

Specification and Verification of Interactors: A Tour of Esterel

Jean-Daniel Fekete, Martin Richard, and Pierre Dragicevic

École des Mines de Nantes
4, rue Alfred Kastler, La Chantrerie
BP-20722, F-44307 NANTES Cedex 3

Abstract. Esterel is a mature reactive language used to specify and implement real time and critical systems. In this article, we investigate its use to specify and verify interactors. We show that Esterel is a mature language with a sound semantics. Esterel programs can be compiled efficiently into several languages and a comprehensive set of tools is available to verify, test, translate into automata and simulate Esterel programs.

In this paper, we compare Esterel with several formalisms popular in the HCI community, and show that it competes favorably, although still sharing common problems. We show that Esterel fills a useful niche in the repertoire of languages and formalisms usable by the HCI community.

Introduction

There have been several articles presenting or comparing existing notations, languages, formalisms and environments [7,9,11]. However, up to now, none has considered the Esterel language and its environment [4].

In this article, we would like to contribute to the search of good formalisms to design and specify interactive systems, not to present new interaction styles or how new problems can be solved. We describe how Esterel can be used to specify and verify the interactor part of an interactive system. By *interactor*, we mean the portion of the interactive program that receives events — from input events as well as from graphic objects and from the kernel of the application — and dispatches them. There are several definitions of what an Interactor is: Accot et al. call that part “low level interaction”, Paternó and Harrison call them interactors too [27,19], although their interactors are more structured internally than what we describe here. In the Garnet toolkit [24], the term interactor is used with a similar meaning as here [23].

This article is organized as follows: section 1 describes the Esterel language and tools. Section 2 shows some examples of its use for specifying and verifying interactors. Section 3 discusses the relative strengths and weaknesses of Esterel compared to other languages and formalisms before concluding.

1 Esterel

Esterel is a reactive synchronous language which was designed for programming real time applications like embedded software in cars, control for power plants or planes, bus controllers or wristwatches.

As Berry says in [3], the *reactive* systems community says that systems fall into two distinct classes:

- *Interactive* (unfortunate naming) systems, where clients ask for accesses or resources that the system grants or allocates if and when possible. This class covers operating systems, data bases, networking, distributed algorithm etc. The computer or the network is the leader of the interaction, and clients wait to be served. The main concerns for specification and verification are deadlock avoidance, fairness, and coherence of distributed information.
- *Reactive* or *reflex* systems, where the computer rôle is to react to external stimuli by introducing appropriate outputs in a timely way. The environment is the leader of the interaction. Reactive systems are prominent in industrial process control, airplane or automobile control, embedded systems, audio or video protocols, bus interfaces, systems, signal processing and man-machine interfaces. In reactive systems, the pace of the interaction is determined by the environment, not by the computers. Most often, clients cannot wait. The main concerns are correctness, safety and timeliness. The term *reactive systems* was initially introduced by Harel and Pnueli in 1985 [17] where the authors propose a distinction between *transformational* and *reactive* systems.

Of course, large scale systems can have components of both kinds. For instance, driving an airplane is mostly reactive, while communicating with the ground is mostly interactive.

Interactive and reactive systems deeply differ on the key issue of behavioral determinism. Interactive systems are naturally viewed as being non-deterministic. Being the master of the interaction, the system is allowed to make hidden internal choice about if and when requests are answered, and the answer to a sequence of inputs needs not be unique. On the other hand, behavioral determinism is a highly desirable and often mandatory property of slave reactive systems: the outputs of the system should be uniquely determined by its inputs and possibly by their timing.

In classical and well-studied concurrent formalism such as Petri Nets or process calculi, non determinism is built-in. The formalisms are well-suited to interactive systems and not to reactive ones.

Synchronous languages are concurrent and deterministic. They are based on the perfectly synchronous concurrency model, in which concurrent processes are able to perform computation and exchange information in zero time, at least at the conceptual level. The synchronous model is well adapted to a very wide spectrum of computer applications, ranging from hardware circuit design to large-scale real-time process control, and including embedded systems, communication protocols, systems drivers, and user interfaces.

