Composed, and in Control: Programming the Timber Robot

Mark P. Jones[1], Magnus Carlsson[1], and Johan Nordlander[2]

[1] Department of Computer Science & Engineering
OGI School of Science & Engineering at OHSU, Beaverton, OR 97006, USA
[2] Department of Computing Science
Chalmers University of Technology, S - 412 96, Göteborg, Sweden
http://www.cse.ogi.edu/pacsoft/projects/Timber/

Abstract. This paper describes the implementation of control algorithms for a mobile robot vehicle using the programming language Timber, which offers a high-level, declarative approach to key aspects of embedded systems development such as real-time control, event handling, and concurrency. In particular, we show how Timber supports an elegant, compositional approach to program construction and reuse—from smaller control components to more complex, higher-level control applications—without exposing programmers to the subtle and error-prone world of explicit concurrency, scheduling, and synchronization.

1 Introduction

This paper describes some programs that we have written to control a small robot vehicle called *Timbot* (the "Timber Robot"). The functionality provided by these programs and the specifications of the vehicle on which they run are not particularly unusual from the perspective of previous and current work on autonomous control and robotics. But the novelty of our work, and the focus of this paper, lies in our use of *Timber*, a new programming language that has been designed to facilitate the construction and analysis of software for embedded systems. In particular, Timber offers a high-level, declarative approach to several of the key areas for embedded systems development—such as real-time control, event handling, and concurrency. By comparison, the languages that have traditionally been used in this domain typically relegate such features to the use of primitive APIs or coding idioms, leading to programs that are harder to analyze, and more difficult to reuse or port to new platforms.

1.1 Background: Project Timber

The work reported here has been carried out as part of a larger effort called Project Timber ("*Time* as a *b*asis for *e*mbedded *r*eal-time systems") at the OGI School of Science & Engineering. One of the goals of Project Timber, and of

particular relevance in this paper, is to investigate the role that advanced programming languages can play in supporting the construction and analysis of high-assurance, portable real-time systems.

Another important goal for the project is to develop techniques and mechanisms for building systems that adapt dynamically to changes in their environment, in resource allocations, or computational load. By comparison, many systems today are brittle, and become unusable, or perhaps even fail outright, if used under conditions that their designers had not anticipated. For example, a live MPEG video stream can quickly become unwatchable if even just a small percentage of the underlying network packets are lost as a result of a drop in available bandwidth. An alternative is to build more flexible systems that can *gracefully degrade* the quality of the services they provide—according to user-specified policies—and still provide useful functionality. In the case of a live video stream, for example, we can respond to a reduction in bandwidth by reducing the frame rate, the image size, the color depth, or some user specified combination of these, and still continue to enjoy a live, real-time video stream. In fact, this very example was one of the motivations for including a video camera on Timbot, and we are investigating a somewhat different application for Timber as a tool for programming and analyzing adaptation policies. Further discussion of these topics, however, is beyond the scope of this paper.

1.2 Outline of Paper

The rest of this paper are as follows. In Section 2, we give an overview of Timbot, describing the hardware components from which it has been assembled.In Section 3, we introduce the Timber programming language by showing how it can be used to provide a software interface to the robot vehicle. In Section 4, we begin to use this interface to develop a library of reusable *controllers*, and then we show how these can be combined to implement more complex control applications in Section 5. Finally, in Section 6, we conclude with a review of future and related work.

2 Introducing Timbot

In this section, we provide an overview of Timbot, the small robot vehicle shown in Figure 1. Built on the chassis of a radio-controlled monster truck, Timbot hosts an on-board embedded PC (an 850MHz PIII, with 256MB ram, and a wireless 802.11b network adapter); an analog video camera on a pan-tilt mounting, which connects to the computer via a PC/104+ frame grabber; sonar and line tracking sensors; and a battery system that allows Timbot to be used either on the desktop or as a standalone vehicle. Recently, we have been using Timbot with a standard Linux distribution installed on a 1GB microdrive, which provides enough headroom to host a fairly rich development environment. However, we have also used Timbot in other configurations, replacing the microdrive with smaller compact flash cards, and using custom built Linux kernels with RTAI

Fig. 1. Timbot, the Timber Robot

for real-time support. Timbot has more memory and more computational power than the machines found in many industrial embedded systems; this reflects its intended use as a platform for experimentation and demonstration. Nevertheless, it still exhibits many of the characteristics—and raises many of the challenges—that occur in the development of a modern, sophisticated embedded system.

