XEVE, an ESTEREL Verification Environment

Amar Bouali

INRIA, B.P. 93, F-06902 Sophia-Antipolis cedex
amar@sophia.inria.fr

Abstract. We describe the verification methods and tools we are currently developing around the language ESTEREL. This language is dedicated for the development of synchronous reactive systems such as hardware or software controllers for which the control handling aspects are predominant. The language has a strong mathematical semantics in terms of Finite State Machines. Automatic verification is then possible on this model in which we represent exhaustively all the possible behaviors of a system. Our methods are based on model minimization coupled with unrelevant behaviors masking and model checking techniques to verify correctness properties like safety and liveness ones by means of synchronous observers

1 Introduction

ESTEREL [1] belongs to the family of synchronous languages, dedicated for the development of reactive systems. ESTEREL syntax is based on a set of primitives to express sequencing, parallelism, instantaneous broadcast, exceptions and watchdogs. It is defined by a mathematical semantics that relies on the *perfect synchrony hypothesis* that assumes a program reacts to its environment fast enough to be considered as instantaneous. The language compiles into two main formalisms according to its mathematical semantics: *explicit* Finite State Machines (FSMs) or *implicit* FSMs as a set of boolean equations with latches. In the first case, the compiler generates the FSM in some internal format, then translated in FC2[1]. In the second case, the compilers generates the equations in BLIF[2]. The FSM model allows automatic analysis and exhaustive verification of the control part of ESTEREL programs. This activity, called model-checking, is becoming ever more popular in the hardware community, [5, 6]. We shall describe in this paper our own approaches to the problem, some partially borrowed from similar work on our part in asynchronous process algebras, [2], some original and dedicated to the synchronous model. Roughly, they shall be split amongst: (compositional) *reduction* techniques, and (side) *observer* monitoring.

2 The ESTEREL Language

ESTEREL programming style allows to describe a systems as a set of parallel interacting modules, which in turn may be composed of interacting submodules.

[1] This is a textual format for graph-like object used to interface our verification tools
[2] *Berkeley Logical Interchange Format* for boolean sequential circuits

A module has an input/output interface which is declared on top of it. The example below is a small program for a synchronous bus arbiter, a simplified version described in [8]: users (`Cell` modules) try to access a bus concurrently by setting a request (`RequestIn` signals) and obtain the access when the arbiter emits a corresponding acknowledge (`AckOut` signals).

```
% The Init module
module Init
output Token;

emit Token;
end.

% The cell module
module Cell:
input Req, GrantIn, TokenIn;
output GrantOut, TokenOut, Ack;

every immediate [GrantIn or TokenIn] do
   present Req then emit Ack
   else emit GrantOut end
end
||
every immediate TokenIn do
   pause; emit TokenOut
end.
```

```
% The main module
module Arbiter4:
input Req1, Req2, Req3;
output Ack1, Ack2, Ack3;

signal G1, G2, G3, G4, T1, T2, T3, T4 in
     run Init [signal T1/Token]
 ||  run Cell [signal Req1/Req, Ack1/Ack,
                      G1/GrantIn, G2/GrantOut,
                      T1/TokenIn, T2/TokenOut]
 ||  run Cell [signal Req2/Req, Ack2/Ack,
                      G2/GrantIn, G3/GrantOut,
                      T2/TokenIn, T3/TokenOut]
 ||  run Cell [signal Req3/Req, Ack3/Ack,
                      G3/GrantIn, G4/GrantOut,
                      T3/TokenIn, T4/TokenOut]
 ||  run Cell [signal Req4/Req, Ack4/Ack,
                      G4/GrantIn, G1/GrantOut,
                      T4/TokenIn, T1/TokenOut]
end.
```

The protocol consists in putting the desired number of cells (4 in the example) in parallel, connecting them to form a ring. The `run` instruction calls an instance of a sub-module: the `signal` declaration that follows is a renaming instruction, that renames the sub-modules proper signals into signals of the environment they are put in. More details on the ESTEREL programming can be found at `http://www.inria.fr/meije/esterel/`.

