

# Programming with Time-Constrained Reactions

Johan Nordlander\*    Magnus Carlsson†    Mark P. Jones‡    Jan Jonsson§

December 20, 2004

## Abstract

In this paper we argue that a programming language for real-time systems should support the declaration of *time-constraints*, and that those constraints should attach to a well-developed notion of *reactions*. To make our claims more precise, we introduce *Timber*, which is a concurrent programming language based on a model of non-blocking, reactive objects. Timber supports both upper and lower time constraints on a reaction, where upper constraints correspond to classical deadlines, and lower constraints constitute the systematic capturing of well-defined points in the future as events in their own right. A series of programming examples illustrates how these mechanisms can be used to express simple solutions to common problems in practical real-time programming, and the relation between Timber and deadline-based scheduling theory is explored.

## 1 Introduction

Look up just about any definition of the term *real-time system*, and it is bound to be based on some variant of the notion of "meeting a deadline". The exact wordings will of course vary, but there is little controversy in saying that in the *problem domain* of real-time systems, in particular the so called *hard* ones, the deadline concept plays a fundamental role.

It is therefore rather surprising that the real-time *solution* domain — by which we mean real-time operating systems, languages, middleware, etc — is frequently free from any references to deadlines. This is indeed a paradox, but it is a widely accepted one; its basis is a common understanding that deadline-driven approaches are yet not mature enough to be applied in anything but perhaps abstract system models. The real-time system designer is thus presented with a *semantic gap* to cross: what the problem domain refers to as events, reactions, and deadlines must be mapped onto the threads, synchronization points and static priorities that have long dominated real-time system implementation practice. It is no exaggeration to claim that bridging this gap is one of the major challenges that face anyone involved in real-time system design.

This paper takes as its starting point the observation that our common programming languages play a fundamental role in the mismatch between real-time programming practice and its problem domain. Specifically, we note that a major reason why deadlines cannot be comfortably handled in these languages is simply that the corresponding notions of events and reactions are cumbersome

---

\*Luleå University of Technology, email: nordland@csee.ltu.se.

†Carmen Systems, email: magnus@carlsson.org.

‡Oregon Health & Science University, email: mpj@cse.ogi.edu.

§Chalmers University of Technology, email: janjo@ce.chalmers.se.

to express as well. The cause can be found in the I/O model that underlies modern languages, which presupposes that event handling is an internal affair of the operating system, and that the best way of interfacing an application to the real world is by means of "services" that silently block execution whenever the world is out of sync. What the programmer at best can hope for is some middleware abstraction that provides an *encoded* notion of events, and perhaps a design pattern that gives the ramifications of how reactions to events must be coded in order to fit the framework. By necessity, the possibilities of maintaining a deadline-oriented view of such a real-time system — from its specification, via an implementation, down to the actual scheduling of code at run-time — are limited.

For an analogy, consider the prospect of maintaining the type safety of function calls in a language that lacks any notion of types as well as functions! Few would argue that such an arrangement is the best way of promoting the idea of type-correct programming. It is the core message of this paper that the issue of real-time correctness can, and should, be addressed from a programming language perspective with the same level of interest as type-correctness has received in the past.

In previous work we have described the event-driven *reactive objects* programming model that forms the basis upon which our real-time programming language Timber is built [14]. The key aspect of the reactive objects model is that events and reactions are unified with messages and methods, respectively, and that methods are non-blocking by construction. The present paper extends this work by the following contributions:

- We provide a detailed definition of and motivation for the *time-constrained reactions* that give Timber its unique character.
- The pragmatics of programming real-time systems in Timber is illustrated in a series of archetypical examples.
- We include a discussion on the relationship between the time-constrained reactions of Timber and the task models applied in existing deadline scheduling research.

The perspective taken in this paper is that of the working real-time programmer with an interest in language design and programming models. The formal semantics of Timber, which might be of primary interest to language implementors and scheduling scholars, is defined elsewhere [4]. A third, more application-oriented view of Timber is presented in a companion paper, where the implementation of components for robot control is described [11].

The rest of this paper is organized as follows. In section 2 we review the reactive objects model of Timber, and introduce the reader to the basic structure of the language. Section 3 defines the notion of time-constraints in Timber, and shows the basic constructs by which the programmer can control the timing behavior of a Timber system. Then follows a section that exemplifies the use of time-constraints in practice (section 4). In section 5 the relation between Timber and existing deadline scheduling theory is discussed. Some related real-time programming language approaches are reviewed in section 6, while section 7 concludes with pointers to future work.

## 2 Background

### 2.1 Reactivity

The idea that the primary role of a computer in its environment is to react to events is by no means new – it has been around in various forms since the dawn of computers more than half a century ago. Constructors of embedded systems in particular have no problems embracing this view, as it clearly emphasizes the computer as a sub-component of a more diverse and general system.

The reactive view is however dwarfed by the much more persistent idea that a computer is the active part, and any equipment with which it may need to interact primarily exist as sub-components of the computing system. This view certainly holds for the large class of external devices that deal with data storage; for example, hard disks, RAM memory, card readers, and line printers. Born within the batch-oriented era of computing, the active, computer-centric perspective naturally leads to the idea of synchronization by transparently blocking input operations, dressed up as subroutines.

Of course, treating interactive users, physical processes, or the internet as mere sub-components of a particular computer is not viable, but neither were such systems commonly connected to computers when batch-oriented computing was the rule. Unfortunately, modern programming languages prevail in upholding the batch-oriented view of the world when it comes to environment interaction, something which, for example, manifests itself in the common modeling of input devices as data files.

The result is that in order to implement real event-driven applications, the programmer has to go to some length in establishing a reactive program structure out of the suggested active programming model. As an illustration, consider the typical top-level *event loop* structure that forms the core of nearly every modern interactive application. An outline of the event-loop pattern is shown in Figure 1(a). The goal of this pattern is to enable the core parts of an application to be written as a set of *event handlers* that are called when an event occurs. In order to achieve this, however, the calls to the blocking event delivery service of the operating system (tentatively called *GetNextEvent* in this example) must be centralized to a top-level dispatch loop, with the implicit side-condition that none of the event-handling subroutines make any call to this function, or to any indefinitely blocking service general. Thus, the role of the event-loop pattern is to convert the active *compute-until-something-happens* style of the operating system service into the passive *call-me-when-something-happens* style of the application.