The block diagram in Figure 2 shows the connections between the main components of Timbot in more detail. At the center, the CPU is connected over

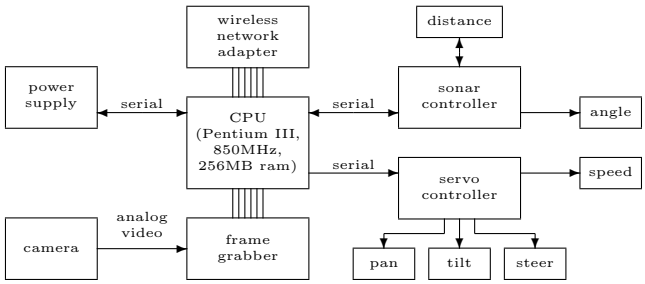

Fig. 2. A Block Diagram of Timbot

standard PC buses to a network adapter and to the frame grabber/camera combination in the lower left hand corner. Toward the top left, a power supply unit generates regulated voltage supplies for each component and handles charging of the batteries when Timbot is connected to a desktop power supply. The power supply is also connected to the CPU by an RS232 serial line that can be used, among other things, to query for an estimate of remaining battery power.

For this paper, however, our attention will be focused on the components in the right hand portion of the diagram, centered around the servo and sonar controllers. These two devices are also accessed via RS232 serial connections, and respond to multi-byte messages from the CPU by setting the physical position of a servo and, in the case of the sonar controller, pulsing an ultrasonic range finder to measure the distance to an obstacle. There are four servos on Timbot, and these can be set independently to adjust camera direction (pan), camera

attitude (tilt), steering direction (steer), and sonar direction (angle). In addition, an electronic speed controller (speed), which has the same electrical interface as a mechanical servo, is used to control the speed of Timbot's motors, and hence the speed at which it crosses the floor.

3 A Timber interface for Timbot

In this section, we describe the interface that Timber programs use to access and control the robot. This interface provides a bridge from the description of hardware in the previous section to the control programs that will come later on. We will also use this to introduce the Timber language and the idioms of reactive programming that it adopts. In the tradition of declarative languages, the goal of Timber is to allow programmers to describe *what* they want to accomplish, without having to worry about *how* it is achieved in terms of low-level concepts such as concurrent threads, scheduling, interrupts, and synchronization.

In discussing Timber code, we will comment on important details, but we do not attempt to provide a full tutorial or reference; further details may be found elsewhere [1]. In fact, Timber was derived from Nordlander's O'Haskell [3], which was, in turn, based on the functional language Haskell [7]. Where O'Haskell modified Haskell by providing a high-level approach to concurrency and reactivity via stateful objects, asynchronous messaging, and subtyping, the Timber language modifies O'Haskell by adding constructs to specify timing behavior and by adopting strict evaluation to facilitate analysis of worst-case execution times. We hope that the key aspects of our code will be clear from the accompanying text, but experience with Timber, O'Haskell, Haskell, or similar languages (in decreasing order of applicability) will be needed to understand the details.

