ECOLE DES MINES DE PARIS

INRIA

Compiling ESTEREL circuits into finite states machines

Stage de DEA d'Informatique 1998/1999

BRES Yannis
Yannis.Bres@sophia.inria.fr

Responsable : Gérard Berry

UNIVERSITÉ NICE SOPHIA ANTIPOLIS

TABLE OF CONTENTS

Table of contents 3
Introduction 5
1 Statement of this work 5
2 ESTEREL : programs, circuits and automata 5
The ESTEREL language 5
The ESTEREL constructive semantics 5
3 Boolean circuits 6
4 Automata 7
Comparing circuits and automata 8
5 The ESTEREL compilation chain 8
Former automaton generation 11
1 Brief historical review 11
2 sscoc 11
New automaton generation 13
1 A different view of circuits 13
2 Basic algorithm 13
3 Algorithm refinements 15
Test delaying 15
Save–and–restore optimization 15
Undo block stacks allocation optimization 15
Using signal relations 15
Smart circuit evaluation 16
Signal partitioning 16
Signal sorting 17
DAG sharing through hash-cache 18
4 Performances 20
Conclusion 23
Bibliography 23
Résumé 24
Abstract 24

INTRODUCTION

1 Statement of this work

Our work consists in developing a new compiler from ESTEREL programs to automata. This compiler will make obsolete the current ESTEREL compiling modules that perform the same task but rely on old technology, and are not fully up-to-date with the ESTEREL semantics (they do not handle all cyclic ESTEREL programs).

We have developed a new cyclic circuit analysis algorithm based on the constructive interpretation algorithm already used in ESTEREL v5.21. We have implemented it in a new ESTEREL compiler module called `scoc`, and we have performed various optimizations of the generated automata.

Our module will be part of the next ESTEREL compiler release.

2 ESTEREL : programs, circuits and automata

The ESTEREL language

ESTEREL is an imperative, synchronous, signal-oriented programming language, developed in the Centre de Mathématiques Appliquées of the Ecole des Mines de Paris in a team lead by Gérard Berry. This language is designed to program real-time systems, communication protocols, software or hardware controllers or man-machine interface drivers. A brief historical of ESTEREL is available in [1] ; ESTEREL is fully described in [2].

ESTEREL is currently used in industry for embedded systems (by Dassault Aviation, Motorola, etc.), integrated circuit design (by Cadence, Synopsys, Intel, Texas Instruments, etc.), and by numerous communication protocols analysts, control theoreticians, etc.

The ESTEREL constructive semantics

Unlike too many programming languages, ESTEREL is provided with its complete mathematical semantics, call the constructive semantics. The semantics is defined by a set of logical rules, introduced in [1] and improved in [5]. For each statement of the language, a set of rewrite rules defines the behavior within a specified environment. The rule are of the form :

$$p \xrightarrow[E]{E',\, k} p'$$

where :

- *p* is the statement being rewritten into *p’* (the *derivative*) ; which is the statement to be executed on next step

- *E* is the input environment (present signals)
- *E'* is the resulting environment (emitted signals)
- *k* is a numerical statement completion code (0 = terminated, 1 = paused till next clock tick, *k*+2 = trap exit at level *k*)

The *circuit semantics* translates ESTEREL programs into Boolean circuits :

- *E* becomes a set of wires carrying present statuses
- *E'* is a set of wires encoding emitted signals
- *k* is a set of wires encoding completion codes ; these wires activate others parts of the circuit

3 Boolean circuits

Boolean circuits are directed graphs in which nodes are Boolean gates or registers, and edges are wires, or nets in electronics terminology. Boolean gates perform logic operations like or-ing or and-ing while registers delay values for one instant (clock cycle). Instants are defined by the pulses of a clock. Circuits are cyclic if their graph holds at least one cycle not located on a register.

Boolean gates are usually drawn using this convention :

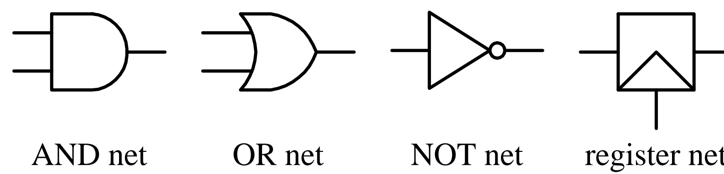

Figure 1 – Boolean gates drawing convention

Electric current flows from left to right, the initial value of the register is written under the register. Small circles on the inputs or outputs of gates indicate negation.

