ELSEVIER Science of Computer Programming 36 (2000) 5–25

Science of Computer Programming

www.elsevier.nl/locate/scico

ESTEREL: a formal method applied to avionic software development

Gérard Berry[a, *], Amar Bouali[b], Xavier Fornari[a], Emmanuel Ledinot[c], Eric Nassor[c], Robert de Simone[b]

[a] *Ecole des Mines de Paris/CMA, 2004, route des Lucioles, B.P 93 F-06902, Sophia-Antipolis cedex, France*

[b] *INRIA Sophia-Antipolis, 2004, route des Lucioles, B.P 93 F-06902, Sophia-Antipolis cedex, France*

[c] *Dassault Aviation, DGT/DPR/DESA, 78 Quai Marcel Dassault F-92214, Saint-Cloud cedex, France*

Abstract

Dassault Aviation is a French aircraft manufacturer building civil business jets (the Falcon family) and military jet fighters (the Mirage and Rafale families). It has been concerned with formal methods inside the development process of avionic software since 1989. In this paper, we give a comprehensive account of three industrial-size studies carried out at Dassault Aviation using the reactive synchronous language ESTEREL and its toolset, in collaboration with the public research team that develops ESTEREL at Ecole des Mines de Paris and INRIA Sophia-Antipolis. We deal with software engineering issues related to compilation, optimization and verification of safety-critical embedded software. The goal is to ensure production of efficient and reliable code. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Avionic software; Synchronous reactive systems; Safety-critical systems; ESTEREL; Modularity; Automatic code generation; Optimization; Verification

1. Formal methods for avionic software

1.1. The application domain

Avionic systems consist of three main classes of subsystems:

- *Aircraft management computers*: The fueling computer, the air conditioning computer, the landing computer, etc.
- *Mission management systems*: Roughly half a dozen computers, communicating through redundant real-time local networks. About 500 000 lines of Ada source code are loaded in the Mission Computer (MCs), 50% up to 80% of which are *reactive* software: event-driven computations, sequential logics, or finite-state automata.

* Corresponding author.

• *Flight control systems* (*FCS*): Three computers, one of which takes responsibility of the aircraft control laws (stability), and is therefore highly *safety critical.*

All embedded software is *specified* at Dassault Aviation, where resulting equipments and systems are also *integrated* and *validated* by ground tests and flight tests. Thus, Dassault Aviation locally masters the initial and final phases of the development process. Between these phases, everything is sub-contracted to equipment suppliers, except for FCS's equipments, whose hardware and software are built in-house.

Because of this organization, there is a twofold interest in formal methods:

(1) The high level, abstract, and precise style of expression enforced by formal methods should lead to better quality specifications than pure paper-pencil ones, or even semi-formal notations. Furthermore, since formal semantics yields at least an interpreter, formal avionic software specifications may be tested *before* they are realized by the equipment supplier. This first motivation applies primarily to the aircraft management computers and to the equipments of the mission management systems.

(2) In the context of flight control systems, where Dassault Aviation is its own supplier, the main goal is to *generate*, test, and possibly *prove* safety critical embedded code. The focus is on formal verification and *automatic safe code* generation from the specification formalism. The word *automatic* is crucial here: while high-level formal specifications are beneficial in the prototyping and specification testing phases, they contribute extra costs, and they should not be simply discarded in later phases. Rewriting the actual embedded code by hand from scratch, using the executable formal specification only as a reference model, is interesting in theory but too expensive in practice.

These industrial needs entail a set of requirements (see Section 2) that any formal method candidate for use as everyday industrial practice have to comply with.

We now briefly outline three real size experiments, drawn from different classes of avionic subsystems, successfully conducted by R&D engineers at Dassault Aviation. In two cases, the results were hard to achieve: the difficulties prompted for evolution in the design methods, as they pointed out particular locations where further theoretical work was needed. Work was carried out by Esterel (Section 3) specialists at Dassault Aviation, who had little knowledge of the specific application domains, so that "specification analysis" included training to the domain.

1.1.1. The maintenance and test computer

The first application is an executive-level software module of one of the three computers in a jet fighter flight control system: the maintenance and test computer (MTC). This computation unit starts testing sequences when the aircraft is on the ground, under orders from the pilot or the maintenance crew. In flight, the MTC monitors all of the FCS's critical parts (main computer's digital rack, actuators motion, etc.), and saves any detected failure in a stable memory space (black box) for post-flight analysis.

The existing MTC software is a real-time multi-tasking code. Nine tasks driven by both periodic and non-periodic events have to be scheduled. The shortest cycle duration is 10 ms, and there is another scheduling cycle of 40 ms. The target is a 68020 microprocessor programmed in C, with no real-time operating system. The actually embedded software has been designed to be scheduled on-line, according to an Highest Priority First (HPF) policy. The tasks handle the events: timer interrupts, pilot commands, aircraft events, etc.

The main objective of this case study was to assess ESTEREL's capability of generating truly embeddable safe and efficient code. ESTEREL was used to replace the HPF on-line scheduler by an off-line formally verified scheduling automaton. All reactive aspects of the original eleven tasks were then gathered into a single reactive kernel, programmed and verified with ESTEREL. We spent approximatively 12 man/months on this case study:

- 6 months by an ESTEREL expert to analyze the detailed software specification and to design a new software architecture better suited to the use of ESTEREL. An MTC software engineer helped the ESTEREL specialist, accounting for one extra man/month of work. The detailed specification was a 500 pages document.
- 2 months by the ESTEREL expert to program the reactive part of the MTC software with ESTEREL, to test it in interpreted mode on a workstation, and to compile the global reactive application into a safe embeddable C code.
- 1 extra month to split the global ESTEREL program into two separately compiled and optimized modules (*Kernel* and *MP* of Table 1), and to test them again.
- 2 months by an FCS software engineer to integrate the two generated C codes on the target 68020 embedded microprocessor, to test them, and to measure their actual performance.

1.1.2. The landing computer

The second application deals with one particular function of a landing computer: the control of the opening and closing of the aircraft traps, in synchronization with the up and down motions of the landing gears. This function also monitors the command execution, to send alarms to the pilot whenever relative traps and gears' position sensors do not deliver the expected information in a certain amount of time after the pilot's command. Position sensors are replicated (double or triple contact sensors) so that a single sensor failure may be unambiguously discovered and eliminated by software fault-tolerance mechanisms.

The main objective of this experiment was extensive formal verification. It took place in 1995 and 1996, and assessed primarily the performances of the TIGER[1] BDD package [18] for exhaustive state space search (see Section 2).

