XEVE : an ESTEREL Verification Environment : (Version v1_3)

Amar Bouali

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

XEVE: *an* ESTEREL *Verification Environment (Version v1_3)*

Amar Bouali

N° 0214

Décembre 1997

THÈME 1

ISSN 0249-0803

XEVE: an ESTEREL Verification Environment (Version v1_3)

Amar Bouali

Thème 1 — Réseaux et systèmes
Projet Meije

Rapport technique n° 0214 — Décembre 1997 — 23 pages

Abstract: XEVE is a verification environment for ESTEREL programs modeled as Finite State Machines (FSMs) with a user-friendly graphical interface. The ESTEREL compiler translates a program into a system of boolean equations with latch that defines a FSM implicitly. XEVE works on these *implicitly* defined FSMs. It is based on the TIGER library which provides efficient data structures and algorithms to manipulate FSMs symbolically using BDDs. It takes as input the set of boolean equations of a FSM described in the BLIF format (*Berkeley Logical Interchange Format*). XEVE provides two main verification functions.

The first function is the FSM state minimization using a notion of *bisimulation* equivalence that relates states indistinguishable when exploring the FSM graph from them. The minimization is made modulo a set of input/output signals declared as hidden. Minimized FSMs are generated explicitly in a textual format called FC2 that can be loaded in the tool ATG for graphical exploration.

The other verification function is the checking of the status of output signals: one can verify if an output is possibly emitted or not. This checking can be made modulo the fixing of input signals to some value (0 for absent or 1 for present). When an output is found possibly emitted or not, a minimal execution trace leading to a state emitting or not the signal is saved in the CSIMUL format. The sequence can played within XES the ESTEREL graphical simulator.

Key-words: Automatic verification of synchronous reactive systems, Esterel, Symbolic bisimulation minimization, Verification by observers, Finite State Machines (FSM), Binary Decision Diagrams (BDD).

The realization of XEVE has been partially supported by the CTI CNET n° 951B182 "Vérification de systèmes synchrones".

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 93 65 77 77 - International : +33 4 93 65 77 77 — Fax : 04 93 65 77 65 - International : +33 4 93 65 77 65
à partir du 05/01/1998
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 — Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

XEVE: un environnement de vérification pour ESTEREL (Version v1_3)

Résumé : XEVE est un environnement de vérification de programmes ESTEREL modélisés par des machines d'états finis (FSMs) comprenant un interface graphique conviviale. Le compilateur ESTEREL traduit un programme en un système d'équations booléennes avec des registres définissant implicitement une FSM. XEVE opère sur ces FSMs définies implicitement. Il est basé sur la librairie TIGER qui fournit des structures de données et des algorithmes efficaces pour manipuler les FSMs en utilisant les diagrammes de décision binaire (BDDs). Il prend en entrée un système d'équations booléennes relatif à une FSM, décrit le format BLIF (*Berkeley Logical Interchange Format*). XEVE offre principalement deux fonctions de vérification.

La première est la minimisation de FSM au moyen d'une notion d'équivalence de *bisimulation* qui lient les états que l'on ne peut distinguer lorsqu'on parcours le graphe à partir d'eux. La minimisation est réalisée modulo un ensemble de signaux d'entrée/sortie déclaré non visibles. Les FSMs minimisées sont générées explicitement dans un format textuel appelé FC2, qui peut être chargé dans l'outil ATG pour une exploration graphique du graphe de la FSM.

L'autre fonction de vérification est l'analyse du statut de signaux de sortie : on peut vérifier si un signal de sortie peut être émis ou non. Cette analyse peut être faite modulo un ensemble de signaux d'entrée dont la valeur est fixée a priori (0 pour absent ou 1 pour présent). Quand un signal de sortie est trouvé comme étant possiblement émis ou non, une séquence d'exécution minimale menant à l'état où cette sortie est trouvé comme tel, est extraite et sauvée dans le format CSIMUL. La séquence peut être jouée dans XES, le débogueur/simulateur graphique d'ESTEREL.

Mots-clés : Vérification automatique de systèmes réactifs synchrones, Esterel, Minimisation par bisimulation symbolique, Vérification par observateurs, Machines d'états finis (FSM), Diagrammes de décision binaire (BDD).

Contents

1 Introduction **3**

2 The Verification Techniques **4**
2.1 The FSM Minimization 4
2.2 The Output Signals Status Checking 5

3 The Graphical Interface **6**
3.1 Inputting a FSM 6
3.2 Selecting a Verification Function 8
3.3 Selecting the Signals 8
3.3.1 Colors for FSM Minimization 9
3.3.2 Colors for Output Status Checking 9
3.4 Applying the FSM Minimization 9
3.5 Applying the Output Signals Status Checking 10
3.6 Options 11
3.7 Getting Help 12

4 Interfaced Tools **13**
4.1 ATG: Drawing Minimized FSMs 13
4.2 XES: Simulating Execution Sequences 13

5 A Tutorial Example **14**
5.1 The Example: a Small Bus Arbiter 14
5.2 Generation of the BLIF File 16
5.3 Staring up a XEVE Session 16
5.4 The FSM Minimization and Visualization 16
5.5 Checking Output Status and Verifying Properties 17

1 Introduction

This paper is a technical presentation of XEVE, an graphical environment for ESTEREL programs. We assume the reader is familiar with synchronous reactive programming, in particular with the ESTEREL language, its compiler and its software environment. Though we recall its main characteristics, we suggest the reader to refer to [2, 1] for a deeper presentation of the language and its compilation environment.

ESTEREL is a language designed for the programming of synchronous reactive systems (SRSs). Programs are made of processes with an input/output signals interface running in parallel and synchronously. Communication is modeled by signal broadcasting (no rendezvous mechanism). The language compiles into two main formalisms according to its mathematical semantics: Finite State Machines (FSMs) or Boolean Circuits.

The ESTEREL compiler produces FSMs either explicitly, where states and transitions are enumerated, or implicitly as a set of boolean equations with latches (or sequential digital circuit). The equations are generated in the BLIF (*Berkeley Logical Interchange Format* [6]) format which provides intermediate description at the level of boolean equations and in which both output signal and next latch values are expressed as propositional formulae on input signals and current latch values. This format is the input format of several academical and commercial synthesis and verification tools for sequential circuits.