The following ESTEREL example emits continuously the O signal until I is present. It is translated into the circuit of Figure 2 :

```
abort
  sustain O
when I
```

Example Code 1 – aborted sustain

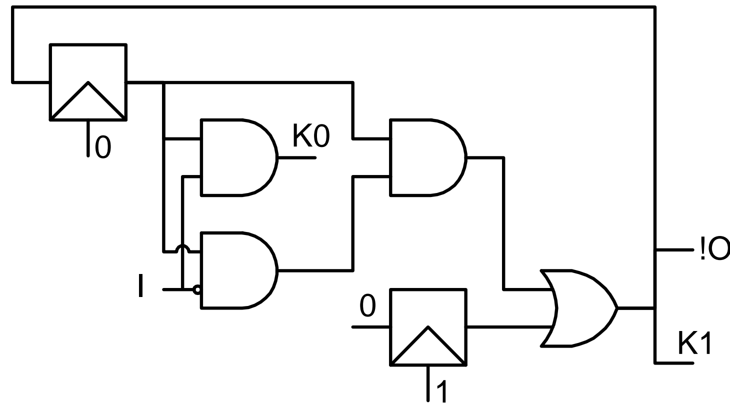

Figure 2 – Example Code 1 corresponding circuit

The register at the bottom of the circuit is called the *boot register*. It outputs 1 at the first instant, and 0 from then on. The circuit has only one input I, one output O, and the two completion code wires K0 (terminated) and K1 (paused), mutually exclusive. The top-left register stores whether the circuit did not terminate at previous instant.

4 Automata

Automata or Finite State Machines (FSM) are transition machines which can only be in a finite number of states. In this paper, we actually consider Mealy Machines where transitions read and emit external signals.

Returning to Example Code 1, we can draw its corresponding finite state machine :

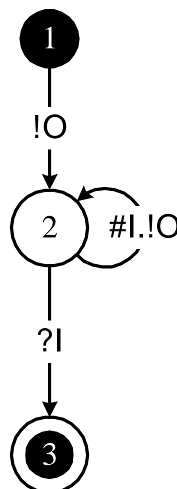

Figure 3 – Example Code 1 corresponding automaton

The automaton starts in initial state 1, drawn as a black circle. The O signal is emitted independently of the presence of I (the `abort` statement being not immediate), and the automaton transitions to state 2. From state 2, if I is absent (#I), we emit O (!O) and stay in state 2 ; otherwise, if I is present (?I), O is not emitted and the automaton falls in final state 3, drawn as a filled circle inside an empty one. State 3 is called *sink state*, because no transition gets out of it.

Comparing circuits and automata

Unlike circuits, automata are apparently very readable. This is particularly true if there is not much parallelism in the circuit. Otherwise, parallel statements create duplication of subautomata leading to size explosion. In the worst case, the automaton for a circuit can be exponentially bigger than the circuit.

As far as software implementation is concerned, the comparison between circuits and automata can be stated as follows :

- One can generate linear-size C code from the circuit, but all the gates have to be computed at each cycle.
- One can generate linear-size C code from the automaton, with no runtime overhead. A transition acts by testing inputs signals and generating outputs, without intermediate computations.
- However, the code size for the automaton can be exponentially bigger than the code size for the circuit.

In practice, automata are excellent until they get too big. Applications such as protocols or simple device control lead to very efficient automata that remain reasonable in size. This is why automata are still used by ESTEREL users. Large applications cannot be implemented by automata, and will not be handled here.

To be more precise, we can notice that to each statement of the language correspond a piece of circuit of almost fixed-size, while, for a circuit with m registers and n inputs/tests, we can expect a 2^m states automaton, each transitions being a tree having 2^n forking nodes ; in fact, not all states are reachable from initial one, and, parts of circuit being disconnected according to current state, not all inputs/tests are tested, leading to more simple automata.

5 The ESTEREL compilation chain

The ESTEREL compiler is C composed of several independent compiling modules, each performing compilation step. ESTEREL compiler usage is described in [3].

The compilation flow is detailed in Figure 4 :

