# Fast Redundancy Elimination Using High-Level Structural Information from Esterel

Dumitru Potop-Butucaru

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *Fast Redundancy Elimination Using High-Level Structural Information from Esterel*

Dumitru Potop-Butucaru

**N° 4330**

November 2001

THÈME 1


ISSN 0249-6399 ISRN INRIA/RR--4330--FR+ENG

INRIA
SOPHIA ANTIPOLIS

# Fast Redundancy Elimination Using High-Level Structural Information from Esterel

Dumitru Potop-Butucaru*

Thème 1 — Réseaux et systèmes
Projet Tick

Rapport de recherche n° 4330 — November 2001 — 15 pages

**Abstract:** Esterel programs and SyncCharts hierarchical automata are compiled into flat sequential circuits. The current compiling process often generates too many latches and gates. We propose a compositional technique based on structural information that efficiently removes redundant latches and gates, without adding extra logic. The transformation works in linear time and gives good practical results. The simplified circuit can be used for simulation, verification, and optimisation.

**Key-words:** Esterel , synchronous language, latch oprimization, logic optimization, compositionality, hierarchical optimisation

* Ecole des Mines de Paris, CMA

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

# Utilisation d'informations structurelles de haut niveau pour la simplification de specifications Esterel

**Résumé :** La compilation de programmes Esterel, ainsi que celle de spécifications visuelles SyncCharts, produit des circuits séquentiels non structurés qui ont souvent beaucoup trop de registres et de portes logiques. Ce rapport propose une technique compositionnelle pour l'élimination d'éléments de circuit redondants en utilisant de l'information structurelle de haut niveau. Les algorithmes sont (sous-)linéaires dans la taille du circuit et les résultats sont bons. Le circuit simplifie peut être utilise a des fins de simulation, vérification, et optimisation.

**Mots-clés :** Esterel, langage synchrone, optimisation sequentielle, compositionalite, optimisation hierarchique

# 1 Introduction

We present a new latch and logic removal algorithm for the circuits generated by the Esterel compiler [3, 4, 5]. The algorithm also applies to graphical hierarchical automata descriptions such as SyncCharts [1].

The algorithm simplifies the state-encoding part of the circuit by removing redundant circuit components. The process is based on structural information directly available from the source Esterel program and already computed by the compiler.

The simplification algorithm is linear in the size of the state-encoding part of the circuit. Therefore, the simplification is quasi-instantaneous even for the largest examples available.

The simplification essentially consists of removing redundant latches and the logic that drives them. Decreasing the number of latches is useful for simulation, since the size of the compiled simulation code decreases, and for verification and optimisation, since BDD-based reachability algorithms are sensitive to the number of latches [9].

Using structural information from Esterel to improve the state representation is not a new idea. Over-approximation of the reachable state space is used to multiplex registers in [12, 14]. Efficient state encoding for C code generation is presented in [8]. Our technique is orthogonal to those and it may improve their results.

In the next section we present the structural information we use and we show how to extract it from the initial Esterel specification. Section 3 shows how to use this information to remove latches and gates. Section 4 presents benchmarks and Section 5 concludes.

# 2 Structural information

Esterel is a synchronous language. The execution of an Esterel program consists in a sequence of reactions driven by the clock ticks. At each clock tick, input signals are read and output signals are generated.

The language is imperative and the state is implicitly encoded in the program text. The kernel constructs are as follows:

- empty statement:

  `nothing`

- sequential and parallel composition, loop:

  $p$`;`$q$ $p$ `||` $q$ `loop` $p$ `end`

- signal emission, signal presence/absence test, local signal declaration:

  `emit S` `present S then` $p$ `else` $q$ `end`
  `signal S in` $p$ `end`

- exception raising and handling

  `exit T` `trap T in` $p$ `end`

- preemption

  `suspend` $p$ `when S` `abort` $p$ `when S`

- delay

  `pause`

The full language contains many more user-friendly statements that are macro-definitions over the kernel language. Therefore, we can reason on the kernel language only. For our purposes, we shall use the following (artificial) example:

```
await A;
await B;
abort
    await D
||
    present E then
        pause
    else
        await F
    end
when C
```

where the `await S` macro-statement expands into

```
trap T in
   loop
      pause;
      present S then exit T end
   end loop
end trap
```

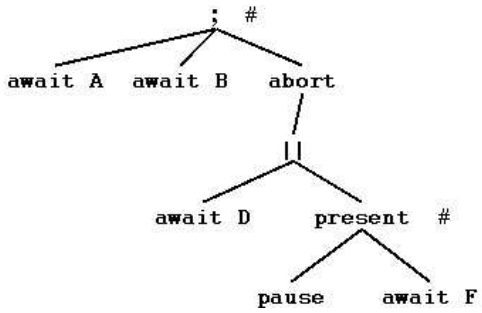
The program awaits the signal `A` from the environment, then awaits `B`. Then, it executes the branches of the parallel statement. The first parallel branch awaits `D` and terminates. When started, the second branch checks if signal `E` is present. If yes, it pauses for one clock tick and terminates. If not, it awaits signal `F` and terminates. The parallel statement instantly terminates when both branches are terminated. The `abort...when C` statement preempts the execution of the parallel statement when `C` occurs, whichever state the parallel is in.