XEVE takes as input FSMs described in BLIF. It is built above the TIGER library and thus uses the library data structures and efficient symbolic state space construction algorithm by means of *Binary Decision Diagrams* (BDDs).

This document is organized as follow: section 2 presents briefly the verification techniques used and provided by XEVE; section 3 exposes the graphical interface and how to use it; section 4 shortly gives the list of interfaced tools; section 5 treats a complete example.

2 The Verification Techniques

The internal representation of an FSM is roughly speaking made of a BDD for the reachable state space and a transition function partionned along the individual latches: the transition is actually a vector of boolean function, each being the boolean function defining a single latch.

XEVE provides two verification functions:

1. The minimization of the number of states of a FSM with respect to an equivalence called bisimulation. The minimization is made modulo input and output signal hiding.

2. The other is the checking of the emission status of output signals. The checking is made on the selected output signals and modulo fixed or hidden input signals.

The functions were initially implemented in two processors, BLIFFC2 and CHECKBLIF, respectively. These processors and XEVE form now the same program, but they can still be called separately. Both functions require the computation of the reachable state space of the FSM. This one is computed symbolically and represented as a BDD.

2.1 The FSM Minimization

We use an equivalence notion called *symbolic bisimulation*, [9, 5]. Roughly, this equivalence relates states that are not distinguishable when exploring the FSM graph. Reduction of FSMs consists in collapsing equivalent states and building the underlying quotient. This relation can be widened by hiding unrelevant signals with respect to the observation one wants to make, which may introduce nondeterministic behavior if some inputs are hidden. Nonetheless, bisimulation gives a minimal canonical form for non deterministic FSMs and has an efficient polynomial time algorithm. The implemented algorithm uses the BDD

representation of the state space and a BDD partitionned representation of the transition relation as a vector of transition functions, one per latch, [3].

Minimized FSMs are generated explicitly in a textual format called FC2.

2.2 The Output Signals Status Checking

There are two status for output signals that can be checked:

1. if an output signal is *possibly* emitted, i.e. there exists a reachable configuration of latches from which the signal is emitted for some configuration of the inputs;
2. if an output signal is *possibly not* emitted, i.e. there exists a reachable configuration of latches from which the signal is not emitted for some configuration of the inputs.

This checking can be made modulo a restriction over the input configurations. Indeed, one can fix the value of some input signals to 0 or 1 (always absent or present) and perform the checking taking into account these constraints.

The output status checking can be used to verify *observers*. These are ESTEREL programs that that catch misbehaviors and property violation from the program to verify. Typical properties are in the form

> if SIGNAL1 and SIGNAL2 are present then emit PROPERTY_VIOLATED

or

> if SIGNAL1 is present then before k occurrences of SIGNAL2, SIGNAL3 must be emitted, otherwise emit PROPERTY_VIOLATED

Observers are then put in parallel with the program under analysis. Model-checking consists just in verifying the status of the observers signals (such as PROPERTY_VIOLATED). Users can directly write observers by hand in ESTEREL. However, there exists a tool called TEMPEST [7] offering a (linear time) temporal logic to express properties; this tool provides a translator from logic formulae into ESTEREL observer programs. TEMPEST then performs the model-checking in the expanded FSM (using its explicit automata description). This is a majors limitation since for large programs, the ESTEREL compiler cannot generate expanded FSM. However, generating implicit version of FSMs using the boolean circuit formalism is much more less problematic. We have developed a processor that interface TEMPEST with the XEVE output checking. This processor is called HURRICANE and is distributed together with XEVE. It takes as input any number of files defining temporal properties and a set of ESTEREL source files forming the program to model-check. Then:

1. it calls the Tl2strl processor from TEMPEST that translates the formulae into ESTEREL observers. It extracts the output signal names from the observers in an obvious way as the translator adopts a naming convention for the observer signals;
2. it forms a new program with the original program put in parallel with the observers;

3. it calls the ESTEREL to get the BLIF file of this new program;
4. it calls the CHECKBLIF processor to check the extracted output signals above on the FSM represented by the generated BLIF file.

3 The Graphical Interface

Figure 1 shows the main graphical panel of XEVE.

