Modeling Reactive Systems in Java

Claudio Passerone, Roberto Passerone, Claudio Sansoè
Politecnico di Torino, Cadence European Labs
Torino, Italy

Jonathan Martin, Alberto Sangiovanni-Vincentelli
University of California at Berkeley
Department of EECS

Rick McGeer
Cadence Berkeley Laboratories
Berkeley, CA

Abstract

We present an application of the Java^{TM} programming language to specify and implement reactive real-time systems. We have developed and tested a collection of classes and methods to describe concurrent modules and their asynchronous communication by means of signals. The control structures are closely patterned after those of the synchronous language *Esterel*, succinctly describing concurrency, sequencing and preemption. We show the user-friendliness and efficiency of the proposed technique by using an example from the automotive domain.

1 Introduction

A reactive system continuously interacts with the environment, generally under some timing constraints. A convenient modeling paradigm for embedded systems is based on the notion of decomposition into a set of *concurrent processes* (see e.g. [7]). Processes can communicate with each other and with the environment either by means of synchronous protocols, such as *rendezvous* or Remote Procedure Call, or by means of exchange of signals. The latter method has been mostly used so far in Hardware Description Languages (e.g. VHDL or Verilog), and in the synchronous languages such as Esterel or StateCharts ([6, 11]). The signaling paradigm seems to be more flexible, because it also admits an *asynchronous* interpretation, that seems better suited to a heterogeneous implementation than strict synchronization (implied by rendezvous) or caller suspension (implied by RPC).

We thus considered an important feature allowing the designer to use the signal/process paradigm to specify the reactive control aspects of the design. In [3] a HW/SW co-design methodology is presented that uses a combination of Esterel and C as the specification languages; in the underlying model of computation, the modules are internally synchronous, and externally asynchronous, communicating using the signal/process paradigm. The functional verification is currently obtained by translating the specification to an intermediate format, and then by generating the appropriate description in the Ptolemy ([1]) simulation framework, where a graphical language is used to interconnect modules together. The non homogeneity in terms of specification languages and simulation environment has proven very inconvenient, especially in the earliest phase of the design, when only the functionality is to be tested and timing information is less of an issue. Moreover, a more structured programming language would be highly desirable, so that common mistakes would be caught and corrected earlier in the design.

Different programming languages have already been considered as alternatives. We have chosen Java as the language to specify the object hierarchy implementing the data computation, and we have added a class library to specify concurrent processes communicating *asynchronously* via a set of multicast signals. The system can therefore be simulated directly in the language of the specification. Choosing Java as the implementation language has several advantages over, e.g., C or C++: portability is not an issue, since it was taken into account very carefully during the design of the language itself; moreover, Java is a truly object oriented language, with all the advantages of object encapsulation and without the numerous historical legacy problems that plague C and C++.

Of course, the current performance of Java interpreters on a standard processor is worse (generally by an order of magnitude) than that of native object code. However, our goal is to use Java as a *specification* language for embedded systems, the actual code to be obtained by optimized software synthesis technique similar to the ones applied in the Esterel and C approach. In addition, there are emerging architectures, such as the *picoJava* processor ([10]) that can execute Java byte code at speeds that are competitive with state-of-the-art micro-processors running their own native code, and Just In Time compilers are making rapid performance improvements. There are also proposed real-time extensions to the Java Virtual Machine ([9]) that could extend the range of applicability of the modeling paradigm advocated in this paper to hard real-time applications.

In this paper hence we present an extension to the Java language ([2]) which provides methods to describe a reactive systems. The extension itself is written in Java and is a superset of the language; it can therefore be used with any architecture for which there exists an implementation of the Java Virtual Machine. Our goals were

1. to be able to have different concurrent modules,
2. to describe their interconnections and provide an event driven communication scheme, with methods to suspend and resume a process, or abort a certain task when a given condition occurs.
3. to address the problem of scheduling concurrent processes in a more predictable way than that provided by the language itself.

The approach is inspired, as discussed above, by the family of Synchronous Languages ([6]), and in particular by the Reactive C language developed by Boussinot *et al.* [4]. Our work differs from the former because we did not develop a new language, but rather extended an existing one. We can thus build on top of a wealth of experience and software tools. Our work also differs from the latter, because we tried to add a minimal amount of new constructs, in order to keep the extended language as simple and close to the original as possible.

The paper is divided as follows: Section 2 briefly outlines some of the features of the Java language used to develop this extension. Section 3 describes in detail the system and how we implemented it. Section 4 presents an application example. Section 5 concludes the paper and discusses opportunities for future work.