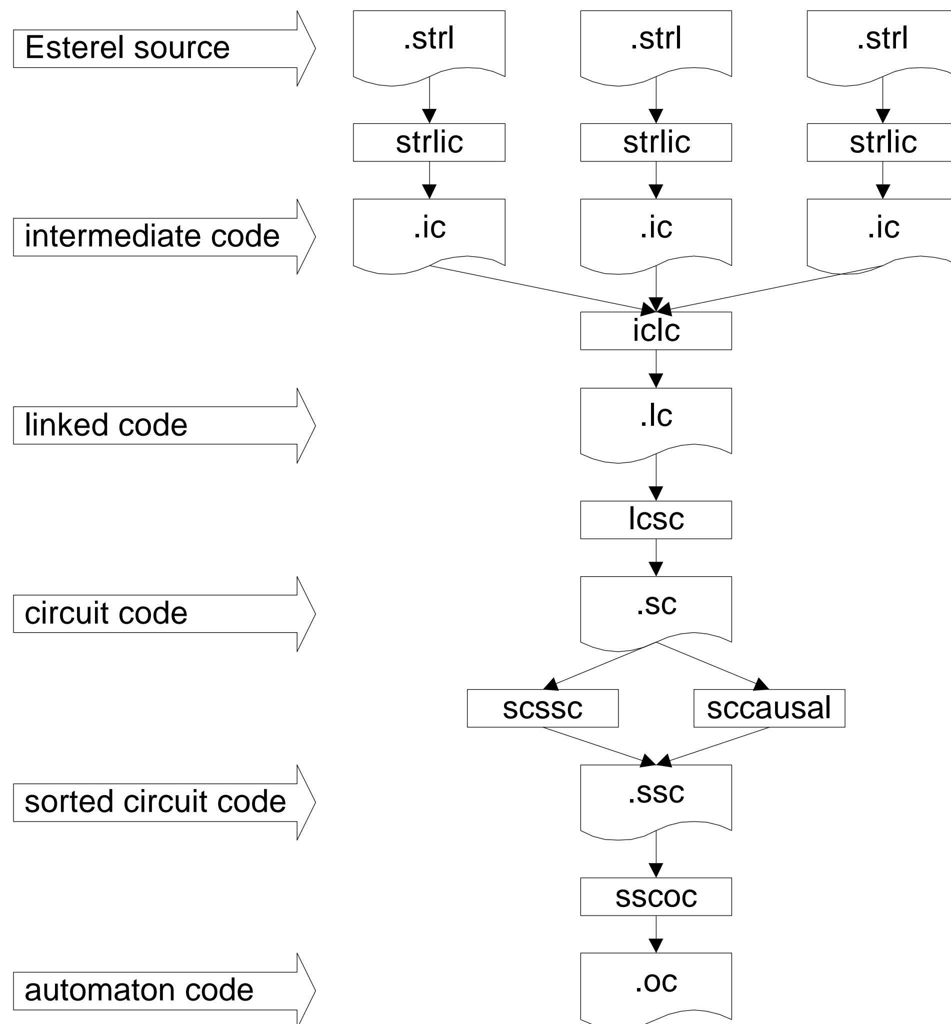

Figure 4 – ESTEREL v5.21 compilation flow

From ESTEREL source files, three steps are required to obtain circuits :

1) Compiling separately source files into intermediate code, with `strlic` ;
2) Linking intermediate code files, with `iclc` ;
3) Compiling linked files to circuits, with `lcsc`.

Then, as discussed later, two steps are currently required to compile circuits into automata :

4) Topologically sorting circuit equations, either :
   - directly with `scssc` for acyclic circuits
   - or with `sccausal` that checks constructiveness of cyclic circuits, and remove cycles.
5) Building the automaton, with `sscoc`.

The main goal of the present work is to suppress these two steps, particularly the `sccausal` pass, which can be very time consuming. We shall use a new direct algorithm, that will work indifferently with cyclic or acyclic circuits.

Another reason for discarding the former `sscoc` compiler module is that it is poorly documented and did not follow the evolution of the rest of the compiler.

FORMER AUTOMATON GENERATION

1 Brief historical review

The ESTEREL compiler has passed through five major steps, each one starting a new major version number :

- Till version 3, the output code was finite-state machines based. Approximate algorithms were able to compile a class of cyclic programs (see [4]).
- Industrial ESTEREL users started developing programs so big that automata kept exploding. A new technology ESTEREL v4 was developed to translate ESTEREL programs into circuits in a roughly linear way. Automata were still useful, and Frédéric Mignard developed a `sscoc` module to translate circuits into automata. Version 4 worked only with acyclic circuits.
- Current version 5 is based on the constructive semantics, which can handles cycles in a very neat way. A module called `sccausal` analyses cycles and returns an acyclic equivalent version of a constructive cyclic program, which makes it possible to use `sscoc` to build automata.

2 sscoc

The `sscoc` compiler was written by Frédéric Mignard during the second half-year of his Computer Science PhD, concurrently with an automata optimizer, embeded within `sscoc`.

`sccoc` takes circuits in `ssc5` format ([6]) as input, and output finite state machines in `oc5` format ([10]). The structure of the compiler is described in [8] ; underlaying research topics are stated in [9].

The `sscoc` former automata compiler viewed acyclic circuits as sorted Boolean equations and used a single-pass algorithm, to solve these equations in linear time. The algorithm works by elimination, substituting variables by results of previously solved equations.

By nature, this algorithm cannot work with cyclic circuits. Cycles have to be removed beforehand by the Binary Decision Diagrams (BDD) based `sccausal` module. However, `sccausal` is expensive in time and memory, and this path is impractical.

NEW AUTOMATON GENERATION

1 A different view of circuits

The new compiler views circuits as arbitrary graphs of Boolean gates, or *netlists*, possibly cyclic. The basic evaluation algorithm mimics electric current propagation in wires (*nets*) and gates.

Technically speaking, nets are implemented as objects that are waiting for messages from their predecessors, compute their value from those messages, and acquaint their successors with this value. Given an input vector to the circuit, elementary propagation steps are iterated until all nets have been evaluated :

- an OR gate can be set to 1 if any of its predecessors has been set to 1, or to 0 when all of its predecessors have been set to 0 ;
- dually, an AND gate can be set to 0 if any of its predecessors has been set to 0, or to 1 when all of its predecessors have been set to 1 ;
- a negation negates its input value whenever it is available.