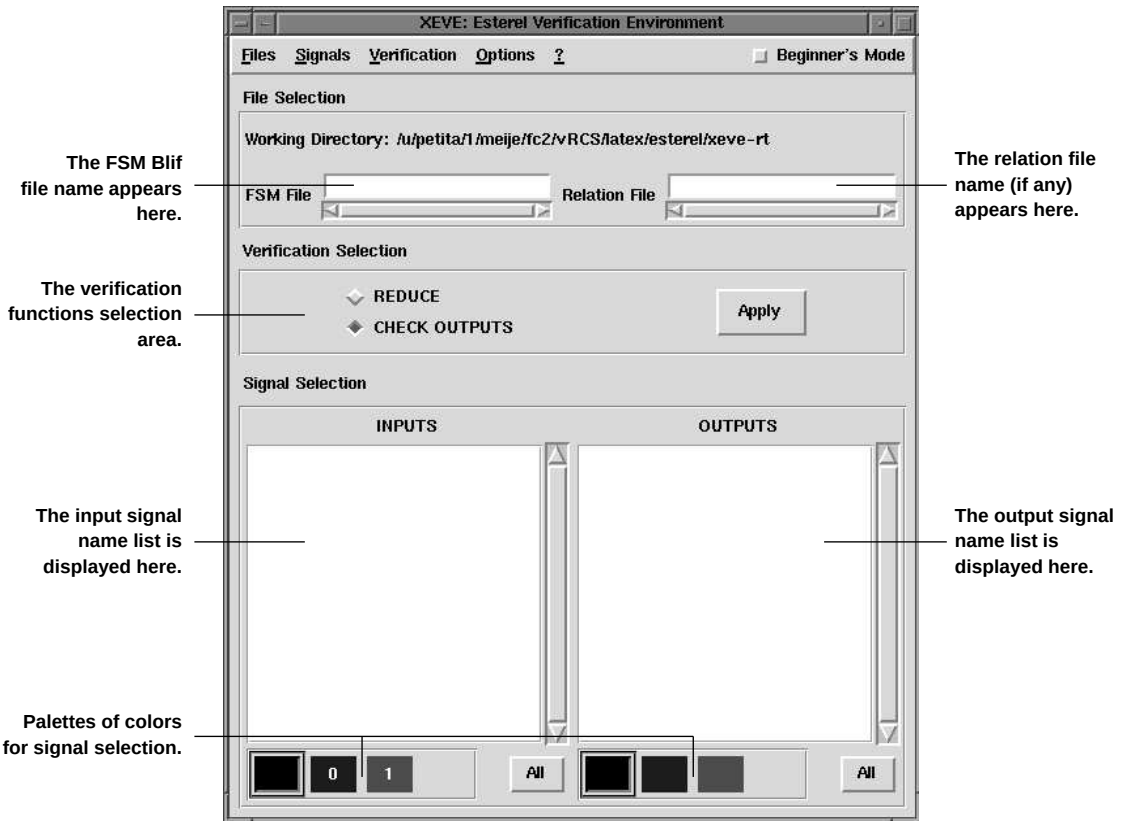

Figure 1: XEVE main panel

3.1 Inputting a FSM

A FSM can be loaded from two types of file:

1. BLIF file: This is the Berkeley Logical Interchange Format to describe a set of boolean equations with latches, hence an implicit FSM. Using the ESTEREL compiler one can compile any ESTEREL program into a set of boolean equations with latches. The compiler can generate the equations in the BLIF format. BLIF is useful as it is accepted by a lot of research synthesis and verification tools such as TIGER. We use TIGER to represent FSMs and to manipulate them symbolically using BDDs from their BLIF description.

2. BDDF file: BDDs are a powerful way to manipulate very large FSMs efficiently. However, BDD construction may take time, particularly the BDD of the reachable state space of the FSM. The TIGER library provides an ASCII format called BDDF to save the FSM's BDDs in this format, once these BDDs have been computed. You can save any FSM in this format using the menu entry "Files:Save FSM" and load any saved BDDF file. This avoids the re-computation of the FSM BDDs and reachable state space.

To load a BLIF file, use the entry menu `Files:Open FSM (blif format)`. A file selection box browser appears for file selection, as shown if figure 2. After a file selection, a Yes/No

Figure 2: XEVE file browser

window is displayed asking you if you need to load some input constraints over the inputs of the program. This constrains must be also described in a BLIF file as a combinational circuit over the inputs. These constraints are the input relations you have defined on your ESTEREL program. If so, click on `Yes` button to select a relation file.

To load a BDDF file, use the entry menu `Files:Open FSM (bddf format)`. In this case, no relation file is asked (it is supposed to be already computed in).

The selected file names are reported in the main panel. Selection can still be made by hand by directly typing in this entry space validating the choice with the `<Return>` key.

At opening time, the described symbolic FSM is internally built and its input/output interface is displayed on the main panel. If a bad file is given, an error message window is popped-up. Figure 3 shows the main panel with an opened FSM.

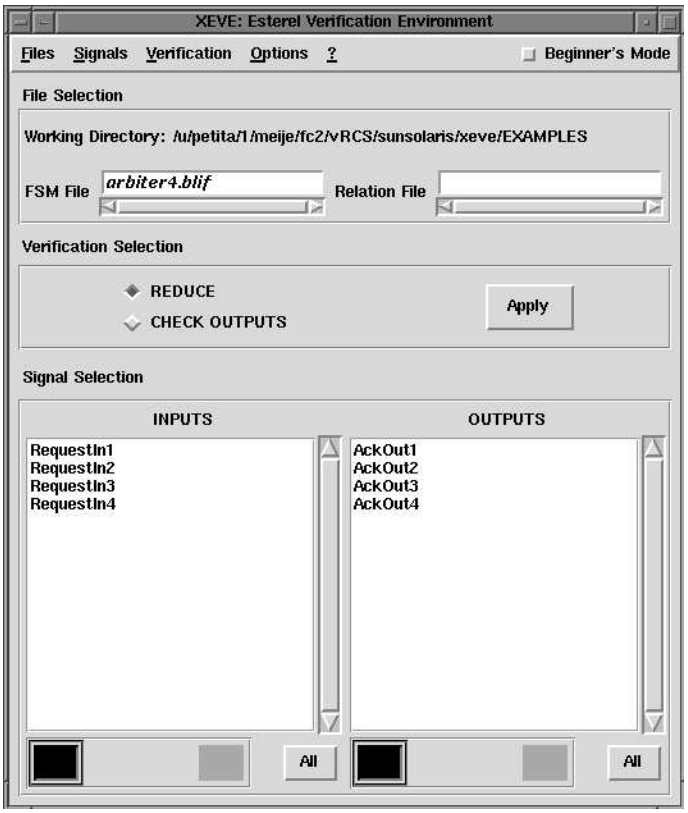

Figure 3: An opened FSM in XEVE

3.2 Selecting a Verification Function

To select the FSM minimization, just click on the `REDUCE` radio-button to set it on. To select the output status checking, click on the `CHECK OUTPUTS` radio-button.

Each function displays a particular palette of colors below the signal list areas. These colors are used to select or fix the signals (see section below).

3.3 Selecting the Signals

Before applying any of the two verification function presented above, you may need to select or fix the signals that are relevant for the verification you want to perform. Selection is made by modifying the signal names foreground to some preselected color in the underlying color palette. Initially, signal names appear in black. Each function has its own palette. We give for each the meaning of the colors.

3.3.1 Colors for FSM Minimization

Both input and output signals may be either *visible* or *hidden*, i.e. taken and not taken into account during behavior comparison of states, respectively. The will not appear in the transition labels of the minimized FSM. The same palette underlies the input and output signal areas, with the same meaning for the colors:

black means the signal is kept visible

grey means the signal is hidden

3.3.2 Colors for Output Status Checking

An input signal can be given a fixed value or let free:

black to let the signal have any value;

blue to fix the signal to 0 (always absent)

red to fix the signal to 1 (always present)

For output signals, we use colors to select the output signals that have to be checked and what kind of checking has to be done:

black means not selected;

blue to check if the signal is possibly not emitted

red to check if the signal is possibly emitted

3.4 Applying the FSM Minimization

If the `REDUCE` radio-button is set on, they you may invoke the FSM minimization by clicking on the `Apply` button. Once this button is invoked, the file browser (see figure 2) is popped-up to select a FC2 file name in which to save the minimized FSM. After the file name selection, a panel is popped-up to display the results of the minimization (see figure 4). The functional steps are:

1. The computation of the reachable state space *only if* it has not been computed before.
2. The minimization of the FSM modulo the signal selection.
3. The generation of the result in the selected FC2 file.