Event-loops are so common in event-driven programming that many people consider the notions synonymous, despite the inherent weaknesses of the event-loop pattern: (1) operating system dependency and limited extensibility, (2) brittleness in presence of changes that break the blocking/non-blocking invariants, and (3) unclear interaction with concurrency. Moreover, the packaging of real-world events as the result of a transparently blocking operating system service is likely to be questioned by very few. Still, there is a backside to the event-loop pattern hidden inside the implementation of such an service, that has high relevance in a discussion on the structure of event-driven programs.

Consider the extended software outline shown in Figure 1(b). Here we have complemented the application event-loop with parts of the operating system as well, in order to illustrate the path

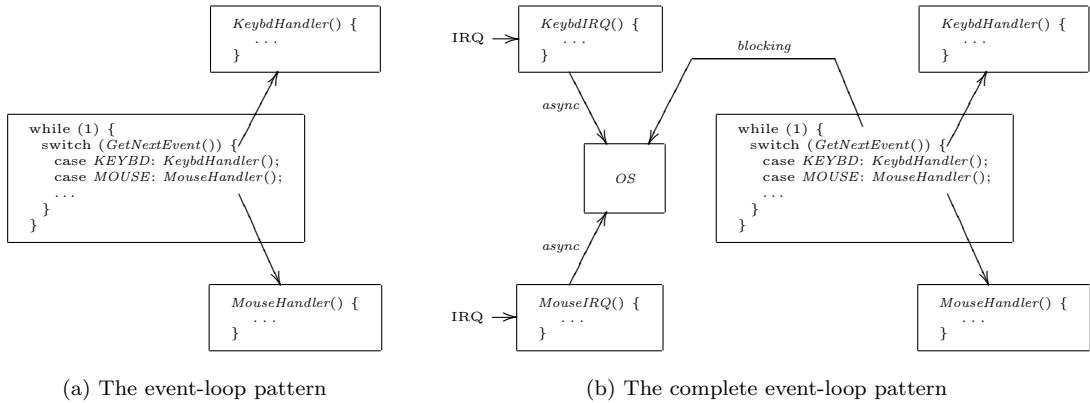


Figure 1: Event loops

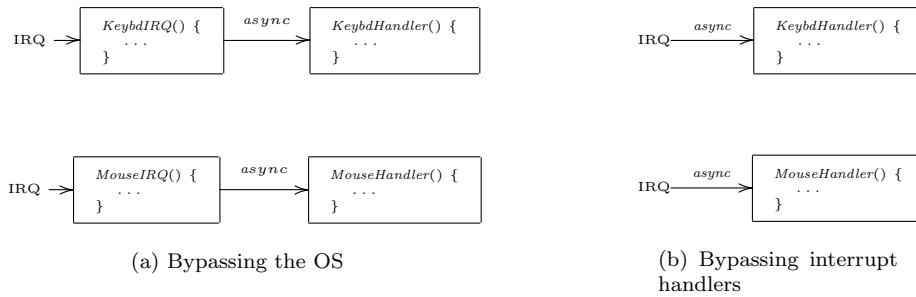


Figure 2: Eliminating event-loops

an event takes from the initial hardware interrupt to the designated event handler. The main job of the operating system is here to use scheduling to implement blocking of the application, on basis of data generated by the low-level interrupt handlers. A key technicality is the need for some asynchronous signaling mechanism, that decouples the execution of the interrupt handlers from the rest of the system. Notice that the effort of the application to switch polarity of an active event service into a reactive program structure is mirrored within the operating system, as a mechanism for converting the reactive nature of interrupt handling into an actively blocking operation!

The irony is that so much operating system effort goes into establishing the batch-oriented ideal of transparently blocking subroutines, when the reactive programmer subsequently will have to work hard in order to predict, control and avoid this very feature. The question that has spurred the development of Timber and its programming model may seem obvious in this context: Why can't we simply bypass the double switching of polarities in traditional event-driven programming, and pipe the internal asynchronous signals of the operating system directly to the corresponding event handlers? A sketch of a reactive program structure along those lines is shown in Figure 2(a). Indeed, by modeling interrupt requests as asynchronous signals to begin with, one could even go one step further, as in Figure 2(b).

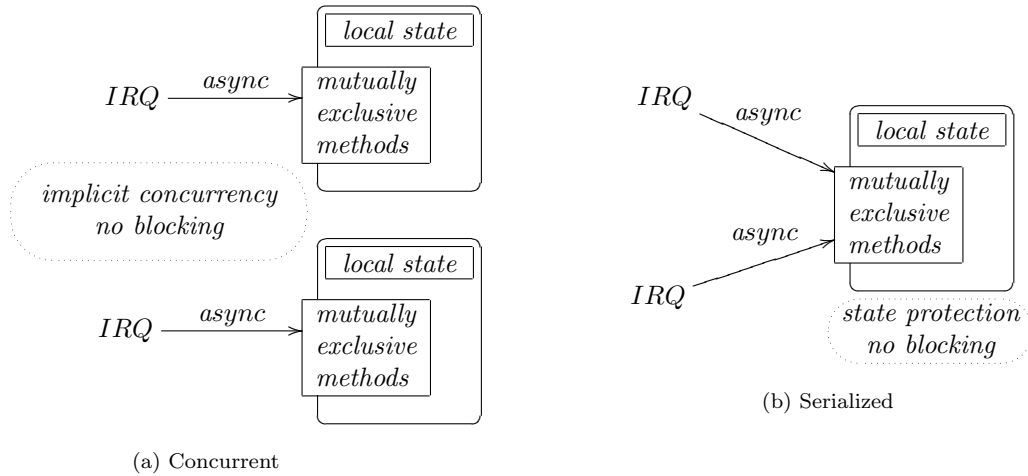


Figure 3: Event-handling

## 2.2 Reactive objects

Of course, the real issues in a move that bypasses the operating system must concern the semantics of the intended programming language, especially regarding concurrency and global state management. Ada and RT Java are both examples of concurrent languages with abilities to run programs on bare metal CPUs; however, both these languages presuppose that their run-time systems take over the operating system role of implementing indefinite blocking. Other, more experimental languages exist that take on a higher level view of interaction, but their programming models are often far removed from the perspective of the embedded systems programmer (see Section 6 for some further discussion).