2 Java and Embedded Systems

Java already has many features that make it easy to program algorithms for embedded systems: in fact it supports multi–threading, synchronization among different threads when accessing shared resources and exception handling. Moreover, Java is a fully object oriented language, with all the constructs to handle complex data structures and flow control; being close to the C++ language, software developers don't have to learn a completely new paradigm.

What Java lacks is an easy way to make it react to stimuli: this can be achieved by using thread synchronization, but as the number of signals increases it becomes less tractable, especially when more than one input is expected at the same time; in fact communication among threads should be made explicit by instantiating a shared object and providing mechanisms to access it. The problem gets even more complex when we want a process to react when it is performing a long computation.

There are also several problems with the Java thread implementation:

1. it provides designers with a low-level control of parallelism and a great deal of freedom for developing parallel applications, which is usually not required, and often leads to bad designs with difficult-to-find bugs,
2. it lacks control structures for communication between threads: Java does support object locks which allow for the creation of synchronization points using *wait* and *notify*; however, Sun Microsystem's Java Virtual Machine (JVM) Specification ([8]) does not specify an order in which threads receive an object lock, so the behavior of the application may be dependent on the JVM being used,
3. it lacks a standard scheduling algorithm, so that the thread scheduling order may be different depending on the scheduling algorithm implemented by the JVM being used. This variation in thread scheduling results in designs that may execute normally on one system, but execute erroneously on another system.

Our work provides an alternative thread package for Java (PureSR) to support Synchronous/Reactive programming constructs, but with an asynchronous overall communication mechanism. A new scheduler is provided that produces always the same behavior across different platforms.

3 Reactive Java and PureSR

When developing the set of features that we wanted the language to include, we looked for a minimal set that would make it possible to derive others, and we wanted to implement them efficiently by leveraging the constructs already available in Java. We therefore developed a library of classes to program in Java with a reactive flavor. This essentially meant supporting some of the main features found in reactive languages, such as Esterel, Signal or Lustre ([6]). Given the imperative nature of both Java and Esterel, this language was chosen as a main reference, and in fact this work is strongly influenced by the available constructs in Esterel. However, it should be noted that although Esterel is a synchronous language, Java is not: processes are in fact not supposed to execute instantly, as in the synchronous hypothesis, and they do take time. Also no causality analysis is performed, and processes are free to perform data-dependent loops.

To make the language reactive we need ways to:

1. define, instantiate and interconnect modules,
2. send and receive events,
3. abort computation in case a given event is received.

We will describe how these points are accomplished in the rest of this section.

The package, called PureSR, consists of a collection of classes which implements the reactive methods by providing a framework for better threads. This framework is depicted in figure 1 and it consists of the classes `Braid` and `Fibre` plus two interfaces `Reactive` and `Shared` (not shown).

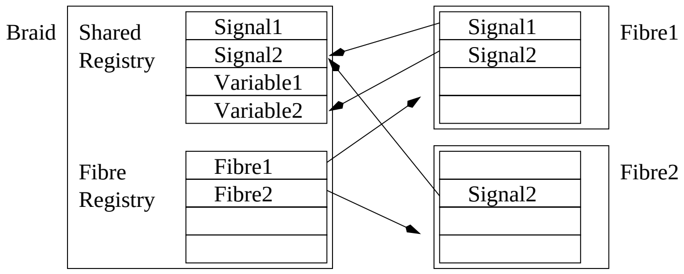

Figure 1: Reactive framework

The first point is obtained simply by the mechanism of class definition and object instantiation. A typical system using PureSR will contain a single object of class `Braid` and several objects of class `Fibre`: a fibre is analogous to a Java Thread, and implements one of the task of the system, and the braid is used to manage the fibres and communication, like an RTOS. `Fibre` implements the `Reactive` interface that defines the methods to be used to describe the behavior of the process and to register its objects with the braid, and can be subclassed by the developer to provide the required behavior. The `Braid` class contains method to register the fibres, to interconnect their objects, to communicate data and to control the execution using an equal-priority, round-robin scheduler. More scheduler methods can be defined later to support real time requirements. Shared objects can be connected in two ways:

- automatically, by using the braid methods `autoConnect` and `autoConnectAll`, which connect shared objects with the same name.
- manually, by using the `connect` method which allows explicit connection of two objects, regardless of their names.

Fibres communicate via shared objects. A shared object is obtained by instantiating any class that implements the `Shared` interface, which defines methods to set and retrieve the value. As shown in figure 1, a braid contains two registries: the first registry contains an entry for each shared object in the system, the second registry contains an entry for each fibre. Each entry in the first registry contains a shared object and a list of all fibres waiting on or watching the shared object, while entries in the second registry contain a pointer to the fibre and a list of all shared objects that the fibre is watching. In the example in figure 1, `Fibre1` communicates with `Fibre2` by setting the value `Fibre1.Signal1`; `Fibre2` then receives the information by reading the value of `Fibre2.Signal2`.