3.1 The Timbot Interface

Our first fragment of Timber code is the definition of the `Timbot` type, which describes the software interface to the special hardware features of the robot:

```
record Timbot < Truck, CameraControl, Sonar

record Truck =
  speed :: Speed -> Action
  steer :: Angle -> Action

record CameraControl =
  pan   :: Angle -> Action
  tilt  :: Angle -> Action

record Sonar =
  angle :: Angle -> SonarListener -> Action
```

The first line tells us that `Timbot` is a combination of three interfaces for controlling vehicle movement (`Truck`), camera orientation (`CameraControl`), and

sonar usage (`Sonar`), respectively. The `Truck` and `CameraControl` interfaces are straightforward, with methods that take either a single angle or speed parameter and return an action that will move the corresponding servo to the specified position. More generally, actions represent "asynchronous message sends," and are an implicit trigger for concurrent execution, allowing new tasks to be executed (or, at least, scheduled for later execution) without delaying further execution of the code from which the action was invoked. (Note that there is no need to wait for the result of an action because actions do not return values.)

The definition of the `Sonar` interface requires more explanation. In a traditional programming language we might expect to access the sonar by calling a function that: sets the sensor direction; pulses the ultrasonic transducer; listens for an echo to obtain an estimate of distance; and returns the result to the caller. It has long been recognized, however, that so-called *blocking* operations like this are a significant source of complexity in the coding of concurrent and distributed systems [4], often requiring programmers to make assumptions, or to use encodings that can lead to deep but subtle bugs—such as deadlock—if they are not correct or if they are not applied correctly. Timber avoids these problems by eliminating blocking computations. There are no blocking primitives in the Timber library—no `getchar`, `read`, or `delay` methods, for example—nor any means for a programmer to construct a blocking method. Instead, Timber adopts a purely event-driven approach in which a program advertises methods (or "*callbacks*") that its environment can invoke to inform the program when input becomes available. Of course, this event-oriented approach is widely used in other languages, but it is weakened in many cases by the continuing presence of blocking operations, which makes it much harder to use in a reliable fashion.

For our interface to the Timbot's sonar, we avoid the need for blocking by passing two parameters to the `angle` method. The first is the angle for the sonar, while the second is a "listener" object that determines how the resulting distance reading will be passed back to the program.

```
record SonarListener =
  distance :: Maybe Distance -> Action
```

Notice that the `distance` method takes an argument of type `Maybe Distance`, indicating that it will either be a value of the form `Just x`—when a distance `x` is measured—or a value `Nothing`—when no measurement is obtained. This could occur if there is no object within range (approximately 2.7m) or if the ultrasonic signal from the sonar is absorbed instead of being reflected. By distinguishing this case explicitly within the type system, we may incur a little extra work in decoding and applying distance readings. Nevertheless, this is clearly preferable, at least if one is interested in reliable control programs, to encoding an out-of-range reading as a particular distance value, and hoping that programmers will always remember to check for the special case.

3.2 Implementations of the Timbot Interface

Every top-level Timber program is parameterized by an `env` argument of type `StdEnv` that allows the program to interact with the external environment in

which it is running. For example, the environment provides a `putStrLn` method that can be used in a command of the form `env.putStrLn msg` to display a message on the console. In the future, we plan to extend the environment with a `timbot` method that allows the interface to Timbot's hardware to be accessed in a similar way as `env.timbot`. Our current prototype, however, uses the following `getTimbot` function instead, whose implementation is discussed in Section 3.4.

```
getTimbot :: StdEnv -> Cmd Timbot
```

Of course, it is useful (and possible) to have other implementations of the `Timbot` interface. For example, we use one such interface for simple development and testing on machines other than Timbot. We are also constructing a more sophisticated implementation of the interface that connects Timber control programs to a simulator that can accurately model the motion and sensors of Timbot.

Now we can begin to write simple programs to control Timbot! The following program, for example, uses `getTimbot` to obtain an interface to Timbot (binding the variable `timbot` to the result), and then starts the robot moving at a constant speed with the steering turned all the way to the left. The result, of course, is to drive Timbot anticlockwise around the perimeter of a circle.

```
circle env = do timbot <- getTimbot env
                timbot.steer hardLeft
                timbot.speed 30
                after (20*seconds) (timbot.speed 0)
```

The `do` keyword introduces a sequence of four commands. The symbolic constant `hardLeft` gives the maximum angle to which the steering can be set for a left turn. (There is, of course, a corresponding value, `hardRight`, for right turns.) The speed setting of `30` corresponds to a (slow) forward speed; speed values range between -128 (reverse, at speed!) and +127 (forward, with haste!), but we have not yet attempted to calibrate speed in more traditional units.