The Timber approach has been to leverage on the object-oriented paradigm, in order to achieve a concurrent programming model that lies very close to what is already familiar to anyone acquainted with event-driven programming. Thus, the *reactive objects* semantics of Timber is a model where

- **Each object is independent**; that is, method execution in one object is implicitly concurrent with activity in other objects.
- **State is local and protected**; that is, each piece of program state belongs to some particular object, which is the only entity that can read and modify its value. Furthermore, object state cannot be concurrently accessed, as the execution of methods of a particular object is mutually exclusive.
- **Method execution is non-blocking**; that is, there is no way for a method to halt execution midway, waiting for a future event. Events are uniformly interpreted as method calls, and in the absence of diverging calculations, each method will eventually terminate and return its object to a resting state.

We illustrate the impact of these characteristics by continuing our event-driven example from above. In Figure 3(a) the two objects can handle their respective interrupt signals in any interleaving, as they only have access to their own private parts of the program state. The enforced

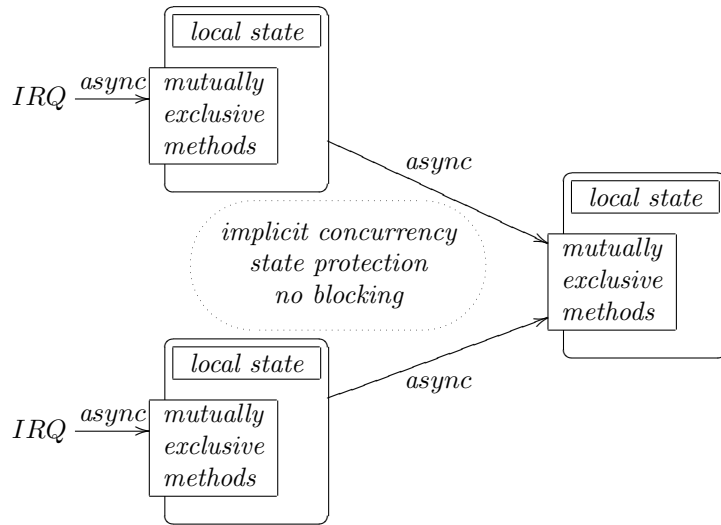


Figure 4: Delegated event-handling

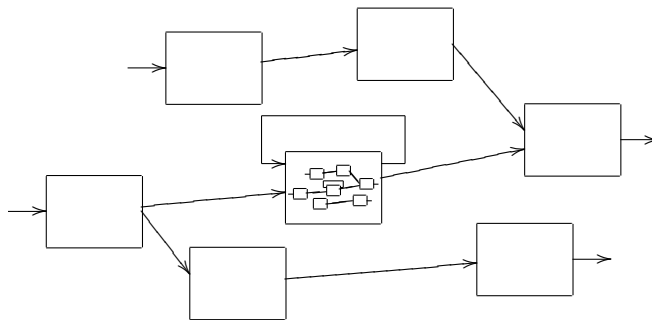


Figure 5: Generic structure of Timber programs

non-blocking nature of methods also coincides very well with the informal requirements that usually constrain both low-level interrupt handlers and more general handlers of event-driven frameworks.

Figure 3(b) illustrates the case where two handlers share their state. This is expressed in the reactive objects model by letting the handler methods be part of the same object, which then guarantees that their executions will not be interleaved. The underlying asynchronous message mechanism of each object captures both any interrupt queuing facilities offered by some hardware, and the event queues normally found on a per-program-basis inside operating systems.

In Figure 4, the possibilities of event-handling using a combination of objects is exemplified. The two leftmost objects in the figure are able to handle events concurrently, while handling of the secondary events in the right object will be serialized. What we see here is the beginning of a program structure set up to handle more complex tasks. An even more general case is shown in Figure 5, where each box may be a single object, or a compound structure of the same kind as the whole figure itself. The important characteristic is that the arrows represent *message-based communication* in both the object-oriented and the concurrent programming sense, and that the nodes stand for a system partitioning according to state as well as control.

In general, a structure of reactive objects connects a set of input events to a set of output events, where the definition of the nodes determine how an individual input event propagates through the graph and modifies its state. At the hardware/software boundary, the input events correspond to interrupts, and output events are denoted by device register accesses. The model is however equally applicable to any level of software component abstraction, as well as to communicating systems in general, of an arbitrary scale.

## 2.3 Timber

With *Timber* we concretize the reactive objects model in a programming language that is further characterized by

- Synchronous as well as asynchronous communication.
- First-class methods and object generators.
- Strong type safety.
- A syntax and so called *monadic* semantics influenced by functional languages.

Here we will just briefly overview the constructs of Timber that are most relevant to the rest of the paper. For a more complete account, we refer to the draft language report [3] and the formal semantics definition [4].

On the top level, a Timber program is a set of bindings of names to expressions, of which some may be *templates*, which is the Timber term for object generators (i.e., traditional *classes*). In concrete Timber syntax, a generator of simple counter objects can be defined as follows:

```

counter = template
    value := 0
    return {
        read = request
            return value
        incr = action
            value := value + 1
    }

```

This definition reveals that *counter* objects consist of a local state variable *value* initialized to 0, and a public interface with methods for reading and incrementing the state. The interface is a record in this case, but can in principle be any data structure. However, interfaces need to contain at least one method in order to be practically useful.