Then, the automaton transitions can be built. If some nets remain unevaluated, the program is not constructive and it is rejected.

A directed graph of nets is built at parse time. The algorithm works on this graph without sorting it.

The algorithm converges in linear time with respect to the size of the graph : it cannot loop endlessly.

This new compiler, called `scoc`, takes circuits in `sc7` format ([7]) as input, and output finite state machines in `oc5` format ([10]).

2 Basic algorithm

The basic algorithm to compile circuits into finite states machine, which is explained in the Figure 5 flowchart, consists of two nested loops : the first one builds states, starting from initial state, until all discovered states have been analyzed ; the second one analyses an individual state and build transitions, by computing nets values as above for all possible input vectors.

For each state, input decisions build a tree that encodes the Mealy transitions set in an efficient way.

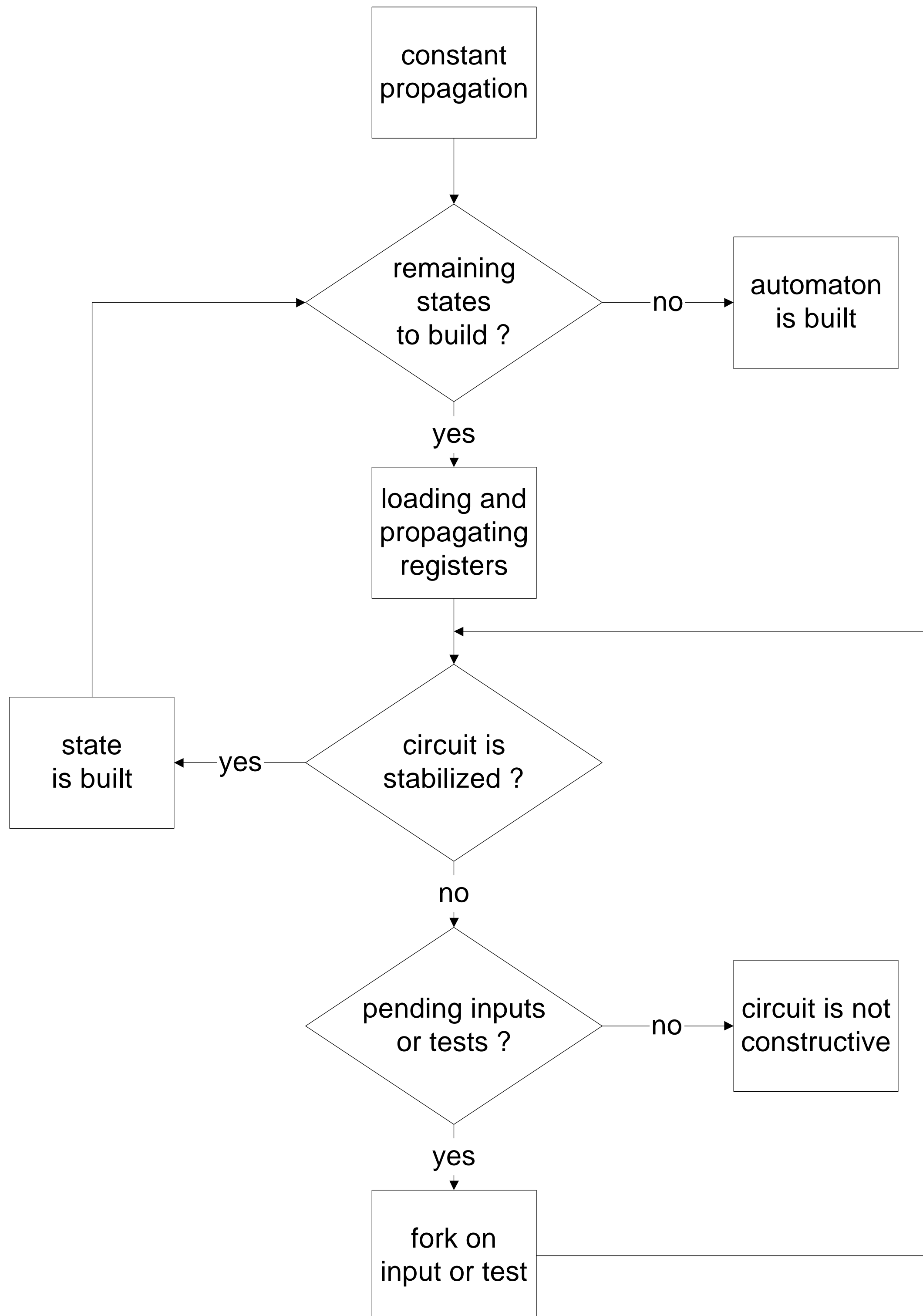

Figure 5 – Basic algorithm diagram

Nets can be of different types, therefore having different behaviors, for instance :

- Registers propagate their current value, which comes from the previous instant ; their computed value in the current instant will replace it for the next instant.
- Nets driving data actions, when evaluated to 1, store their action index in the current transition thread, then propagate their value ;
- …

These behavioral differences are handled through class specialization.

