Reactive Objects

Johan Nordlander, Mark P. Jones, Magnus Carlsson, Richard B. Kieburtz, and Andrew Black[*]
OGI School of Science & Engineering at OHSU, 20000 NW Walker Road, Beaverton, OR 97006.

Abstract

Object-oriented, concurrent, and event-based programming models provide a natural framework in which to express the behavior of distributed and embedded software systems. However, contemporary programming languages still base their I/O primitives on a model in which the environment is assumed to be centrally controlled and synchronous, and interactions with the environment carried out through blocking subroutine calls. The gap between this view and the natural asynchrony of the real world has made event-based programming a complex and error-prone activity, despite recent focus on event-based frameworks and middleware.

In this paper we present a consistent model of event-based concurrency, centered around the notion of reactive objects*. This model relieves the object-oriented paradigm from the idea of transparent blocking, and naturally enforces reactivity and state consistency. We illustrate our point by a program example that offers substantial improvements in size and simplicity over a corresponding Java-based solution.*

1 Background

In the traditional view of programming, the program is assumed to be the master of its environment, and interaction with the environment is accordingly expressed in terms of the subroutine abstraction. This programming model dates back to the early age of batch-oriented computing, when programmers saw a need to abstract away from low-level details of peripheral devices—such as card-readers and line-printers—that were used at the time. The key idea here is that blocking of execution is made *transparent*; that is, the programmer is supposed not to be interested in knowing whether a subroutine obtains its result by some internal computation, or by means of synchronization with an external device.

Despite many advances in language design, this simple, traditional view of I/O still prevails in contemporary object-oriented languages. But modern software executes in much more complex environments, with interactive point-and-click graphics, ubiquitous networks, multiple threads of activity with inter-thread communication and sharing, and so on. Embedded systems fit this description perhaps even better, with their rich variety of asynchronous input sources, and often clock-driven concurrent processes. In such environments, the utility of the traditional, batch-oriented view of interaction deteriorates rapidly.

The central problem is that, if external input is obtained as the results of certain subroutine calls, a program must make a choice as to what subroutine to call, and hence make a premature commitment to which event it will accept next. But the order of external events is seldom under program control, so a naive adherence to the batch-oriented I/O model quickly leads to programs in which events are either missed, or else randomly reordered in time.

The common pattern of an *event loop* in sequential object-oriented software is an attempt to reduce the rigidity of the traditional I/O model. However, an event-loop solution requires that all events of interest are already encoded externally and posted to a common queue, which may not always be the case. For example, the Java Abstract Window Toolkit (AWT) uses a common event queue for mouse-clicks, key-presses, and other GUI-related events [6]. Network sockets, on the other hand, are not handled by the AWT. So a program that needs to simultaneously monitor GUI events as well as network packets must still battle the limitations of the traditional I/O model: how to wait for multiple asynchronous events – all modelled as the results of distinct method calls to the environment – at the same time?

A standard approach in scenarios like this is to allocate a unique *thread of execution* for each potentially blocking operation, writing the code for each thread as if only one future event mattered. On the surface, such a strategy might appear to simplify the design task, because each thread now fits the traditional, batch-oriented I/O model quite well. However, in any non-trivial application, where the specified reactions are not completely independent, the original problem of coordinating inputs from multiple, asynchronous sources has now moved to another part of the program: namely, the thread in which the results of the simple blocking threads must be coordinated. To solve this problem one must of course face all the well-known problems of concurrent thread programming — assuring thread safety and state consistency, while also

[*]email: {`nordland,mpj,magnus,dick,black`}`@cse.ogi.edu`

ensuring liveness and avoiding deadlocks.