During these steps, you may interrupt the minimization procedure at any time by clicking on the `Abort` button. Aborting the computation makes XEVE returning to the state it was just before the launching of the procedure. Once the procedure has completed you may:

- invoke ATG to explore graphically the minimized FSM. To this end, click on the `Draw Automaton` button.
- dismiss the result window by clicking on the `Dismiss` button.

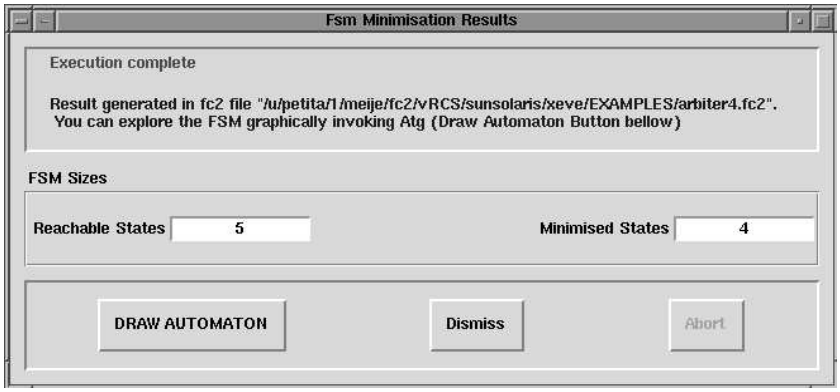

Figure 4: The minimization result panel

3.5 Applying the Output Signals Status Checking

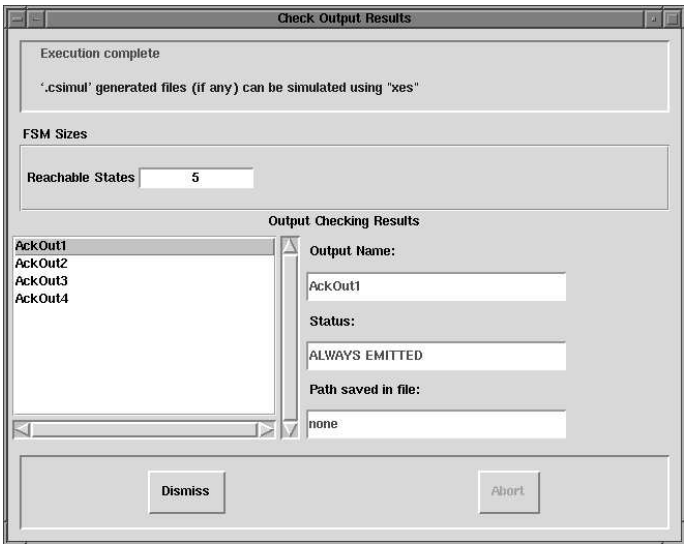

Figure 5: The output status checking result panel.

When the `CHECK OUTPUTS` radio-button is set on, you may invoke the output status checking by clicking on the `Apply` button. As in the previous case, a result window is popped-up, as illustrated in figure 5. The list of signals displayed in this panel are the signals you have selected: they are colored in red or blue in the main panel. Here are the steps that are performed:

1. The reachable state space is computed if it has not been already computed.

2. For each selected output signal, it is checked if it is possibly emitted if it is colored in red or not emitted if it is colored in blue. If a red (resp. blue) signal is found emitted (resp. not emitted), an execution trace leading to the state where the emission (resp. non emission) is found, is extracted. The trace in saved in the CSIMUL format. The name of the file is `signal-name.csimul` if *signal-name* is the name of the signal under consideration.

As in the previous case, the computation can be aborted at any time using the `Abort` button. At the end of the procedure, you may dismiss the result window using the `Dismiss` button. CSIMUL files you may have generated can be played within XES, the ESTEREL graphical simulator.

3.6 Options

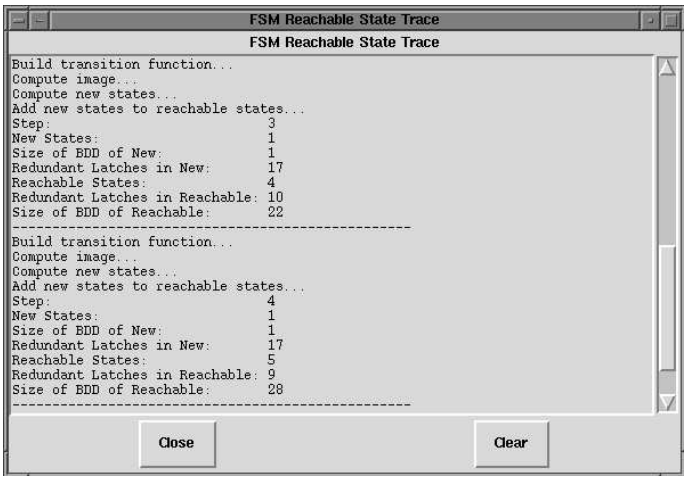

Figure 6: The reachable state trace window.

Options are available in the `Options` menu. The menu entries are three check-buttons, each to set/unset a particular option. These check-buttons are:

`FSM Statistics` if set on, then a window is popped-up that displays the FSM main characteristic values: the number of inputs, outputs, latches and literals of the FSM. It gives moreover the number of reachable states if it has been computed. Figure 7 shows this window. At start-up, this option is set off.

`Reachable State Trace` if set on, then a window is popped-up as soon as the computation of the reachable states is started. In this case, the window displays a set of informations for each step of the reachable state space algorithm. At start-up, this option is set off.

`Balloon Help` if set on, the balloon help system displays a small window with a help text for the main buttons and areas of the graphical interface. At start-up, this option is set on.

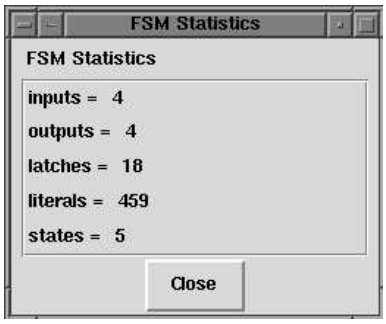

Figure 7: The FSM statistics window.