[1] Commercial package developed at DEC PRL by O. Coudert, J.C. Madre, H. Touati, distributed by XORIX.

This case study required a 7 man/months effort:

- 2.5 months to analyze the various system and software specification documents (about 300 pages).
- 2.5 months to write and simulate the Esterel program in interpreted mode, including the development of a dedicated graphical user prototyping interface.
- 2 months to formalize and verify the safety properties, including the time needed to write a introductory document to verification by model-checking, meant to be read by non-expert software engineers.

1.1.3. The side displays management

Man-machine interfaces in military aircraft are very complex, as seen from peering inside the cockpit of any modern airplane. In the latest combat aircraft at Dassault Aviation, tactile screens are included in two side displays. This increases by far the actual number of commands simultaneously available to the pilot. About 80 graphical pages can be displayed on these two tactile side screens, each page including numerous virtual command buttons.

The control logic which decides which two pages out of the 80 are displayed on screens depends on 125 physical or virtual commands, as well as on the current activation state of operational functions (navigation, attack, etc.) and on devices availability.

The side-display management logic was specified, programmed, and tested using the Esterel environment, augmented with a Statecharts-like graphical editor and object-oriented language extensions. The main objective was to experiment with Esterel programming in the large.

The side-displays case study required 2 man/month for the *Functions* module (of Table 1), 1.5 man/month for the side *Displays* (of Table 1) management module, and an extra time of up to 5 days to manage to compile both together, because of resulting cyclic circuits (see Section 4.2).

2. Requirements for industrial strength formal methods

Getting a formal method to become adopted by a reasonably large number of engineers in a company is a long and painstaking process, even when the method is based on high-quality academic research results and when its tools ensure high productivity. In this section, we list specific requirements which Dassault Aviation feels a formal method has to meet to be well accepted and successful in the development of actual products under time-to-market constraints. We also briefly discuss how far Esterel meets these requirements.

- *Expressive power*: The formal method (FM) must add significant value compared to general-purpose programming languages (*C*, C^{++}, Ada, etc.), while remaining *easy to use*. This implies that the formalism should be *small*, *high-level*, and *specialized*, with primitive notions chosen for their adequacy to the application domain and its existing design methods. Engineers used to paper-pencil specifications and unfamiliar with actual programming will require *graphical* formalisms. Graphical representation support may also be useful when developing large applications, for instance to help browsing the hierarchical structure of the model at a glance, and to make communication amongst designers easier. Graphical notations should retain sound semantics just as textual programs. Otherwise, they can become misleading or fail to automatically generate simulation or implementation code.

- *Testing facilities*: *Formal* should be compatible with *flexible*. The "program a little, simulate a little" paradigm of rapid application development (RAD) tools is also appropriate to formal methods. A formal method should provide an interpretor or a fast incremental compiler in order to be attractive and cost-effective in the industrial production cycle.

  Still, quick simulation is not enough. Source-level debugging, test-case recording and automatic replay of tests should also be available. High-level dedicated programming style on one-hand, verification on the other hand do not supersede traditional simulation, but they should rather *complement* it. In particular, verification counterexamples should produce test sequences.
- *Automatic code generation*: Development process seamlessness is at stake here. It is now well established that formal methods significantly improve specification quality and productivity. However, if there is no further link from the specification phase to the programming and final testing phases, the benefits are severely reduced or impaired.

  Automatic embedded-code generation from high-level dedicated formalisms is one of the most effective way to ensure that the program is conform with its specifications. The conformity relies on the compiler correctness. Formally verified refinements [1] or realizability theory and program extraction from constructive proofs [16] are other means to ensure process seamlessness.

  For Dassault Aviation FCS activities, process seamlessness is certainly *crucial* due to the importance of safety-critical aspects.
- *Formal verification*: From an industrial point of view, the main requirement regarding formal proof techniques is probably *simplicity* of use. At Dassault Aviation, as certainly in many other companies, software engineers fear routinely that formal methods are too complex and esoteric, that too long training or different skilled people are required. Automatic methods based on model-checking [15] or abstract interpretation [10] are more easy to introduce in a company than partially automated ones based on theorem proving.

  The second requirement is *scalability*. In our context the starting point of verification and analysis of programs consists quite often in the efficient computation of the (finite but extra-large) space of reachable configurations. Enormous progress were achieved in the past five years thanks to symbolic BDD-based algorithms [14, 17]. But even powerful BDD packages such as the TiGeR library extensively used in the current collaboration between Dassault Aviation and INRIA, sometimes fail to

complete on medium-size programs. Brute force global verification is certainly not sufficient, and compositional approaches must be derived to cut down on complexity [15].
- *Support*: Industrial companies cannot rely confidently on software products whose maintenance support is not guaranteed, or taken up by a sufficiently large community of industrial users. This applies as well to formal methods. This question was put forth in 1995, when Dassault Aviation decided to use Esterel operationally.

3. Synchronous reactive programming

3.1. Concepts

Reactive systems are mostly meant to communicate and react in collaboration with their environment. They usually possess both a reactive interaction aspect and a transformational aspect for data computation that update internal memory states. Here, we are concerned by control-dominated systems for which data-handling is fairly simple. *Synchronous reactive systems* (SRS) [19] behave according to discrete *instants* where inputs are provided. The system reacts to inputs in function of its current state by computing the outputs and the new state. The *synchrony hypothesis* simply states that instants are shared throughout the system and that reaction takes no time; in practice, it means that reaction time can be safely neglected.

SRS are constructed as networks of parallel subcomponents that react simultaneously and communicate by broadcasting signals, which can be explicitly scoped to range only on subsystems. *Preemption* structures control the life and death of a process; they are essential construct for control-dominated systems.

See [6, 7, 12, 19] for a general presentation of SRS, including motivation, discussion and justification of concepts.

