Circuit Generation for Verification of ESTEREL Programs

Alain Girault, Gérard Berry

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Circuit Generation for Verification of ESTEREL *Programs*

Alain Girault and Gérard Berry

N° 3582

1998

THÈME 4

ISSN 0249-6399

Circuit Generation for Verification of ESTEREL Programs

Alain Girault* and Gérard Berry†

Thème 4 — Simulation et optimisation
de systèmes complexes
Projet BIP

Rapport de recherche n°3582 — 1998 — 19 pages

Abstract: We propose in this paper a method that takes external Boolean variables into account for the verification of ESTEREL programs. The intermediate code that we use is a circuit that drives an action table. The circuit represents the control of the program, and the action table manipulates its external variables. The method transforms the actions into Boolean gates and registers acting on nets instead of variables. This involves encoding the input variables into the circuit and decoding output variables. This expansion method has been implemented within the `scdata` processor and can be used in conjunction with the ESTEREL compiler.

Key-words: Code generation, ESTEREL, formal verification, model checking, parallel languages, reactive systems, sequential circuits, synchronous programming.

(Résumé : tsvp)

This work has been partially supported by DASSAULT AVIATION.

* INRIA BIP project, ZIRST - 655, avenue de l'Europe 38330 Montbonnot Saint-Martin, France, tel: (33) 476 61 53 51, Alain.Girault@inrialpes.fr

† Ecole des Mines de Paris, Centre de Mathématiques Appliquées, INRIA meije project, 2004 Route des Lucioles, 06560 Sophia-Antipolis, France, tel: (33) 492 38 79 63, Gerard.Berry@sophia.inria.fr

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN (France)
Téléphone : 04 76 61 52 00 - International: +33 4 76 61 52 00
Télécopie : 04 76 61 52 52 - International: +33 4 76 61 52 52

Génération de Circuits pour la Vérification de Programmes ESTEREL

Résumé : Nous proposons dans cet article une méthode qui permet de prendre en compte les variables Booléennes dans la vérification de programmes ESTEREL. Le code intermédiaire que nous utilisons est un circuit qui contrôle une table d'actions. Le circuit code la structure de contrôle du programme, et la table des actions manipule les variables externes du programme. Notre méthode transforme les actions en portes logiques et registres qui agissent sur des fils au lieu des variables. Ceci suppose de coder les variables d'entrée dans le circuit et de décoder les variables de sortie. Cette méthode d'expansion a été mise en œuvre dans le processeur `scdata` qui peut être utilisé conjointement avec le compilateur ESTEREL.

Mots-clé : Génération de chemin de données, ESTEREL, vérification formelle, programmation parallèle, systèmes réactifs, circuits séquentiels, programmation synchrone.

1 Introduction

1.1 Reactive Systems

Reactive systems are computer systems that react continuously to their environment, at a speed imposed by the latter [18]. This class of systems contrasts on one hand with transformational systems, i.e., classical programs whose inputs are available at the beginning of their execution, and which deliver their outputs when terminating, and on the other hand with interactive systems, i.e., which react continuously to their environment, but at their own speed: for instance, operating systems. Among reactive systems are most of the industrial real-time systems: control, supervision and signal-processing systems. One of their most important features is *dependability*, since these systems are often critical ones. For instance, the consequences of a software error in an aircraft automatic pilot or in a nuclear plant controller are disastrous. Therefore these systems require rigorous design methods as well as formal verification.

1.2 The Synchronous Approach

Synchronous languages [14] were introduced in the 1980's to make the programming of reactive systems easier. They are based on the *synchrony hypothesis*: all parallel activities share the same discrete time scale. Each activity can then be dated on this scale; this has the following advantages:

- Temporal reasoning is made easier.
- Interleaving-based non-determinism disappears, which makes program debugging and verification easier.

The main languages based upon the synchrony hypothesis are STATECHARTS [18], ESTEREL [7], LUSTRE [15], SIGNAL [20], ARGOS [22], and SYNCCHARTS [1].

1.3 Formal Verification of Reactive Systems

Since reactive systems often concern critical applications, verification is a key issue. It has been shown in [24, 26] that the main goal concerning reactive systems is to verify a set of "safety" properties. These are properties which express that something bad will never happen.

The main formal verification technique is performed on a model of the program: hence it is called "model checking" [25, 9]. Basically, for a given safety property, it checks that this property is satisfied for each reachable state of the program model. The verification can be explicit, i.e. performed on a explicit model of the program, or implicit, i.e. performed on an implicit model of the program. The automaton encoding OC format (for "object code") is such an explicit model: the automaton transitions correspond to the system reactions to an input. OC is produced by the ESTEREL, LUSTRE, ARGOS, and SYNCCHARTS compilers. Several explicit verification tools have been designed for LUSTRE programs [26] as well as ESTEREL programs [19].

