Esterel Meets Java:
Building Reactive Synchronous Programs in Java

Jean-Daniel Fekete and Martin Richard
École des Mines de Nantes
4 rue Alfred Kastler, BP 20722, 44307 Nantes Cedex 03, France
{Jean-Daniel.Fekete,Martin.Richard}@emn.fr

December 15, 1998

Abstract

Esterel is a mature reactive language used to specify and implement real-time and time-critical systems. In this article, we investigate its use with Java and describe a Java binding for Esterel. We show that Esterel programs translate and integrate well into Java.

The real-time, critical part of the code can be specified with Esterel and verified with a large set of tools, before being turned into a Java class.

We believe embedded Java applications should adopt this kind of marriage to achieve a high level of reliability without sacrificing the comfort of the language and its environment, despite the limitations of Java in term of real-time support.

Introduction

The popularity of the Java language [AG96] is growing in the field of real-time and embedded systems. However, as a general purpose language, Java does not offer yet any specification or verification tools, nor the level of formalization required for safety-critical applications. Esterel [BG92] is a reactive language designed for real-time systems with a wide range of supporting tools.

In this article, we describe :

- the relationship and complementarity between Java and Esterel, or how to integrate a reactive program into an object oriented system;
- a Java binding for Esterel;
- results and limitations of that coupling.

This article assumes basic knowledge of the Java language but not of the Esterel environment. It is organized as follows: Section 1 describes the Esterel language and tools, section 2 describes the concepts and implementation choices of the Java binding, section 3 discusses results and limitations.

1 Esterel

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

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 80s in different places. Esterel was defined by G. Berry in Sophia-Antipolis. It is sound and has a semantics written in several forms: *behavioral* [BG92], *operational*, and *constructive* [Ber96]; this last-one having also a *operational* and *denotational* form.

1.1 The language

We will introduce Esterel through a small example. Consider the following controller specification written in natural language coming from [Ber98]:

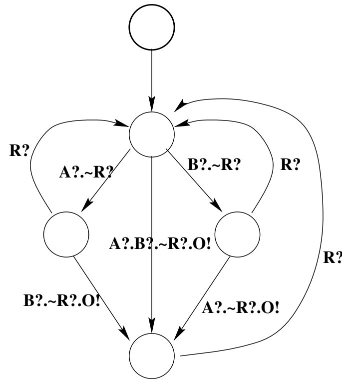

Figure 1: A Mealy machine.

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

Figure 2: ABRO sample program.

> *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.

A common for formalizing such a specification is to draw the picture of an automaton (also called 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 determined by the environment. For other signals, the status

is not defined, and it is set present by executing an **emit** statement. The **await** instruction 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 [Ber93]. 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* further executed. 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 formulation is much clearer.

1.1.1 Nested Preemptions

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 [Ber98]. 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.

1.2 Execution Environment

Esterel programs can be compiled into several languages: C, ADA, C with Tcl/Tk library for simulation, Lisp and probably others.

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

```
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.
```

Figure 3: The runner program.

several input events were awaited in parallel.) With the new constructive semantics Esterel programs can be compiled into sequential circuits [Ber92, Ber96] expressed in term of *Binary Decision Diagrams* (BDD) developed by Bryant [Bry92]. With this model it is now possible to compile and execute almost any Esterel program.

Furthermore, Esterel places 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). User-defined types and functions can be manipulated through external declarations: components generated from Esterel can easily be integrated in most existing environments.

1.3 Tools

Since reactive software is most frequently 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 ob-

servers [Ber98].

Bisimulation reduction of Esterel programs is performed by the FcTools system [BRRdS96]. 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 [JPO96]. It allows the user to specify an observer using temporal logic properties [MP92]. These properties are then merged and compiled into an Esterel program and the generated observer is executed in parallel with the program to observe.

There are other tools around Esterel: autograph allows automata produced by the Esterel compiler to be visualized. The XEVE tool can reduce automata according to some criterion in order to have multiple views of the automata. XEVE can also compute reachability properties on both the automata and the sequential circuits produced by the Esterel compiler. The Esterel compiler is supplied with a Tcl/Tk [Ous94] 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 Implementation

Traditionally, Esterel is translated into an imperative language like C. It relies on global functions to implement its interface and on global variables to implement its state. Efficiency is at premium.