The communication is coordinated by using reactive constructs similar to those used in Esterel. They are implemented in the `Braid` class and are:

emit sets the value of the specified shared object and notifies all awaiting fibres of the change.

await halts the execution of a fibre until a new value has been emitted for the spcified shared object.

The third point above required us to be able to interrupt a process during the computation to deliver the event; this in turn may be dangerous, since it could leave a process in an unknown state. Our solution is to provide methods to establish *safe recovery* points (see e.g. [5], in which safe recovery points were also used for preemption), where actions can be taken to ensure a consistent behavior. This is accomplished by using the exception mechanism: the `AbortException` class implements an `Exception` that is thrown each time a shared object which is being watched by a fibre is emitted. Since an exception can be thrown only from within the process, the condition is only checked each time a signal is emitted or awaited in the block of code executed under the watching of an event. `AbortException` contains a single method called `check`, which allows the fibre to see which shared object caused the abortion. This feature allows the use of nested abortion clauses: the inner clause is called first, and if the emitted shared object does not match the one the clause is watching, then the exception is simply re-thrown for the outer clause to catch. The constructs implemented in the `Braid` class to handle exceptions are the following:

abortOn notifies the braid that the fibre is watching the specified shared object. Any subsequent call to reactive constructs made by the fibre will throw an AbortException if the shared object has been emitted.

endAbort notifies the braid that the fibre is no longer watching the specified shared object.

synch throws an AbortException if any shared objects that the fibre is watching have been emitted.

The **synch** is used to explicitly check the watched signals, and *must* be used whenever a block of code that should be run while watching a certain condition does not contain any **emit** or **await**. One can also insert **synch** methods automatically, in order to satisfy given abortion latency constraints.

Several classes that implement the `Shared` interface are included in the package. These classes implement the more common types of shared data: `SharedBoolean`, `SharedInt` and `SharedFloat` are used to share `Booleans`, `Integers` and `Float`, respectively, while `SharedSignal` objects share no data, but are used to signal other fibres that a pure event has occured.

4 A real life example

As an example of the application of our rapid prototyping methodology, we consider a car dashboard. The typical elements are:

- The *Speedometer* which displays the vehicle speed in the range of 0 km/hr to 260 km/hr.
- The *Fuel Gauge* which displays the fuel level from empty (E) to full (F).

- The *Coolant Gauge* which displays the water temperature ranging from ambient to 250 C.
- The *Lifetime Odometer* which displays the distance in the range between 0.0 km and 999,999 km.
- The *Resettable Trip Odometer* which displays the distance in the range between 0.0 km and 999.9 km.
- The *Seat Belt Warning Light* which is illuminated if the seat belt has not been fastened.
- The *Fuel Warning Light* which is illuminated when the fuel level becomes low.
- The *Coolant Temperature Light* which is illuminated when the coolant temperature is too high.

The system is modeled in a hierarchical fashion. There are five computation chains: the speedometer, the odometer, the fuel sensor, the temperature sensor, and the seat belt controller. Each computation chain is implemented in a fibre or a network of fibres, using the primitives described in the previous section.

The speedometer and odometer use the same strategy to display the speed and the kilometers traveled: a proximity sensor placed near the wheel shaft detects the passing of an indentation; at each passage a pulse is sent to the dashboard. There are usually four indentations on the shaft so that each pulse represents 1/4 of a revolution. The dashboard senses the pulses from the wheel and displays the current speed and the total amount of kilometers traveled.

The implementation style that we have chosen is closely patterned after an existing *Esterel* specification of the dashboard controller, that in turns derives by functional decomposition from a real industrial specification. The methods that have been described in the previous section have greatly helped in making this specification easy and faithful to the original. As an illustration of the style consider the *Seat Belt Warning Light* controller. The informal specification is as follows:

1. When the ignition key is turned on, wait for x seconds.
2. If during these x seconds the key is turned off or the seat-belt is fastened, then go to 1.
3. After x seconds have elapsed turn the alarm on and wait for y seconds.
4. If during these y seconds the key is turned off or the seat-belt is fastened, then turn the alarm off and go to 1.
5. After y seconds have elapsed, then turn the alarm off and go to 1.