3.2. Structured programming of reactive systems

3.2.1. Esterel

Esterel [6, 7, 31] is an imperative language dedicated to structured programming of control-dominated SRS. The core set of primitives contains powerful orthogonal constructs for concurrency, communication, and preemption. The full language adds a number of user-friendly derived statements. Data handling is imported from classical procedural languages such as C. An Esterel program starts by an interface header, where data-handling objects and input/output signals are declared. The program body then defines the behavior. An example Esterel program drawn from one of the Dassault Aviation applications is given in Fig. 1. The programming constructs are as follows:

```
module GESTION_TEST_BM:

type ZONE;

constant TEMPO_MAUV : integer;

function PRESENCE_PANNE_DANS_PLOT_COURANT(): boolean;

input TEST_BM_EFFECTIF, ZONE_DE_DEPART_BM(ZONE),
      ZONE_D_ARRIVEE_BM(ZONE), DEMARRAGE_TEST_BM,
      FIN_DU_TEST, RESET_TEST_BM,
      TRAITEMENT_FIN_DE_PLOT, TOP_TEMPO_MAUV,
      ASP, RELANCE_BM;
output
      DEMARRAGE_TEST, ZONE_DE_DEPART(ZONE),
      ZONE_D_ARRIVEE(ZONE), RESET_TEST,
      MAUV, FIN_DE_PLOT_TRAITEE;

  loop
    await TEST_BM_EFFECTIF;
    abort
      loop
        await DEMARRAGE_TEST_BM;
        emit ZONE_DE_DEPART(?ZONE_DE_DEPART_BM);
        emit ZONE_D_ARRIVEE(?ZONE_D_ARRIVEE_BM);
        emit DEMARRAGE_TEST;
        trap FIN_DU_TEST in
          [
            await FIN_DU_TEST;
            exit FIN_DU_TEST
          ||
            await RESET_TEST_BM;
            sustain RESET_TEST
          ||
            every TRAITEMENT_FIN_DE_PLOT do
              if PRESENCE_PANNE_DANS_PLOT_COURANT() then
                signal FIN_DE_PLOT in
                  [
                    await TEMPO_MAUV TOP_TEMPO_MAUV
                  ||
                    present ASP then
                      await RELANCE_BM
                    end present
                  ];
                  [
                    emit FIN_DE_PLOT
                  ||
                    abort
                      sustain MAUV
                    when FIN_DE_PLOT
                end signal
              end if;
              emit FIN_DE_PLOT_TRAITEE
            end every
          ]
        end trap
      end loop
    when [not TEST_BM_EFFECTIF]
  end loop
end module
```

```
module GESTION_MAUV_EN_TEST:

input MAUV,
      MAUV_ON,
      MAUV_OFF;
output
       ALLUMER_MAUV,
       ETEINDRE_MAUV;

  emit ETEINDRE_MAUV;
  [
    loop
      await MAUV;
      emit ALLUMER_MAUV;
      await [not MAUV];
      present [MAUV_ON or MAUV_OFF] else
        emit ETEINDRE_MAUV
      end present
    end loop
  ||
    every MAUV_ON do
      emit ALLUMER_MAUV
    end every
  ||
    every MAUV_OFF do
      emit ETEINDRE_MAUV
    end every
  ]
end module
```

Fig. 1. Examples of ESTEREL modules.

- *Control flow statements*: classical assignment, sequential composition '`;`', `if-then-else` test , the looping construct "`loop-end`", and parallel composition '`||`'.
- *Communication statements*: signal emission "`emit` *S*", signal presence test "`present` *S* `then` *body1* `else` *body2* end", and local signal scoping "`signal` *S* `in` *body* end".

- *Division of discrete instants*: "`pause`" explicitly closes the current behavior until next reaction; "`halt`" is a short-hand for "`loop pause end`", which idles forever (unless preempted).
- *Preemption operators*: "`suspend` *body* `when` *S*" inhibits (freezes) the body for the current reaction if *S* is present; "`abort` *body* `when` *S*" kills the body and instantaneously terminates upon *S*. A "`trap` *T* `in` *body* `end`" construct makes it possible to self-preempt *body* by executing "`exit` *T*". The `trap-exit` pair constitutes an exception mechanism fully compatible with concurrency.

Practice has shown that the above statements are very handy for writing control applications where the difficulty is in launching, killing, and coordinating tasks.

An essential feature of Esterel is *determinism*: a program generates the same output sequences from the same input sequences, which is desirable in most applications. This contrasts with the classical use of non-deterministic operating systems, which make debugging and analysis much harder. Furthermore, in many cases, no heavy dynamic task creation is required, so that using an operating system would be overkill.

Synchronous reactive programs are better viewed as (big) finite-state machines. The case study of the visualization interface is an excellent example: not all possible displays are active at the same time, but they must all be present and have state; the protocol for their activation is a sophisticated finite state machine.

3.2.2. Graphical representation

Statecharts [21] is a popular graphical description language in the domain of embedded systems. However, this formalism sometimes lacks accuracy to represent the subtle semantic preemption mechanisms offered by ESTEREL, and this reflects in the lower ability to produce efficient code. The graphical language SYNCCHARTS [2] is based on the same hierarchical automata view but provides graphical counterparts to the Esterel preemption constructs, see Fig. 5 Graphical SyncCharts are compiled into textual ESTEREL.

4. From Esterel to Boolean equation systems

The ESTEREL compiler translates the control core of programs into *Boolean Equation Systems* (BES) with *Boolean memory registers*, also called *latches* here. Such systems are widely used for the logical description of synchronous hardware circuits. As far as data handling is concerned, ESTEREL data actions are viewed as additional side-effects triggered by appropriate Boolean values.

The translation into BES makes it possible to apply existing hardware CAD techniques to embedded software. Conversely, one can view synchronous languages as high-level hardware description formalisms and transport their techniques there. Several steps of the ESTEREL compilation process benefit from the relation with circuit design: *causality* analysis, combinational and sequential *optimization*, and *symbolic* *model-checking* for analysis and debugging. All these steps involve crafted dedicated algorithms, mostly using BDDs (Binary Decision Diagrams) [13]. Some of them are original and have been developed in the context of the collaboration with Dassault Aviation, which provided larger and larger examples. For the time being, program compiling and analysis is done in a global way, which is a strong limitation. Larger programs will require a *compositional approach*, see Section 4.5.

In the sequel, we briefly present the semantics of BES, the constructive causality analysis, BES optimization, and BES verification.

4.1. Formal semantic interpretation

4.1.1. Boolean equation systems and the Esterel translation

Formally, a BES is a structure: (*In*, *Out*, *Reg*, *Loc*, *Eqs*), where *In*, *Out*, *Reg*, *Loc* are disjoint finite sets of *input*, *output*, *register*, and *local* Boolean variables, and where $x_i = F(x) \in Eqs$ is an equation defining the variable $x_i \in Out \cup Reg \cup Loc$ as a Boolean formula on variables $x \in In \cup Reg \cup Loc$. (In hardware vocabulary, the variables are called *wires* or *nets*, and BES are often called *netlists*. They also correspond to Mealy machines.) Execution is done in cycles, starting from the initial state where all registers have value 0. In each cycle, the outputs, next state, and locals are computed from the inputs and Boolean memory registers. The number of states a BES can take is obviously finite and of cardinality less than 2^n if there are n registers. An essential characteristic of a BES is its *reachable state space* or *RSS*, which is the set of states that can be reached by a computation from the initial state.