## 2.1 Esterel program state

The only kernel statement that takes time is `pause`. When the control reaches a `pause` statement, it is retained there until the next clock tick. All the other kernel statements handle the control in a purely combinational way and do not generate control flow delays. Therefore, the state of the Esterel program between clock ticks is determined by the list

of the explicit or macro-generated `pause` statements where the control has paused. We say that these statements are *selected*.

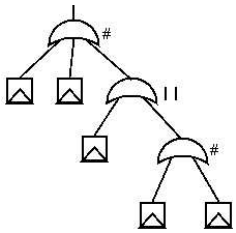
The notion of selection inductively extends to arbitrary statements. We say that a statement is selected if and only if it contains a selected `pause`. The parallel statement in our example is selected if control pauses in one of the delay statements "`await D`", `pause`, or "`await F`". The *selection status* is 1 if the statement is selected, 0 otherwise. It is easily computed using the syntactic tree of the statement, which is as follows for our example:




The selection status at a node is the disjunction of those of the children.

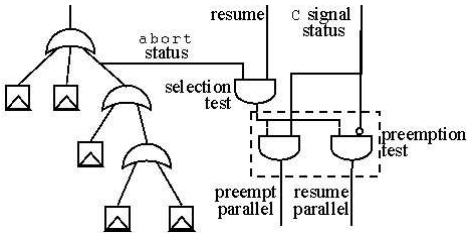
There is an important difference between parallel nodes and the other nodes: the statuses of parallel branches are independent, while the statuses of children are exclusive for all the other nodes. One cannot pause at the same time in the alternate branches of a test or in different components of a sequence. This is pictured by the `#` signs on the nodes with exclusive children.

In the circuit generated by the Esterel compiler, the selection structure is directly mapped into a tree of registers and *or*-gates:




A `pause` statement generates a register that delays the control flow. The other nodes generate *or*-gates (the one-input *or*-gate generated by the `abort` statement has been simplified away). Notice that *or*-gates associated with nodes having exclusive children can be replaced by *xor*-gates.

The selection wires guard the execution of the resumption code – preemption tests and state changes triggered by control resuming its flow from the registers where it paused. The figure below shows how the selection status of the `abort` statement drives the preemption test on signal `C`:



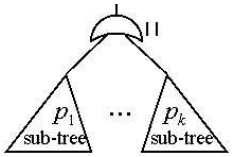

When we resume the `abort` statement, we first check if the statement is selected. If it is, we trigger the preemption test. If `C` is present, we preempt the parallel statement. Otherwise, we resume it.

## 2.2 Redundant selection nodes

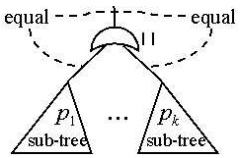
A commonly found structure inside Esterel programs is:

$$p = p_1||p_2||\ldots||p_k$$

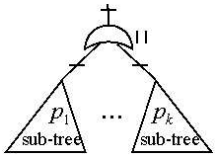
where the $p_i$'s are non-terminating statements (*e.g.* infinite loops). The selection tree of the statement $p$ has the form:




When we start $p$, the $p_i$'s start instantaneously. The statement $p$ stops only when it is globally preempted, for example by an enclosing `abort` statement. The $p_i$'s all stop at the same time. Therefore, the selection statuses of $p$ and of the $p_i$'s are always equal:

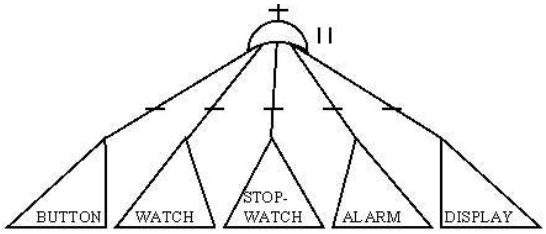



We use a traditional geometry notation to mark the equality relations in the selection tree: equal wires are decorated with the same sign:



We will say that a selection node always equal to its parallel parent is *redundant.*

We usually find equality relations at all levels in the selection tree. A good example is the digital wristwatch programmed in Esterel [2].