Methods are of two kinds: *actions* that imply asynchronous communication, and *requests* for expressing synchronous rendezvous. The general forms for method and template expressions are

<b>action</b>	<b>request</b>	<b>template</b>
<i>statements</i>	<i>statements</i>	<i>statements</i>

Each statement is basically either an assignment to local state, a template instantiation, or (with the exception of template expressions) an asynchronous or synchronous method call. The usual selection of looping and branching constructs is also available, as well as a statement that introduces local bindings:

```

test env = action
    c ← counter
    c.incr
    c.incr
    n ← c.read
    env.putStr (show n)

```

This example says that *test* is an asynchronous method which instantiates a *counter* object, increments it twice, reads its current value, and prints a string representation of the value to any object supplied as the parameter *env*. The left-arrow syntax introduces a local name for the result of a side-effecting command, in contrast to the symbol `=`, which is reserved for declarative definitions. In this context it is worth noting that `:=` denotes a destructive update, something which is only available for state variables introduced in an enclosing template scope.

Types play an important role in Timber programming, still most type information can be left out from programs due to the automatic type inference facilities supported by the language. A cornerstone in this process is however the definition of interface signatures, which for our counter above would look as follows:

```

record Counter =
    incr :: Action
    read :: Request Int

```

From this information, the inference process will be able to deduce that *counter* has the type *Template Counter*. We will not pursue the type system aspect any further in this paper, though, and the coming examples can if necessary be read as if Timber were an untyped language.

### 3 Time constraints

Given a programming model that identifies methods with reactions, and method calls with events, it becomes natural to associate methods with real-time constraints. Timber actually supports the declaration of both upper and lower time bounds, where an upper bound is the equivalent of a classical deadline, and a lower bound corresponds to the generation of clock-based events typically expressed using delays, timers, and timeouts in a traditional setting.

#### 3.1 Deadlines

The deadlines that constrain a real-time program under execution are typically derived from the desired timing behavior of the system as a whole, by subtracting end-to-end deadlines by any latencies and delays introduced by external components along the signal path. This pattern could potentially be applied for the purpose of deriving deadlines for individual software components of a reaction chain as well, but we have chosen to adapt a different strategy with Timber. Our arguments are threefold:

Firstly, the points in time when software components in a reaction chain call each other are purely artificial. They will all approach the time of the original event as CPU speed increases.

Secondly, the time when an event first enters a Timber system can readily be made available to subsequent reactions, just like any ordinary parameter. This makes it unnecessary to compensate for latencies introduced by initial segments of a reaction chain.

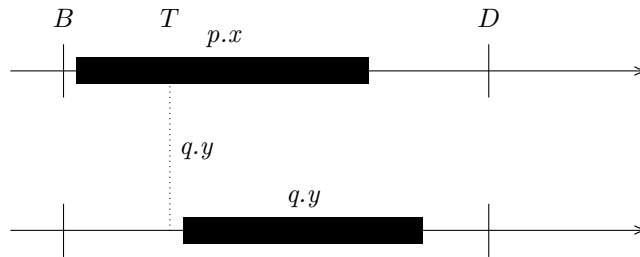


Figure 6: Inheriting time-constraints.

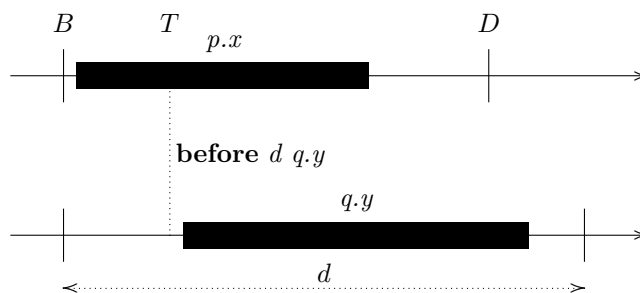


Figure 7: Extending the deadline.

Thirdly, if a Timber system is at all schedulable, it is implied that the first, as well as all subsequent components in a reaction chain will meet their deadlines. Compensating for latencies in latter components by tightening the deadline for the first one will not improve the schedulability of the system.

For these reasons, the model used in Timber is that a deadline by default applies to *all* reactions that might result from an event entering the system. Internal events are thus no more observable than the individual statements executed by reactions; all that matters is that every component involved in the handling of a particular event is able to complete before the given end-to-end deadline, measured from a common reference point in time.

We illustrate this idea with a timeline of two coupled method executions in Figure 6. Here an event enters the system at time  $B$  as an interrupt coupled to some asynchronous method  $p.x$ . The reaction to this event must be finished before time  $D$ , which of course implies that  $p.x$  must complete before  $D$ . However,  $p.x$  also makes a call to an asynchronous method  $q.y$  at some point  $T$  during its execution. Now, instead of making this point the start of a new timeframe with its own deadline—which would inescapably lead to the question of what the deadline is for reaching point  $T$ —Timber simply lets  $q.y$  inherit both  $D$  as well as its reference point  $B$ . Notice that if any of  $p.x$  or  $q.y$  misses the common deadline under these circumstances, the system must be overloaded. In particular, specifying an artificial deadline for  $T$  would not have a chance of rectifying the problem, as the total amount of processor work required by the methods would still remain the same.

There are however situations where only *part* of a reaction is associated with a tight deadline, but other parts have much looser constraints. This need is addressed in Timber by a means for extending the deadline at each asynchronous method call. An example of such a scenario is shown

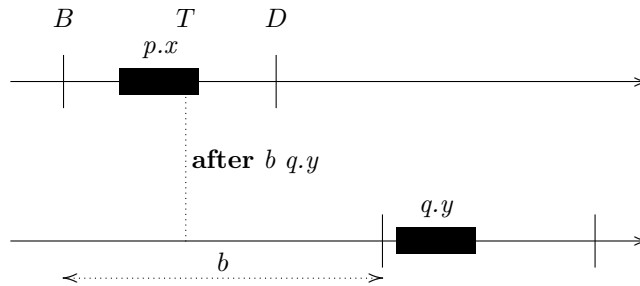


Figure 8: Extending the baseline.

in Figure 7. By prefixing the call to  $q.y$  with the construct **before**  $d$ , the deadline of  $q.y$  is adjusted to  $d$ , measured relative the common reference point  $B$  (note: *not* the time of the call). The new effective deadline is however dynamically limited to values greater than or equal to the current deadline, in order to avoid paradoxical situations where a secondary reaction ends up more urgent than the activity that has caused it. The ability to extend deadlines is furthermore only available for calls to asynchronous methods. A synchronous call, which by definition occurs interjected within activities by the caller, can neither be more urgent than the code preceding the call, nor less urgent than the code that follows.