As an alternative to the explicit automaton format, the ESTEREL compiler can produce a Boolean sequential circuit written in the SC format (SC standing for "sequential code"), which can be considered

as an implicit automaton [2]. An SC program is a circuit driving an action table. The circuit represents the program's control, while the action table performs computations over the program's data. When the source program is pure ESTEREL (that is, without valued signals and variables), the SC code has no action table. This circuit representation yields both compact code and good execution time. In particular, it avoids the model size explosion problem often encountered in the explicit approach.

Formal verification of the program is also possible using the sequential circuit format [10, 8]. Several tools based on model-checking already exist that permit verification of sequential circuits: for example SMV [23], TIGER [11], XEVE [6], and VIS [29]. For instance, VIS checks that a given safety property written in the temporal logic CTL [9] is satisfied by the program. However, verification can only be performed on the program's control part, that is, it cannot take into account the actions performed by the circuit.

1.4 Extended Verification

We propose a method that extends formal verification to ESTEREL programs with Boolean variables. We call this *data expansion*. We have implemented the data expansion within the `scdata` processor: it takes as input an SC circuit and gives as result the expanded corresponding circuit, still under the SC format. The `scdata` processor can thus be used in conjunction with the other ESTEREL tools.

After a short presentation of constructive circuits in Section 2 and the SC format in Section 3, we present the method in Section 4. Then we present the steps in detail in Section 5. In Section 6 we illustrate the formal verification with an example. Finally, in Section 7, we show how to extend data expansion to enumerated types.

2 Constructive Circuits

If an ESTEREL program contains no cyclic instantaneous signal dependencies, then the circuit obtained by the translation has no combinational cycle. Since we also want to deal with dependency cycles in ESTEREL programs, we have to understand combinational cyclic circuits. To schematize, two semantics have been used in the ESTEREL compilers to produce sequential circuits: the Boolean semantics and the constructive semantics [27, 4, 5]. They differ on the analysis of combinational cycles, also known as *causality analysis*. The goal of the causality analysis is to be consistent with the electrical behavior of circuits. The Boolean semantics uses the classical Boolean calculus to solve equations. The constructive semantics propagates the known value of nets.

Consider the following cyclic program and the corresponding circuit:

```
x := x or not x
```

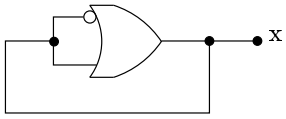

Figure 1: The `hamlet` cyclic circuit

Under the Boolean semantics, `x or not x` will be simplified and replaced by `1`, leading to the equivalent program `x := 1`. This is inconsistent with the electrical semantics which tells that the `hamlet` circuit has no stable behavior for some gate delays. On the other hand, under the constructive semantics, such a simplification is forbidden and the construct `x or not x` propagates a `1` if the net `x` is defined. In the `hamlet` circuit, since the value of `x` is not known a priori, the circuit has no behavior: we say that it is not constructive.

The ESTERELV5 compiler[1] implements this constructive semantics.

Now, handling data in finite domain is standard in SMV and VIS. Our goal is to provide the same facility for ESTEREL, considering only Boolean variables for simplicity. Compared to the standard approach, the difficulty lies in the fact that the control circuit can have combinational cycles. Since the data part can interfere with the constructive causality analysis, we must deal with data expansion beforehand.

3 The SC Format

Basically, an SC [3] program is a control circuit driving an external action table. Several tables describe the objects manipulated by the program (types, constants, variables, signals, and so on), and a net table describes the circuit itself. Each net has a number and is defined by an "access list" and an "expression":

- The access list indicates sequential dependencies: the value of any net in the access list must be known before the value of the net that bears the access list can be computed. Remember that a net does not compute an output but propagates a value as soon as it has enough inputs [27]. According to the constructive semantics (see Section 2), a gate controlled by an access net could be implemented with only regular gates. Indeed, the `c or not c` gate propagates `1` if and only if the net `c` is defined! Hence, the circuit of figure 2(a), where the link between the gate and its access net is represented by a doted line, is equivalent to 2(b), but contains less gates. When `c` is not defined, no matter what the output of the gate `1` is, it cannot propagate itself past the gates `3` and `4`. In other words, access nets are shortcuts.

