Modeling Reactive Systems in Java

C. PASSERONE and C. SANSOÈ
Politecnico di Torino
L. LAVAGNO and R. McGEER
Cadence Berkeley Laboratories
J. MARTIN, R. PASSERONE and A. SANGIOVANNI-VINCENTELLI
University of California at Berkeley

We present an application of the Java™ 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.

Categories and Subject Descriptors: B.6.3 [**Hardware**]: Design Aids—*hardware description languages; simulation*; I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis; J.2 [**Computer Applications**]: Physical Sciences and Engineering—*electronics*; J.6 [**Computer Applications**]: Computer-Aided Engineering—*computer-aided design* (*CAD*)

General Terms: Design, Languages

Additional Key Words and Phrases: High level design, Java, prototyping, simulation

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, for example, Hoare [1978]). 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 signaling paradigm has been mostly used so far in Hardware Description Languages (e.g., VHDL or Verilog),

Authors' addresses: C. Passerone and C. Sansoè, Politecnico di Torino, Dipartimento di Elettronica, corso Duca degli Abruzzi 24, 10129, Torino, Italy; L. Lavagno and R. McGeer, Cadence Berkeley Laboratories, 2001 Addison Street, Berkeley, CA 94704; J. Martin, R. Passerone, and A. Sangiovanni-Vincentelli, Department of EECS, University of California at Berkeley, Berkeley, CA 94720.

and in Synchronous Languages such as Esterel or StateCharts [Halbwachs 1993; v.d.Beek 1994]. It is quite flexible, because it also admits an *asynchronous* interpretation that is 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 Balarin et al. [1997], 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 nonhomogeneity 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.

We have chosen Java [Arnold and Gosling 1996] 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 has several advantages: portability is not an issue, since it was taken into account 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++. It also performs extensive type checks at compile time, thus reducing the chances of runtime bugs. Modeling and debugging can be performed on almost any platform. However, Java is not suitable as it is to describe a reactive embedded system, and so we developed new features in the form of a class library to give designers new constructs to be used in the specification phase.

We envision a design flow in which the designer first uses Java with our classes to define an executable specification of the embedded software and of the environment in which it is going to be used. This has a number of advantages in several of the successive steps:

Documentation. Having such a description is very useful because it provides an unambiguous interpretation of the informal specification of the system.

Simulation. The system and its environment can be simulated to verify the correctness of the design. At this level, it is very important to allow partially nondeterministic specifications, at least for the environment. This can be achieved by free inputs to the environment model, that are tied to random number generators at run time. One of the main goals of this work is also to provide a consistent simulation when using different implementations of the Java Virtual Machine (JVM).

Implementation. After functional specification is complete, one can either use a different implementation language (e.g., when code and memory size are at an absolute premium), or use various compilation techniques on the same Java source. While standard Just in Time (JIT) compilers can achieve good optimization levels, we believe that better results can be obtained by restricting the set of Java constructs to those that have an Extended Finite State Machine interpretation (a good idea anyway in embedded system specification), which allows one to use superoptimization techniques, like the one described in Balarin et al. [1997]. Automatic synthesis can then produce C code as the software for the final implementation.

Garbage collection is a difficult issue in embedded systems, both because of memory consumption and because of problems in satisfying real-time constraints. We thus advocate a *fully static* object creation policy, thus making garbage collection unnecessary, while we wait for proposed real-time extensions to the JVM [Nielsen 1997] to make real-time garbage collection feasible.

Hence, in this paper we present an extension to Java which provides methods to describe a reactive systems. Our goals were to be able to have different concurrent processes and describe their interconnections in an event driven communication scheme, with methods to suspend and resume them. We also wanted 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 [Halbwachs 1993], and in particular by the Reactive C language [Boussinot et al. 1996]. 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.

Javatime [Young and Newton 1997] has a somewhat similar goal to our proposal, in that it is also using Java as a vehicle for system-level design. However, the Javatime approach is significantly different from ours. In particular, the Javatime Model of Computation is aimed at subsuming several others (among which Data Flow and Synchronous), and hence is just *reactive*, but does not make the synchronous hypothesis. Javatime "components" work asynchronously in "fundamental mode", that is they must react faster than their environment can perceive. However, there is no global scheduling imposed, and hence the overall specification is nondeterministic in general (the behavior depends on nondeterministic scheduling choices). The Javatime approach requires the designer to *successively refine* the Javatime MOC into an *implementation MOC* (e.g., synchronous for hardware implementation), that is defined as a Java package on top of the basic Javatime package. This refinement, in the case of synchronous implementation, effectively schedules the initial specification and determinizes it.