Given only the first three lines, the definition of `circle` would, quite literally, send Timbot into an infinite loop: once a setting is made, the servo controller will work, even against physical pressure, to maintain it. To prevent the infinite loop, we included the final line to specify that `timbot.speed 0` should be executed 20 seconds after the program begins. The symbolic constant `seconds` is a multiplier that can be used to express time values. (There are similar multipliers for `milliseconds` and `microseconds`.) The use of symbolic constants makes it possible to express times in a platform independent manner, recognizing that multiplier values are likely to vary from one machine to the next.

This is also our first example of a timing annotation. In code like this, the `after` construct is intuitive and simple, but we will see that it is also powerful. As a first hint, we note that the `after` construct in this example is emphatically *not* the same as a 20 second (blocking) delay followed by the `timbot.speed 0` command. (Remember: there are no blocking operations in Timber!) Instead, it should be read as a high-level declaration of the time at which a specific action is to be performed. It would make no difference if the `after` construct were moved to the first line after the call to `getTimbot`; the semantics, and for all practical purposes, the observational behavior of the program would not be changed.

3.3 Monadic Programming in Timber

While Timber adopts a strict evaluation strategy like ML [2], it also relies on monadic programming [8] to encapsulate side-effects, and to facilitate analysis and optimization. Many of the programs in this paper run in the `Cmd` monad, meaning that they have a type of the form `Cmd t`, and correspond to a command that can be executed to obtain a result of type `t`. When no particular return value is needed, we typically substitute the unit type, written `()`, for `t`. For example, the `circle` program in the previous section is a function of type `StdEnv -> Cmd ()`.

In practice, most Timber programs use several different but related monads that distinguish, for example, between *methods*—which have access to the local state of an object—and *commands* (i.e., values with types of the form `Cmd t`)—which can invoke the methods of an object, but do not themselves have any local state. The use of different monads provides documentation and more precise information for programmers and program analysis tools alike. For example, the type system allows us to distinguish several special types of command using subtypes of `Cmd t`: commands that execute a synchronous request have a type of the form `Request t`; actions—which are commands that execute an asynchronous method call—have type `Action`; and commands that instantiate a template to construct an object of type `t` have types of the form `Template t`.

Timber also provides special syntax for these different kinds of command. Requests are written as a sequence of commands prefixed by a `request` keyword instead of the `do` that we saw in the definition of `circle`. Return values are specified by commands of the form `return e`. Actions are written in a similar way, but prefixed by the `action` keyword. Of course, actions do not (and cannot) specify return values. The syntax for templates uses expressions of the form `template` *local* `in` *e* to denote a command that, when executed, will construct a new object whose local state variables (if any) are initialized by the (possibly) empty list of statements in *local*, and whose interface is specified by the expression *e*. The interface to an object will often be a record, but this is not required; unlike other object-oriented languages, Timber treats objects and records as orthogonal language features.

3.4 Implementation Details

It is quite easy to implement the `getTimbot` function of Section 3.2 using standard Timber libraries that work with character devices. For reasons of space, we do not include the code here, and restrict ourselves to a brief discussion of the most important issue: synchronization. For example, from the description of Timbot in Section 2, it is clear that truck control settings and camera control settings must be multiplexed through the same serial link to the servo controller. Some kind of synchronization is needed in situations like this to avoid giving concurrent tasks simultaneous access to the same device. The sonar controller might easily become confused, for example, if the multi-byte messages for two different control tasks were accidentally interleaved. Many languages, however, require

explicit coding of synchronization, which works against the goals of abstraction because it assumes that programmers will have enough information about the underlying implementation to understand when synchronization is required and to know how it should be achieved. In Timber, these problems are solved at the language level—its semantics guarantee that each object is treated as an implicit critical section, meaning that at most one of its methods is active at a time. As such, synchronization is implicit in Timber code, with the task of determining where it is actually necessary being left to the underlying implementation.