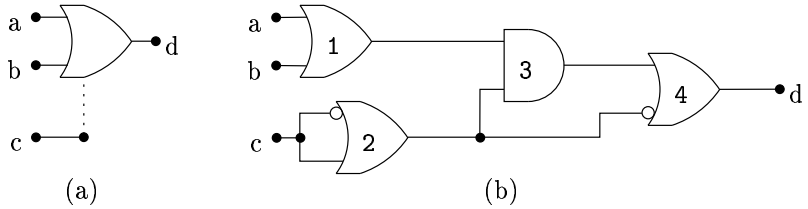

Figure 2: A gate with an access net and the equivalent circuit

[1] ESTEREL is available at `http://www.inria.fr/meije/esterel`.

- The expression indicates the inputs of a net. It is either a conjunction or a disjunction of nets or negated nets. Expressions cannot be nested. Two expressions are predefined: `$0` for `0` and `$1` for `1`.

We will not list all the different nets that exist, but just describe the most useful of them:

- A `standard` net defines a net with an expression and an access list. It builds basic Boolean expressions. For instance, the net `26` computes `24 and not 25`, but only after the net `4` has been computed to either `0` or `1`:

```
26: ELSE_5_K0_
access: (4 )
and: (24 !25)
```

- An `action` net drives an action defined in the action table. This action is called whenever the net bears the value `1`. An action is a variable assignment[2]. For instance, the net `18` launches the action `4` when the net `12` takes the value `1`, but only after the net `15` has been computed:

```
18: ACTION_1_OUT_K0_ act: 4
access: (15 )
12
```

- An `ift` net drives a test defined in the action table. This test action is called whenever the net takes the value `1`. The `ift` net is assigned the result value of the test. For instance, if the net `24` is at `1`, then the net `25` launches the test action `6` and sets the net `25` to the test result value:

```
25: THEN_5_K0_ ift: 6
24
```

- An `input` net corresponds to an input signal defined in the signal table. It behaves like an `ift` net. Its value is determined by calling the presence action associated with the input signal. That call returns the Boolean presence value of the signal, i.e., `1` if the input is present, and `0` otherwise, and sets the variable associated with the input signal. Note that an `input` net does not have an expression. For instance, the `input` net of `X` can be:

```
3: X__I_ in: 1 ift: 1
```

- An `output` net corresponds to an output signal defined in the signal table. It behaves like an `action` net by calling the `output` action whenever it bears the value `1`. For instance, the `output` net of `Y` can be:

```
9: Y__O_ out: 3 act: 3
```

[2] There are also external procedure calls.

- A `register` net defines a register, with a single `input` net and an initial value. For instance, here is a register with net `51` as input and `0`, i.e. `$0`, as initial value:

```
49: REG_19_
reg: $0 51
```

For instance, consider the following ESTEREL program:

```
module foo :
input c : boolean;
input x : boolean;
input y : boolean;
output o : boolean;
var a : boolean in
  a := false;
  loop
    if not ?c then
      a := (?x and ?y) or (not ?x and not ?y); % xnor
    else
      a := (?x and not ?y) or (not ?x and ?y); % xor
    end;
    emit o (a);
  each tick
end var
end module
```

Figure 3 shows the SC circuit obtained after compiling the program `foo`. `Action` nets are shown with the action on top of them. Action `a:=xor(x,y)` stands for `a:=(x or y) and (not x or not y)`. For the sake of readability, `register` nets are represented by registers, and `standard` nets are represented by logical gates. The control flows from the boot register to the last `output` net. After the `ift` net, it divides in two, a `then` branch and an `else` branch. Finally, the dotted lines represent the sequential dependencies indicated by access lists. They ensure that an input value is never used before the input is actually read by the `input` net.

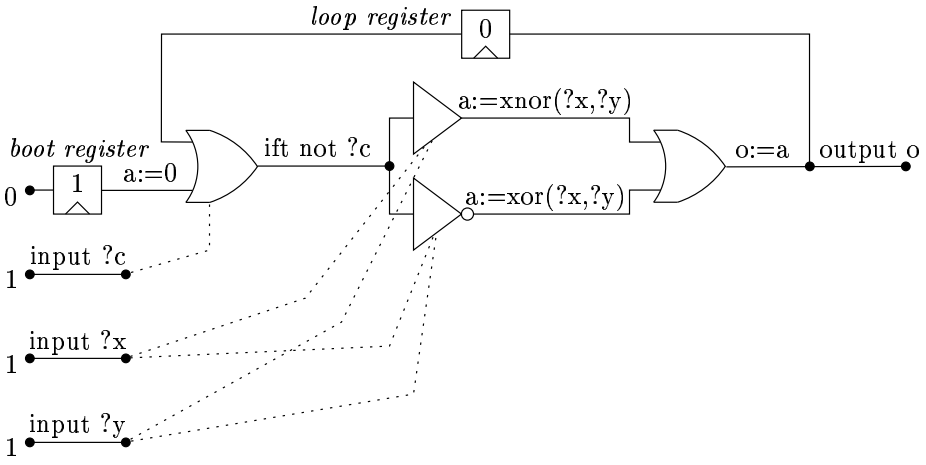

Figure 3: The `foo` SC circuit

The program `foo` has a sequential behavior. Remember that reactive systems react continuously to their environment. This is reflected by having a sequential circuit with registers. At each tick of its clock, the circuits reads the new values of its inputs, computes the new values of its internal variables, and emits the new values of its outputs.