Notice, though, that in the AWT-plus-sockets scenario sketched above there is nothing that actually suggests concurrent *execution*; what the extra threads achieve is really the ability to perform concurrent *blocking*. While concurrency is an important tool in real-time programming, being forced to use it just to circumvent an inappropriate I/O model is not satisfactory. Also note that an abstract object model of the example would probably consist of just one simple object, equipped with methods corresponding to the events it is required to handle: mouse clicks, key presses, and packet arrivals. A central argument of this paper is that the heavy encodings needed to turn even simple event-driven models into working code is most unfortunate, and is an important factor behind the perceived complexity of concurrent object-oriented programming.

The current practice of using threads to circumvent the traditional I/O model is extremely fragile, in the sense that an accidental call to a blocking operation in the middle of an event-handler will immediately destroy the responsiveness of that thread. That is, transparent blocking makes responsiveness a delicate property that can only be upheld by careful programming, requiring knowledge not only of the complex APIs that encode events and threads, but also of all operations which potentially might block, and which thus must be avoided. As an illustration, consider the following quote from the documentation for Java's New I/O library [9]:

> That a selection key indicates that its channel is ready for some operation is a hint, but not a guarantee, that such an operation can be performed by a thread without causing the thread to block. It is imperative that code that performs multiplexed I/O be written so as to ignore these hints when they prove to be incorrect.

Clearly some special skill or rigorous discipline is required to navigate safely through such dangerous waters. It is especially noteworthy that this comment concerns a new library with pretensions to make event-driven Java programming significantly easier than before.

The idea of transparent blocking takes its most sophisticated form in the remote-procedure-call paradigm, or remote-method-invocation (RMI) as it is called using Java terminology. Again, hiding the intricacies of synchronization with a remote machine under a familiar subroutine-like interface seems attractive at first, because it makes the code of distributed programs look quite similar to code written for use in a strictly local context.

However, the similarity is deceptive. A defining aspect of a distributed system is usually that it is subject to *partial failure*; that is, programs are expected to continue running even if a remote server is down, broken, or otherwise unaccessible. Contrast this to failures directly affecting the local node: here the failure of one component is equivalent to the whole node going down. So in order to hide distribution, the RMI paradigm must also hide the possibility of partial failures. What this means is that failure of any remote machine in an RMI setup is equivalent to a total system failure. In practice, RMI-based programs can only regain some form of robustness by protecting the remote invocations by timeouts and exception handlers. However, this of course also makes the RMI paradigm considerably less convenient (and distribution less transparent) [8].

2 Reactive objects

The conclusion we have drawn from the problems associated with transparent blocking and batch-oriented I/O is that significantly more robust event-based software can be obtained by abandoning indefinite blocking altogether, and letting an event-driven design permeate the whole programming model. Thread packages, design patterns, and various middleware layers can only do so much to alleviate the programmer from a fundamentally computer-centric view, and they cannot help at all with enforcing responsiveness as long as every method call has a potential of blocking.

Our alternative programming model takes as its starting point the intuition behind the classical object-oriented paradigm: objects are autonomous, objects maintain a state, objects have methods, methods execute in response to messages. The main step towards a reactive variant of this model is to relieve the classical model from any ties to the traditional way of viewing I/O. In its place, a more orthodox object-oriented scheme of interaction can be devised:

- *input*—the environment calls a method of a program object
- *output*—the program calls a method of an environment object

Method calls in these categories do not just carry data, they can also be seen as representing the actual input and output *events* themselves. Notice in particular the asymmetry between input and output that results from this scheme: output is a concrete *act* of the program, while input is modelled as a passive capability to *react*. In other words, objects have full control over the output events, but leave the input events to be scheduled by the environment.

The autonomy and integrity of objects is essential to this view, though. Just as it is usually beneficial to view real-world objects as having a certain level of atomicity of operation, so is it essential to keep software objects from becoming invaded by multiple method invocations at the same time.