3.7 Getting Help

There are three levels of help.

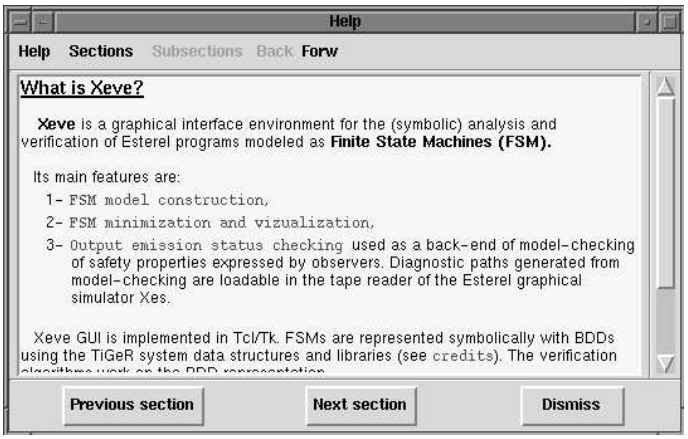

Figure 8: The XEVE help window.

A help text window cut into sections. To invoke it, use the entry menu `?:Xeve Help`. The help window that is popped-up is shown in figure 8.

An on-line help for beginners: This help is enabled when the check-button `Beginner's Mode` on top of the main panel is set on. A help text is displayed on a message window at each main function calls. The information window is automatically discarded at each new function call. The user can also dismiss it manually. At start-up, this help is set off.

A Balloon help: A balloon help window appears on the main buttons and areas of the interface. It appears a short delay (about half a second) after the cursor has entered and moved inside a button or an area where a balloon help is defined. The balloon help is displayed off when the cursor leaves the button or the area, or if you type any key on the keyboard. At start-up, the balloon help is activated. To set it off, disable the check-button entry menu `Options:Balloon Help`.

4 Interfaced Tools

As previously mentioned, the tools ATG and XES are interfaced with XEVE through the file formats FC2 and CSIMUL respectively.

4.1 ATG: Drawing Minimized FSMs

Once minimized, you can call ATG to draw the FSM graph by clicking on the `Draw Automaton` button. ATG appears as a menu-bar and displays a window with a single state and no transition. ATG invites you to explore step by step the FSM graph. To do so, press the mouse left button on a state to explore and while keeping the button pressed, drag the mouse to some position where ATG shall draw around *all* the successors of the state and the transitions leading to them. Explored state are drawn with a single circle while unexplored ones are drawn with two circles. Figure 10 shows several drawings of the same window corresponding to successive steps of exploration.

4.2 XES: Simulating Execution Sequences

XES is a X-window simulator for ESTEREL programs, which supports interactive mouse-based simulation and source code symbolic debugging. The ESTEREL code must be compiled with the `-simul` option, which generates C code. Then compile the C code to get an object code file. You can build and run your own executable simulator in two different ways. You can directly link your ESTEREL program object code to the `libxes.a` library and run the produced executable, or you can use the XES command. Since simulation object files can be quite big, using XES is preferable whenever possible. When inside XES, you can invoke the tape player/recorder allowing ESTEREL session saving and playback. A tape is a file in the current directory. It contains recorded events in csimul format. Its name is the tape

name with extension `.csimul`. The execution sequences generated by the output status checking are such files. So you can play the sequence in the simulator. Moreover, the tape player/recorder provides different play modes (step by step, silent, fast, ...).

5 A Tutorial Example

In this section, we show how to verify a small synchronous bus arbiter written in ESTEREL using XEVE.

5.1 The Example: a Small Bus Arbiter

This example was originally treated in Mc Millan's thesis, [8] and adapted in ESTEREL in [4]. Consider the following informal specification: A range of users may claim access to a bus through input wires $Request_1, \ldots, Request_n$. At most one requested access must be granted (at output wire Ack_i, $1 \leq i \leq n$) at each step, and exactly one in case there is effectively an actual request.

Each user shall be connected to a *cell*. Each cell receives a request through an input signal `RequestIn`. It acknowledges the access to the bus through the output signal `AckOut`. A token ring is introduced for a faith and symmetric solution. A latch is introduced in each cell for the token passing. The cell and a network of 3 cells are illustrated in figure 9. We

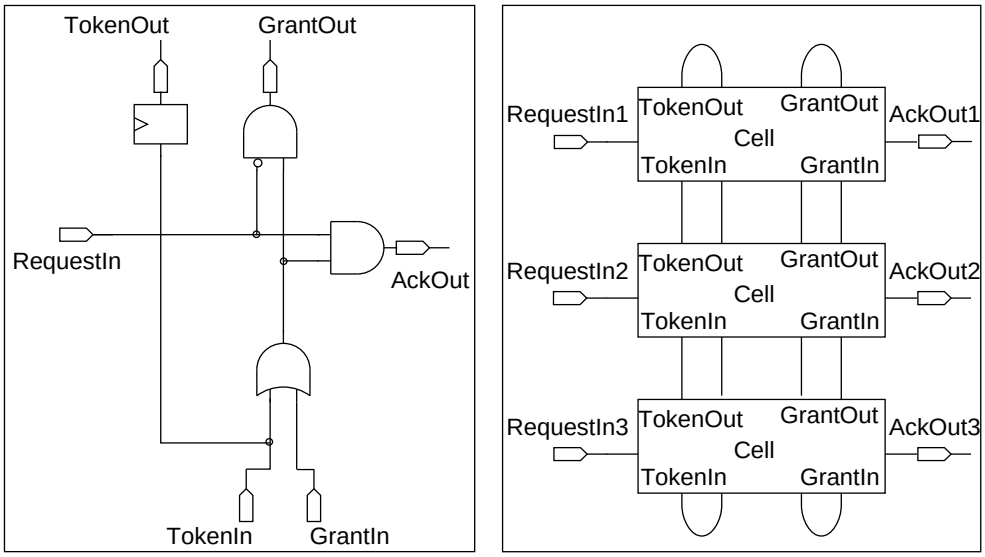

Figure 9: The circuits for a cell and a ring of 3 cells.

have to ensure that:

1. Two users can not access to the bus simultaneously..
2. If there is some request in the ring, then an access to the bus is obtained by some user.
3. If a user is constantly requesting then it eventually accesses to the bus at some point in the future..