The perfectly synchronous model and languages appeared independently in the beginning of the 80's in different places. Esterel was defined by G. Berry in Sophia-Antipolis, Lustre was defined by P. Caspi and N. Halbwachs in Grenoble [16], Signal was developed by A. Benveniste and P. Le Guernic in Rennes [15]. In Israel, D. Harel introduced the Statecharts quasi-synchronous graphical formalism [18]. In Grenoble, F. Maraninchi defined the Argos formalism [22] where (restricted) Statecharts are fully synchronous.

1.1 The language

We will introduce Esterel through a small example presented by Berry in [3]: consider the following controller specification written in natural language:

*Emit the output O as soon as both the inputs A and B have been received.
Reset the behavior whenever the input R is received.*

As it stands, this simple specification is a little bit ambiguous. We additionally assume that nothing is to be done at initialization time and that input signals can be simultaneous, as it is common in hardware. Furthermore, in the case where R occurs, the output should not be emitted and only the resetting should be performed.

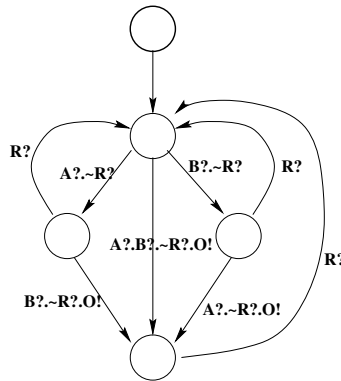


Fig. 1. A Mealy machine.

A common way of making such a specification formal is to draw the picture of an automaton (also call a Mealy machine) as in figure 1. The equivalent Esterel program is shown in figure 2.

In each reaction, each signal has a unique presence/absence status. For input signals, the status is given by the environment. For other signals, the status is absent by default, and it is set present by executing an **emit** statement. The **await** instruction

```

module ABRO:
input A, B, R;
output O;
loop
  [ await A || await B ];
  emit O
each R
end module

```

Fig. 2. ABRO sample program.

waits for A and terminates when A occurs. A parallel combination of two statements terminates instantaneously as soon as both statements are terminated. The time needed for synchronization is conceptually 0. Therefore, **await** A || **await** B terminates instantaneously as soon as both A and B have occurred. The sequencing operator “*p; q*” instantaneously transfers control to *q* when *p* terminates. Therefore O is emitted as soon as both A and B have been received. The **loop** *p* **each** R is a *preemption* operator. Its behavior is the following: the body *p* is immediately started and it runs freely until R occurs. At that instant, *p* is instantaneously killed, whatever its current state is, and *p* is immediately restarted afresh. In **loop** ... **each**, preemption is called *strong* because it has priority over body execution: at preemption time, the body is *not* executed in the instant. Therefore, if A, B, and R are simultaneously present, then O is not emitted, as requested by the specification. The behavior is exactly that of the automaton, but its expression is much clearer.

Synchrony expresses that the internal bookkeeping necessary to execute statement takes no time. The only constructs that take time are the ones explicitly required to do so, here **await** and **loop** ... **each**. Notice that synchrony of all other constructs is necessary to obtain the required behavior with no spurious silent move.

Nested Preemptions As Berry says in [3], in Esterel, the essence of programming consists of controlling the life and death of activities by using preemption structures. The nesting of preemption structures expresses preemption priority in a natural way. Figure 3 describes the basic training of an athlete [3]. The input events are Morning, Second, Step, Meter, and Lap. The full sequence is executed only if the lap is longer than 15 Second plus 100 Meter. If the lap is shorter than 15 Second, the athlete only runs slowly. If the lap is shorter than 15 Second plus 100 Meter, he never runs at full speed. The same happens if Morning arrives too soon.

Notice that any input can serve as a time unit in a preemption. In reactive programming, timing constraints don't have to be expressed in seconds. When driving a car, if there is an obstacle at 30 meters, the timing constraint is “stop in less than 30 meters”, no matter how long it takes.

```

module Runner:
input Second, Meter, Lap;
output ...; % not given here
every Morning do
  abort
  loop
    abort
    RunSlowly
    when 15 Second;
  abort

```

```

  every Step do
    Jump || Breathe
  end every
  when 100 Meter
    FullSpeed
  each Lap
    when 2 Lap
  end every
end module.

```

Fig. 3. The runner program.