The extension mechanism is also used to set the initial deadlines associated with external events—conceptually, the default relative deadline for such events is 0, so any positive value constitutes a legal extension. The actual representation of relative deadlines is abstract, but a variety of constant symbols is available to allow time to be expressed in most commonly used units. A particular value sets the effective deadline to infinity, thus removing the reaction in question from the real-time domain. It must be stressed, though, that while the Timber scheduler bases its run-time decisions on the deadlines attached to each reaction, no attempts are made to automatically detect and handle deadline misses. The correct behavior in such situations is highly application-dependent, and we will see in the following sections how detection of missed deadlines can be easily programmed. Furthermore, we believe that the greatest value of the Timber deadline regime lies in its semantic formalization of what constitutes a time-correct program execution [4]. This is a necessary starting point for any form of static schedulability analysis, something which is of undisputed importance to the field of hard real-time systems. The relation between Timber and existing research in this area will be further discussed in section 5.

### 3.2 Baselines

As a dual to the upper, “must-finish-before” time-bounds we refer to as deadlines, we introduce the term *baseline* for the common reference point of an event we have been introduced in the previous subsection. A baseline can be thought of as a “must-start-after” constraint, even though it is obvious that as long as baselines are solely identified with time-stamps of external events, such constraints will always be met.

However, just as it is meaningful to extend the deadline of selected method calls, so is the concept of increasing baselines. Analogous to the **before** construct, we may write **after**  $b$  in front of an asynchronous call to shift its baseline  $b$  time units into the future, again counting from the

current baseline. An illustration of a baseline shift is shown in Figure 8. Notice especially that in the absence of an explicit deadline prefix, the deadline of a call remains constant *relative its effective baseline*.

By means of the **after** construct, the Timber programmer is given the opportunity to schedule asynchronous method calls at well-defined points in the future, that are insensitive to variations in execution speed of the current statement sequence. Moreover, when combined with the ability to control deadlines, the window for executing these methods can be specified with arbitrary precision. A particular advantage with this mechanism is the maintained ability of a sender to react to other events while calls with extended baselines are in transit. In line with the reactive object model, there is no notion of sleeping or blocking involved, neither by the caller, nor by the callee. We will see in the following section how this feature may be put to good use.

## 4 Using time-constraints

In this section we will illustrate the practical side of programming with time-constrained reactions by means of a series of program fragments.

### 4.1 Periodic tasks

Our first example will be a pattern for constructing a periodic task, by utilizing a recursive asynchronous method with a shifted baseline:

```
tick = action
      stmts
      after (50 * ms) tick
```

This method can be included in any kind of template context; other methods of the same object will peacefully coexist with the periodically activated code. Because the baseline offset  $50 * ms$  is measured relative the current baseline at each invocation (and not the actual time of the recursive call), the implementation does not suffer from accumulating drift. Notice that an initial call to *tick* is required to start the process, so in a sense, the periodic execution that follows can be seen as an infinitely oscillating reaction to that first triggering event.

As an illustration to the usefulness of explicit deadlines, we can easily define a variant of the code above where the time slot available for *stmts* is much less than the period:

```
tick = action
      stmts
      after (50 * ms)
      before (7 * ms) tick
```

If we want to specify that *all* calls to *tick* (including the first one) should have the same relative deadline, we might write as follows instead:

```
tick = before (7 * ms)
      action
      stmts
      after (50 * ms) tick
```

Notice how the **before** prefix can be applied to both the name of a method at a call site, and at its actual definition. With the referentially transparent semantics of Timber, these alternatives actually amount to the same thing!

## 4.2 Implementing delays

A non-repetitive call with an extended baseline is shown in the following example, which implements a simple delay between commands to a slow external device like a plotter.

```
drawTo pos = action
    pen.ctrl Down
    after (100 * ms) (action xy.moveTo pos)
```

As always with the **after** construct, the invoking object is free to continue immediately after the invocation. The example also illustrates the use of an *anonymous* method; i.e., a method not bound to a name.

The case where data must be read at the end of a specified interval must be coded in Timber by means of a *callback* method; i.e., a parameter standing for an unknown method accepting the desired data.

```
readAt pos cont = action
    head.moveTo pos
    after (200 * ms)
    ( action
        data ← head.read
        cont data )
```

A synchronous method that simply returns the data after 200 ms cannot be written in Timber, as there exists no means of blocking execution for that long while a method is active. The minor awkwardness of having to supply a callback parameter should however be contrasted with the maintained responsiveness of both the caller and the callee in the Timber formulation. The readiness to respond to new events during the 200 ms time-slot should have direct consequences for the ease by which (say) monitoring and error handling could be added to the template fragment above.

## 4.3 Timeouts

In general, a Timber object will keep track of what it has done and what to expect by means of local state variables. As an illustration, the following code example implements parts of an object that issues requests over a network, after which it expects replies within a specified interval.

```
template
    state := Idle
    send = action
        network.sendreq reply
        after t timeout
        state := Active
    reply d = action
        if state == Active then
            ...
            state := Idle
    timeout = action
        if state == Active then
            ...
            state := Idle
    ...
```

In the call to *network.sendreq* we see a typical example of how a callback parameter is supplied to handle a reply that depends on an external event. Notice how the two different outcomes—a real reply, or a timeout—are given equal status in the implementation.

In the previous example, it is manifest that both a reply and a timeout might occur while the sending object is in state *Idle*. The fact that an external network node is fully able to send anything it wants at any time cannot be ignored, so the state check performed in *reply* must really be present in any robust implementation. The timeout events, on the other hand, are actually under local node control, so it might seem reasonable to have them cancelled whenever a valid reply is received.