3 The Verification Methods

Verification is based on the FSM model. Due to the state space explosion problem (the FSM size may be exponential with respect to the size of the program sources), our approach consists first is reducing as much as possible the state and transition spaces while keeping the structure and the behaviors of the system. This is achieved by using compositional methods. When this method fails, the compiler can still produce the circuit representation. In this case, we use symbolic representation to represent the state and transition spaces using BDDs.

Verification consists in analyzing the system behaviors to detect deadlock or livelocks, and perform model checking of properties we express through *synchronous observers*. We now describe each in turn:

FSM Minimization: Reduction consists in quotienting the model by state equivalence (bisimulation [9]), after a prior abstraction of signals due to the property checked (unconcerned signals are simply hidden, which can introduce non-determinism). Still, FSMs either deterministic or not, can be minimized canonically using bisimulation. In practice, the original algorithm for coarsest (state) partitioning [7, 4] has to be refined, due to the particular symbolic representation of input events mentioned above. Now *sets* of outgoing transitions have to be matched against *sets* of similar transitions for states to remain equivalent. Strong bisimulation remains a congruence for synchronous product, so that compositional reduction can be performed. The FC2SYMBMIN tool takes a FC2 explicit description of the FSM, minimizes it, and generates the minimal FSM in a new FC2 file. BLIFFC2 works on BLIF files where an FSM is described implicitly as a set of boolean equations with latches. It exploits the TIGER system library[3] to manipulate symbolically the FSM. It first computes the reachable state space (RSS) of the FSM, minimizes the FSM states and transitions symbolically using BDDs. The minimized FSM is generated explicitly in a FC2 file.

Compositional Minimization: Roughly, compositional minimization consists in reducing sub-modules first separately before composing them in parallel. In a hierarchical description, a node module is obtain by composing smaller sub-modules. FC2SYMBMIN can perform the synchronous product of two sub-modules given as reactive automata in the same way the ESTEREL compiler makes it when we put them in parallel. This method is a way to tackle the state space explosion avoiding the evaluation of the full global FSM. It reduces drastically the size of the FSM to store when combined with signal hiding.

Termination Status and Liveness: An important information when dealing with continuously interacting systems is to know if the system is able to stop reaching a deadlock, or to behave silently infinitely (while it change states inside a loop, no visible output can be emitted). This check can be done *modulo* a subset of input signals set to off and output signals not considered as visible. We use this feature to verify some liveness properties expressed in a *always eventually* style. Indeed, we verify it by checking that there's no silent loops before the behavior expressed in the eventually part, if we mask all the signals not concerned with the property. Diagnostic paths are extracted in case.

Verification by Observers: Observers are simply new reactive components set in parallel with the main program, and monitoring some of its outputs while feeding it inputs at times to exercise it towards desired configurations. Then observers usually testify of their results by specific new signals *Success* or *Failure*. We try to use observers to check *safety* properties, but also a growing number of *liveness* properties, and also to develop a range of properties inherent to synchronous languages (and in particular ESTEREL) that are of constant use (as *deadlocks* and *livelocks* in usual asynchronous systems). Treatment of fairness is also an exciting issue in such context (although we shall not deal with it here), especially if one consider that programs are now issued from a "real" language, and not an ad-hoc formalism kept close to the verification model. The CHECKBLIF tool takes a BLIF description of an observed program as input and a set of observer output to check for emission. For each output that can be emitted, one (of the shortest) example path leading to a state from which the

[3] Developed at DEC by O.Coudert, J-C.Madre and H.Touati, it offers data structures for BDDs and FSM manipulations, starting from circuits described in BLIF

output is emitted is extracted and saved in an ascii format called `csimul`. This kind of files are loadable in the ESTEREL graphical simulator called XES.

4 The Toolset

We described the list of software modules we have developed for the analysis and the verification of ESTEREL programs:

XEVE [3] is a graphical panel allowing file selection and application of verification methods, as well as signal (de)selections when they play a parameter role. Programs are supposed to come in BLIF syntax for boolean equation form, and FC2 syntax for explicit FSMs.

CHECKBLIF verifies simple properties from specific signals in a product machine, following the observer paradigm;

BLIFFC2 performs reduction of implicit FSM relative to bisimulation, and provide an explicit (small) FSM as result;

FC2SYMBMIN performs reduction of explicit FSM relative to bisimulation, and allows parallel composition of such FSMs so that it is compatible with the observer approach, allowing compositional verification;

ATG is a graphical display system used in our context to visualize Mealy machines, usually after reduction.