1.2 Formal Semantics

We don't expect Esterel to be the only language and formalism used to specify and implement an interactive applications. We rather hope to use Esterel for describing the reactive part, and link it with other formalisms to describe other aspects. Hopefully, to simplify this linking, Esterel has a semantics that is written in several forms.

The *behavioral* semantics presented in [4] specifies the reaction of a program to an input as a single transition defined by an inference rule system.

The *operational* semantics is also presented in [4]. All programs found correct by that semantics are reactive and deterministic. The Esterel v3 compiler was directly implemented from this operational semantics.

The *constructive* semantics of Esterel [2] controls logical behavioral rules of the language constructs by means of two auxiliary predicates that determine for each input what a program *must* do or *cannot* do in terms of control and signal propagation. The constructive semantics can be presented in an equivalent *operational* form that is adequate for studying the execution of a reaction, or in an *denotational* form that defines the input/output function of a module — abstracting away all possible microstep orderings — and is synchronous and compositional. The Esterel V5 compiler is implemented directly from the constructive semantics.

1.3 Execution Environment

Esterel programs can be compiled into several languages: C, C++, ADA, C with Tcl/Tk library for simulation, Lisp and probably others (we have implemented a translator for Java.)

In the early years of Esterel, the compiler translated programs into an automaton, similar to the Mealy machine of figure 1. With this kind of model, some programs produced an automaton too large to be reasonably executed (usually when several input events were awaited in parallel.) With the new constructive semantics Esterel

programs can be compiled into sequential circuits expressed with BDD (*Binary Decision Diagrams* developed by Bryant [8]). With this model it is now possible to compile and execute almost any Esterel program.

Furthermore, Esterel has very few demands on the execution environment and offers generic mechanisms to bind its requirements into the underlying system (for example for concurrent process control). It is able to manipulate user-defined types and functions through external declarations: components generated from Esterel can easily be integrated in most existing environments.

1.4 Tools

Since reactive software is most often used for safety-critical applications, verification of program properties is fundamental. In real time software, we are interested in safety properties of the kind “wrong things never happen” and in bounded-response properties of the kind “something useful will happen before some time”. Liveness properties of the kind “something useful will happen some day” are usually much less important for reactive systems.

Many useful properties are data-insensitive and can be proved or disproved using only the finite-state control structure of a program. There are two techniques for such pure control properties: bisimulation reduction and verification using observers [3].

Bisimulation reduction of Esterel programs is performed by the FcTools system [6]. The implementation is very efficient for explicit automata, but as yet much less efficient for sequential circuits. *Verification using observers* is a very smart technique implemented by the TempEst system [20]. It allows the user to specify an observer using temporal logic properties [21]. These properties are then merged and compiled into an Esterel program and the generated observer is executed in parallel with the program to observe. We show an example of its use in section 2.3.

There are other tools around Esterel: autograph is for visualizing automata produced by the Esterel compiler. The XEVE tool, can perform automaton reductions according to some criterion in order to have multiple views of the automata. XEVE can also compute reachability on both the automata and the sequential circuits produced by the Esterel compiler. The Esterel compiler is supplied with a Tcl/Tk [25] library that allows inline simulation of Esterel programs.

To summarize this section, the Esterel language is simple and sound, Programs can be translated into several languages and run efficiently in existing environments with modest demands. Furthermore, a comprehensive set of tools is available to verify, visualize and simulate Esterel programs.

2 Examples

Now that we have seen the principles of the Esterel language, we present three examples of increasing complexity, providing an intuition of the language and its associated tools. We have taken our examples from the [10] and [1].

2.1 Click, Type and Display

Figure 4 shows a simple program written in Esterel that places typed text on the display at the points indicated by the mouse. It shows how concurrency is expressed in Esterel by having two processes called “mouse” and “kbd”. It also shows how external types can be expressed and used inside an Esterel program.

In the large, it receives 3 types of input events (DN, M and UP) and produces a typed event containing the position and value of the string to display. Since strings are very primitive in Esterel, the program relies on the user defined **str** type and four associated functions *append*, *empty*, *affect* and *test*.