Timber offers such an opportunity by means of unique *message tags* that are created as the result of each asynchronous call. Normally these tags are just ignored, but in case the ability to cancel a specific call is needed, its tag should be captured and stored. Cancellation amounts to calling the special *cancel* method of the designated tag, as the following fragment shows:

```
tag ← after t timeout
...
tag.cancel
```

The Timber library contains a useful template called *singlecall*, which takes care of managing the storage of a single tag, as well as automatically canceling any old message whenever a new supervised call is requested. A reformulation of the network request example illustrates the use of *singlecall* for cancelling redundant timeouts.

```
template
state := Idle
sc ← singlecall
send = action
    network.sendreq reply
    sc.call (after t timeout)
    state := Active
reply d = action
    if state == Active then
        ...
        sc.cancel
        state := Idle
timeout = action ...
...
```

#### 4.4 Missing deadlines

As deadline misses are not automatically caught by Timber, such detections must be programmed explicitly. Due to the consistent use of baselines for timing reference, it is easy enough to define a task that checks some application-dependent program state when the deadline for some computation is due:

```
handler = before t action
    after t other.checkResult
    ... do heavy computation ...
    other.deliverResult
```

There are however many design decisions to make before this pattern can be turned into concrete code. Should deadline misses be handled exactly when they occur, or is it ok to wait until the over-running computation is done? Should the failing computation be aborted, or is it

sufficient to simply replace or disable the deliverance of its result? What should replace the result if the computation is aborted, and in what state should that object be left? And how do we deal with the fact that deadline-based systems in general are inherently unpredictable in overload situations?

Timber does provide an experimental feature for aborting running computations (in contrast to canceling pending messages), but we have not reached any conclusion whether this is the right level for dealing with missed deadlines. While it is clear that the need for aborting running computations is much less pressing in Timber than in languages based on blocking threads, we wish to point out that a thorough study on overload handling in Timber must remain as a topic for future work.

## 4.5 A time-constrained data collector

As an illustration of a reaction chain where deadlines are gradually relaxed, consider the following sketch of the data collecting part of a program for data acquisition and analysis.

```

collector analyzer inport =
  template
    buf := []
    irq = before (4 * microseconds)
      action
        data ← inport.read
        buf := (baseline, data) : buf
    t = 2 * ms
    tick = before (100 * microseconds)
      action
        before t (analyzer.analyze buf)
        buf := []
        after t tick
  return { irq = irq; start = tick }

```

Here the intended scenario is that the availability of new data is announced through an interrupt, after which a data value must be fetched from a specific register within 4  $\mu$ s. Moreover, every 2 ms the collected data is handed over to an analyzer for further processing, together with an accurate time-stamp for each data value. The deadline for analyzing data is a full cycle of 2 ms; however, the jitter for these cycles must not exceed 100  $\mu$ s.

These timing requirements are expressed in Timber as a deadline of 4  $\mu$ s for the interrupt handler, and a deadline of 100  $\mu$ s for the periodic tick that drives the analyzer. However, since the analyzer itself may take up to 2 ms to complete its task, the hand-over call is assigned a relaxed deadline by means of the **before** construct. Obtaining a time-stamp of each data value is achieved by simply pairing it with the absolute baseline of the corresponding event; a value made available to every method through the pre-defined variable *baseline*.

In this example it is especially important to recall that explicit deadlines in Timber code denote real-time *specifications*, not statements about expected execution times. Since the constrained methods *irq* and *tick* are part of the same object, they are bound to occasionally obstruct each other. That impact is however limited to the actual *execution times* of each method, and it is the task of a schedulability analysis to figure out if these times are short enough to allow both methods to always meet their deadlines.



Figure 9: Sonar baselines and deadlines. The pattern repeats at  $B+3s$ ,  $B+3s$ , ...

## 4.6 A sonar driver

We will end this exposition of time-constrained programming in Timber by showing a simple implementation of a sonar driver that is coupled to an alarm. The specifications we are assuming state that a sonar beep should be 2 ms long, with a maximum jitter of 50  $\mu s$ , and that the required accuracy of the measurements dictate that time-stamps associated with beeps must also be accurate down to the 50  $\mu s$  range (Figure 9). Furthermore, the sonar is supposed to sound every 3 seconds, and the deadline for reacting to off-limit measurements is 5 ms. These specifications look as follows when translated into Timber code:

```

sonar port alarm =
  template
    t := genesis
    ping = before (50 * microseconds)
          action
            port.write beepOn
            t := baseline
            after (2 * ms) stop
            after (3 * seconds) ping
    stop = action
            port.write beepOff
    echo = before (5 * ms)
          action
            distance = k * (baseline - t)
            if distance < limit then
              alarm
    return { irq = echo; start = ping }

```

Here a predefined constant *genesis* is used, which stands for the earliest point in time that the system is able to express.

As an example of how a system like the sonar might be configured to run on a bare-metal embedded system, we show an example of the main declaration for such a system below.

```

main regs =
  template
    s ← sonar (regs!0xac00) a
    a ← alarm (regs!0xa3f0)
    return [
      (resetIRQ, s.start),
      (sonarIRQ, s.irq),
      (buttonIRQ, a.ack)
    ]

```

On this level, the interface to the software, as seen from the hardware, is the array of interrupt handlers it provides. Consequently, the hardware takes the shape of an array of device registers that can be read or written.

## 5 Scheduling

We have already claimed that Timber lends itself especially well to deadline-based scheduling. In this section we will develop this claim a bit further; first by showing how our current Timber implementation makes use of the deadlines manifest in Timber source code, then by discussing the relationship between Timber and common task models that underlie deadline-based scheduling theory.

Recall that every method call in a Timber program is associated with an absolute deadline, either implicitly by inheritance from the caller, or by means of an explicit **before** specification measured relative the present baseline. Our current Timber implementation – a rather direct and non-optimizing interpretation of the formal semantics – naturally maintains these deadlines in the run-time representation of messages. Each object furthermore keeps its incoming messages in a private queue that is sorted according to deadline. Mutual exclusion between methods of the same objects is achieved by mapping every object onto its own lightweight thread, that consumes and executes messages from the private message queue whenever there are any.