The properties 1 and 2 are *safety* one and property3 is a liveness one. In fact, the liveness property 3 can be bounded by the number of users in the ring, which is the maximal time for the token to be received by the constantly requesting user. The ESTEREL source program for a cell is:

```
% Module implementing a Cell process
module Cell:
input RequestIn, GrantIn, TokenIn;
output GrantOut, TokenOut, AckOut;

every immediate [GrantIn or TokenIn] do
   present RequestIn then
      emit AckOut
   else
      emit GrantOut end
end
||
every immediate TokenIn do
   await tick;
   emit TokenOut
end
.
```

The signal `GrantOut` is a way to "pass a grant" along the ring from the token holder to the first user that is requesting in the same instant, whenever the token holder is not requesting in the instant. The grant allows to access the bus without having the token. The grant is received through the signal `GrantIn`. The ESTEREL program source for a ring of 4 cells is:

```
% A dummy module to through a token in the ring at the first instant
module Init:
output Token;

emit Token
.
% The module that implements the bus arbiter with 4 cells process running
% in parallel and concurently for a bus access.
module Arbiter4:
input RequestIn1, RequestIn2, RequestIn3, RequestIn4;
```

```
output AckOut1, AckOut2, AckOut3, AckOut4;

signal G1, G2, G3, G4, T1, T2, T3, T4  in
     run Init[signal T1/Token]
 ||  run Cell[signal RequestIn1/RequestIn, AckOut1/AckOut,
                     G1/GrantIn, G2/GrantOut, T1/TokenIn, T2/TokenOut]
 ||  run Cell[signal RequestIn2/RequestIn, AckOut2/AckOut,
                     G2/GrantIn, G3/GrantOut, T2/TokenIn, T3/TokenOut]
 ||  run Cell[signal RequestIn3/RequestIn, AckOut3/AckOut,
                     G3/GrantIn, G4/GrantOut, T3/TokenIn, T4/TokenOut]
 ||  run Cell[signal RequestIn4/RequestIn, AckOut4/AckOut,
                     G4/GrantIn, G1/GrantOut, T4/TokenIn, T1/TokenOut]
end.
```

We put four instances of the cell in parallel making the proper renaming to distinguish the cells. The module `Init` serves in injecting the token in the at the first instant. All the sources above are put in a file `arbiter4.strl`.

5.2 Generation of the BLIF File

We invoke the ESTEREL compiler to generate the circuit of the program in the BLIF format.

```
> esterel -L blif arbiter4.strl
```

This command generates the file `arbiter4.blif`.

5.3 Staring up a XEVE Session

To start XEVE type the command:

```
> xeve
```

Load `arbiter4.blif` as explained in section 3.1. You may load the file directly by giving to the command line the file name as an argument:

```
> xeve arbiter4.blif
```

Figure 3 shows what you obtained after opening the file.

5.4 The FSM Minimization and Visualization

We have applied the minimization of the loaded FSM. We have kept all the signals visible. In the case of 4 cells, the FSM has 4 states as one may expect. Indeed, each state is a configuration of the latches where the token is at a precise position. Figure 10 shows the graphical exploration of the FSM's graph with the tool ATG.

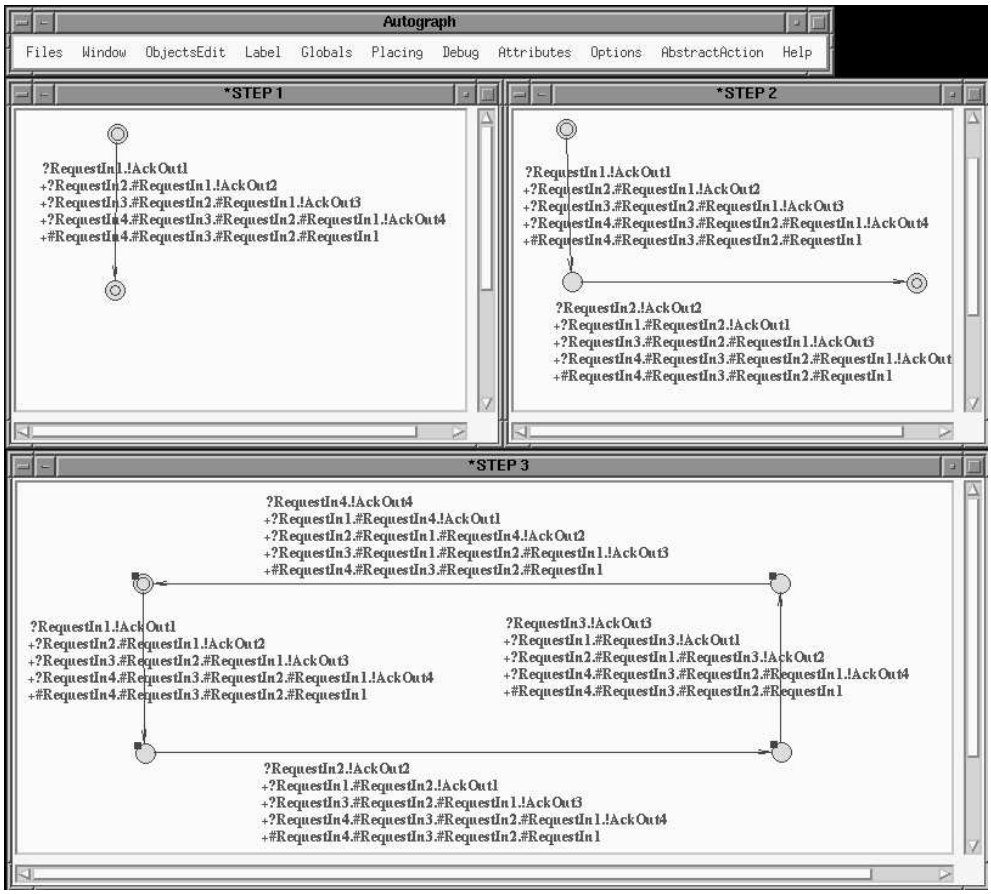

Figure 10: Explored FSM of the 4-users bus arbiter with ATG

5.5 Checking Output Status and Verifying Properties

Checking the status of some output signals under some input constraints is a way to verify simple safety properties. In our example, we can check for instance that if some user i is constantly requesting ($\texttt{RequestIn}_i = 1$) and if all other users are constantly not requesting ($\forall j \neq i$, $\texttt{RequestIn}_j = 0$), then the user i access the bus at each instant ($\texttt{AckOut}_i$ is always emitted). To check this, we set the input interface to fix the input signals as expressed in the condition we have just described, and as illustrated in figure 11. Then, we check if $\texttt{AckOut}_i$ is possibly *not* emitted to obtain a contradiction as this output signal is always emitted. In the illustrated example, we obtain that $\texttt{AckOut}_1$ is always emitted.