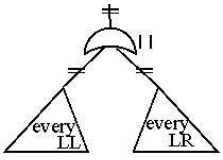



The program itself consists of 5 non-terminating modules running in parallel, as pictured above. For instance, the `BUTTON` module translates button presses into internal commands addressed to the other modules. Here, the root nodes of the 5 sub-trees are redundant.

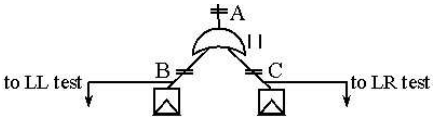
Nested within `BUTTON`, we find the following statement:

```
   every LL do
     emit NEXT_ALARM_TIME_POSITION_COMMAND
   end
||
   every LR do
     emit SET_ALARM_COMMAND
   end
```

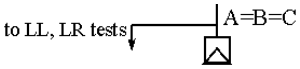
Every push on the lower-left `LL` wristwatch button determines the emission of an internal command signal. Pushing the lower-right `LR` button triggers another command. The two parallel branches never terminate, so the selection tree of the statement is:




Each `every` statement generates exactly one hardware register. Then, the selection tree is:




Since the values generated by the three gates are always equal, we can replace the entire structure by a single latch:



The general transformation will be described in Section 3. Notice it is not a simple retiming, since the selection wires fanout to tests as explained in Section 2.1.

To determine that a parallel branch is redundant we use the following criterion:

- *If a parallel branch cannot terminate, its selection status is always equal to the selection status of the entire parallel statement.*

The Esterel compiler computes the necessary information during the static analysis phase.

Not all parallel branches are redundant. By using our criterion we find parallel nodes that have both redundant and non-redundant children. This will allow us to further simplify the state representation in the next section.

In the following table, we compare the total number of parallel statement branches to the number of redundant branches for several examples. Except for the toy wristwatch, the examples come from industrial applications (avionics and circuit design).

| example | parallel branches | redundant branches |
|---|---|---|
| cabine | 891 | 776 |
| carburant | 362 | 354 |
| global | 1500 | 1048 |
| mmid | 63 | 61 |
| mmip | 33 | 33 |
| sequenceur | 86 | 84 |
| tcint | 39 | 38 |
| trappes | 122 | 121 |
| wristwatch | 34 | 33 |

The table shows that redundancy is very frequent.

# 3 Circuit transformations

We now simplify the selection tree by reducing the redundancy in the state representation. In doing this, we are subject to a strong constraint: most the selection statuses are used somewhere in the complete circuit and cannot be simply discarded.

We use 3 types of transformations:

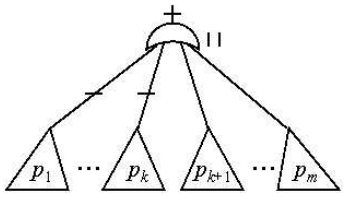
1. redundant fanin simplification
2. removal of gates that are equal to another gate
3. removal of a gate in a set of mutually exclusive gates, if we know their logical OR.

All the transformations can only decrease the number of gates and/or latches.

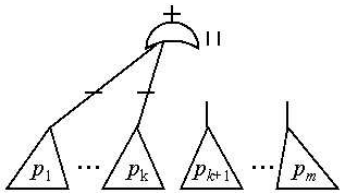
Also, they do not impose side-effects changes of the circuit outside the selection tree - they simply replace the selection tree by a simpler circuit providing the same interface.

## 3.1 Fanin simplification

Consider the following parallel selection node having both redundant and non-redundant children:

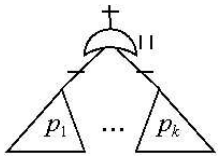



The function computed by the parallel node is the same as the one computed by each redundant child. Therefore, we can simplify the fanin of the parallel selection *or*-gate:



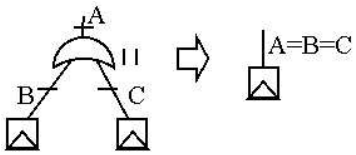

## 3.2 Equal nodes

We are now left with the following pattern:

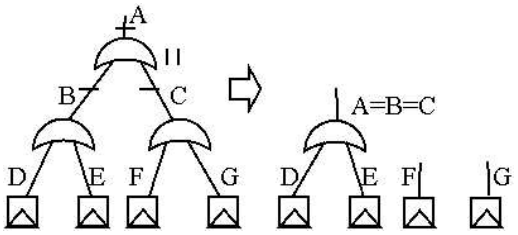



Since all the $p_i$'s selection wires are equal to that of $p$, it is enough to keep only one of them.