On the other hand, the concurrent operation of distinct objects is a natural aspect of the real world, and we wish our reactive objects to support the same intuition. We therefore take it as a semantic foundation of our model that

> *every object is an autonomous unit of execution that is either executing the sequential code of exactly one method, or passively maintaining its state.*

The combination of inter-object concurrency with internal sequential execution effectively makes a reactive object a union of the well-known concepts of an encapsulated state and a critical region.

Because objects are autonomous execution units, it makes sense to distinguish between *asynchronous* and *synchronous* method invocations. In the former case, the sender of a message continues execution in parallel with the receiving object, whereas in the latter case, the sender and receiver perform a rendezvous. Of course, only synchronous methods provide an opportunity to directly return a result from the receiver to the sender of a message.

At a first glance, synchronous methods seem to provide a way of reintroducing traditional input methods like `getc` in the model. We do however make an important restriction that will prohibit such use:

> *no methods—in environments or in programs—must block execution indefinitely.*

This restriction rules out language constructs like selective method filtering, as well as environments that provide naive interfaces to blocking system calls. All that a synchronous method call can do is to compute a reply based on the current state of the receiver, possibly after performing some side-effects. The serialization of all method executions of the receiving object does not change the fact that a synchronous method is essentially an ordinary subroutine, since, by an argument of transitivity, if the receiver is not ready to immediately execute a synchronous call, it must be busy servicing one of the non-blocking calls that stand in the way.

Reactive objects thus alternate between phases of passive inactivity and temporary outbursts of method execution. In contrast to so called active objects, a reactive object does not have a continuous thread of execution; all executable code of an object is defined in terms of its methods. A method of a reactive object is furthermore guaranteed to terminate, provided that it does not deadlock or enter an infinite loop. However, due to the absence of blocking constructs in this model, the only source of deadlock is the synchronous method call, and a cyclic chain of such calls is easily detectable at run-time.

The reactive object model has been realized in the language Timber that we will survey in the next section. The model is general enough, though, that we would like to summarize a few informal claims about its properties before we go into language details.

- Reactive objects is a simple and natural model of event-driven systems on various level of detail, from hardware devices to full distributed applications.
- It is a straightforward integration of concurrency and object-oriented programming, with the added bonus of automatic protection of state consistency.
- A single reactive object can easily handle input from multiple asynchronous sources.
- Under assumptions of freedom from non-termination and a very simple form of detectable deadlock, a reactive object is also guaranteed to be responsive in all states.

3 Reactive objects in Timber

We will now give the model of reactive objects a more concrete form, by showing how it is realized in the programming language Timber [4]. Timber is a strongly typed, object-oriented language with constructs specifically aimed at real-time programming; however, its foundation in the reactive object model makes it suitable as a general purpose language as well. We will introduce Timber by a small programming example, after pointing out some distinguishing details.

- In the syntax of Timber, = denotes a definition, **let** introduces local definitions, `f x y` is a function `f` applied to two arguments, and `:=` is assignment to a state variable.
- Methods have first-class status: they can be passed as parameters and stored in data structures.
- Commands and declarations are grouped using layout.

Our example is a variant of the program Ping.java from the New I/O API in Java 1.4. This program demonstrates how to concurrently measure the time it takes to connect to a particular TCP port on a number of remote hosts. The Timber source is shown in Figure 1, and the output of a typical run looks as follows:

```
dogbert: 20.018 ms
ratbert: 41.432 ms
ratburg: NetError "lookup failure"
theboss: no response
```

Ping is implemented as an *object template* (i.e., a class). It is parameterized over a list of hosts and a port number, and a record `env` that contains methods for interacting with the environment. Templates define a number of state variables as well as an *interface*, which is typically a record of methods that the environment can invoke. All methods in

```
ping hosts port env =
  template
    outstanding := hosts
  in let
    client host start peer =
      record
        connect      = action
          env.putStrLn(host++": "++show(baseline-start))
          outstanding := remove host outstanding
          peer.close
        neterror err = action
          env.putStrLn(host++": "++show err)
          outstanding := remove host outstanding
        deliver pkt  = action done
        close        = action done
    cleanup = action
      forall h <- outstanding do
        env.putStrLn(h++": no response")
      env.quit
  in record
      main = action
        forall h <- hosts do
          env.inet.tcp.open h port (client h baseline)
        after (2*seconds) cleanup
```

Figure 1. Ping in Timber

this example are asynchronous (as determined by the **action** keyword).