The first module, named **mouse**, describes the behavior of the mouse, which repeatedly waits for a DN (down) event, then for M (move) event. This last event carries a value (the position of the mouse) that gets emitted by the `moveto` internal output event. Finally, the module waits for an UP event before resuming the loop. The **relation** line specifies that a DN event cannot happen at the same time as an UP event. It only simplifies the compiler work.

```
module mouse :
type point;
input DN, M(point),UP;
output moveto(point);
relation DN # UP;
loop
  await DN;
  await M;
  emit moveto(?M);
  await UP
end.

module kbd:
type str, result, point;
procedure append(str)(str);
procedure empty(str)();
procedure affect(result)(point, str);
function test(str, string) : boolean;
input K(str), moveto(point);
output typed(result);
var s:str, r:result in
  call empty(s);

loop
  await K;
  if (test(?K, "z")) then
    call affect(r)(?moveto, s);
    emit typed(r);
    call empty(s);
  else
    call append(s)(?K);
  end if;
end loop
end var.

module ClickTypeAndDisplay:
type point, str, result;
constant nulpoint:point;
input DN, M(point), UP, K(str);
output typed(result);
signal moveto(point) in
  emit moveto(nulpoint);
  [ run mouse || run kbd ]
end signal.
```

Fig. 4. The “Click, Type and Display” program.

The module **kbd** deals with the keyboard. Here, characters are appended into a string until “z” is typed. Then, the variable **r** of compound type **result** is filled

with the accumulated string and the last position emitted by the mouse module. The output event `typed` is emitted with this variable.

Finally, the module `ClickTypeAndDisplay` starts in parallel both the mouse and the kbd module after initializing the value of the `moveTo` event. The equivalent program, shown in figure 5 and described in [10] uses tail-recursion and functional programming idioms that we find much harder to read.

```
proc Mouse= DN? . M?p . moveTo!p. UP? . Mouse

proc Kbd(s) = K?c.
  if c==NewLine then
    typed!s . Kbd(emptyString)
  else
    Kbd(append(s,c))
  fi

proc Text(p) =
  < moveTo?p . Text(p)
  :: typed?s . drawString(s,p)? . Text(p)>

type = Mouse & Kdb(emptyString) & Text(nullPt)
```

Fig. 5. The “Click, Type and Display” program in Squeak

When compiled, this program boils down to the C++ class of figure 6, that can be subclassed to implement the output function `typed`.

2.2 Double Click

The “Double Click” program show in figure 7(taken from [10]) detects a single or a double click. It uses two timers to detect the end of a simple click and the beginning of the double click. In contrast to standard “double Click” implementations, this one does not triggers a `down` event when single click occurs and does not generate a `click` when a double click occurs. This kind of behavior is usually hard to program correctly with an imperative language. This program shows an interesting nesting of structures, but is otherwise quite simple.

2.3 Playing Music

This application is taken from [1]. It simulates a piano keyboard with a computer keyboard. Each time the user presses a key, a note is played and maintained until the key is released. Then, the sound decreases quickly and stops. A performer can play

```

class ClickTypeAndDisplay {
public:
    void I_DN ();
    void I_M (point)
    void I_UP ();
    void I_K (str);
    int reset();
    int run();
    ClickTypeAndDisplay();
    virtual ~ClickTypeAndDisplay();
    virtual void typed(result);
};

```

Fig. 6. C++ header from the “Click, Type and Display” program.

<pre> module DoubleClick: input UP, DN, end_clicktime, end_doubleclicktime; output down, click, doubleclick, start_clicktime, start_doubleclicktime; relation UP # DN, end_clicktime # end_doubleclicktime; loop trap T in await DN; emit start_clicktime; emit start_doubleclicktime; do await UP watching end_clicktime timeout emit down; exit T; end timeout; </pre>	<pre> do await DN; emit start_clicktime; watching end_doubleclicktime timeout emit click; exit T; end timeout; do await UP; watching end_clicktime timeout emit click; emit down; exit T; end timeout; emit doubleclick; end trap end. </pre>
---	--

Fig. 7. The “Double Click” program.

a chord by pressing several keys at the same time. The authors of [1] have made a formal model using high level Petri Nets. They implemented their model on the X Window system and discovered a surprising behavior: the expected pattern of events is at the top of figure 8 whereas the actual pattern is at the bottom. This is due to the X server sending synthetic events to implement the auto-repeat functionality. For this application, a plain note is replaced by an annoying vibrato!