3 Algorithm refinements

The basic algorithm can be refined in many ways, either to improve compilation speed, or quality of generated automata.

Test delaying

When a test net is set to 1, forking does not occur immediately ; instead, the net is just appended to a list of pending test nets. When propagation cannot proceed, pending nets are extracted form the list, and fork occurs. This technique reduces redundancy between branches.

Save–and–restore optimization

Once a branch of the transition tree has been built, when backtracking to handle another inputs, the previous context must be restored.

We could save nets states in a big table, to make restoration easy. But, most of the times, transitions branches are short, forking points are very close to each other, and only a subset of nets are altered between two successive backtracking points. Storing in a table would be inefficient. Instead, we chose to store *undo data blocks* into undo block stacks, being themselves stored in a backtracking stack :

- Each time a branch is started, a new undo block stack is created and pushed on top of the backtracking stack.
- Each time a net (or some other data having to be saved and restored) is to be altered, it is pushed on top of the current undo block stack.
- Each time a branch building is over, the current undo block stack is flushed, saved data being restored, then the stack is popped.

This technique ensures minimal data transfers, unaltered nets being not unpatched by the saving/restoring process.

As this technique was implemented since first version, we can't estimate its impact on compilation speed, but it is undoubtedly worth it.

Undo block stacks allocation optimization

Default heap allocation is known as a very inefficient process when subsets of allocated and freed memory blocks interleaves in special ways, which currently occurs with our stacks.

As stacks of stacks can also be seen as a single stack. Instead of allocating undo blocks one at a time and linking them with pointers, we allocate them by chunks of hundred kilo-octets, linked to each other. "stacks" boundaries are (roughly) stored as a reference to a block and an index in the block where the stacks starts.

This modification also requires to turn all undo blocks from polymorphic objects to constant-size unions with a type tag. It yielded an average 200% compilation speed improvement.

Using signal relations

Input signal tests are the main cause of transition size explosion. Reducing the number of required input tests can dramatically impact compilation speed and automaton size.

To achieve this reduction, ESTEREL programmers can specify relations between input signals, which are of two kinds :

- Signal implications (⇒), where the presence of a master signal implies the presence of a slave one ;
- Signal exclusions (#), where the presence of a signal implies the absence of other ones.

Given the following example, the relations are stored in a directed graph, as seen in Figure 7:

```
3 ⇒ 2
2 ⇒ 1
1 # 4 # 5
```

Figure 6 – Signal relations example

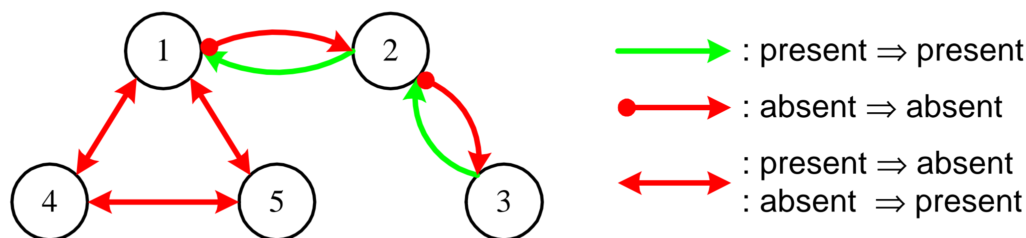

Figure 7 – Signal relation example graph

Storing the relations in a graph allows us to process them in linear time and in any order, which is required for forthcoming optimizations.

Smart circuit evaluation

Concerning resulting finite states machine, the only nets that are really important are registers (which determine next state), K0/K1 (which determine whether the automaton falls into sink state or just goes on), output signals (being emitted or not) and action nets (which are, among other uses, side-effects operating on values) ; all other nets are there only to compute the aforementioned ones.

Hence, we can stop circuit evaluation as soon as all the *required* nets are stabilized. As this would lead to acceptance of circuits that are non-constructive (where some nets can never be valued), this optimization remains optional.

Signal partitioning

An ESTEREL program often contains sequential code that lingers over several instants : the different parts of those sequences are therefore not activated at the same time (in mutual exclusion). Exhibiting the inputs/tests which are only used by those inactivated parts would lead to avoidable branching.

Determining the partition between inputs/tests that can impact on circuits and those which can't is achieved by selecting which input/test nets are reachable from required ones (registers, K0/K1, outputs, actions nets), using a depth-first walk in the circuit graph, which is linear in time.

Combined to smart evaluation, this optimization lead to great compilation speed improvement on several circuits (for instance, on TCINT example, compilation time felt from more than 3 hours to 3 or 4 seconds).

Signal sorting

We can even go further, and sort inputs upon the number of registers that are likely to be set. This appeared to be very effective on circuits where parts form a hierarchical system.