4 Data Expansion's Principle

In the original SC circuit, wires carry the control while variables carry the values of signals. For the data-expansion, first we associate with each Boolean variable a net bearing its value, and then we implement actions by new logical gates.

In the SC circuit of figure 3, it is impossible to formally verify that, for instance, if `c` equals `1`, then the value carried by `o` must be equal to the `xnor` of `x` and `y`, because all actions are external to the circuit.

The data expansion produces one *data-path* for each program's Boolean variable. A data-path is a sequence of Boolean operators driven by control nets. It computes the successive values of one Boolean variable in the current instant, realizing the appropriate assignments according to the action table.

4.1 The Memory of the Circuit

Values must be carried from one instant to the next instant. This is done by having one register for each Boolean variable. The value of the variable at the beginning of the instant is the output of this register. The value at the end of the instant is stored in the register to become the initial value at the next instant.

4.2 The Control Flow

When expanding `action` nets, we must preserve the control flow. Indeed, in the expanded SC circuit, it is the control flow that will ensure that the correct values will be assigned to the variables. Thus, each `action` net in the source SC circuit must be transformed into a `standard` net in the final circuit. Each one of these new `standard` nets will drive a portion of the data-paths.

4.3 Choice of the Variables to be Expanded

Prior to data expansion, we must decide which variables are to be expanded. Only variables that are modified exclusively by Boolean actions can be expanded. Such a decision can easily be made through type checking. Note that external procedure calls are always considered to be non Boolean actions since it is impossible to check their code. Several optimizations are also made at this point. For instance, a variable that is neither used nor modified in the program will not be expanded.

5 Data Expansion's Successive Steps

5.1 Expansion of an Assignment

The expansion of an assignment is done with a multiplexor driven by the same net that drove the assignment in the original SC program. The two inputs of the multiplexor are the net bearing the value on the right side of the assignment, and the net bearing the previous value of the variable on the left side of the assignment. Thus the output net of the multiplexor bears the new value only if the driving net equals `1`.

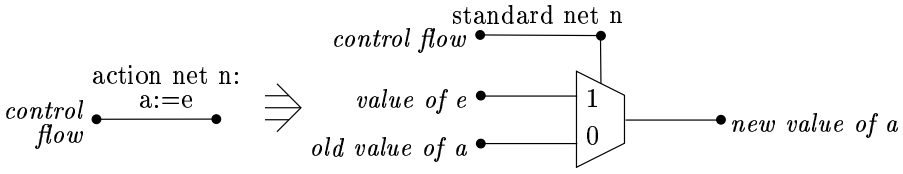

Figure 4: An assignment's data-path

Note that the `action` net `a:=e` has been turned into a `standard` net and now drives the multiplexor that implements the assignment. The multiplexor in figure 4 does not exist in SC. It is implemented with three `standard` nets in the following way (here `94` is the old `action` net, `95` carries the value of `e`, `55` carries the old value of `a`, and `98` carries the new value of `a`):

```
96: NETEXPR_96_
and: (94 95)
97: NETEXPR_97_
and: (!94 55)
98: NETEXPR_98_
or: (96 97)
```

In the sequel, we will draw multiplexors in place of the three `standard` nets.

5.2 Expansion of a Test

A test modifies the control flow of the program. An `ift` net evaluates its expression and takes the result of this evaluation as its new value. Thus, to expand an `ift` net, we expand the expression tested by the `ift`, and we make sure that the control is propagated only in the correct branch:

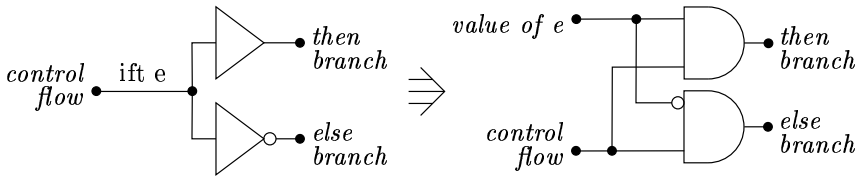

Figure 5: A test's data-path

5.3 Expansion of an Expression

The expansion of a Boolean expression is straightforward: the expression is just replaced by the corresponding Boolean gates. For instance:

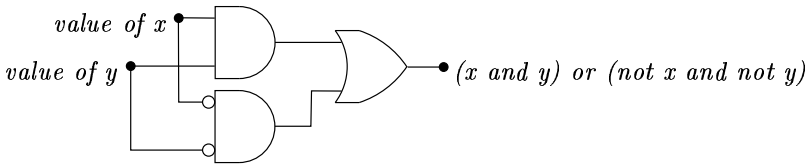

Figure 6: An expression's data-path

5.4 Expansion of an Input Variable

An `input` net is expanded by encoding the value of the input variable into the circuit. This is done by an `ift` net, where the expression tested is just the input variable. Indeed, a `ift` net evaluates its expression and takes the result of this evaluation as its new value. Moreover, as an input variable is never modified in the rest of the SC program, its data-path contains only the encoding part.

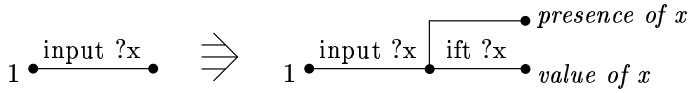

Figure 7: An input's data-path

5.5 Expansion of an Output Variable

An `output` net is expanded by decoding the value of the output variable. For the resulting SC program to be compatible with the original one, we must still use a net invoking an `output` action to emit the variable. But the SC variable bearing the value of the signal is not assigned to anymore since the assignment actions have been replaced by gates. Therefore we must assign it to the value carried by the data-path. This is done by two `action` nets, one to assign the output to `1`, and one to `0`. Finally we must make sure that the assignment is done before the `output` action is invoked. In other words, we must force the `output` net to wait for the two `action` nets. This is achieved with the `access` mechanism provided in SC (represented by the two dotted lines):

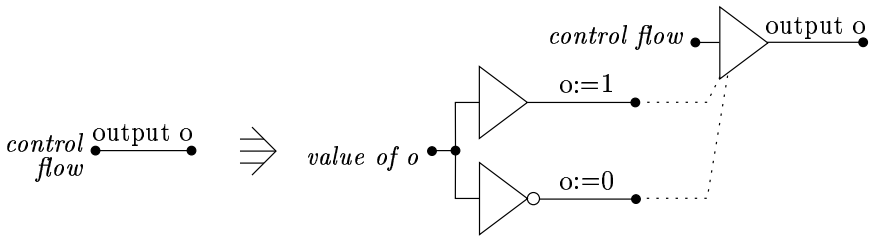

Figure 8: An output's data-path

5.6 Sorting the Actions

A data-path is a sequence of Boolean operators driven by the control flow. The control flow ensures that the actions implemented by the data-path are performed in the correct order. For instance, consider the following sequential portion of ESTEREL program:

```
x := true;
y := x;
x := false;
```

In the corresponding SC program, the control flow makes sure that the three assignments are computed in the correct order. After the data-expansion, we still want these actions to be computed is this order. It is essential because some variables may be assigned more than once, as it is the case for `x` in this example. To make sure that the sequential order will be preserved after the data-expansion, we have to expand the actions in the order of the control flow [21].

The control flow is directly induced by the net dependencies. Here we face the problem that the SC circuit might not be acyclic. An SC circuit is cyclic if it has an instantaneous cycle without any register. If this is the case, it has one or more non empty strongly connected component (SCC for short). If an SCC contains more than one `action` or `ift` nets, then we must decide in which order we have to expand them. A lot of work has been done on the analysis of cyclic circuits [21, 17, 27]. They have led to the definition of the admissible behavior of cyclic circuits, as well as precise algorithms for checking if a circuit has an admissible behavior. This is known as "causality analysis".

Here we will allow for a more rudimentary algorithm since we assume causality analysis has already been been performed by the ESTEREL compiler. To achieve this, we use an algorithm based on Tarjan's algorithm for sorting the SSCs of a circuit [28]. Sorting the SCCs indeed sorts the actions that must be expanded.

If an SCC contains two or more `ift` nets, it means that we cannot sort these tests. Since an `ift` action modifies the control flow of the circuit, there exists no admissible behavior for the circuit and we reject it by exiting.

If an SCC contains two or more `action` nets, it means that we cannot sort these actions. We then have to check that all these actions do not interfere one with each other:

- If they all commute, then we can expand them in any order.
- If among any two actions that do not commute, only one can be invoked in each instant, then we can expand them in any order.
- Otherwise, it means that the program is not causal and we reject it by exiting.

To check that two actions commute, we compare their respective sets of input and output variables. This analysis is performed by the ESTERELV5 compiler.