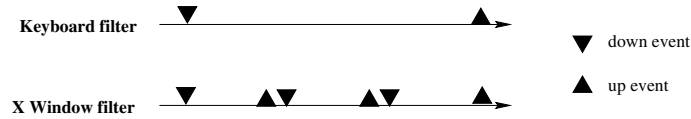


Fig. 8. The stream of events.

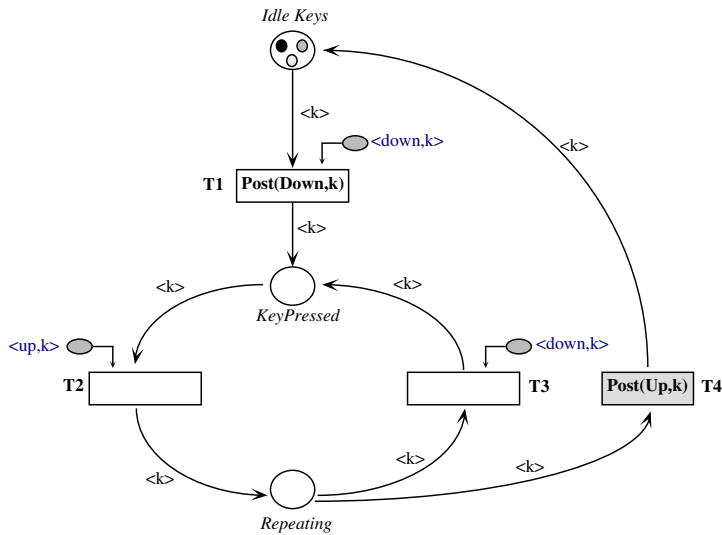


Fig. 9. The Petri Net solution to the playing music problem.

To solve their problem, the authors modeled a new solution using the Generalized Stochastic Petri Net of figure 9. The model must be read as follows: after a down event has been received the token corresponding to the key that has been pressed is set in the place *KeyPressed* and a Down is produced. Then only an up event can be received which removes the token from place *KeyPressed* and sets it into the place *Repeating*. This up event may be either a synthetic event or a real

one. In the case of a synthetic one, a synthetic down event should be received right after and thus the transition T3 will be triggered and the token set back to the place *KeyPressed*. If after t seconds no down event has been received, the application assumes that the up event was a real one. Figure 10 presents the same solution using Esterel.

<pre> module piano: input up(integer),down(integer), end_timer; output play(integer),stop(integer), start_timer; loop await down; emit play(?down); await up; trap T in loop emit start_timer; </pre>	<pre> do await end_timer; emit stop(?down); exit T watching down timeout await up; end timeout end loop end trap end loop. </pre>
--	--

Fig. 10. The “Playing Music” program.

In comparison to the Petri Nets solution, this one is readily compilable and executable, as well as verifiable with a rich set of tools.

Applying Esterel Tools Using XEVE, we found the shortest paths to emit output events:

- to emit `play`, we only need to receive a down event,
- to emit `stop`, we need to receive the sequence down, up and `end_timer`,
- to emit `start_timer`, we need to receive a down and an up.

AUTOGRAPH can produce the graphical representation of figure 10 from the output of the Esterel compiler or from a filtered version of the Esterel program produced by the XEVE tool.

With the TEMPEST tool, we can express some canonical safety properties written in propositional linear time temporal logic. For our example, we have checked the following property:

```

SA := { play -> down }
Safety00 := Always { SA }

```

it expresses the idea that if a down is received, a `play` event should be emitted at the same instant. This property is translated into an Esterel program, merged with the original program and compiled. XEVE can then verify reachability and reduction properties, namely that the event named `Safety00_VIOLATED` produced by

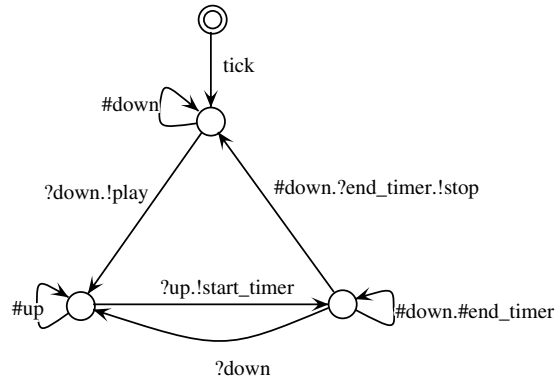


Fig. 11. Automaton produced by Esterel and visualized with AUTOGRAPH.