Scheduling in such a system amounts to determining, at every instant, which of all runnable threads that will be assigned to the CPU. To this end, an effective deadline is maintained for each object, so that the object thread with the *earliest deadline* can be selected for execution [13]. The effective deadline is defined as the minimum of all pending message deadlines and the currently executing message (if any), which actually implements the *priority inheritance protocol* for EDF [18]. Messages whose baselines have yet to occur do not take part in this calculation, but are instead put into a system wide delay queue, for eventual delivery by the aid of an external timer.

The ease by which Timber facilitates a deadline-based implementation is a strong indication that the language indeed represents a substantial reduction of the gap between the problem and solution domains of real-time programming. Furthermore, this is accompanied by an equally short gap between programming practice and formal semantics [4]. Still we believe it is the connection to deadline scheduling theory that offers the most interesting path for further studies of the language.

In the terminology of [19], a Timber action constitutes a *task*, and an asynchronous call a *job*. Timber objects play the role of *shared resources*, at the same time as they also map onto what Baker calls *featherweight processes*; i.e., identified sequences of job executions [1]. As shared resources, objects possess the property that every job will require one during its whole execution. Synchronous calls temporarily increase the number of required resources, whereas asynchronous calls mark the release of new jobs.

The fact that jobs can be generated with release times at programmed points in the future opens up for a generic model that can express both *sporadic* and *periodic* tasks as special cases. Here a periodic task would imply an original job that generates a copy of itself one period into the future, ad infinitum. A sporadic task generates no secondary jobs at all. Obviously there exist many variations between these extremes, including the case where an event gives rise to a finite chain of jobs, all with the same nominal release time (baseline) and the same deadline. We propose the term *chain-reactive* for the generic task model that covers all these cases.

While an in-depth study of the chain-reactive task model remains as future work, we note that many existing schedulability theories are relevant to Timber already as it is. For the (in Timber terms) common case where objects communicate only by means of asynchronous messages, Jeffay’s model of single-resource tasks applies [10]. This model is particularly interesting because

its schedulability conditions lead to a formal proof that the chosen scheduling strategy of EDF with *Dynamic Deadline Modification* is indeed optimal. The special case of *non-preemptive EDF* [9] is also relevant to Timber, as the semantics of the language is liberal enough to allow for non-preemptive scheduling as long as deadlines can be met. Baker’s *Stack Resource Policy* (SRP) is a general preemptive resource scheduling approach that covers Timber’s synchronous methods as well [1]. A particularly strong point with the SRP is that it would allow Timber programs to run using a single stack shared by all objects. We are currently exploring this possibility in the development of our next-generation Timber compiler.

Common to all the cited approaches is that schedulability of a program can be checked *statically*, at compile- or link-time instead of run-time. This possibility comes at the cost of some restrictions on program behavior, though: (1) tasks and resources (methods and objects) must not be dynamically created, (2) resource usage (object communication patterns) must be statically known, and (3) deadlines (**before** parameters) must be statically computable. Neither of these conditions are true for Timber programs in general.

However, Timber offers some clear advantages when it comes to identifying the subset of programs amenable to static schedulability tests. Namely:

1. Programs without dynamic object creation are easily identifiable as those where template instantiations only occur within other templates, never within actions and requests.
2. Static communication patterns can be identified by the property that object interfaces never occur as method parameters, only as parameters to templates.
3. Compile-time computable deadline expressions are precisely those that can be lifted to the top declaration level by means of program transformations (assuming termination is checked by other means).

Formalization of these conditions, as well as development of tools that would support the checking of them, are currently being planned.

## 6 Related work

The complexity of event-driven programming in mainstream languages is illustrated by the existence of formalized real-time models like ROOM and Rational Rose Real-time [17, 5]. The purpose of these models is to automatize as much as possible of the often tedious and error-prone construction of an event-driven infrastructure, by taking abstract system descriptions in terms of actors, messages, etc; and generating the major part of a corresponding implementation in a language like C++ or Java, mapped onto some specific operating system. However, despite references to the term real-time in their names, neither of these models support an especially well developed notion of time. This fact is addressed by Saksena *et al*, who extend ROOM with the concept of deadlines, and provide a set of guidelines for mapping these problem domain notions onto the priority-based thread abstractions of the presumed target system [16]. While these guidelines undeniably identify important areas of real-time system design, they can also be seen as a very concrete symptom of the semantic gap that does exist between the problem and solution domains of real-time systems.

In the realm of more domain-specific, but also more experimental, real-time programming languages we find designs that do provide general forms of timing constraints that, at least in principle,

could open up for deadline-based scheduling. Examples are Real-time Euclid [12], RTC++ [22], and CRL [20]. Hooman and van Roosmalen describe a generic language extension in the same spirit, exemplified with an unnamed language design that even comes with a formal definition [7]. However, a general remark regarding these approaches is that timing constraints apply to very fine grain program units (statement blocks in RTC++, individual statements in CRL and the Hooman/van Roosmalen design); which, by the presence of general threads and blocking constructs, do not correspond very well to the actual schedulable units as they appear at run-time. It is also the case that timing constraint arithmetic on this detailed level risks being rather elaborate, something which further reduces the value of these languages as real-time modeling tools.

Real-time Euclid is moreover an example of a language where resource-awareness has motivated severe restrictions in expressivity in order to simplify execution time and schedulability analysis [12]. This sparseness is also shared by some real-time languages based on the functional programming paradigm [21, 8, 15]. Here, however, the term real-time is actually identified with bounds on resources, not with any means for declaring constraints that specify some *desired* timing behavior.

Languages for reactive programming are often associated with the *synchronous* model of computation (Esterel, Signal, Lustre) [2]. In synchronous languages, computations are assumed to take place at specific instants, rather than being spread out in time. In practice this is achieved by conducting program execution by one or more periodic clocks, where all computations are assumed to terminate within a clock period. The need for scheduling and schedulability analysis is thus removed, at the expense of a fully synchronized system whose responsiveness is determined by the longest running computation. In contrast, each reaction in a Timber system can be individually constrained, and arbitrary approximations to the idea of instantaneous computation can be achieved by placing upper and lower time bounds sufficiently close together.