5.7 Final Result

Figure 9 shows the expanded circuit obtained with the `foo` circuit of figure 3. The original `foo.sc` circuit contains 52 nets and 2 registers. The resulting `foo.data.sc` circuit contains 101 nets and 4 registers. Both figures are of course more compact since only indispensable nets are drawn.

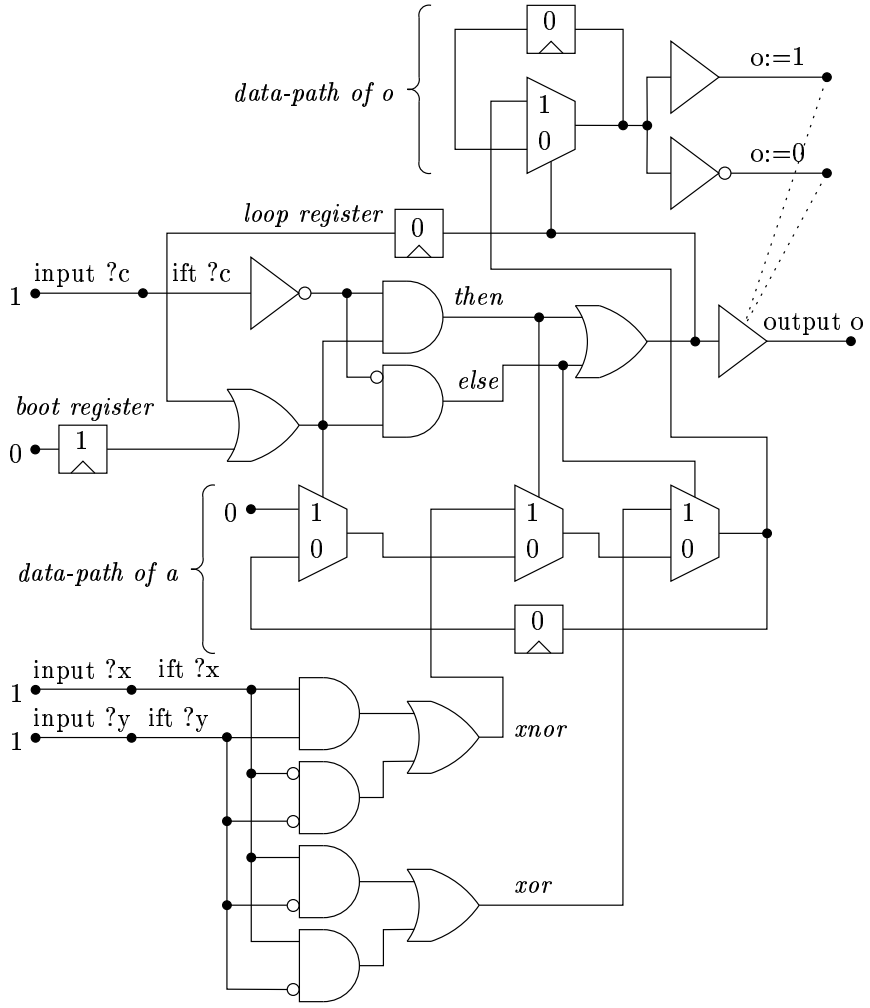

Figure 9: The expanded `foo` SC circuit

6 Formal Verification

In this Section we illustrate the formal verification method on the program `foo` of Section 3. We have used the ESTEREL`V5` compiler, and the VIS tool designed by the CAD group at UC Berkeley [29].

First we compile the program towards an SC circuit:

```
esterel -sc foo.strl
```

Second we expand its Boolean actions. The `scdata` processor takes an SC circuit as input and gives the expanded corresponding SC circuit as a result:

```
scdata < foo.sc > foo.data.sc
```

This produces the file `foo.data.sc`. Now VIS uses the BLIF input format [13] for describing circuits. An option of the ESTEREL compiler allows the translation into BLIF, and the `-Lblif:-soft` option makes sure that `action` nets are correctly translated [12]:

```
esterel -Lblif:-soft foo.data.sc
```

This produces the file `foo.data.blif`, which contains the description, known as "model" in BLIF, of our expanded circuit. To verify properties, we use the model-checking algorithm provided by VIS. This algorithms checks that a given temporal logic formula written in CTL [9] is satisfied by the BLIF circuit. Now the header of `foo.data.blif` is:

```
.model foo
.inputs \
    IfOutWire10_1_1 \
    IfOutWire11_3_5 \
    IfOutWire12_5_6 \
    c \
    x \
    y
.outputs \
    o \
    IfInWire10_1_1 \
    IfInWire11_3_5 \
    IfInWire12_5_6 \
    ROOT_RETURN_ \
    ActWire_13_6_25 \
    ActWire_14_7_26 \
    ROOT_HALTING_
```