4 The Controller Abstraction

While embedded systems may use sophisticated sensors and actuators to engage in complex interactions with their environment, many present a much simpler interface to their human users: an on/off switch! This is typical for systems that are designed to operate autonomously, without frequent user input once they have been turned on. We have already found several examples of this in the programs that we have been writing to control Timbot, both in small components, and in complete programs, which we can express in timber by using the following `Controller` abstraction:

```
record Controller =
  start :: Action
  stop  :: Action
```

4.1 An Acceleration Controller: First Attempt

As an example, the following code defines a controller that will set a `timbot` in motion, accelerating from rest by increasing the vehicle's speed by `incr` units after each period of time `t`, but never exceeding the specified `maxSpeed`.

```
accelControl :: Speed -> Speed -> TimeDiff -> Timbot -> Template Controller
accelControl maxSpeed incr t timbot
 = template running := False
   in let accel s = action if running then
                             timbot.speed s
                             let s' = s + incr
                             if s' <= maxSpeed then
                               after t (accel s')
   in record start = action if not running then
                             running := True
                             accel 0
             stop  = action running := False
                            timbot.speed 0
```

This controller is useful in practice because it reduces the possibility of a damaging jolt that could occur if we set the speed of `Timbot` directly to the target speed. The key here is the `accel` method that is called with a zero speed setting when the controller is first started. Subsequent recursive calls increase the

speed in steps using `after` to ensure that they are distributed correctly over time. Additional logic, using a Boolean state variable `running`, will terminate the acceleration if the controller is stopped before the target speed is reached.

Unfortunately, our definition has a serious flaw: if the controller is turned off, but then turned back on again before the next `accel` step is executed, then we will continue with the sequence of `accel` calls initiated when the controller was first started, while also generating a second sequence of calls for the later start. Such behavior is unlikely to produce satisfactory results!

4.2 An Acceleration Controller: Second Attempt

Clearly, it is not enough for an acceleration controller's `stop` method just to reset the running flag and bring the vehicle to a stop with `timbot.speed 0`; it must also cancel any pending calls to `accel`. It is easy to implement this using standard Timber library functions. But we will go a step further and generalize to obtain an abstraction that can be used in other contexts, while also neatly encapsulating our solution to the bug in the original `accelControl`. The benefit, of course, is that other programmers can then use this more general construct to build new controllers more concisely, without recreating our original bug.

We start with the definition of a new subtype of `Controller` that adds a method called `invoke`. This new method will be invoked immediately after the controller is started. Each time it is called, however, it returns a value of type `Maybe TimeDiff`, indicating when (if at all) it should next be invoked.

```
record RepeatController < Controller =
  invoke :: Request (Maybe TimeDiff)
```

Now we can use an object of this type to build a controller with the correct behavior using the following `startstop` function:

```
startstop :: RepeatController -> Template Controller
startstop rc
 = template running := False
            sc <- singlecall
   in let tick = action mpa <- sc.invoke
                        case mpa of
                           Just t  -> sc.call (after t tick)
                           Nothing -> done
   in record start = action if not running then
                               running := True
                               rc.start
                               tick
              stop  = action if running then
                               running := False
                               sc.cancel
                               rc.stop
```

Note the use of the single call object `sc`, which allows our program to schedule the execution of an action `a` using `sc.call a`, but also allows that action to

be canceled, if it has not already started, using `sc.cancel`. This provides the missing feature that we needed to avoid the original `accelControl` bug, and is included as part of the Timber libraries.