The main advantage of BES is to be both practical and formal. Practically speaking, one can directly generate hardware or C code from a BES. Theoretically speaking, one can use the power of modern symbolic Boolean manipulation techniques such as BDDs to efficiently analyze and optimize BES.

The idea of the translation from Esterel to BES is to encode the control flow by additional wires explicitly propagating activity, encoding sequencing as transmission of activity, concurrency as forking of activity, and preemption as deprivation of activity. For example, the sequence `P;Q` is turned into a parallel product `P|Q` where `P` signals completion through an additional signal on which `Q` is busy waiting for immediate start. The full translation is provided in [4, 5].

4.2. Constructive causality

For the semantics of a BES to be truly founded, one must require that outputs and registers get a unique Boolean value for each input and state. A classical sufficient condition is acyclicity of the combinational logic (excluding registers). However, the Esterel experience at Dassault Aviation has shown that this condition is too strong for large controller programming, especially when integrating code from several programmers. Cyclic circuits must also be considered.

Our analysis of cyclic circuits relies on *constructive causality* [5, 30], a refined notion which ensures that values of local, output, and next-state wires are computed entirely by pure propagation of facts from inputs and current state (as opposed to "guessing" solutions, later validated). A full description of constructive causality and its algorithmic computation is outside the scope of this paper, but we can sketch the main ideas. First, given inputs and a state, there exists a linear-time algorithm to decide constructiveness (i.e. correctness of the BES) and to compute the outputs and next state. This algorithm is implemented in the Esterel interpreter. Second, we have developed BDD-based algorithms to symbolically check for constructiveness for all inputs and all reachable states. The algorithm is presented in [30], improving upon original ideas by Malik [26]. It is based on ternary Boolean encodings where the undefined Boolean $\{\perp\}$ encodes absence or non-stabilization of value, as in Scott's denotational semantics. The algorithm perform a fixpoint computation in the three-valued Boolean domain, symbolically propagating facts (defined wire values). The acyclic version of the circuit is built from the original cyclic circuits and the computed BDDs. The Sccausal [32] component of the Esterel compiler implements this algorithm using the TiGeR BDD library [18].

Three major technical improvements of the constructiveness checking algorithm were required to handle the large Dassault examples. We cite them for specialists. First, we use fixpoint acceleration techniques originally developed for abstract interpretation of programs [11] to reduce the number of steps in the fixpoint computation. Second, we interleave the forward computation of the reachable state space (RSS) of the circuit with the computation of its output and next-state BDDs. These BDDs explode when directly computed from the combinational logic. Using the approximation of the RSS computed so far as a simplifier makes the computation possible. Third, to avoid generating too big an acyclic BES from a cyclic one, we had to carefully keep the structure of connected components, replacing only a few backwards nets by BDDs (notice however that the worst case is truly exponential).

4.3. Optimization

Due to the structural aspects of the translation, BES produced from Esterel (and even more from SyncCharts) are far from optimal. As usual, high-level programming provides comfort and easier software engineering but sacrifices low-level wizard tricks for efficiency. Optimization procedures are meant to correct this situation.

BES optimization divides into *combinational* and *sequential* techniques. Combinational optimization deals with the pure Boolean logic part, while sequential optimization deals with optimizing the state encoding by registers. Combinational optimization for Esterel is currently performed using standard algorithms of the Sis environment developed at UC Berkeley [27]. We have developed new sequential optimization techniques implemented in a tool called Remlatch [28]. Most of them are based on efficient calculation and usage of the RSS. We also use direct information provided by the structured programming style of Esterel, before computing the RSS, which can be expensive. For instance, in a sequence $P; Q$ or in a test *if c then P else Q*, the registers generated by P and Q are mutually exclusive. This information can be exploited both to simplify the logic and to remove redundant registers [29].

The collaboration with Dassault Aviation was a major testbench for the development of our optimization techniques. As a matter of fact, some of the Dassault programs now serve as large-size benchmarks in the CAD community.

4.4. Verification, analysis, debugging

BES form an ideal support for formal verification of program properties, a crucial need in our application range. Verification is directly based on the RSS symbolic computation. The RSS can actually be thought of as the largest useful invariant of the system. Verification queries can be put either in terms of *temporal logic* formulae [23] or of *synchronous observers* [20]. While less powerful, we choose the second approach because of its homogeneity [3, 9]: observers are additional Esterel programs put in parallel with the main program under analysis; an observer is in charge of emitting a specific signal (e.g. `BUG`) in case of program malfunction. Verification of safety properties then consists in symbolically checking that observer outputs can never be emitted. Our Xeve model-checker [8] verifies observer-based properties and generates counter- examples for false properties. Verification is currently extended towards some forms of liveness properties.

Exhaustive verification of programs with data is more difficult (sometimes undecidable) due to the possible complexity of infinite data types and related transformations. We are currently attacking this area, either by developing specific algorithms or by using other existing verifiers.

4.5. Modularity, compositionality

Many of the processing phases presented above can be computationally costly. This is especially the case for the constructiveness and RSS computations. The only way to reduce the complexity is to apply the algorithms in a compositional fashion. This can be done either by automatically detecting specific substructures in the BES, such as strongly connected components, or by exploiting the modular syntax of the programs. For instance, in a large program, the same module is often instantiated many times. Optimizing this module once for all is a big plus, provided that optimization preserves the constructive semantics.

The current implementation is not compositional. The bigger Dassault examples have shown that the brute-force approach has found its limits. The Dassault engineers have successfully performed compositional analysis on large examples, chaining steps by hand. We are now developing compositional compiling techniques that will make this kind of analysis automatic.