We want to verify that the property:

```
o = if not c then xnor(x,y) else xor(x,y)
```

holds forever, which can be expressed by the following CTL formula:

```
AG ( (!(c=1)->(o=xnor(x,y))) * ((c=1)->(o=xor(x,y))) )
```

Here we face two technical problems:

- Because of the encoding for input values (Section 5.4), the values of inputs `c`, `x` and `y` are respectively carried by the wires `TEST_c_`, `TEST_x_` and `TEST_y_`. For similar reasons (Section 5.5), the value of output `o` is carried by the wire `ActWire13_6_25`. To observe these inputs, we must add their corresponding wires to the output list of the `foo` model (except `ActWire13_6_25` that exists already).
- In the VIS tool, temporal formulae can only instanciate registers outputs and local variables, i.e., no inputs. Although this restriction should disappear in a future version of VIS, it forces us to use the synchronous observer method [16]. It consists in computing separately our formula, and feeding the result into a register. Note that a BLIF file can contain several models, one for each circuit described, and circuits can be hierarchically nested.

We thus add to the expanded circuit a small `observer` circuit which computes the property we want to verify. The `observer` model must have the same input list as the `foo` model. Moreover, it must call the `foo` model as a sub-circuit:

```
.model observer
.inputs \
    IfOutWire10_1_1 \
    IfOutWire11_3_5 \
    IfOutWire12_5_6 \
    c \
    x \
    y
.subckt foo \
    IfOutWire10_1_1=IfOutWire10_1_1 \
    IfOutWire11_3_5=IfOutWire11_3_5 \
    IfOutWire12_5_6=IfOutWire12_5_6 \
    c=c \
    x=x \
    y=y \
    o=o \
    IfInWire10_1_1=IfInWire10_1_1 \
    IfInWire11_3_5=IfInWire11_3_5 \
    IfInWire12_5_6=IfInWire12_5_6 \
    ROOT_RETURN_=ROOT_RETURN_ \
    ActWire_13_6_25=ActWire_13_6_25 \
    ActWire_14_7_26=ActWire_14_7_26 \
    ROOT_HALTING_=ROOT_HALTING_ \
    TEST_c_=TEST_c_ \
    TEST_x_=TEST_x_ \
    TEST_y_=TEST_y_
.latch comp bad 0
```

```
.names bad TEST_c_ TEST_x_ TEST_y_ ActWire_13_6_25 comp
1---- 1
00110 1
00000 1
01100 1
01010 1
.end
```

As a consequence, the CTL formula becomes `AG(bad=0)`, and is easily checked by VIS:

```
vis> read_blif foo.data.blif
Warning: Some variables are unused in model observer.
vis> init_verify
vis> model_check foo.ctl
# MC: formula passed --- AG(bad=0)
```

7 Enumerated Types

Data expansion can be extended to enumerated types. For each enumerated type, we need to define a Boolean encoding of the possible values of the type. Then such a variable's data-path will have several nets instead of one (i.e., one net for each booelan of the encoding). Moreover, we have to redefine the encoding of inputs, the decoding of outputs, and to extend the translation of computations into logical gates.

For instance, consider the type `color` with the following possible values `red`, `blue`, and `white`. We encode these values with two Booleans in the following way:

`red`	0	0
`blue`	0	1
`white`	1	0

Then, figure 10 shows the portion of the expanded SC circuit corresponding to the assignment `x := blue`.

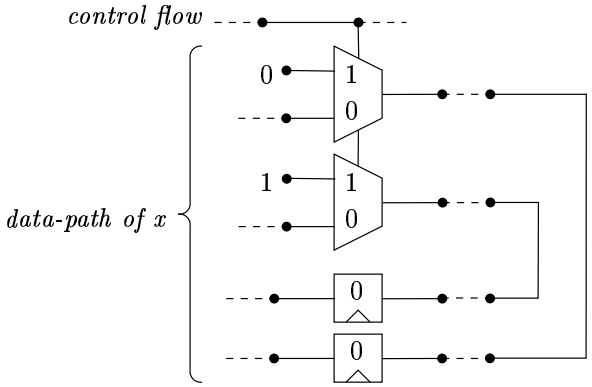

Figure 10: Data-path for enumerated types

8 Conclusion

To perform formal verification on an ESTEREL program, we compile it into a sequential circuit and then use hardware verification tools. However, this verification cannot take into account the external variables manipulated by the program.

We have presented a method that transforms computations over Boolean variables into portions of the sequential circuit. The transformation builds one data-path for each Boolean variable of the program. Each data-path remains driven by the control part of the program, which ensures that the computations are performed in the correct order. This data expansion allows formal verification on complete ESTEREL programs with Boolean variables. We have presented an example of formal verification performed with the VIS tool. We have also shown how it can be extended to enumerated types.