TEMPEST is reachable. If it is, the property is not verified. In our case, it is not reachable and the property is therefore true. If the formula had been $up \rightarrow stop$, the property would have been false.

3 Discussion

This tour cannot be complete without placing Esterel in the context of other notations, formalisms and languages used in the HCI community. We will rely on two previous articles : Philippe Brun and Michel Beaudouin-Lafon’s Taxonomy [7] and José Campos’s Review [9].

3.1 Esterel in a Taxonomy

In [7], the authors propose a taxonomy of formalisms starting from three domains or disciplines : Cognitive Science, Calculus Theory and the Theory of Categories. They further refine each domain to specific formalisms. Following their taxonomy, Esterel is in between Squeak [10] and Full LOTOS, *i.e.* classified as the join of two paths : Calculus Theory \rightarrow Graph Theory \rightarrow Automata \rightarrow Communicating Processes Algebra \rightarrow Esterel; and Theory of Categories \rightarrow Communicating Processes Algebra \rightarrow Esterel.

Each formalism is evaluated using an grid made out of 12 criterions, rated roughly from 0 to 1. Small squares give the ratings of Esterel, from white (0) to black (1):

- The *expressive power* is the ability of the formalism to describe the larger possible set of features of interactive systems. It corresponds to the ability to describe:
 1. the user’s tasks and actions, □

- 2. the interface state and system's feedback, □
- 3. the sequencing of actions, including time constraints, ■
- 4. the parallelism of actions, ■
- 5. the presentation of the interface, □
- 6. the management of user errors. □
- The *generative capabilities* is the ability of the formalism to generate a predictive performance analysis, or the code of the final system. It depends on the ability to:
 - 7 generate all or part of the final system (e.g, code generation), ■
 - 8 conduct a predictive analysis (e.g., cognitive load, task time execution), □
 - 9 prove system properties (e.g., termination, accessibility), ■
 - 10 derive generic interface functions (e.g. contextual help, cut and paste). . . □
- The *extensibility and usability* are two important criteria. They determine to a great extent the survival of the formalism in the real-world:
 - 11 extensibility, □
 - 12 usability. ■

Esterel compares here favorably to Full LOTOS [5], which was among the best formalism, along with XUAN [14] and Object-Oriented Petri Nets (OOPN) [26]. Esterel cannot faithfully be compared to XUAN because it does not deal with tasks at the human level. Esterel cannot be used to specify a whole application whereas Full LOTOS and OOPN can; Esterel is therefore much more focused and has a much more expressive syntax for interactors. Compared to Full LOTOS alone, Esterel is simpler and integrates better into existing environments since it is not aimed at running alone. Compared with OOPS, it can express much better nested preemptions. For example, adding exceptions in a “drag and drop” interactor specified in Esterel only requires a watchdog whereas it requires a substantial modification of the equivalent Net (a new transition should be created for each state during the manipulation).

3.2 Esterel in a review

In [9], Campos and Harrison “*review progress in the area of formal verification of interactive systems and propose a short agenda for further work.*” While evaluating strengths and weaknesses of four approaches, they provide a comprehensive list of arguments. We have been studying the validity of these arguments when applied to Esterel in similar contexts, *i.e.* if Esterel had been used in place of LOTOS as used by Paternó or Lustre in place of d’Ausboug et al. [12].

As stressed in the article, neither LOTOS, nor Lustre can really be used to verify high level properties because associated tools lack information about the underlying system. In this respect, interactive applications built using Esterel suffer from the same problem, inherent to the separation of interactors and their underlying environment. The problem can only be solved by merging the semantics of the interactors and of the underlying environments and use a unified approach. In that respect, Esterel has the advantage of its simplicity, compared to LOTOS or Lustre: its semantics are relatively small (each semantics is 4 to 10 pages long, see [13,2]).