The compositionality issue is connected to the “assume/guarantee” style of assertional verification [25]. The goal is to replace a parallel component by a simpler *specification*,

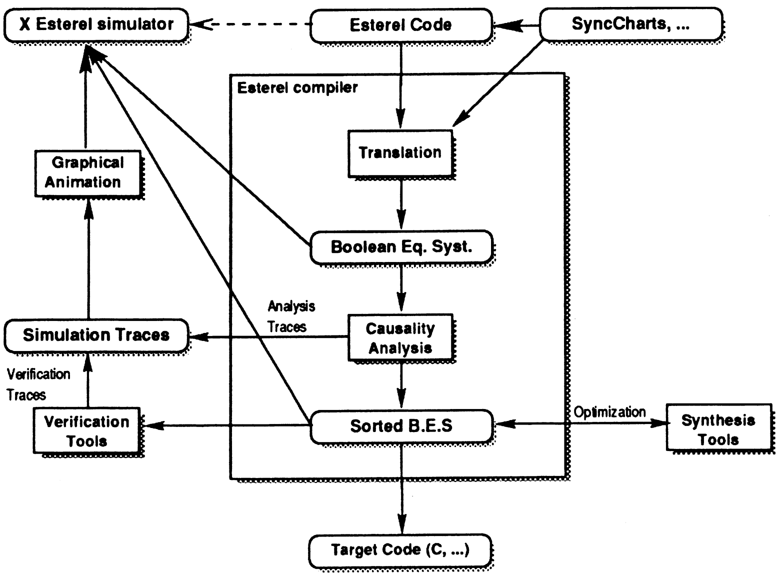

Fig. 2. ESTEREL compiler architecture.

with a division of the proof in two: prove that the component can indeed be abstracted by this specification regarding the rest of the system; prove that the simpler global system indeed satisfies a given property of interest. Further research is needed to develop good algorithms to extract the specification automatically when feasible, and to help performing the right abstractions, for example in the style of [24].

4.6. Compiler architecture and environment

Fig. 2 shows an overview of the ESTEREL environment. The main stream of compilation is framed in the central box. The left part concerns simulation and verification. The XES simulator and symbolic debugger displays simulation information by animating the source code. It also displays information generated by the constructiveness analyzer and the verifier, such as counter-example traces.
The ESTEREL environment is available on the web at
`http://www.inria.fr/meije/esterel`

5. The results of the experiments

5.1. Quantitative analysis

We report in Table 1 on complexity benchmarks for the three case studies. The meaning of columns and rows is explained below. In the sequel, we provide more

Table 1
Applications statistics

	MTC		Landing system			Side displays		
	Kern.	*MP*	*G&T*	*Contacts*	*Glob.*	*Funct.*	*Displ.*	*Glob.*
Inputs	35	20	20	60	160	55	90	110
Outputs	65	30	40	160	300	180	125	190
Modules	35	30	30	45	65	15	70	105
Instances	80	40	100	180	280	50	300	425
Lines	3200	2000	1800	3000	6500	3500	6700	15000
Reg	180	110	150	540	950	620	370	1360
Loc	3650	2500	3700	17000	25000	15000	15000	45000
Reachable	2×10^6	8×10^5	3×10^5	10^8	10^{14}	$2{\times}10^6$	$2{\times}10^8$	
Opt Reg	40	65	44			45	110	280
Opt Loc	980	1500	1600			2500	3500	20000
Properties	3				17			5
Transition	0.9 ms							0.8 ms

qualitative results and comments on the adequacy of Esterel for the treatment of these case studies, as found from experience.

The MTC final version of the Esterel code was split in two main modules: the scheduling *Kernel* and the Maintenance Panel (*MP*) management module.

For the Landing System case study we have three columns. The *G&T* column gives statistics about the smallest code: only the controller part of the application with consolidated sensor information as input. The *Contacts* code still contains the controller part only, but all the input signals from replicated contact sensors and the corresponding voter modules are present in the code. The *Global* code contains the controller and the *controlled* parts for closed-loop simulation and verification. A discrete event model of the mechanical devices (traps and landing gears) was also programmed with Esterel and linked to the controller part. This model of the environment is a faulty model, containing about a hundred input signals for simulation of faults (contact lock/unlock and motion freeze/unfreeze events). This modeling of the environment was mandatory to exercise the logic of alarms in the controller part.

The Side Displays case study consists of the integration (*Global* code) of two interacting modules: the *Functions* module and the side *Displays* management module. The Inputs and Outputs lines show that the interfaces contain respectively 150, 460 and 300 signals. For mission management systems, Dassault Aviation intends to develop Esterel programs with up to 2000 input/output signals. For instance, the man-machine interface of Dassault Aviation latest combat aircraft involves 700 commands available to the pilot.

The line named Modules gives the number of library generic modules defined in the application, whereas the line Instances gives the number of statically instantiated calls to these modules. Some generic modules such as voters are instantiated many times. Lines gives the number of Esterel source code lines. In the Side Displays management experiment, the 15 000 lines were generated using various preprocessors, including a prototype SyncChart editor [22] (see Fig. 5). Automatic code generation of Esterel programs inflates the number of source lines with respect to hand written applications.

The Reg line gives the number of registers (or latches) before any optimization was performed. This information is available after compilation into BES. Loc denotes the number of local signals used for instantaneous broadcast communication between modules at the circuit level. These numbers are large because the programs exclusively use pure signals. No data types or data handling functions were used in these programs because formal verification is presently limited to control parts of Boolean variables. A more concise style of Esterel programming could have been adopted, but at some expense regarding verification.

The Reachable line gives the number of reachable states computed for safety properties verification or cycles removal in the case of cyclic circuits. The state space depends on whether timer values are taken into account or not. For the landing system global model, the TiGeR system failed to compute the reachable states when the states included timer values. The computation were performed on a SUN Sparc 20 workstation with 128 MB of RAM and on a DEC Alpha with 500 MB. The 10^{14} value is the order of magnitude of the highest number of analyzed states, when every timer value is abstracted by one Boolean value. This kind of computation needed about 20 h.

The Opt Reg line gives the number of remaining latches after optimization. One notices drastic reductions, such as 575 removed registers out of 620, or 1080 out of 1360. Then the size shrinks down to (and often far lower than) this of hand-written low-level code.

The Opt Loc line gives the number of remaining local signals or wires of the circuit. One also notices spectacular decreases for the programs of the experiment based on the side displays.

We then give the number of formally verified properties. Dassault Aviation used the Path Linear Time temporal logic of TiGeR and Esterel synchronous observers to verify the properties.

The Transition line gives the average transition time of the generated C code. It was carefully measured only on the 68020 microprocessor during the MTC experiment. Before optimization the average transition time was 3.25 ms. The 0.8 ms value for the global side displays model was measured on an Ada generated code running on an Ultra Sparc 2 workstation. These two values are not directly comparable.

The size of the most optimized version (circuit level and C level optimizations) of the *Kernel* C code was 16.5 kB. Before optimization, the size was 49.2 kB. In the case of the landing system experiment, the size before optimization was 213 and 71 kB after. The optimization was performed by Cadence Research Lab as an informal collaboration.

So these experiments demonstrate that optimization plays a crucial role. A factor of 2–3 in time and space may be gained by various kinds of optimizers from crude Esterel generated code.