In contrast, objet-oriented languages are more concerned with reuse and modularization, achieved by objects described by classes. In our binding, an Esterel program is translated into a Java abstract class. Table 1 shows mechanisms provided by Esterel and their realization into Java.

This is further refined in the remainder of this section. We start by describing the output of the Esterel to Java translator. We then show how to integrate the generated class into the Java environment and into a Java program.

The Java binding of Esterel is made of four layers:

1. the **Esterlet** class, root of all Esterel-generated classes,

Esterel	Java
program	class
input event	method
output event	abstract method
external function	inherited method
external procedure	inherited method
external type	inherited or global type
Task	thread and inherited method

Table 1: Translation of Esterel constructs into Java mechanisms

2. an adaptation class, inheriting from Esterlet, that defines all the required types, procedures and functions used by the Esterel program,

3. the Esterel-generated abstract class *per se*,

4. a concrete Java class that can be instantiated into a Java program.

2.1 The Esterel-generated class

Figure 4 shows a simple program written in Esterel, inspired from [CP85]. It uses all the special communication mechanisms allowed in Esterel to interact with the implementation language and runtime environment: external types, constants, functions, procedures and tasks.

The program places typed text on the display at the points indicated by the mouse. In the large, it receives 4 kinds of input events (`DN`, `M` and `UP` and `K`) 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 **StringBuffer** type defined by Java) 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 an `M` (move) event. This last event carries a value (the position of the mouse) that gets emitted by the `moveto` internal output event. The module then starts an asynchronous task called *scratch*

that returns a signal R when it finishes. This asynchronous task implements a time-out and is terminated either by the arrival of the R event or by the mouse UP event. In the first case, a string is also emitted. The **relation** line specifies that a DN event cannot happen at the same time as an UP event. This only simplifies the compiler's work.

The module kbd deals with the keyboard. Here, characters are appended into a **StringBuffer** until "z" is typed. Then, the variable r of the compound type **result** is filled with both the accumulated string and the last position emitted by the mouse module. The output event typed is emitted with this variable.

Finally, the module test2t starts the mouse and the kbd module in parallel after having initialized the value of the moveto event.

The resulting generated Java class is shown in figure 5. Input signals are mapped to methods calls (starting with the "I_" prefix), that the environment should perform. Output signals are mapped to abstract methods (here *typed*) called by the Esterel-generated program. The method *call_start_R* is used to start the asynchronous process associated with the R signal and produce the signal at the end of its execution (in this example, it starts the procedure called *scratch*).

The *run_once* method computes the next state from the received input events and fires output events according to the Esterel program. This method should be called by the environment after all the input methods have been called. It returns a boolean value indicating whether it has finished execution. An application has to derive this class and define the abstract *typed* method to be able to use it, as described in section 2.2.1.

2.1.1 The adaptation class

The Esterel program of figure 4 uses several user-defined types (**Point**, **StringBuffer**, **result**), a constant value (nulPoint), a function (*test*), procedures (*append*, *empty*, *affect*) and a task (*scratch*). Java has no notion of "top-level" variable, procedure or function, it only knows about methods and variables in classes. We therefore use a class derived from Esterlet to define the required types, constants, functions, procedures and tasks (except for Java primitive types and classes, defined in their own Java source file, like the **request** class defined here in its own file). Figure 6 shows the actual implementation.

```
module mouse:
type Point;
constant nulPoint:Point;
input DN, M(Point),UP;
output moveto(Point), K(string);
task scratch(Point)(string);
return R;
relation DN # UP;
var p := nulPoint : Point in
  loop
     await DN;
     await M;
     emit moveto(?M);
     abort
        exec scratch(p)("foo") return R;
     when
        case R do emit K("z")
        case UP
     end abort;
  end loop
end var.

module kbd:
type StringBuffer, result, Point;
procedure append(StringBuffer)(string);
procedure empty(StringBuffer)();
procedure affect(result)(Point, StringBuffer);
function test(StringBuffer, string) : boolean;
input K(string), moveto(Point);
output typed(result);
var s:StringBuffer, r:result in
   call empty(s)();
   loop
      await K;
      if (?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 test2t:
type Point, StringBuffer, result;
constant nulPoint:Point;
input DN, M(Point), UP, K(string);
output typed(result);
return R;
relation DN # UP;
signal moveto(Point) in
   emit moveto(nulPoint);
   [ run mouse || run kbd ]
end signal.
```