A strong point mentioned in [9] concerns the inability to faithfully translate Full LOTOS into an automaton for verifying temporal and reachability properties (Full LOTOS guards are discarded in the process). The same problem arises with Esterel, because a conditional statement turn an Esterel program into a non-deterministic automaton where few things can be proved. In our point of view, a solution to this problem is to let the underlying environment do the conditionals and transform the results into Esterel input events, and apply conditionals after Esterel output events. For example, assume an Esterel interactor has to react in some way when the left mouse button is pressed and in some other way when the right mouse button is pressed. It is essential that two different Esterel events be produced (say DN_LEFT and DN_RIGHT) and not let Esterel do a test from the value carried by the DN event. Although this problem seems easy to solve in most cases and produces Esterel programs that are verifiable, it shifts part of the verification problem into the underlying environment.

3.3 Esterel as an Effective Tool

Some Esterel programs can be translated into several languages and rely on simple mechanisms to interact with the environment. The standard Esterel distribution comes with a compiler of Esterel into C. We have also built a translator of Esterel into classes both in C++ and Java. All the examples shown in this article have been translated into classes and instantiated into running programs. This feature is crucial to the acceptance of formalisms in the development process: Esterel can be used both to specify, verify and program interactors. Not only is the specification easier than with an imperative language, but the corresponding imperative program comes for free. A simple advantage is the fact that the “Playing piano” example has been specified and verified and then, an array of interactors (Java objects) have been instantiated and used inside a Java application without changing a line of the Esterel specification, using only object-oriented mechanisms (such as derivation) to specialize interactors to their running environment. Furthermore, a generic mechanism has been integrated within Esterel version 5 to start a concurrent process, be notified when it finishes and suspend or resume it. We have translated this facilities into Java and C++ and used this mechanism to express the behavior of long tasks or interactive feedback (like a noise occurring during a direct manipulation).

Finally, as far as we know, Esterel seems to have the richest set of tools available now, for testing, proving properties and generating code in several languages. A major advantage of Esterel compared to LOTOS is the availability of tools based on BDD (*Binary Decision Diagrams*) rather than automata to verify properties and generate code. Programs that used to produce hundreds of thousands of automaton states can now be efficiently checked and compiled. This changes the kinds of tools available to verify interactors. A classical example used by Xeve is to simulate the running environment using Esterel code and checking the resulting system for real-time properties of reachability, etc. This is possible with almost no limitation due to the use of BDD instead of an explicit automaton.

4 Conclusion

We have presented the language Esterel and some of its applications in the context of interactors specification and verification. Esterel offers some important advantages to LOTOS or Lustre in similar contexts, it is:

- simple,
- language neutral,
- offers several formal semantics,
- imposes almost no restriction on the size of programs,
- offers an extensive set of tools.

Unlike other formalisms, Esterel produces executable programs that are both small and efficient. We are currently using interactors programmed in Esterel, translated into C++ and Java classes.

Still, as a reactive language, Esterel cannot be used to specify, program and verify whole interactive systems. It should be used where it is best suited: the interactor part. In this respect, the relative small size of its formal semantics should simplify the task of connecting it formally to an underlying system, but this point still needs to be done.

References

1. Johnny Accot, Stéphane Chatty, Sébastien Maury, and Philippe Palanque. Formal transducers: Models of devices and building bricks for the design of highly interactive systems. In *Design, Specification and Verification of Interactive Systems (DSV-IS'97 Proceedings)*, Berlin, June 1997. Eurographics, Springer-Verlag.
2. G. Berry. *The Constructive Semantics of Pure Esterel*. Draft book, <ftp://ftp-sop.inria.fr/meije/esterel/papers/constructiveness.ps.gz>, 1996.
3. G. Berry. The foundations of ESTEREL. In *Proof, Language and Interaction: Essays in Honour of Robin Milner, G. Plotkin, C. Stirling and M. Tofte, editors*. MIT Press, 1998. To appear.
4. G. Berry and G. Gonthier. The ESTEREL synchronous language: design, semantics, and implementation. *Science of Computer Programming*, 19(2):87–152, 1992. The Esterel compiler is available at <http://www.inria.fr/meije/esterel/>.
5. T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems, North-Holland*, 14:25–59, 1987.
6. A. Bouali, A. Ressouche, V. Roy, and R. de Simone. The fc2tools set. In *AMAST'96*, volume 1101 of *Lecture Notes in Computer Science*, Munich, Germany, 1996. Springer Verlag. The fc2tools are available at <http://www.inria.fr/meije/verification/>.
7. Philippe Brun and Michel Beaudouin-Lafon. A taxonomy and evaluation of formalisms for the specification of interactive systems. In *People and Computers X - Proc. Conference on Human-Computer Interaction, HCI'95, Huddersfield (UK)*, pages 197–212. Cambridge University Press, August 1995.
8. R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, september 1992.