Now, for example, we can recode our `accelControl` controller more concisely, and without the bug, using the following definition:

```
accelControl maxSpeed incr t timbot
 = template s    := 0
            ctrl <- startstop (record
                      start  = action s := 0
                      stop   = action timbot.speed 0
                      invoke = request
                        timbot.speed s
                        s := s + incr
                        return (if s <= maxSpeed
                                  then Just t else Nothing))
   in ctrl
```

Notice that all of the logic associated with the `running` flag has been captured and hidden away by the use of `startstop`.

4.3 Imperative versus Declarative: A Matter of Style?

Our definitions of `accelControl` have an imperative feel, which some readers may feel detracts from the goals of Timber as a declarative language. This is subjective, but we note also that it often comes down to a debate about programming style. The following alternative definition, for example, while retaining some imperative elements, avoids the explicit recursion in the original:

```
accelControl maxSpeed incr t timbot
 = let profile = zip [0,t..] [0,incr..maxSpeed]
   in template mc <- multicall
   in record start = action
                       forall (ti,si) <- profile do
                         mc.call (after ti (timbot.speed si))
             stop  = action
                       mc.cancel
                       timbot.speed 0
```

This code defines a list of (time,speed) pairs called `profile` that describes the complete acceleration process. This list is used when the controller is started to generate a corresponding sequence of (time-delayed) actions. (The `forall` construct—which might suggest a loop in an imperative language—is really just convenient syntactic sugar for a standard operation on lists.) The only real difference here is the use of a `multicall`, which behaves much like a `singlecall`, except that it enables us to call (and subsequently cancel) a collection of multiple pending actions. The `multicall` method used here is not included in the current Timber libraries, but will perhaps be added in a future version.

4.4 Other Controller Components

As we write programs to control Timbot, we are collecting a library of reusable components, like `accelControl`, that are useful in other applications. In this section, we describe three representative examples from this growing collection.

Our first example is a `periodicControl`, which will execute a particular command at regular intervals for as long as the controller is turned on. Its definition is a simple application of `startstop`:

```
periodicControl      :: TimeDiff -> Cmd a -> Template Controller
periodicControl t cmd = do rc <- template in
                                 record start  = action done
                                        stop   = action done
                                        invoke = request cmd
                                                         return (Just t)
                           startstop rc
```

No special actions are needed (beyond those already handled by `startstop`) when a `periodicControl` component is either started or stopped, so the trivial action, `done`, is specified for these two methods.

Our second example is `sweepControl`, which can be used to sweep a device (such as the sonar, or the camera) across a range of different angles (between `minA` and `maxA`), changing the angle by some fixed `incr` after each `t` units of time, and triggering an appropriate action at each point. For this example, we use an object with a state variable `angle` that records the current angle, and a Boolean state variable `incr` to indicate if the angle is currently increasing or not (i.e., moving from `minA` to `maxA` or from `maxA` to `minA`). As one might hope, the periodic stepping of `sweepControl` is captured naturally using `periodicControl`.

```
sweepControl (minA, maxA, stepA, t) act
 = template
     angle := minA
     incr  := True
     ctrl  <- periodicControl t
               (action act angle
                       if incr then angle := angle + stepA
                                    if angle > maxA then
                                      angle := maxA
                                      incr  := False
                               else angle := angle - stepA
                                    if angle < minA then
                                       angle := minA
                                       incr  := True)
    in ctrl
```

Almost all of the code here deals with the specific needs of a `sweepControl` component—much of which has to do with calculating how the sweep angle should change from one step to the next. There is, by comparison, very little in the way of boilerplate code, because that has already been packaged away for us in abstractions like `periodicControl`.