Figure 4: The “Click, type and display” Esterel program

```
import java.awt.*;
import java.lang.*;
import java.applet.*;
abstract class test2t extends Test2Utils
{
        public void I_DN () ...
        public void I_M (Point __V) ...
        public void I_UP () ...
        public void I_K (String __V) ...
        public void I_R () ...
        public int number_of_execs () ...
        public int number_of_execs_of_scratch () ...
        public ExecStatus getExec(int i) ...
        void call_start_R()
        public void reset_input () ...
        public void reset_exec () ...
        public test2t () ...
        public boolean run_once () ...
        public boolean reset() ...
        public abstract void typed(result dummy);
}
```

Figure 5: Java class produced from the “Click, type and display” program

```
import java.awt.*;
import java.lang.*;
import java.applet.*;

abstract class Test2Utils extends Esterlet {
  protected static Point _Point(Point s1, Point s2) {
      if (s1 == null)
        s1 = new Point(s2);
      else
        s1.setLocation(s2);
      return s1;
    }
  protected static Point nulPoint = new Point();
  protected static result _result(result r1, result r2) {
    if (r1 == null)
      r1 = new result(r2);
    else
      r1.affect(r2);
    return r1;
  }
  class empty_ref { StringBuffer p1; }
  protected static void empty(empty_ref r) {
    if (r.p1 == null)
      r.p1 = new StringBuffer();
    else
      r.p1.setLength(0);
  }
  class affect_ref { result p1; }
  protected static void affect(affect_ref r, Point p,
                               StringBuffer s) {
    if (r.p1 == null)
      r.p1 = new result();
    r.p1.p = new Point(p);
    r.p1.s = s.toString();
  }
  class append_ref { StringBuffer p1; }
  protected static void append(append_ref _ref,
                               String s) {
    _ref.p1.append(s);
  }
  class R_ref { Point p1; }
  void scratch(R_ref p, String s) {
    try {
      Thread.currentThread().sleep(1000);
    } catch(InterruptedException e) {}
  }
}
```

Figure 6: Adaptation class for the “Click, type and display” program.

The naming of types and functions is straightforward. Procedures and tasks keep their names, but sometimes require a special processing due to the call by reference semantics they should obey. For each procedure taking parameters by reference, a special class is defined that contains one instance variable per parameter. An instance of this class is created and filled before the procedure or task call. Its contents is then copied back into the Esterel-generated program afterwards. While not elegant, this mechanism correctly implements the call-by-reference semantics.

Finally, Esterel requires special procedures to implement the copying of user-defined types (the := operator). For the type **Point**, a function called *_Point* is called. Instead of using the same call by reference mechanism as for procedures, the method simply returns its “call-by-reference” argument, which is assigned back to the variable in the Esterel-generated program.

To help defining this adaptation class, the Esterel to Java translator generates a Java skeleton class, declaring all the used types, constants, functions, procedures and tasks.

```
import java.awt.*;
import java.lang.*;
import java.applet.*;

abstract class Esterlet {
  abstract class ExecStatus implements Runnable {
    public Thread getThread() ...
    public void run_once() ...
    public void reset() ...
  };
  public boolean run_all_once() ...
  public abstract int number_of_execs();
  public abstract ExecStatus getExec(int i);
  public abstract void reset_input();
  public abstract boolean run_once();
  public abstract boolean reset();
  public void run_execs () ...
}
```

Figure 7: The Esterlet class

2.2 The Esterlet class

Figure 7 shows the abstract **Esterlet** class, root of all Esterel-generated programs. It defines two methods to compute the next step of the Esterel program: *run_once* and *run_all_once*. The former is abstract and generated by the Esterel compiler. It triggers all the required output events but only the latter also triggers the asynchronous tasks, using the *run_execs* method. Asynchronous tasks are naturally implemented using the standard Java classes **Thread** and **Runnable**. All the other methods are auxiliary methods used to control an Esterlet object.