The algorithm for data expansion has been implemented within the `scdata` ESTEREL`V5` compiler module. It runs on SUN-Sparc, DEC-Alpha workstations, and LINUX. It contains 11,000 lines of C code, for a 300 Kb executable. Finally, for debugging purposes, it provides an option for visualizing the values of the expanded variables within the ESTEREL symbolic debugger XES.

Acknowledgment

Many thanks to Stephen Edwards and Xavier Fornari for carefully reading this article and improving it!

References

[1] C. André. Representation and analysis of reactive behaviors: A synchronous approach. In *CESA'96*, Lille, France, July 1996. IEEE-SMC.

[2] G. Berry. ESTEREL on hardware. *Philosophical Transaction Royal Society of London*, 339:87–104, 1992.

[3] G. Berry. *The SC Format Net Table.* CMA and INRIA, Sophia-Antipolis, France, October 1995. Unpublished Report; Available at `http://www.inrialpes.fr/bip/people/girault/Documentations/Sc6`.

[4] G. Berry. The constructive semantics of ESTEREL, 1998. Available at `http://www.inria.fr/meije/personnel/Gerard.Berry.html`.

[5] G. Berry. The foundations of ESTEREL. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.

[6] A. Bouali. XEVE: An ESTEREL verification environement. In *International Conference on Computer Aided Verification, CAV'98*, LNCS, Vancouver, Canada, June 1998. Springer-Verlag.

[7] F. Boussinot and R. de Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[8] J. Burch, E. Clarke, D. Long, K. McMillan, and D. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on CAD*, 13(4):401–424, April 1994.

[9] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2):244–263, April 1986.

[10] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*. Springer-Verlag, 1989.

[11] O. Coudert, J.-C. Madre, and H. Touati. TIGER *Version 1.0 User Guide.* DEC PRL, January 1994.

[12] X. Fornari. *Optimisation du Contrôle et Implantation en Circuits de Programmes* ESTEREL. PhD Thesis, Ecole des Mines de Paris, Paris, France, March 1995.

[13] The SIS group. *Berkeley Logic Interchange Format (*BLIF*).* UC Berkeley, Berkeley, CA, /july/ 1992.

[14] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Pub., 1993.

[15] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[16] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *International Conference on Algebraic Methodology and Software Technology, AMAST'93*, Twente, NL, June 1993. Springer-Verlag.

[17] N. Halbwachs and F. Maraninchi. On the symbolic analysis of combinational loops in circuits and synchronous programs. In *Euromicro'95*, Como, Italy, September 1995.

[18] D. Harel and A. Pnueli. On the development of reactive systems. In *Logic and Models of Concurrent Systems, NATO*. Springer-Verlag, 1985.

[19] L. J. Jagadeesan, C. Puchol, and J. E. Von Olnhausen. Safety property verification of ESTEREL programs and applications to telecommunications software. In P. Wolper, editor, *International Conference on Computer-Aided Verification, CAV'95*, volume 939 of *LNCS*, pages 127–140, Liege, Belgium, July 1995. Springer-Verlag.

[20] P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMaire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[21] S. Malik. Analysis of cyclic combinational circuits. In *ICCAD'93*, Santa Clara, CA, November 1993. IEEE Computer Society Press.

[22] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In W.R. Cleaveland, editor, *Third International Conference on Concurrency Theory, CONCUR'92*, volume 630 of *LNCS*, pages 550–564, Stony Brook, USA, August 1992. Springer-Verlag.

[23] K. McMillan. *Symbolic Model Checking.* Kluwer Academic Publisher, Boston, MA, 1993.

[24] A. Pnueli. How vital is liveness? Verifying timing properties of reactive and hybrid systems. In W.R. Cleaveland, editor, *Third International Conference on Concurrency Theory, CONCUR'92*, number 630 in LNCS, Stony Brook, USA, August 1992. Springer-Verlag.

[25] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, volume 137 of *LNCS*, pages 337–351. Springer-Verlag, April 1982.

[26] C. Ratel, N. Halbwachs, and P. Raymond. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Transactions on Software Engineering*, 18(9):785–793, September 1992.

[27] T. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *European Design and Test Conference*, pages 328–333, Paris, France, March 1996.

[28] R.E. Tarjan. Depth first search and linear graph algorithms. *SIAM J. Comput.*, 2(1), 1972.

[29] The VIS group. VIS: A system for verification and synthesis. In *International Conference on Computer-Aided Verification, CAV'96*, LNCS, New Brunswick, NJ, August 1996. Springer-Verlag.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

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