Another interesting variant of the fully time-driven approach is the language Giotto and its underlying model in the shape of the E-machine [6]. Here the programmer can specify that a reaction to an event should be delivered precisely at a certain point in time. Consequently, the output of a task will be queued until it is time to react, and so the E machine becomes a deterministic system. We take it as an axiom that a fully time-driven and deterministic approach constitutes an over-specification when used for the purpose of modeling general time-constrained systems.

## 7 Conclusion and future work

In this paper we have argued that a programming language for real-time systems should support the declaration of *time-constraints*, and that those constraints should attach to a well-developed notion of *reactions*. To make our claims more precise, we have introduced *Timber*, which is a concurrent programming language based on a model of non-blocking, reactive objects. Timber supports both upper and lower time constraints on a reaction, where upper constraints correspond to classical deadlines, and lower constraints constitute the systematic capturing of well-defined points in the future as events in their own right. A series of programming examples has illustrated how these mechanisms can be used to express simple solutions to common problems in practical real-time programming, and the relation between Timber and deadline-based scheduling theory has been explored.

Several directions for future work exist. One obvious path concerns finalizing a public Timber implementation, in order to enable gathering of experiences from a more diverse field of programmers. We have also mentioned that the overload behavior of Timber, not to mention deadline-based systems in general, is an area that requires further study.

In terms of program analysis for Timber, we plan to attack the general schedulability problem and its program structure preconditions, as well as the problem of establishing method execution times on various platforms. For both problems, we expect Timber to offer some interesting opportunities; the fact that state variables are guaranteed to be free from concurrency interference should for example simplify the task of establishing flow information.

Finally, even though the deadline-based semantics of Timber lies very close to the established field of EDF scheduling theory, there does not seem to be a straight-forward way of reasoning about resource requirements for the generic chain-reactive tasks made possible by our **after** construct in any existing analysis framework. This is an interesting new research problem that we are looking forward to study in more detail.

## Acknowledgments

The work reported in this paper was sponsored in part by DARPA, contract #F33615-00-C-3042, as part of the PCES program (Program Composition for Embedded Systems). This work has benefited from the comments of members of the Project Timber team and of the PacSoft center at OGI. Additional thanks to Björn von Sydow for insights and helpful discussions.

## References

- [1] T. P. Baker. Stack-based scheduling for realtime processes. *Real-Time Syst.*, 3(1):67–99, 1991.
- [2] A Benveniste, P Caspi, S A Edwards, N Halbwachs, P Le Guernic, and R de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.
- [3] A.P. Black, M. Carlsson, M.P. Jones, R. Kieburtz, and J. Nordlander. Timber: A programming language for real-time embedded systems. Technical Report CSE-02-002, Dept. of Computer Science & Engineering, Oregon Health & Science University, April 2002.
- [4] Magnus Carlsson, Johan Nordlander, and Dick Kieburtz. The semantic layers of Timber. In Atsushi Ohori, editor, *Programming Languages and Systems, First Asian Symposium, APLAS 2003, Beijing, China*, volume 2895 of *Lecture Notes in Computer Science*. Springer, November 2003.
- [5] G. Gullekson. Designing for concurrency and distribution with rational rose realtime. White paper, Rational Software, 2000.
- [6] Thomas A. Henzinger and Christoph M. Kirsch. The embedded machine: Predictable, portable real-time code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2002.
- [7] J. Hooman and O. van Roosmalen. An Approach to Platform Independent Real-Time Programming. *Real-Time Systems, Journal of Time-Critical Computing Systems*, 19(1):61–112, 2000.

- [8] John Hughes and Lars Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *International Conference on Functional Programming*, pages 70–81, 1999.
- [9] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *IEEE Real-Time Systems Symposium*, pages 129–139, 1991.
- [10] Kevin Jeffay. Scheduling sporadic tasks with shared resources in hard-real-time systems. In *IEEE Real-Time Systems Symposium*, pages 89–99, 1992.
- [11] Mark Jones, Magnus Carlsson, and Johan Nordlander. Composed, and in control: Programming the Timber robot. <http://www.cse.ogi.edu/~mpj/timbot/ComposedAndInControl.pdf>, 2002.
- [12] E. Kligerman and A.D. Stoyenko. Real-Time Euclid: A Language for Reliable Real-time Systems. *IEEE Transactions on Software Engineering*, SE-12(9), 1986.
- [13] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.
- [14] Johan Nordlander, Mark Jones, Magnus Carlsson, Dick Kieburtz, and Andrew Black. Reactive objects. In *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*, Arlington, VA, April 2002.
- [15] Alvaro Rebon Portillo, Kevin Hammond, Hans-Wolfgang Loidl, and Pedro Vasconcelos. Granularity analysis using automatic size and time cost inference. In *Proceedings of IFL '02—Implementation of Functional Languages*. Springer Verlag, September 2002.
- [16] M. Saksena, P. Freedman, and P. Rodziewicz. Guidelines for Automated Implementation of Executable Object Oriented Models for Real-Time Embedded Control Systems. In *IEEE Real-Time Systems Symposium*, 1997.
- [17] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.
- [18] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, pages 1175–85, 1990.
- [19] J.A. Stankovich, editor. *Deadline Scheduling for Real-time Systems, EDF and Related Algorithms*. Kluwer, 1998.
- [20] A.D. Stoyenko, T.J. Marlowe, and M.F. Younis. A Language for Complex Real-Time Systems. *The Computer Journal*, 38(4), 1995.
- [21] Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *International Conference on Functional Programming (ICFP '01)*, Florence, Italy, September 2001.
- [22] Y. Ishikawa et al. Object-Oriented Real-Time Language Design: Constructs for Timing Constraints. *SIGPLAN Notices*, 25(10):289–298, Oct 1990.