The bus arbiter, a token-ring based protocol, which was originally treated in Mc Millan's thesis [11] and adapted in ESTEREL in [12], is a very good example for this optimization (see Figure 8) : it consists of a set of cells requesting access to a data bus ; only one cell can access the bus at a time. To prevent from starvation and ensure fairness, cells are assigned a priority order which rotates through time : at first instant, the first cell would get the access before all other (if it was claiming it), the second one would get it –in the instant– if the first one didn't claim it, and the last one would only get the access –still in the instant– whether no other cells claimed it ; at second instant, the second cell would have the highest priority, and the first one the lowest, and so on…

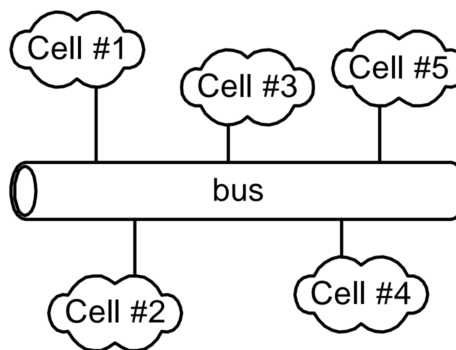

Figure 8 – Cells requesting access to a bus

In ESTEREL, cell *i* requesting an access emits a REQ*i* signal, and the cell being granted access receives an ACK*i* signal. The resulting automaton has as many states as there are cells, each states indicating which cell has the current highest priority. Since each request signal can be present or not independently, we can expect each transitions to be 2^n sized (n being the number of cells). However, as soon as a high priority cell emits a request, all other cells of lower priority cannot get access to the bus, so that it is unnecessary to test their request signal ; furthermore, requests from high priority cells should be tested first, since their presence lead immediately to a new state. The corresponding circuit behavior of this property is that, depending on current state, the request signal coming from the cell of highest priority will set all the registers (indicating the next state), and so on for the next signals…

Applying this input sorting algorithm arranges the transition trees to reduced ones, comb-shaped, therefore linear (see Figure 9) :

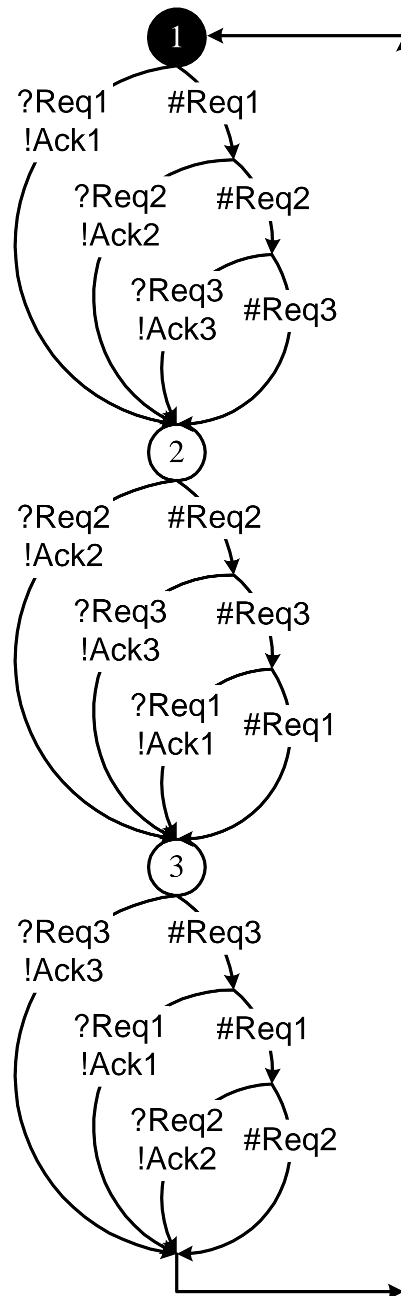

Figure 9 – Arbiter bus automaton (3 cells)

This algorithm is implemented as linear in time depth-first walks within the graph from registers.

DAG sharing through hash-cache

One of the drawbacks of finite states machines is that unrelated blocks are merged, or at least do not obviously appear. Also, some transition parts get duplicated in different places among automata, while they could be shared to save memory or at least to reduce output size.

These two goals, making unrelated blocks reappear in some ways and having output size reduced, are achieved by manipulating transitions no more as trees but as directed acyclic graphs (DAGs), all those dags being accessible through a hashtable : once a transition node is built, it is looked for a previously built twin (identical node with identical heirs), then destroyed –on successful search– to be replaced by this pre-existent node, or stored in the hashtable –otherwise.

By the way, this also allows to compare test branches in constant time (by pointer comparison), which may conduct to remove *–a posteriori–* tests which were avoidable.