5.2. The MTC case study

The use of ESTEREL was initially to design a prototype specification, using the interpreter and the graphical user interface for early simulation. Fig. 3 shows the simulation panel of the scheduler module (*Kernel*) once the partitioning of the first global model was completed. This reactive program schedules nine tasks, represented by nine rectangles in the middle of the figure. Each line of these rectangles denotes an activation state of the task. EN_COURS means running, BLOQUEE means suspended, PRETE means ready, and so on. The grayed lines show the current state of the program. Several functional problems were uncovered during the prototyping phase of the specification. They were corrected on the real embedded software that was developed in parallel with the ESTEREL experiment. This early simulation of the scheduling policy and of all event handling aspects was impossible on the embedded software before final integration was reached. This was because the task activation logic was scattered in the nine tasks, and thus could not be activated until the final linking phase with the HPF on-line scheduler was completed. Separating control and data-handling aspects and early simulation of the control part were considered by the FTC software engineers as one of the main benefits of the synchronous programming methodology.

Once the specification was frozen, automatic embeddable code generation was attempted. The old compiler based on symbolic execution and explicit automata production ran out of memory. The new compiler based on BES production first rejected the program because of cycles in the combinational part of the circuit. Guided by

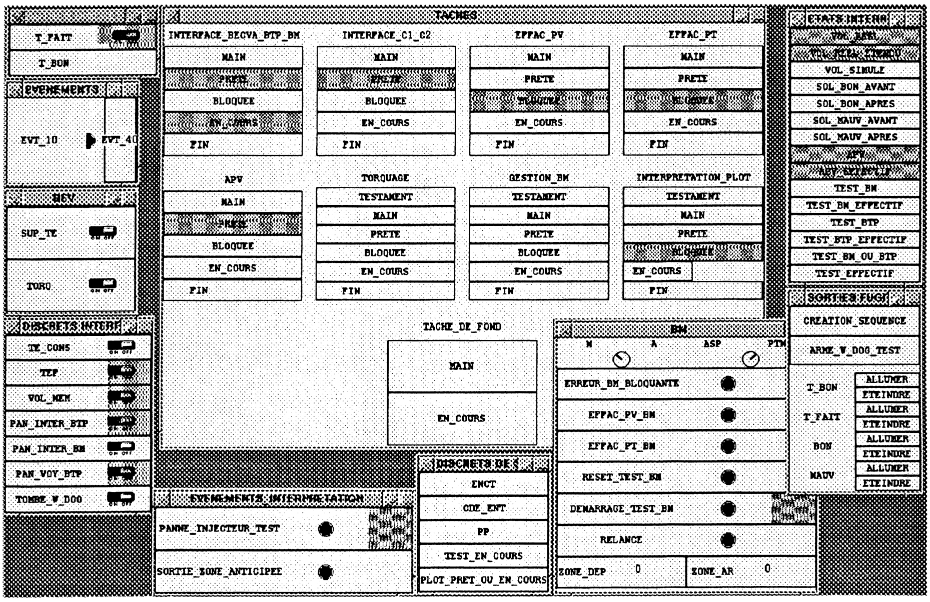

Fig. 3. Simulation panel of the *Kernel* module.

symbolic debugging informations on the cycles, the program was modified to remove them without changing its functionalities. Then, a safe sequential code was produced in a few seconds.

This code was analyzed by a commercial development tool for real-time programming (an emulator of the 68020). The average transition time estimation was 5.5 ms before optimization. This was far too much. The uncertainty on this estimation was 10%. The objective was less than 1 ms since the scheduler was designed to switch the running task about 10 times per 20 ms cycle. To meet this requirement, the code was split into a pure scheduling automaton *Kernel* and the rest of the reactive part of the MTC software (the *MP* module which is one of the nine scheduled tasks, named `GESTION_BM` in Fig. 3). The other eight tasks are pure data-handling (or "transformational") tasks. As was explained above, optimization succeeded in decreasing the automaton transition time until a value below the 1 ms objective was reached.

The two modules generated by the Esterel compiler were then integrated with the code of the tasks on the target embedded computer and the whole software was validated on a ground-based real-time testbench. The FCS software engineer involved with this experiment found the integration process straightforward. The behavior of the two modules observed in interpreted mode (prototyping phase of the specification) were identical to the behavior of the respective compiled and optimized versions of these modules. This behavioral invariance along the compilation and optimization process is *absolutely crucial* and was regarded as such by the FCS engineers and the technical managers at Dassault Aviation. What is at stake here is the seamlessness of the development process, from specification to coding and unit testing. This continuous process supported by automatic generation of full-fledged code from high level executable specifications (the Esterel instructions and the Statecharts diagrams) leads to a significant productivity increase. No quantification was performed, but a 30% reduction of the overall development time is guessed.

A few formal properties were verified with TiGeR on the global model of the specification (*Kernel* + *MP*), and then on the *Kernel* module. The mutual exclusion of the running tasks, given for free in the on-line approach, had to be proved on the global automaton. It was also verified that no task capable of triggering tests may be activated in flight. These verifications were made in early 1994 at Digital Equipment Paris Research Laboratory on a finite state machine with more than 10^{12} reachable states. This experience convinced the engineers at Dassault Aviation that BDD based symbolic model checking was a technological breakthrough to verify Esterel programs. Previous explicit methods were limited to automata with at most 10^5 reachable states.

5.3. The landing computer case study

This experiment was carried out in 1995 and was focused on formal verification. The main objective was to assess the power of the TiGeR BDD package since brilliant results had been obtained from the MTC case study. Starting from a risk analysis and a system level dependability study, 17 safety properties of the landing software were identified.

This time, complexity limits were reached in spite of TiGeR efficiency. The few properties (2 out of 17) needing closed-loop verification *and* counter values simultaneously failed be verified. TiGeR ran out of memory. The problem was circumvented by reducing the range of counters while preserving the ratios between the different timing constants of the program. This was not really satisfactory, but thanks to this trick no property remained unproved. A non-trivial error in the Esterel program was found by formal verification.

Two examples of formally verified properties are:

- no up (resp. down) motion of the landing gears without a previous up (resp. down) command. These two properties can be checked on the controller part only (open-loop verification), with abstraction of the value of the counters.
- Assuming that there is no breakdown in the environment (controlled part), the landing gears are locked in the down (resp. up) position at most 15 s after the up (resp. down) command was issued by the pilot. These two properties require closed-loop verification (hypothesis on the behavior of the environment) and the timer values in the states of the global finite-state machine.

The experiment was judged conclusive, but the opinion about the absolute power of BDD algorithms was slightly revised. The difficulties motivated R&D engineers at Dassault Aviation to start paying some attention to assume guarantee reasoning and compositional model checking.