Our third example demonstrates a different style of definition. In this case, a `multiControl` controller can be used to start/stop each of the elements in a list of controllers from a single start/stop command.

```
multiControl   :: [Controller] -> Template Controller
multiControl cs = template in record
                    start = action forall c <- cs do
                                    c.start
                    stop  = action forall c <- cs do
                                    c.stop
```

The following simple program illustrates how the components described above can be combined to construct simple control programs for Timbot. This particular section of code, for example, constructs independent sweep controllers, each operating at a different frequency, for the camera pan and tilt, and then uses `multiControl` to package them into a single controller.

```
do cam <- getTimbot env
   pc  <- sweepControl (-60, 60, 6, 50*milliseconds) cam.pan
   tc  <- sweepControl (-30, 30, 2, 80*milliseconds) cam.tilt
   multiControl [pc, tc]
```

In this example, we are simply using Timber to describe, at a high-level, the construction and connections between a group of reusable control components. What the language hides are the subtle and sometimes complex concurrency, scheduling, and synchronization issues that are needed to weave the code from each component together with the intended timing.

5 Control Applications

In this section we describe some simple control programs that can be constructed from the components in previous sections. In particular, these programs are designed to use information obtained from sonar. The most important details to notice in these examples are the ease with which components can be combined, and the clarity that results from the implicit treatment of concurrency. Note also that each example is packaged using the same controller abstraction as the components from which they are built. As a result, these examples could in turn be reused as components in a larger, more complex system.

5.1 Simple Obstacle Avoidance

In this section we show the code for a simple obstacle avoidance program that drives the robot forward while sweeping the sonar across the path in front of it to look for obstacles. (For the purposes of this paper, an obstacle is any object that is within 1m of the robot on its forward path. In practice, we often test Timbot by stepping into the path of the robot and using ourselves as obstacles!) If an obstacle is detected, then the robot will stop, but continue scanning in the

hope that the obstacle will move. If a full second passes with no obstacle being detected, then the robot will once again begin accelerating forward again.

The code for `simpleObstacleAvoid` is straightforward, obtaining a `timbot` interface; building an accelerator controller, a listener for the sonar, and a controller to sweep and trigger the sonar; and gluing these pieces together.

```
simpleObstacleAvoid env
 = do timbot <- getTimbot env
      accel  <- accelControl 50 10 (500*milliseconds) timbot
      lstn   <- obstacleLstn accel
      sweep  <- sweepControl (-12, 12, 2, 100*milliseconds)
                  (\a -> timbot.angle a lstn)
      return sweep
```

The main control logic is provided by the listener that receives distance measurements from the sonar, which we construct using the `obstacleLstn` function:

```
obstacleLstn :: Controller -> Template SonarListener
obstacleLstn ctrl
  = template lastTime := 0
    in record distance d
                = action t <- currentBaseline
                          case d of
                            Just x | x<1.0 -> lastTime :=t
                                              ctrl.stop
                            _              -> if t > lastTime+1*seconds
                                                then ctrl.start
```

This listener records the time at which an obstacle was last detected in a private state variable `lastTime`. Each time the sonar reports a distance, the listener checks to see if it indicates the presence of an obstacle. Note that an out-of-range reading is treated, perhaps rather dangerously, as an indication that no obstacle has been seen! The careful reader might also spot that `obstacleLstn` is a candidate for reuse because it can be connected to an arbitrary controller, and not just to the `accelControl` that is used in `simpleObstacleAvoid`.

5.2 Wall Following

In this section, we show the Timber code for another well-known example of autonomous robot control: wall-following. The goal of this application is to drive the robot along at a fixed distance from a wall on its right hand side. If the robot gets too close to the wall (below a distance `minD`), then we steer the robot to the left, and away from the wall. On the other hand, if it gets too close (greater than a distance `maxD`), then it will steer to the right, and toward the wall. (For readers not familiar with this particular problem, we should note that this simple strategy will only work correctly within certain parameters—we assume that the vehicle begins parallel to the wall at a distance within `minD` and `maxD`, and that it does not move or turn too quickly.)