Given the following example, a set of two unrelated blocks which are run concurrently, we can see on the standard automaton (Figure 10), that these unrelated blocks are not apparent, the second one being duplicated. On the opposite, we can see on the automaton with shared DAGs (Figure 11), that these unrelated blocks are still visible. Of course, this would have resulted in greater improvements with a higher number of unrelated blocks.

```
  present A
    emit A'
  end present
||
  present B
    emit B'
  end present
```

Example Code 2 – unrelated parallel blocks

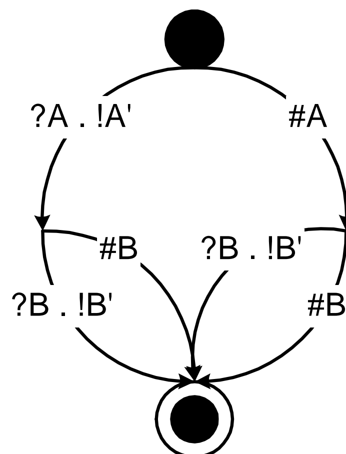

Figure 10 – Example Code 2 corresponding automaton

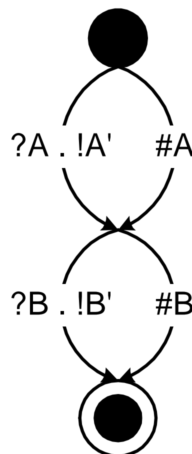

Figure 11 – Example Code 2 corresponding automaton, with shared DAGs

4 Performances

`scoc` has been tested and timed on the following circuits :

- Arbiter12, the bus arbiter mentionned about signal sorting refinement, with 12 cells ;
- ATDS-100-C2, an interface protocol between a VME bus and an ATDS bus ;
- Control, the Renault 5 dashboard ;
- TCINT, a Turbo Channel device interface ;
- Wristwatch, Gérard Berry's ESTEREL implementation of a wristwatch.

Table 1 summarizes the different metrics we can notice about these examples.

	nets	registers	inputs
Arbiter12	507	25	12
ATDS-100-C2	796	54	8
Control	1 383	66	15
TCINT	1 051	90	19
Wristwatch	830	35	6

Table 1 – Example circuits metrics

Keep in mind that :

- the number of registers exponentially drives the number of potential states ;
- the number of inputs (with the number of test nets, not stated) drives exponentially drives the size of transitions ;
- the number of nets drives the time taken for propagation algorithm.

Table 2 states the number of processor cycles taken for each example by `scssc` and `sscoc`, then the new `scoc`, with several options. As Arbiter12 is a cyclical circuit, `sccausal` was required instead of scssc ; `sccausal` being very time expensive, it would not have been fair to consider it.

This measures were done on an Intel Pentium II processor : with such a processor, running at 250MHz (which was even not the Intel bottom-of-the-range, at the time this paper was written), 250 000 000 cycles would only take 1 second.

	Arbiter12		ATDS-100-C2		Control		TCINT		Wristwatch	
SCSSC			12 655 140		16 914 682		16 914 682		15 528 094	
SSCOC	3 548 953 106		21 085 486		303 894 941		2 415 424 115		24 947 579	
SCSSC + SSCOC	3 548 953 106	100.00%	33 740 626	100.00%	320 809 623	100.00%	2 432 338 797	100.00%	40 475 673	100.00%
SCOC -simul -nocache	539 765 735	15.21%	192 575 710	570.75%	288 204 933	89.84%			30 572 158	75.53%
SCOC -simul	443 666 466	12.50%	123 921 908	367.28%	298 244 182	92.97%			31 095 716	76.83%
SCOC -simul -smarteval	296 605 212	8.36%	26 987 096	79.98%	292 358 295	91.13%	724 519 316	29.79%	30 789 945	76.07%
SCOC -simul -smarteval -sortinputs	13 877 086	0.39%	29 045 996	86.09%	294 008 621	91.65%	230 017 026	9.46%	30 828 684	76.17%
SCOC -simul -smarteval -sortinputs -dags	13 820 412	0.39%	28 994 608	85.93%	595 940 993	185.76%	235 028 491	9.66%	31 615 023	78.11%
SCOC -simul -sortinputs	219 868 773	6.20%	128 319 646	380.31%	298 911 819	93.17%			31 112 758	76.87%
SCOC -simul -sortinputs -dags	220 401 545	6.21%	128 274 224	380.18%	600 886 155	187.30%			31 875 884	78.75%

Table 2 – Performance measures (processor cycles)

From this table, we could compare `scoc` versus `scssc+sscoc`, and draw the Figure 12 graph :