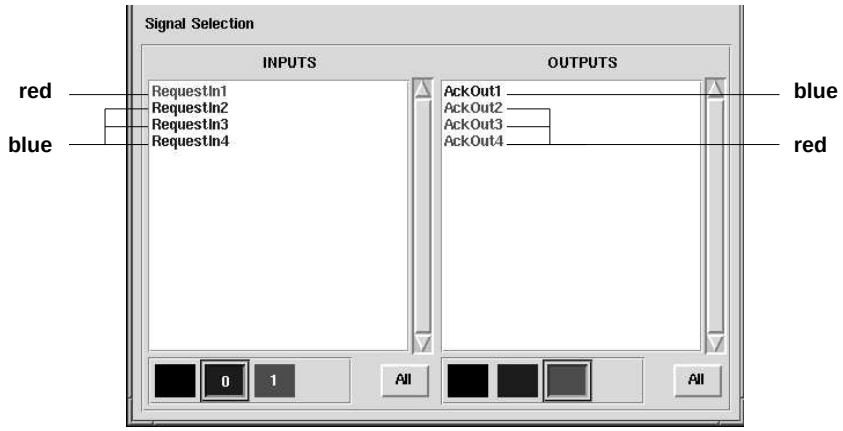

Figure 11: Signal selection for output signal status checking

There are simple correctness properties to be checked on the bus arbiter. While expressed in temporal logic, the idea here it not to use this formalism but instead use the ESTEREL languages to define these properties by means of synchronous observers. The properties are:

- There is at most one acknowledge: `always` $(i \neq j \Rightarrow \neg(AckOut_i \wedge AckOut_j))$.
- There is only an acknowledge for someone requesting: `always` $(AckOut_i \Rightarrow RequestIn_i)$.
- There is at least one acknowledge in case someone is requesting: `always` $(RequestIn_i \Rightarrow \exists j, AckOut_j)$.
- (weak fairness) someone constantly requesting will eventually be acknowledged: `always eventually` $(AckOut_i \vee \neg RequestIn_i)$.

The ESTEREL source for these observers is:

```
% Observer module implementing the temporal property
% "always ( i =/= j ==> not(Ack_i and Ack_j) )"
% expressing there is at most one acknowledge
module AtMostOneAck4:
input AckA, AckB, AckC, AckD;
output AT_MOST_ONE_ACK_VIOLATED;

loop
   present [(AckA and AckB) or (AckA and AckC) or (AckA and AckD) or
            (AckB and AckC) or (AckB and AckD) or
            (AckC and AckD)] then
      emit AT_MOST_ONE_ACK_VIOLATED
   end;
```

```
   await tick
end.

% Observer module implementing the temporal property
% "always ( Requ_i ==> exists j, Ack_j)"
% expressing there is at least one ack in case of request.
module AtLeastOneAck4:
input ReqA, ReqB, ReqC, ReqD, AckA, AckB, AckC, AckD;
output AT_LEAST_ONE_ACK_VIOLATED;

loop
   await immediate [ReqA or ReqB or ReqC or ReqD];
   present [AckA or AckB or AckC or AckD]
   else
      emit AT_LEAST_ONE_ACK_VIOLATED;
   end;
   await tick
end.

% Observer module implementing the temporal property
% "always eventually ( Ack_i or (not Requ_i))"
% expressing someone constantly requesting will eventually be ackwoledged.
module AckWhenAllRequest:
input ReqA, ReqB, ReqC, AckD;
output ACK_WHEN_ALL_REQUEST_SUCCESS;

await immediate [ReqA and ReqB and ReqC and AckD];
emit ACK_WHEN_ALL_REQUEST_SUCCESS
.

% Observer module implementing the temporal property
% "always  (always Requ_i implies eventually Ack_i )"
% under the knowledge that a constant request must be statisfied in 4 cycles
% (4 being the number of Cell processes in the ring). It expresses
% someone constantly requesting will eventually be ackwoledged in 4 cycles.
module Fairness:
input RequestIn, AckOut;
output Fairness_FAIL;

loop
   trap SUCCESS in
      loop
         trap RESET in
            loop
               present RequestIn then
                  present AckOut then
```

```
                  exit SUCCESS;
               end present;
            else
               exit RESET;
            end present;
         each tick
      ||
         [
          % await 4 tick (4 = number of Cells in the ring)
          await tick; await tick; await tick; await tick;
          emit Fairness_FAIL
          ]
        handle RESET do await tick;
        end trap; % RESET
     end loop;
  handle SUCCESS do await tick;
  end trap; % SUCCESS
end loop
end module


module ObsArbitrer4:
input RequestIn1, RequestIn2, RequestIn3, RequestIn4;
output AT_MOST_ONE_ACK_VIOLATED, AT_LEAST_ONE_ACK_VIOLATED,
       ACK_WHEN_ALL_REQUEST_SUCCESS,
       Fairness_FAIL1, Fairness_FAIL2, Fairness_FAIL3, Fairness_FAIL4;

signal AckOut1, AckOut2, AckOut3, AckOut4 in
   run Arbiter4

|| run AckWhenAllRequest[signal RequestIn1/ReqA, RequestIn2/ReqB,
                         RequestIn3/ReqC, AckOut4/AckD]

|| run AtLeastOneAck4[signal RequestIn1/ReqA, RequestIn2/ReqB,
                            RequestIn3/ReqC, RequestIn4/ReqD,
                            AckOut1/AckA, AckOut2/AckB,
                            AckOut3/AckC,AckOut4/AckD]

|| run AtMostOneAck4[signal AckOut1/AckA, AckOut2/AckB,
                           AckOut3/AckC,AckOut4/AckD]

|| run Fairness[signal AckOut1/AckOut, RequestIn1/RequestIn,
                       Fairness_FAIL1/Fairness_FAIL]

|| run Fairness[signal AckOut2/AckOut, RequestIn2/RequestIn,
                       Fairness_FAIL2/Fairness_FAIL]
```

```
|| run Fairness[signal AckOut3/AckOut, RequestIn3/RequestIn,
                      Fairness_FAIL3/Fairness_FAIL]

|| run Fairness[signal AckOut4/AckOut, RequestIn4/RequestIn,
                      Fairness_FAIL4/Fairness_FAIL]
end.
```