5.4. The side displays management case study

This experiment represents nearly a third of a large reactive program, the development of which is undergoing at Dassault Aviation. This future program will gather in a single synchronous machine the cockpit management logic and the operational functions activation logic. Operational functions are air to air combat or ground attack for instance. These two logics are tightly coupled, hence the need for global analysis.

Fig. 4 shows two pages displayed on the two side screens of the cockpit. These screens are represented by the windows named `VTLG` and `VTLD`. The lower part of the figure shows other man-machine devices involved in the management of the side displays. Squared or grayed labels in these two pages are activated tactile commands. These commands control the activation of operational functions and the activation of sensors. For instance, the squared `RDR` label means “activation of the RaDaR”, “A/A” is the abbreviation of Air to Air combat.

In this experiment, three difficulties were encountered. The first one is related to memory mechanisms based on stacks. Pages formerly displayed on a screen as well as activated operational functions are memorized in some sort of three places “stacks”. These stacks may have easily been programmed in C, but this would have banned

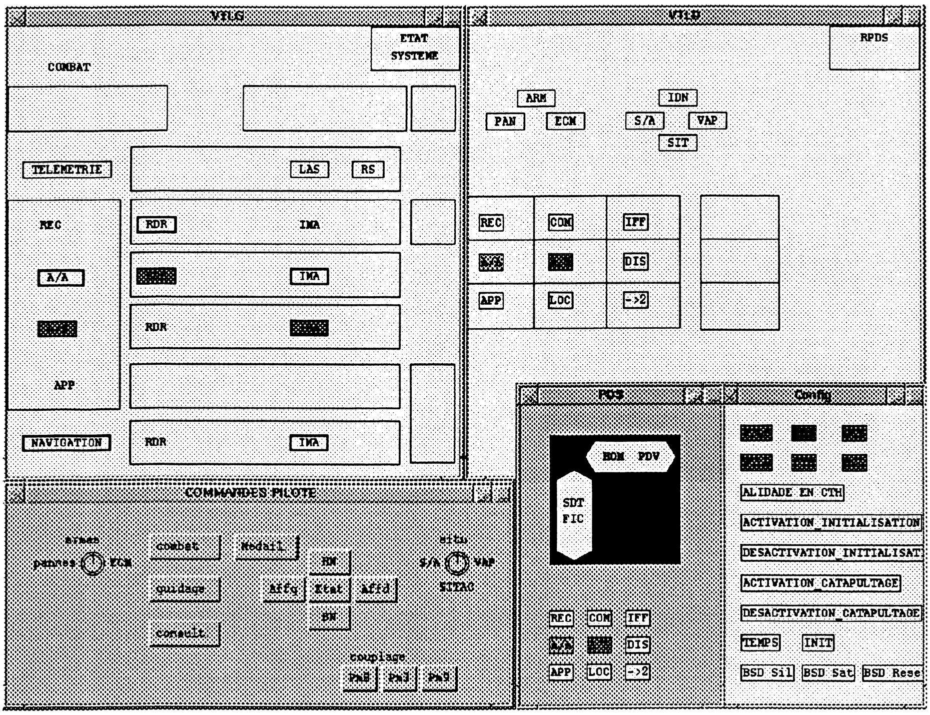

Fig. 4. Cockpit side displays.

formal verification. The Boolean encoding of the three so-called stacks in pure Esterel signals and latches was possible, but it was awful (see Fig. 5).

The second difficulty was related to cyclic BES generation. Several cyclic circuits were generated and the Sccausal causality analysis processor ran out of memory while attempting to transform them into acyclic circuits. This situation compelled Dassault Aviation to prototype modular compilation facilities to plug acyclic circuits in larger programs that Sccausal did not manage to process as a whole. This demonstrated the need for modular compilation of large Esterel programs in case of combinational cycles.

The third difficulty was the following one: global formal verification prior to any optimization phase turned out to be impossible. Using the compositional compilation utilities, a global program composing *optimized* partial circuits together was produced from the non-optimized component circuits and from the composition context. The reachable state space of the composition of the optimized components (a global locally optimized program) turned out to be computable. This observation motivated compositional optimization before verification.

The main conclusion of this case study was the absolute need for compositionality in the Esterel compilation and optimization processes. This was recognized as a critical issue for Esterel programming in the large by all the authors.

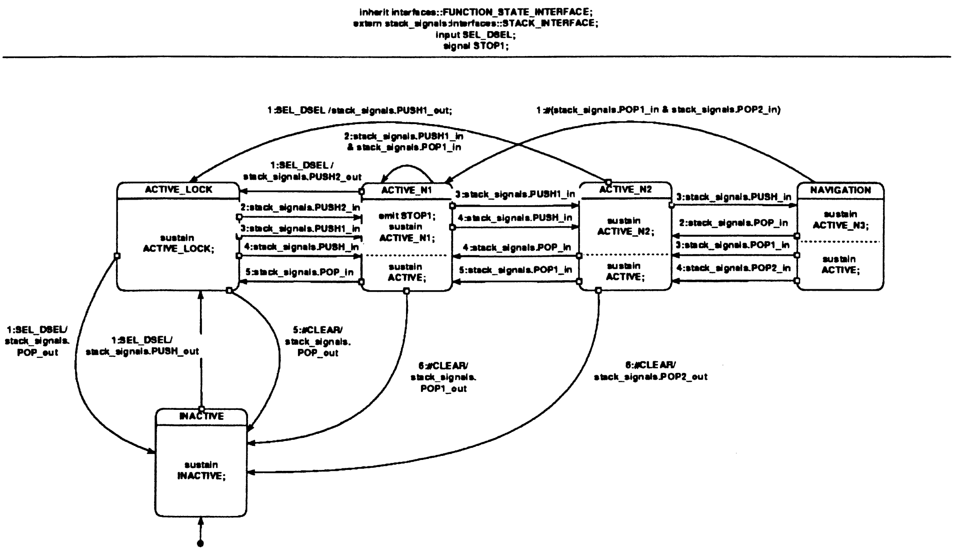

Fig. 5. A sync-chart specification example.

6. Conclusion and future work

Esterel is an industrial-strength formal method for medium-size reactive applications. Medium size means a few hundreds of I/O signals, a hundred of module instances, a few thousands of Esterel code lines, about 100 kB of embedded code per synchronous software unit. Nearly ten years of effort were needed to reach this point. The main challenge is to produce safe code from high-level specifications. The crucial steps were the translation into BES and the exploitation of the constructive semantics.

Beyond the limits of medium-size applications, compilation, optimization, and verification turn out to be intractable in a brute force global way. Compositionality is required and has proved promising on prototype experiments.