The inputs to the belt controller are the events `ignition`, and `beltOn`; it interacts with the RTL model of the microcontroller timing unit to wait for a given number of seconds (specified by the `FIVE_SECONDS` constant in the code). A code fragment is shown in Figure 2. In lines 3-4, the shared objects are declared and assigned names. The `register` method (6-9) automatically registers all of the shared objects with the braid. The behavior of the `Seatbelt` fibre is provided by the `run` method (10-39). After initially setting the alarm to the OFF state (12), the fibre enters an infinite loop in which it waits for the ignition key to be turned on (16). It then starts a timer for five seconds and waits for it to finish (18,19). If the ignition key is still on and the seat belt is still unfastened after five seconds, a seatbelt alarm signal is emitted (26). The fibre again starts a timer for five seconds (28) and waits for it to finish while watching the ignition key and the seat belt (29-30). If the ignition key is turned off, or the seat belt is fastened before the timer runs out, an `AbortException` is thrown, and the `try...catch` clause (29-33) is halted, and the belt alarm is turned off (35).

The PureSR alternative thread package gives us true independence of the thread mechanism implemented by the particular Virtual Machine being used. This allows us to have a sort of execution sequence equivalence between different implementations, so that a virtual prototype running on a PC and the actual system would behave the same. This could not be accomplished by using the built-in Java thread scheduling mechanism, which would arbitrarily preempt and execute threads, leading to non determinism.

5 Conclusion and future work

An extension of the Java programming language towards reactive systems programming has been presented. It provides means of describing different concurrent modules and how they interact through an asynchronous event passing communication mechanism. Concurrency and scheduling are handled by the new thread package PureSR; reactiveness is achieved using thread synchronization and exception handling. Given the close correspondence of the Reactive Java constructs to those of the synchronous language Esterel, it is conceivable to use the latter for the implementation by using automatic synthesis technicques, like the one applied in [3].

In the future we would like to be able to experiment with different scheduling policies, rather than only the round-robin one. Moreover, currently designs using PureSR must be flat; if communication between braids was implemented, hierarchical designs would be feasible. This would facilitate the design of larger systems that re-use smaller systems as components.

References

[1] See http://ptolemy.eecs.berkeley.edu.

[2] K. Arnold and J. Gosling. *The Java programming language*. Addison Wesley, 1996.

```
public class Seatbelt implements Reactive {
   // Shared objects.
   public SharedBoolean ignition = new SharedBoolean("ignition");
   ...
   Braid b;
   public void register(Braid braid) {
      b = braid;
      b.autoRegisterObjects(this);
   }
   public void run() {
      // Initialize the alarm to off.
      beltOn.setValue(new Boolean(false));
      // Loop forever.
      while(true) {
         // Wait for the ignition key to be turned on.
         b.await(ignition);
         // Ignition key has been turned on, so wait for five seconds.
         b.emit(startTimer, FIVE_SECONDS);
         b.await(timerFinished);
         // If the ignition key is still on and the seat-belt has not
         // been fastened, turn on the alarm.
         Boolean belt = (Boolean)beltOn.getValue();
         Boolean ig = (Boolean)ignition.getValue();
         if(!belt.booleanValue() && ig.booleanValue()) {
            // Ignition is on and the seatbelt is off, so emit the alarm
            b.emit(beltAlarm, new Boolean(true));
            // Start the timer, and wait for it to time out.
            b.emit(startTimer, FIVE_SECONDS);
            try { b.abortOn(beltOn);
                  b.abortOn(ignition);
               b.await(timerFinished);
            } catch (AbortException e) {}
            finally { b.endAbort(beltOn); b.endAbort(ignition);}
            // Turn the alarm off.
            b.emit(beltAlarm, new Boolean(false));
            // Loop back and wait for conditions to change again.
         }
      }
   }
}
```

Figure 2: Reactive Java code for seat belt controller

[3] F. Balarin, E. Sentovich, M. Chiodo, P. Giusto, H. Hs ieh, B. Tabbara, A. Jurecska, L. Lavagno, C. Passerone, K. Suzuk i, and A. Sangiovanni-Vincentelli. *Hardware-Software Co-design of Embedded Systems – The POLIS experience.* Kluwer Academic Publishers, 1997.

[4] F. Boussinot, G. Doumenc, and J.-B. Stefani. Reactive objects. *Annales des Telecommunications*, 51(9-10):459–473, September 1996.

[5] P. Chou, E.A. Walkup, and G. Borriello. Scheduling for reactive real-time systems. *IEEE Micro*, 14(4):37–47, August 1994.

[6] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers, 1993.

[7] C. A. R. Hoare. Communicating Sequential Processes. In *Communications of the ACM*, pages 666–677, August 1978.

[8] T. Lindholm and F. Yellin. *The Java$^{\mathrm{TM}}$ Virtual Machine Specification.* Addison Wesley, 1996.

[9] K. Nielsen, 1997. See http://www.newmonics.com.

[10] SunSoft, 1996. See http://java.sun.com.

[11] M. v.d.Beek. A comparison of Statecharts variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems, Third International Symposium Proceedings*, September 1994.