We illustrate the transformation with two small examples. We already saw the first one in section 2.2. Here, we erase one register and the root selection node:



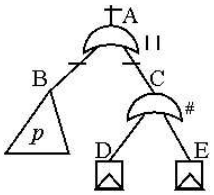

In the second example, we erase two *or*-nodes:



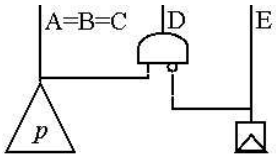

The choice of the node to keep is not obvious. For the benchmarks presented in Section 4, we kept unchanged the sub-tree of minimal depth. Other choices may be explored, *e.g.* choosing the sub-tree of minimal support.

### 3.3 Exclusive gates

A second gate-removing transformation erases one in a set of mutually exclusive selection nodes of which we know the disjunction. We apply this transformation on exclusive selection nodes equal to their parallel parent. Here is an example:




Here, we have $A = B = C$, while $D$ and $E$ are in exclusion and $D \vee E = C$. Thus, we can compute $D = B \wedge \neg E$. The simplified circuit is:




The transformation of Section 3.2 can actually be seen as a particular case of the exclusive gate transformation, by considering that a single node is a singleton set of mutually exclusive nodes. This simplifies the implementation.

## 4 Results

First, we compared the resulting circuits with those of produced using other re-encoding techniques.

In `remlatch` [11], determining redundant registers is based on the computation of the reachable state space (RSS) of the circuit. Other latch optimisation techniques are used in addition.

In `regtree` [12], the exclusions given by the selection tree are used to re-encode the state of the Esterel circuit. Basically, if two latch sets are in exclusion in the original circuit, they are multiplexed on only one of the two original sets by using one extra latch for the choice. The re-encoding process is applied in a bottom-up way.

As far as our techniques are concerned, we applied two simplification procedures:

**Procedure 1** removes redundant fanin and then applies the equal nodes removal transformation in a bottom-up way.

**Procedure 2** removes redundant fanin and then applies the transformation of section 3.3 in a bottom-up way.

The number of latches in the initial and simplified circuits is presented in the following table:

| example | init. | proc.1 | proc.2 | remlatch | regtree |
|---|---|---|---|---|---|
| cabine | 910 | 703 | 450 | – | 896 |
| carburant | 465 | 378 | 195 | – | 412 |
| global | 1359 | 1110 | 893 | – | – |
| mmid | 110 | 90 | 68 | 72 | 81 |
| mmip | 45 | 35 | 24 | 18 | 38 |
| sequencer | 154 | 130 | 100 | – | 118 |
| tcint | 82 | 68 | 59 | 47 | 35 |
| trappes | 157 | 123 | 81 | 43 | – |
| wristw. | 35 | 24 | 15 | 7 | 27 |

We also measured the variation in the number of literals (factored) in the simplified circuits, after a sweep by SIS [10]:

| example | init. | proc.1 | proc.2 | remlatch | regtree |
|---|---|---|---|---|---|
| cabine | 25377 | 19736 | 19642 | – | 24145 |
| carburant | 9006 | 6399 | 6577 | – | 6422 |
| mmid | 2986 | 2556 | 2657 | 2889 | 2839 |
| mmip | 1175 | 943 | 980 | 1099 | 1064 |
| sequencer | 4451 | 3776 | 3841 | – | 3880 |
| tcint | 1201 | 928 | 966 | 1200 | 1424 |
| trappes | 4637 | 3845 | 3779 | 4241 | – |
| wristw. | 961 | 759 | 794 | 837 | 1077 |

The RSS computation limits the scope of `remlatch` to circuits of small and medium size. On `sequencer`, it ran out of memory after 15 minutes of processing on a machine with 1Gbyte of memory. On `mmid`, it ran for more than 2 hours before giving the results. Its results in terms of latch removal are generally better than the results of our procedures. However, the logic is significantly larger. We also recall that both the procedures 1 and 2 are quasi-instantaneous even on the largest examples available.

An implementation limitation prevents `regtree` from being applied on circuits containing combinational cycles. This is the case for `global` and `trappes`. Preliminary results show that the results of `regtree` can be improved by using the redundancy property we identified.

The tradeoff between logic and latch removal in our procedures is visible if we compare the results of the procedures 1 and 2. The extra gates generated by Procedure 2 are the *and*-gates produced by the circuit transformation of section 3.3. For the examples `cabine` and `trappes` both the size of the logic and the number of latches decrease. In these cases, the sweep of logic that used to feed erased latches is more effective.