2.2.1 Using it in a Java program

When the Esterel program has been translated and the adaptation class has been defined, the Esterel-generated class can be used by simply sub-classing it like this:

```
test2t esterel = new test2t() {
  public void typed(result r) {
  processResult(r); }
};
esterel.I_DN();
esterel.run_all_once();
```

The Java program is responsible for mapping real input events into calls to the Esterel-generated objects. Several instances of such objects can coexist at the same time and be connected together to build a multi-level reactive program.

The Java binding is simple to use and maintains a nice separation of concerns between the reactive part of the program and the imperative part.

3 Results and discussion

We have applied this translator to several sample applications, from tens to thousands lines of Esterel and Java code. Our target application domain was human-computer interaction but we have also translated the Wristwatch Esterel sample program, originally written in Esterel and C.

Java and Esterel are indeed complementary:

- all the requirements of Esterel are fulfilled by Java (as seen in table 1),
- Java has an event model but no powerful mechanism to manage it, whereas Esterel is strong at managing events,
- some properties of Java sequential constructs could be verified using traditional methods, but concurrent constructs — such as event handling — are too difficult to verify directly. In contrast, Esterel programs can be verified before being translated into Java.

From a programmer's point of view, Esterel fills an important gap between the reception of Java events and their interpretation. The Java event model is well defined, but managing several events is typically hard to do in an imperative language. Mapping Java events into Esterel input events is trivial with our binding. Esterel is designed to translate events into actions. Then again, Java's imperative style is well adapted to performing actions. In this respect, Java and Esterel interact very well.

From a real-time point of view however, current Java implementations do not offer enough guarantees in term of time accuracy and performance. Notable problems are caused by the Java scheduler and the garbage collector. The current Java scheduler has no notion of real-time, and a process can be interrupted for an arbitrary amount of time. Also, garbage collection can occur at any time and last arbitrary long. Even if some tricks can be used to limit its effects (like explicitly calling the garbage collector at regular intervals to avoid long unexpected interruptions), Java offer no upper bounds for the delay.

Even in a controlled environment, where only one known Java program runs, garbage collection occurs, due for example to the production of events by Java.

Our experience shows that current Java programs can be used with a time accuracy around the tens of a second without resorting to any optimization trick. This accuracy is sufficient for interactive applications for example.

For real-time or safety-critical applications, some enhancements have to be made to the Java scheduler and garbage collector. However, we believe this could be added with very limited changes to the current Java interface.

Conclusion

In this paper, we have summarized the main characteristics of the Esterel environment and showed that it was very adequate to specify real-time critical systems. We have then described the Java binding of Esterel programs. This binding has four layers: the Esterlet root class, an intermediary adaptation class that derives from Esterlet, the Esterel-generated class and then finally the application of this last class into a Java program.

The Java run-time environment includes all the mechanisms required to run Esterel programs, enabling a seamless integration of Esterel-generated objects.

The marriage of Esterel and Java is simple yet effective and should provide embedded Java applications with the reliability and quality level they require.

However, to achieve real-time performance, some mechanisms should be added to Java:

- real-time threads,
- control of the triggering of the garbage collector.
- control the amount of time taken by the garbage collector.

The translator of Esterel C into Java, with sample code, can be downloaded at:
`http://www.emn.fr/info/perso/fekete/Esterel/`.
It requires the Esterel environment, available with its own license at:
`http://www.inria.fr/meije/esterel/`.

References

[AG96] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, 1996.

[Ber92] G. Berry. ESTEREL on hardware. *Philosophical Transactions Royal Society of London A*, 339:87–104, 1992.

[Ber93] G. Berry. Preemption and concurrency. In *Proc. 13th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 93)*, volume 761 of *Lecture Notes in Computer Science*, pages 72–93, Bombay (INDIA), dec. 1993. Springer Verlag.

[Ber96] G. Berry. *The Constructive Semantics of Pure Esterel*. Draft book, ftp://ftpsop.inria.fr/meije/esterel/papers/constructiveness.ps.gz, 1996.

[Ber98] 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.

[BG92] 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/.

[BRRdS96] 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/.

[Bry92] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, september 1992.

[CP85] L. Cardelli and R. 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.

[JPO96] L. Jategaonkar Jagadeesan, C. Puchol, and J.E. Von Olnhausen. Safety property verification of ESTEREL programs ans 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/.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer Verlag, 1992.

[Ous94] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, USA, 1994.