The paper is organized as follows: Section 2 briefly outlines some of the problems of Java implementation of threads that motivated us to develop this extension. Section 3 describes in detail the package and how we implemented it, showing its application in an example. Section 4 finally concludes the paper.

2. JAVA AND EMBEDDED SYSTEMS

Java already has many features that make it easy to program algorithms for embedded systems: in fact, it supports multithreading, 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 a program 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. There are also several problems with the thread mechanism in general, and with its implementation in Java in particular:

(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; however, Sun Microsystem's JVM Specification [Lindholm and Yellin 1996] 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.

For example, consider the differences between the "green thread" package supported for the Solaris platform, and the Windows NT/95 implementation of multithreading: in the first case, a thread is allowed to run until it voluntarily stops, or a higher priority thread takes control of the system; in the second case, each thread is allotted a time slice, after which it is preempted to allow others to run. Also among Posix thread (`pthread`) compliant implementations there may be differences: the IBM port of the JVM on the AIX Operating System has no support for priorities, while other implementations (i.e., IRIX) do. Moreover, even in the same JVM port one can sometimes choose between different thread packages, as in the

Solaris case, where a "native" preemptive implementation, with support for multiprocessor systems, can be used instead of the already mentioned "green thread" package.

It should be noted that some of these problems are inherent in threads, and some are inherent in portable threads. Efficient scheduling of threads is integral to the operating system on a given platform, and it is thus unsurprising that the Java language designers did not fix a scheduler in the language design. The resulting phenomenon of "write once, debug everywhere" is a natural consequence.

The problem with threads in general has long been known. Indeed, this problem led to the development of the "Synchronous/Reactive" languages in the 1970's and 1980's. The analysis of the day was simple, and still holds: most bugs in parallel software are due to uncoordinated updates of shared variables. Complex interactions, in particular locking and semaphores, are attempts to fix this problem, but are only partly successful. In a semaphore-based programming environment, the semaphores themselves become the shared variables, and the locking action the update. In this case, uncoordinated access to the semaphores leads to the characteristic problem of semaphore-based parallel programming, deadlock.

In contrast, if threads run in zero time, updates to a variable by a single thread are inherently coordinated. Semaphores are unnecessary—perhaps it is easiest for a programmer schooled in that tradition to imagine that, in reactive programming, all shared variables are controlled by a *single* semaphore, automatically obtained by a thread when it resumes execution and automatically relinquished when it waits on a signal.

Since concurrency is often a requirement in embedded software, and given the differences between JVMs, we wanted to address this problem by providing an alternative thread package for Java (PureSR), a robust mechanism for execution and update that produces always the same behavior across different platforms. In doing this we also decided to support Synchronous/Reactive programming constructs, but with an asynchronous overall communication mechanism. We achieved this goal with a package which is itself written in Java, and therefore can run on any platform; it was possible to do this because we provide our own classes for process instantiation, management and synchronization, which do not depend on the JVM thread scheduling algorithm.

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 [Halbwachs 1993]. Given the imperative nature of both Java and Esterel, this language was chosen as a main reference,

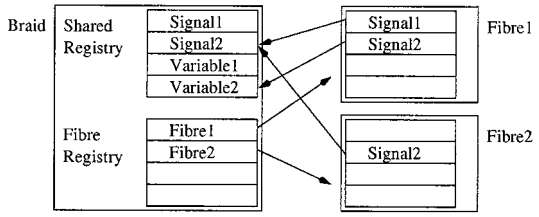

Fig. 1. Reactive framework.

and in fact this work is strongly influenced by the available constructs in Esterel. It should be noted that although Esterel is a synchronous language, Java is not; however, we do preserve synchronicity by running threads in cycles: in each cycle, all threads execute the code between two successive `await` statements, but the emitted values are not seen until the beginning of the next cycle. Therefore, the results do not depend on the order in which the threads are run within a cycle.

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 and interfaces 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).

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 tasks of the system, and the braid is used to manage the fibres and their intercommunication, 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 methods to register the fibres, to interconnect their objects, to communicate data and to control the execution using an equal-priority, round-robin scheduler. Each fibre is allowed to run until it calls one of the braid's communication methods. More scheduling methods can be defined later to support real time requirements.

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. Shared object can be connected either automatically or manually using methods provided by the braid.

The braid contains two registries: one holds an entry for each shared object in the system, and a list of all fibres waiting on or watching it; the other contains a pointer to each fibre and a list of all shared objects that the fibre is watching. The braid also implements the reactive constructs used to coordinate communication:

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 specified 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 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 to a watching fibre each time another fibre emits to a shared object. 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 rethrown 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 **await**. One could also insert **synch** methods automatically, in order to satisfy given abortion latency constraints.