Again, the code breaks into two pieces, the first of which constructs and connects components, while the second encodes the main logic in a listener.

```
wallFollow minD maxD env
 = do timbot  <- getTimbot env
      accel   <- accelControl 40 10 (500*milliseconds) timbot
      lstn    <- wallLstn timbot accel minD maxD
      trigger <- periodicControl (200*milliseconds)
                                 (timbot.angle 70 lstn)
      multiControl [trigger, accel]

wallLstn :: Timbot -> Controller -> Distance -> Distance
                -> Template SonarListener
wallLstn timbot ctrl minD maxD
 = template in record
     distance d = action
       case d of Just x  | x < minD  -> timbot.steer hardLeft
                         | x > maxD  -> timbot.steer hardRight
                         | otherwise -> timbot.steer 0
                 Nothing             -> ctrl.stop
```

The listener that we have used in this example responds a little differently, and perhaps too cautiously, to an out-of-range reading from the sonar (the case for `Nothing` in the definition above) by assuming that this indicates the end of the wall, and so bringing the vehicle to rest. In fact, an out-of-range reading might also have been the result of a transient glitch. We leave it as an exercise to the reader to extend the definition here to delay stopping the vehicle until several consecutive out-of-range readings have been received, and so reduce the chance that the robot might stop prematurely,

6 Future and Related Work

The examples in this paper have demonstrated how Timber can be used to support an elegant and compositional approach to the construction of simple control algorithms for the Timbot robot vehicle. The high-level treatment of concurrency is particularly useful in avoiding the need for programmers to deal explicitly with the thorny issues of synchronization, scheduling, etc. As we continue to develop more interesting and more sophisticated control programs, we are also building a useful library of flexible and reusable control components.

There have been several other attempts to explore the use of declarative languages in similar application domains. Rees and Donald [9], for example, showed how the abstraction mechanisms of Scheme can be used in robot control, but also relied on explicit concurrency and synchronization. Wallace and Runciman [10] showed how functional languages can be used to describe an embedded controller for a lift shaft, but adopted a more primitive process model that allows processes to receive only one type of message. Most recently, *Functional Reactive Programming* (FRP) has been used to provide declarative specifications of

event-based programs with continuously time-varying *behavior* functions. The FRP style has been used in a number of applications including robot control [6, 5], where a special *task* monad is used to sequence tasks and track the robot state. More detailed comparison of Timber and FRP is a topic for future work.

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 and SySL centers at OGI. Particular thanks: to Perry Wagle for considerable assistance in building Timbot, and for suggesting and prototyping interesting control applications; and to Andrew Black, Dick Kieburtz, and James Hook for helpful insights and encouragement.

References

[1] Andrew P. Black, Magnus Carlsson, Mark P. Jones, Richard Kieburtz, and Johan Nordlander. Timber: A programming language for real-time embedded systems. http://www.cse.ogi.edu/PacSoft/projects/Timber/, April 2002.

[2] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised.* MIT Press, 1997.

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

[4] 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.

[5] John Peterson, Gregory D. Hager, and Paul Hudak. A language for declarative robotic programming. In *Proceedings of the IEEE International Conference on Robotics and Automation*, Detroit, MI, May 1999.

[6] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. In *Proceedings of Principles and Applications of Declarative Languages (PADL '99)*. Springer-Verlag, 1999.

[7] Simon Peyton Jones and John Hughes, editors. *Report on the Programming Language Haskell 98, A Non-strict Purely Functional Language*, 1999. Available from `http://www.haskell.org/definition/`.

[8] Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th Symposium on Principles of Programming Languages (POPL '93)*. ACM, January 1993.

[9] Jonathan A. Rees and Bruce R. Donald. Program mobile robots in scheme. In *Proceedings of ICRA '92, the IEEE International Conference on Robotics and Automation*, 1992.

[10] Malcolm Wallace and Colin Runciman. Lambdas in the liftshaft - functional programming and an embedded architecture. In *Proceedings of Functional Programming and Computer Architecture, (FPCA '95)*, La Jolla, California, June 1995. ACM Press.