. Fig.5 and fig.6 show the results obtained for these benchmarks on a sparc processor with SUN C-compiler in terms of reaction time and code size.

V3 compiler (automata compilation) does not terminates on very large examples, due to state space explosion. Saxo-rt obtains reaction time between 1.5 and 100 times better than boolean equations method (optimized with blifopt). On the largest examples (mca200 and Chorus), we obtain reaction time between 5 and 25 times better than Esterel V5 compiler while keeping code size about 4 times smaller. Comparison with EC compiler shows surprisingly similar results.

10 Conclusion

We have presented an original way of interpreting Esterel semantic into a event graph. This graph is transformed by the Saxo-rt compiler in a sequential control flow graph from which efficient sequential code is produced, enabling the use of Esterel on embedded systems with tight energy and memory constraints. On large examples, reaction speed obtained with Saxo-rt is usually from 5 to 20 times better than with boolean equations compilation. The Saxo-rt compiler accepts any acyclic Esterel programs and has been validated on several industrial prototypes. It has now reached an industrial quality and is being integrated into the CAD tool EsterelStudio, developed by Esterel Technologies.

Moreover, although we use a very different execution structure, the similar

results obtained by S. Edwards suggest that performance could be easily improved by combining the advantage of the two compiling methods.

References

[1] N. Halbwachs, *Synchronous programming of reactive systems*, Kluwer academic publication, 1993.

[2] G.Berry, G. Gonthier, *The* Esterel *Synchronous Programming Language: Design, Semantics, Implementation*, Science of Computer Programming, vol. 19-2, pp. 87-152, 1992.

[3] G. Berry, *The* Esterel *Primer*, Ecole des Mines, Centre de Mathématiques Appliquées, Sophia-Antipolis, *http://www.esterel.org*.

[4] G. Berry, *The Foundations of* Esterel, in Proof, Language and Interaction: Essays in Honour of Robin Milner. G. Plotkin, C.Stirling and M. Tofte, editors, MIT Press, Foundations of Computing Series, 2000

[5] G. Berry, *The Constructive Semantic of Pure Esterel* draft book available on *http://www.esterel.org*

[6] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, *The synchronous data flow programming language LUSTRE*, Proceedings of the IEEE, vol. 79, n°9, septembre 1991.

[7] G. Gonthier, *Sémantiques et modèles d'exécution des langages réactifs synchrones: application à* Esterel, Thèse de l'université de Paris-sud, centre d'Orsay, mars 1988.

[8] F. Mignard, *Compilation du langage* Esterel *en systèmes d'équations booléennes*, Thèse de l'Ecole des Mines de Paris, octobre 1994.

[9] S.A. Edwards, "*Compiling Esterel into Sequential Code*", 7^{th} International Workshop on Hardware/Software Co-Design, CODES'99, Roma, Italy, May 1999.

[10] S.A. Edwards, "*An Esterel Compiler for Large Control-Dominated Systems*", to appear in IEEE Transationcs on Computer-Aided Design of Integrated Circuits and Systems, 2002.

[11] V. Bertin, M. Poize, J. Pulou, "*Une nouvelle Méthode de Compilation pour le Langage Esterel*", Journées Thématiques Universités/Industries sur l'Adéquation Algorithme-Archietecture pour les Applications Temps-Réel Complexes, Lille, March 1999.

[12] D. Weil, V. Bertin, E. Closse, M. Poize, P. Venier, J. Pulou, *Efficient Compilation of* Esterel *for Real-Time Embedded Systems*, Proceeding of CASES'2000, pp. 2-8, San Jose, November 2000.

[13] E. Closse, M. Poize, J. Pulou, P. Venier, D. Weil, Saxo-rt*: Efficient Compilation of* Esterel *for Real-Time Embedded Systems*, Proceeding of IWACT'01, pp. 83-86, Bucarest, July 2001.

[14] J. Ferrante and M. Mace, "*On Linearizing Parallel Code*", in proceeding of POPL85, NewOrleans, January, 1985