We also were interested in evaluating the impact of our transformations, seen as a cheap pre-processing phase, on some heavy optimization and verification algorithms. First, we applied the `blifopt` circuit area optimization script of SIS on several examples and on their simplified counterparts. The results showed no general significant difference in size, depth, number of registers, or optimization time.

However, `blifopt` can only be applied on small examples, since it relies on RSS computation. So, we did the same comparisons using the `combopt`[13] combinational logic optimization script. The results are good, as seen in the following table. The pairs give the number of registers and the number of literals (factored).

| example | original circuit | blifopt -area | proc.2+ combopt | combopt |
|---|---|---|---|---|
| cabine | 910/25377 | - | - | - |
| carburant | 465/9006 | - | 171/2457 | 465/5997 |
| mmid | 90/2556 | - | 68/1322 | 110/1900 |
| mmip | 45/1175 | 12/236 | 24/456 | 45/741 |
| sequenceur | 154/4451 | - | 100/1457 | 154/2421 |
| tcint | 82/1201 | 43/220 | 58/474 | 82/808 |
| wristwatch | 35/961 | 11/195 | 15/415 | 35/660 |

Similar results are obtained if we use `basicopt` as optimizer. `basicopt` is a sequential logic optimizer that is not as aggresive as `blifopt` and that takes less time to execute.

Next, we were interested in reachable state space (RSS) computation, using the TiGeR BDD library. The RSS size is not changed, but its encoding, yes, as BDDs tend to be smaller - having less variables.

Also, the simplifications we propose can transform certain sequential properties into combinational ones, as proved by experiments conducted at Esterel Technologies.

# 5 Conclusion

By using the hierarchical nature of the Esterel program state encoding we highly simplified the state representation in the automatically generated circuit.

The cost of the simplification is negligible. We only modified the sub-circuit which holds the state representation. The transformations we applied are simple enough for the correspondence between the initial and the simplified state representations to be straightforward. Also, they are compositional.

Our simplification procedure can be applied to other synchronous formalisms, and in particular to the graphical formalism SyncCharts [1]. The `carburant` example in our tables is actually an Esterel program automatically generated from a UML/SyncCharts specification [6].

We also intend to use the information presented in section 2.2 to generate a faster and smaller sequential (`C`) code from the Esterel programs, in the spirit of [8].

Other Esterel compiling techniques [7, 8] may take advantage of the selection tree properties we identified in section 2.2.

## Acknowledgements

I wish to thank Gérard Berry et Robert de Simone for taking the time of reading draft versions of this paper and making lots of constructive remarks.

## References

[1] Charles André. SyncCharts: A visual representation of reactive behaviors. RR 95-52, I3S, 1995.

[2] Gérard Berry. Programming a digital watch in Esterel v3.2. Available at http://www.esterel.org/.

[3] Gérard Berry. Esterel on hardware. *Philosophical Transactions of the Royal Society of London, Series A*, 19(2):87–152, 1992.

[4] Gérard Berry. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[5] Gérard Berry. The constructive semantics of pure Esterel. Draft book available at http://www.esterel.org/, July 1999.

[6] Gérard Berry, Amar Bouali, Xavier Fornari, Emmanuel Ledinot, Eric Nassor, and Robert de Simone. Esterel: a formal method applied to avionic software development. *Science of Computer Programming*, 36:5–25, 2000.

[7] V. Bertin, M. Poizé, and J. Poulou. Une nouvelle méthode de compilation pour le langage Esterel. In *Proceedings GRAISyHM-AAA*, Lille France, March 1999.

[8] Stephen Edwards. Compiling Esterel into sequential code. In *Proceedings CODES'99*, Rome, Italy, May 1999.

[9] Gary Hachtel and Fabio Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, 1996.

[10] Ellen Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, hamid Savoj, Paul Stephan, Robert Brayton, and Alberto Sagiovanni-Vincentelli. Sis: A system for sequential circut synthesis. Memorandum M92/41, UCB, ERL, 1992.

[11] Ellen Sentovich, Horia Toma, and Gérard Berry. Latch optimization in circuits generated from high-level descriptions. In *Proceedings of the International Conference on Computer-Aided Design(ICCAD)*, 1996.

[12] Ellen Sentovich, Horia Toma, and Gérard Berry. Efficient latch optimization using exclusive sets. In *Proceedings of the 34th Design Automation Conference (DAC)*, Anaheim, CA, USA, June 1997.

[13] The Esterel Team. The basicopt package documentation. Package available at http://www.esterel.org/, June 1998.

[14] Horia Toma. *Analyse constructive et optimisation séquentielle des circuits générés à partir du langage synchrone réactif Esterel.* PhD thesis, Ecole des Mines de Paris, September 1997.

# Contents

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399