The Ping program is started by creating an instance of the template, and then invoking its `main` method. Internally, Ping objects maintain the state variable `outstanding`, a list of hosts from which a response has not been seen. The main method calls `inet.tcp.open` of the environment in order to initiate a TCP connection to the designated port on every given host (**forall** expresses a loop construct, with `h` as a loop variable). These calls do not wait for the connection to complete, though; instead, the local method `connect` is set up to be invoked when the connection has been established. The record `client` used for this purpose is parameterized over the host we are trying to connect to, the start time of the program, and an environment-provided record containing methods for communicating with the peer host. For timing purposes, actions can refer to the pre-defined variable `baseline`, which is set to the arrival time of the event that triggered the action.

After all connections have been initiated, the asynchronous method `cleanup` is scheduled to be called two seconds after the baseline of `main`. Note that nowhere does the execution of `main` block, it just sets up other actions to react to future events.

All methods of an object can safely manipulate its state variables in a single-threaded fashion. E.g., it is irrelevant here that some methods are defined within the record `client`, and others elsewhere. This flexibility allows us to express a straightforward solution using only one object, with a single state variable. In contrast, the Java program is 287 lines of code, has ten class variables, and needs three threads: one for the demultiplexing of connection events, one for single-threaded printing, and one for timeout.

4 Related work

The reactive object model described in this paper was first developed for the programming language O'Haskell [7]. The current work is, however, our first attempt to distill the programming model as a contribution in its own right. Our language Timber inherits much of its basic design from O'Haskell, but adds several important features, of which the notion of a time-constrained reaction is the most relevant to this paper.

Current work on reactive languages is mostly concerned with the *synchronous* approach to reactivity [3, 5]. The main hypotheses made in this model are that computations take zero time and event delivery is instantaneous. From these assumptions it follows that events received or generated somewhere between two clock ticks are actually occurring *simultaneously*, and hence, for example, multiple invocations of a particular method during an instant must be indistinguishable from just a single invocation.

The unification of the object and process concepts is an idea that stems from the *Actor* model [1, 2]. However, the state of an actor is identified with its current mapping from names to method bodies, and messages to undefined methods are simply queued. Hence actors do not possess the responsiveness property we are emphasizing with our model. Moreover, the Actor model lacks anything similar to our synchronous methods, and asynchronous message delivery is not order-preserving.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

[3] A. Benveniste and G. Berry. The Synchronous Approach to Reactive and Real-time Systems. Technical Report 1445, INRIA-Rennes, 1991.

[4] A. P. Black, M. Carlsson, M. P. Jones, R. Kieburtz, and J. Nordlander. Timber: A programming language for real-time embedded systems. Technical Report, http://www.cse.ogi.edu/PacSoft/projects/Timber/, April 2002.

[5] F. Boussinot, G. Doumenc, and J. Stefani. Reactive Objects. *Annals of Telecommunications*, 51(9-10):459–473, 1996.

[6] P. Chan and R. Lee. *The Java Class Libraries, Second Edition, Volume 2*. Addison-Wesley, 1997.

[7] J. Nordlander. *Reactive Objects and Functional Programming*. PhD thesis, Department of Computer Science, Chalmers University of Technology, Göteborg, Sweden, May 1999.

[8] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A Note on Distributed Computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, Inc., Nov. 1994.

[9] J. Zukowski. New I/O Functionality for Java 2 Standard Edition 1.4. http://developer.java.sun.com/developer/technicalArticles/releases/nio/, October 2001.