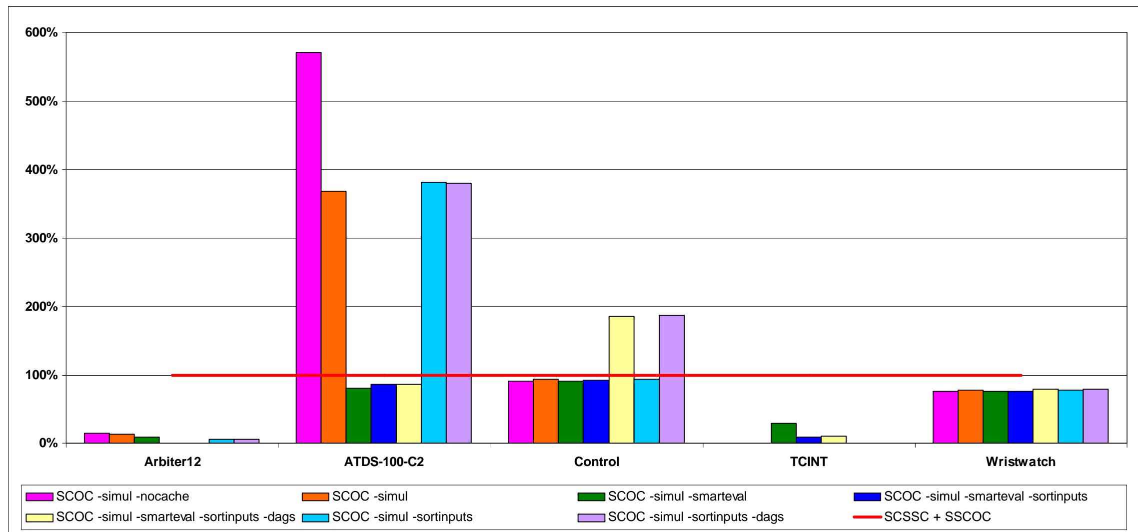

Figure 12 – Performance comparisons

From these comparisons, we can state miscellaneous observations (neither ordered nor exhaustive) :

- `scoc` can always be faster than `scssc+sscoc`, and sometimes more than 10 times, with appropriate options.
- Not all options but smart evaluation are always more efficient ; with full evaluation, compiling TCINT would take about 3 hours, and 3 seconds with.
- Altough transition nodes caching increase treatments, it is most of the time worth it, for it speeds memory allocation up.

Output quality was not considered, altough it could be interesting. Anyway, we can notice that sorting inputs (which dramatically increases compilation speed in the best case, and only causes a slight overhead in other cases) is supposed to have a good impact on transitions size.

CONCLUSION

In order to conclude, we could state that scoc is a rather efficient compiler, able to process cyclical circuits, and up-to-date with the ESTEREL language.

Our future researches will probably focus on looking for ways to turn scoc able to compile unlinked circuits separately, then link the resulting sub-automata together.

BIBLIOGRAPHY

[1] The Foundations of ESTEREL. G. Berry. CMA, Ecole des Mines and INRIA.

[2] The ESTEREL v5 Langage Primer, version 5.21 release 2.0. G. Berry. CMA, Ecole des Mines and INRIA. April 6, 1999.

[3] ESTEREL v5 documentation. G. Berry and the ESTEREL Team. CMA, Ecole des Mines and INRIA. April 6, 1999.

[4] The ESTEREL Synchronous Programming Langage : Design, Semantics, Implementation. G. Berry, G. Gonthier. CMA, Ecole des Mines and INRIA.

[5] The Constructive Semantics of Pure ESTEREL, Draft 2. G. Berry. CMA, Ecole des Mines and INRIA. 22 may, 1996.

[6] The ssc format sorted net table (release ssc5). The ESTEREL Team. CMA, Ecole des Mines and INRIA. March 26, 1998

[7] The sc format net table (version sc7), Draft 1. G. Berry. CMA, Ecole des Mines and INRIA. 21 juin 1998.

[8] Le processeur sscoc du compilateur ESTEREL v4_4x. F. Mignard. CMA, Ecole des Mines. 28 décembre 1994.

[9] Compilation du langage ESTEREL en systèmes d'équations booléennes. F. Mignard. Thèse de doctorat de l'Ecoles des Mines de Paris. 19 octobre 1994.

[10] The LUSTRE-ESTEREL portable format, version oc5. The ESTEREL Team. CMA, Ecole des Mines and INRIA. September 11, 1998.

[11] Symbolic Model Checking. Mc Millan. Kluwer Academic Publishers. 1993.

[12] Symbolic bisimulation minimisation. A. Bouali and R. de Simone. In Fourth Workshop on Computer-Aided Verification, volume 663 of LNCS, pages 96–108, Montreal, 1992. Springer-Verlag.

RESUME

Ce document présente les travaux effectués dans le cadre de mon stage de DEA d'Informatique, concernant la compilation de circuits booléens en automates.

Nous commencerons par introduire le cadre des langages temps réels, notamment ESTEREL, puis nous présenterons les circuits logiques et les automates d'états finis. Nous discuterons brièvement de l'ancienne technique de compilation des circuits en automates, basée sur des systèmes d'équations triées (donc non cycliques), puis nous expliciterons la nouvelle, ne nécessitant pas de tri et indifférente aux éventuels cycles. Nous terminerons par une évaluation comparative des performances du nouveau compilateur sur quelques exemples.

ABSTRACT

This document presents the work done during the second half-year of my Computer Science PhD, dealing with Boolean circuits compilation into finite states machines.

We will start by introducing the framework of real time languages, focusing on ESTEREL, then we will present Boolean circuits and finite states machines (FSM). We will briefly discuss of former circuits to FSM compilation technique, based upon sorted equation systems (therefore non cyclical), then we will explicit the new one, which does not require any more sorting and is unaffected by potential cycles. We will end by a comparative performance evaluation on several examples.