9. José C. Campos and Michael D. Harrison. Formally verifying interactive systems: A review. In *Design, Specification and Verification of Interactive Systems (DSV-IS'97 Proceedings)*, Berlin, June 1997. Eurographics, Springer-Verlag.
10. Luca Cardelli and Robert Pike. Squeak: A language for communicating with mice. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 199–204, July 1985.
11. David A. Carr. Toward more understandable user interface specifications. In *Design, Specification and Verification of Interactive Systems (DSV-IS'96 Proceedings)*, Berlin, June 1996. Eurographics, Springer-Verlag.
12. Bruno d'Ausbourg, Guy Durrieu, and Pierre Roche. Deriving a formal model on an interactive system from its uil description in order to verify and to test its behaviour. In F. Bodard and J. Vanderdonck, editors, *Design, Specification and Verification of Interactive Systems '96*, pages 105–122, Springer-Verlag/Vien, June 1996. Springer Computer Science.
13. Georges GONTHIER. *Sémantiques et modèles d'exécution des langages réactifs synchrones ; application à Esterel*. PhD thesis, Thèse de l'École des Mines de Paris, March 1988.
14. P. Gray, D. England, and S. McGowan. Xuan: Enhancing uan to capture temporal relationships among actions. In S. W. Draper Eds. G. Cockton and G. R. S. Weir, editors, *Proceedings of HCI'94: People and Computers IX*, pages 313–326, Glasgow, 1994. Cambridge University Press.
15. P. Le Guernic, M. Le Borgne, T. Gauthier, and C. Le Maire. Programming real time applications with SIGNAL. *Proceedings of the IEEE, Special Issue on Another Look at Real Time Programming*, september 1991.
16. N. Halbwachs, P. Caspi, and D. Pilaud. The synchronous dataflow programming languages LUSTRE. *Proceedings of the IEEE, Special Issue on Another Look at Real Time Programming*, september 1991. The Lustre distribution is available at <http://www.imag.fr/VERIMAG/SYNCHRONE/lustre-english.html>.
17. D. Harel and A. Pnueli. On the development of reactive systems. *Logics and Models of Concurrent Systems*, F13:477–498, 1985. NATO ASI Series.
18. David Harel. STATECHARTS: a Visual Formalism for Complex Systems, volume 8, pages 231–274. North Holland, june 1987.
19. Michael D. Harrison. A model for the option space of interactive systems. In *Engineering for Human-Computer Interaction: Proc. IFIP WG2.7*, Elsevier, 1992.
20. L. Jategaonkar Jagadeesan, C. Puchol, and J.E. Von Olnhausen. Safety property verification of ESTEREL programs and applications to telecommunications software. In *Proc. CAV'95, Lecture Notes in Computer Science*, Munich, Germany, july 1996. Springer Verlag. TempEst is available at <http://www.cs.utexas.edu/users/cpg/TempEst/>.
21. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer Verlag, 1992.
22. F. Marainchi. The ARGOS language: graphical representation of automata and description of reactive systems. In *International conference on visual Languages*, Kobe, Japan, 1991.
23. Brad A. Myers. Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, UI Builders, pages 211–220, 1991.
24. Brad A. Myers, Dario Giuse, and Roger Dannenberg et al. GARNET: Comprehensive support for graphical, highly interactive user interfaces. *COMPUTER magazine*, November 1990.

25. John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, USA, 1994.
26. P. Palanque, R. Bastide, C. Sibertin-Blanc, and L. Dourte. Design of user-driven interfaces using petri nets and objects. In F. Bodard, C. Rolland, and C. Cauvet, editors, *Conference on Advanced Information System Engineering CAISE'93*, Paris (France), June 1993. Springer Verlag, LNCS n.685.
27. Fabio Paternó. *A Method for Formal Specification and Verification of Interactive Systems*. Phd thesis, Departement of Computer Science, University of York, 1995.