As an example of the coding style using PureSR, consider the *Seat Belt Warning Light* controller of a dashboard, whose code is shown in Figure 2. The informal specification is as follows:

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

```
1:  public class Seatbelt implements Reactive {
2:    ...
3:    public void run() {
4:      beltOn.setValue(new Boolean(false));              // Initialize the alarm to off
5:      while(true) {                                     // Loop forever
6:        b.await(ignition);                              // Wait ignition key on
7:        b.emit(startTimer, FIVE_SECONDS);               // Start timer ...
8:        b.await(timerFinished);                         // ... and wait 5 seconds
9:        Boolean belt = (Boolean) beltOn.getValue();     // Check belt ...
10:       Boolean ig = (Boolean) ignition.getValue();     // ... and ignition
11:       if (!belt.booleanValue() && ig.booleanValue()) { // If key on and belt not fastened
12:         b.emit(beltAlarm, new Boolean(true));         // Turn the alarm on
13:         b.emit(startTimer, FIVE_SECONDS);             // Start timer ...
14:         ABORTON(beltOn, ignition) {
15:           b.await(timerFinished);                     // ... and wait 5 seconds, but ...
16:         } ENDABORT(beltOn, ignition);                 // ... watch beltOn and ignition
17:         b.emit(beltAlarm, new Boolean(false));        // Turn the alarm off
18: } } } }
```

Fig. 2. Reactive Java code for seat belt controller.

(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 another fibre, modeling a programmable timer, to wait for a given number of seconds. The behavior of the `Seatbelt` fibre is provided by the run method (3–18). After initially setting the alarm to the OFF state (4), the fibre enters an infinite loop in which it waits for the ignition key to be turned on (6). It then starts a timer for five seconds and waits for it to finish (7,8). If the ignition key is still on and the seat belt is still unfastened after five seconds, a seatbelt alarm signal is emitted (12). The fibre again starts a timer for five seconds (13) and waits for it to finish while watching the ignition key and the seat belt (14–16). If the ignition key is turned off, or the seat belt is fastened before the timer runs out, the alarm is immediately turned off (17). Note that ABORTON (14) and ENDABORT (16) are just macros used to simplify the syntax, and are expanded by a preprocessor to a `try...catch` clause, which uses the standard Java exception mechanism.

The entire dashboard application in Java runs approximately four times slower than a similar implementation in C, automatically synthesized from the original Esterel specification.

4. CONCLUSIONS

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.

Our alternative thread package gives us true independence of the thread mechanism implemented by the particular JVM 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 nondeterminism. Moreover, in our implementation there is no penalty with respect to the Java language itself regarding the time needed to perform a context switch. In fact, we rely on the Java mechanism, but we ensure that at most one thread is active at any given time in order to achieve a deterministic schedule.

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 techniques.

REFERENCES

ARNOLD, K. AND GOSLING, J. 1996. *The Java programming language.* Addison-Wesley, Reading, Mass.

BALARIN, F., SENTOVICH, E., CHIODO, M., GIUSTO, P., HSIEH, H., TABBARA, B., JURECSKA, A., LAVAGNO, L., PASSERONE, C., SUZUKI, K., AND SANGIOVANNI-VINCENTELLI, A. 1997. *Hardware-Software Co-design of Embedded Systems—The POLIS experience.* Kluwer Academic, Boston, MA.

BOUSSINOT, F., DOUMENC, G., AND STEFANI, J. 1996. Reactive objects. *Ann. Telecommun. 51*, 9–10 (September), 459–473.

HALBWACHS, N. 1993. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers.

HOARE, C. A. R. 1978. Communicating Sequential Processes. *Commun. ACM 21*, 8 (Aug.) 666–677.

LINDHOLM, T. AND YELLIN, F. 1996. *The Java™ Virtual Machine Specification.* Addison-Wesley, Reading, Mass.

NIELSEN, K. 1997. See http://www.newmonics.com.

V.D. BEEK, M. 1994. A comparison of Statecharts variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems, Third International Symposium Proceedings* (Sept. 1994). Springer Verlag, New York, 128–148.

YOUNG, J. AND NEWTON, R. 1997. Embedding programs in the Java language in the synchronous model of computation through the process of successive, formal refinement. In *Proceedings of the International Conference on Computer-Aided Design* (Nov. 1997).

Received February 1998; revised August 1998; accepted September 1998