XES is the ESTEREL graphical simulator used in our setting to play counter-example sequences generated from verification failures.

XEVE is available by anonymous ftp and is largely diffused. It is also used in an industrial context, particularly at Dassault Aviation where ESTEREL has been chosen to implement critical pieces of embedded softwares.

Figure 1 shows the software architecture of the XEVE environment. Figure 2 shows its main graphical panel and some of result windows.

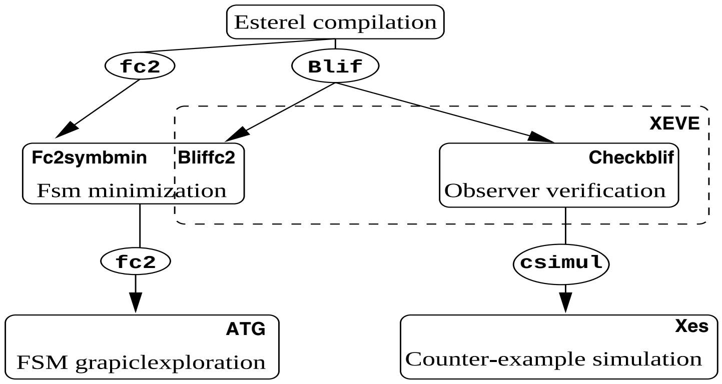

Fig. 1. Xeve internal and external tool modules

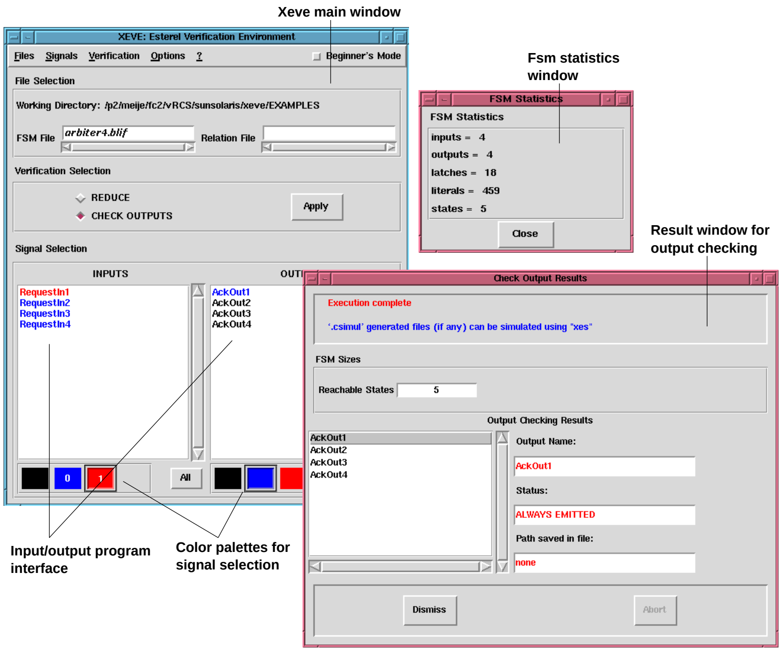

Fig. 2. Xeve graphical interface

References

1. G. Berry, and G. Gonthier The Esterel Synchronous Programming Language: Design, Semantics, Implementation *Science Of Computer Programming*, 19(2):87–152, 1992.
2. A. Bouali, A. Ressouche, V. Roy, and R. de Simone. The FC2Tools set. In *Proceedings of CAV'96* vol. 1102 of *LNCS*, 1996.
3. A. Bouali. XEVE: an ESTEREL Verification Environment (Version v1_3). Technical Report 214, INRIA, December 1997. http://ftp-sop.inria.fr/pub/rapports/RT-214.ps.
4. 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.
5. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.
6. O. Coudert, C. Berthet, and J.C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Automatic Verification Methods For Finite State Systems*, Grenoble, France, 1989. LNCS 407, Springer-Verlag.
7. P.C. Kanellakis and S.A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86:43–68, 1990.
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*, vol. 104 of *LNCS*, 1981.