Esterel is used operationally at Dassault Aviation for parts of FCS software since applications are of small or medium size in this context. The use of Esterel for mission management systems is progressively introduced at Dassault Aviation too. As mission management reactive modules turn out to be large, compositionality in compilation, optimization and verification is now on the critical path.

References

[1] J.R. Abrial, M.K.O. Lee, D.S. Neilson, P.N. Scharbach, I.H. Sørensen, The B-method, in: VDM'91: Formal Software Development Methods, vol. 2, Lecture notes in Computer Science, Vol. 552, Springer, Berlin, 1991, pp. 398–405.

[2] C. Andre, Representation and analysis of reactive behaviors: a synchronous approach, in: Computational Engineering in Systems Applications, Lille (France), (1996) IEEE-SMC pp. 19–29.
[3] C. André, M. Bourdellès, S. Dissoubray, SyncCharts/Esterel: un environnement graphique pour la spécification et la programmation d'applications réactives complexes, Rev. du Génie Logiciel 46 (1997).
[4] G. Berry, Esterel on hardware, Philos. Trans. Roy. Soc. London A 339 (1992) 87–104.
[5] G. Berry, The Constructive Semantics of Esterel, Draft book available at: http://www.inria.fr/meije/esterel, 1998.
[6] G. Berry, The Foundations of Esterel, in: G. Plotkin, C. Stirling, M. Tofte (Eds.), Proof, Language and Interaction: Essays in Honour of Robin Milner, MIT Press, Cambridge, MA, 1998, to appear.
[7] G. Berry, G. Gonthier, The Esterel synchronous programming language: design, semantics, implementation, Sci. Comput. Programming 19 (2) (1992) 87–152.
[8] A. Bouali, XEVE, an Esterel Verification Environment, in: Proc. Tenth Int. Conf. on Computer Aided Verification CAV'98, Lecture Notes in Computer Science, UBC, Vancouver, Canada, June 1998.
[9] A. Bouali, J.-P. Marmorat, R. De Simone, H. Toma, Verifying Synchronous Reactive Systems Programmed in Esterel, Lecture Notes in Computer Science, Vol. 1135, Springer, Berlin, 1996.
[10] F. Bourdoncle, Abstract debugging of higher-order imperative languages, ACM SIGPLAN Notices 28 (6) (1993) 46–55.
[11] F. Bourdoncle, Efficient chaotic iteration strategies with widenings, Lecture Notes in Computer Science, Vol. 735, Springer, Berlin, 1993.
[12] F. Boussinot, R. de Simone, The Esterel language, Another Look at Real Time Programming, Proc. IEEE 79 (1991) 1293–1304.
[13] R.E. Bryant, Graph-based algorithms for boolean function manipulation, IEEE Trans. Comput. C-35(8) (1986) 677–691.
[14] J.R. Burch, E.M. Clarke, L. McMillan, D.L. Dill, J. Hwang, Symbolic model checking: 1020 and beyond, in: 5th IEEE Symp. on Logic in Computer Science, Philadelphia, 1990, pp. 428–439.
[15] E.M. Clarke, D.E. Long, K.L. McMillan, Compositional model checking, in: Proc. Fourth Ann. Symp. on Logic in Computer Science, Pacific Grove, CA, IEEE Computer Society Press, Silver Spring, MD, 5–8 June 1989, pp. 353–362.
[16] Th. Coquand, G. Huet, The calculus of constructions, Inform. Comput. 76 (1988) 96–120.
[17] O. Coudert, C. Berthet, J.C. Madre, Verification of synchronous sequential machines based on symbolic execution, in: Automatic Verification Methods For Finite State Systems, Grenoble, France, Lecture Notes in Computer Science, Vol. 407, Springer, Berlin, 1989.
[18] O. Coudert, J.-C. Madre, H. Touati, TiGeR Version 1.0 User Guide, Digital Paris Research Lab, December 1993.
[19] N. Halbwachs, Synchronous Programming of Reactive Systems, Kluwer, Dordrecht, 1993.
[20] N. Halbwachs, F. Lagnier, P. Raymond, Synchronous observers and the verification of reactive systems, in: Proc. 3rd Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93, Workshops in Computing, Springer, Berlin, June 1993.
[21] D. Harel, Statecharts: a visual formalism for complex systems, Sci. Comput. Programming 8 (3) (1987) 231–274.
[22] L. Holenderski, A. Poigné, Synchronie workbench, in: Proc. ATOOLS'98 Workshop on Analysis Tool Support for System Specification, Development and Verification, Malente, Germany, June 1998.
[23] L.J. Jagadeesan, C. Puchol, J. Von Olnhausen, Safety property verification of Esterel programs and application to telecommunications software, in: Computer-Aided Verification, Lecture Notes in Computer Science, vol. 939, Liege, Belgium, July 1995.
[24] K.L. McMillan, Verification of an Implementation of Tomasulo's Algorithm by Compositional Model Checking, Lecture Notes in Computer Science, Vol. 1427, Springer, Berlin, 1998.
[25] J. Misra, K. Mani Chandy, Proofs of networks of processes, IEEE Trans. Software Engng. 7 (4) (1981) 417–426.
[26] S. Malik, Analysis of cyclic combinational circuits, in: IEEE/ACM Int. Conf. on CAD, Santa Clara, CA, ACM/IEEE, IEEE Computer Society Press, Silver Spring, MD, November 1993, pp. 618–627.
[27] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, R.K. Brayton, A.Sangiovanni-Vincentelli, SIS: a system for sequential circuit synthesis, Technical Report, U.C. Berkeley, May 1992.
[28] E.M. Sentovich, H. Toma, G. Berry, Latch optimization in circuits generated from high-level descriptions, Proc. IEEE/ACM Internat. Conf. on Computer-Aided Design, November 1996.

[29] H. Sentovich, H. Toma, G. Berry, Efficient latch optimization using incompatible sets, Proc. 34th Des. Automat. Conf., June 1997.

[30] T. Shiple, G. Berry, H. Touati, Constructive analysis of cyclic circuits, Proc. Int. Des. Testing Conf. (ITDC), Paris, 1996.

[31] Esterel Team, The Esterel v5 Language Primer, Ecole des Mines/INRIA, available at `http://www.inria.fr/meije/esterel/`, Esterel Compiler Documentation, 1998.

[32] H. Toma, Analyse constructive et optimisation séquentielle des circuits générés à partir du langage synchrone réactif ESTEREL, Ph.D. Thesis, École des Mines de Paris, 1997 (in French).