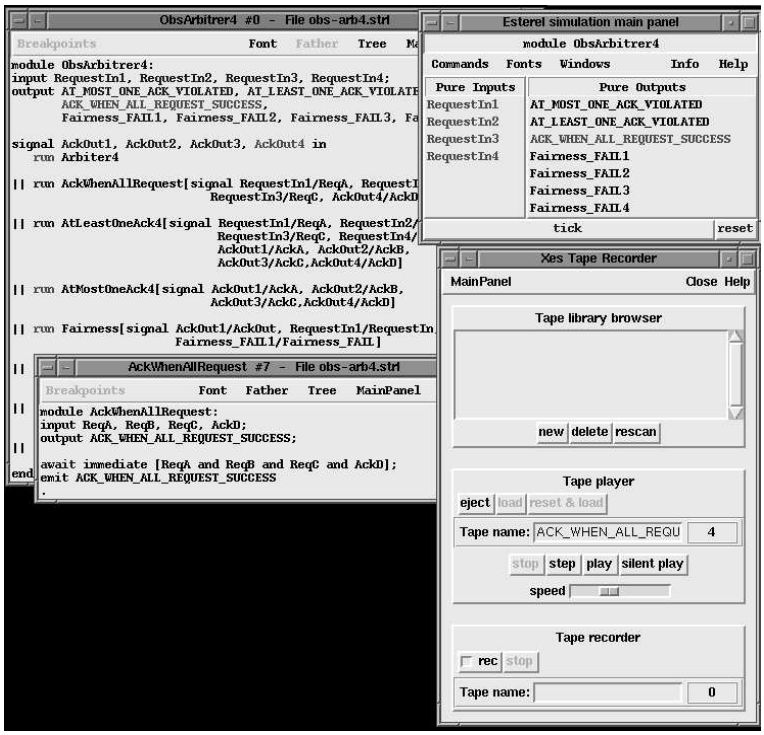

Figure 12: The XES simulation session

The observer modules are put in parallel and compiled in the BLIF file `arbiter4.blif`. The three first observers implement the safety properties we have specify in the beginning. The last observer implements a *bounded* liveness property.

To check these properties, we set the input/output signal lists as follow: we set all the inputs to black (they can t have any value; we set to red all the observer output signals. When we check them, we find that all these signals can never be emitted but the `ACK_WHEN_ALL_REQUEST_SUCCESS` which can. This means, in the example, that the last user in the ring order may have access to the bus though all its predecessors are

requesting the access at the same time, thanks to the token which impose a priority. We could have checked this property for all the users by putting as many instances of the module `AckWhenAllRequest` as there are users, with a particular signal renaming for each. In the example, as the signal is found emitted, an input event sequence is extracted that leads to a state where the signal can be emitted. This sequence is save in the file `ACK_WHEN_ALL_REQUEST_SUCCESS.csimul`, called a tape, and can be simulated on the ESTEREL source using XES, the ESTEREL graphical simulator. Figure 12 shows the graphical interface of XES: the tape reader and recorder is popped-up, playing the tape file previously generated, called `ACK_WHEN_ALL_REQUEST_SUCCESS.csimul`.

Acknowledgments and Credits

The author would like to thank Gérard Berry for his helpful comments and suggestions on the design of XEVE, and Robert de Simone for the discussions on the verification algorithms.

XEVE algorithms and data-structures are written in C++. They are based on the TIGER system C libraries, a DEC product commercialized by Xorix. It is developed by O. Coudert, J.C. Madre, H. Touati (`madre@synopsys.com`). The graphical part is written in TK, a scripting language developed at Sun Microsystems in the Sunscript development team leaded by John Ousterhout. The help part is a patched version of the help implemented in in the JSTOOLS by Jay Sekora (`js@priceton.edu`). The balloon help system is a modified version (thanks to Xavier Fornari) of an initial implementation in TKDESK by Christian Bolik (`zzhibol@rrzn-user.uni-hannover.de`).

List of Figures

1 XEVE main panel 6
2 XEVE file browser 7
3 An opened FSM in XEVE 8
4 The minimization result panel 10
5 The output status checking result panel. 10
6 The reachable state trace window. 11
7 The FSM statistics window. 12
8 The XEVE help window. 12
9 The circuits for a cell and a ring of 3 cells. 14
10 Explored FSM of the 4-users bus arbiter with ATG 17
11 Signal selection for output signal status checking 18
12 The XES simulation session 21

References

[1] G. Berry. A hardware implementation of pure Esterel. *Sadhana, Academy Proceedings in Engineering Sciences, Indian Academy of Sciences*, 17(1):95–130, 1992. Rapport Centre de Mathématiques Appliquées de l'Ecole des Mines de Paris, numéro 06/91.

[2] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

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

[4] A. Bouali, J.-P. Marmorat, R. De Simone, and H. Toma. Verifying Synchronous Reactive Systems Programmed in Esterel. *Lecture Notes in Computer Science*, 1135, 1996.

[5] R. De Simone and A. Ressouche. Compositional Semantics of Esterel and Verification by Compositional Reductions. *Lecture Notes in Computer Science*, 818, 1994.

[6] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, U.C. Berkeley, May 1992.

[7] L.J. Jagadeesan, C. Puchol, and J. Von Olnhausen. Safety Property Verification of Esterel Programs and Application to Telecommunications Software. In *Computer-Aided Verification*, Liege, Belgium, July 1995.

[8] Mc Millan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[9] D.M.R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, 5^{th} *GI Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1981.

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

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 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 - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803