Proceedings of the 2002 IEEE
International Conference on Robotics & Automation
Washington, DC • May 2002

Specifying Behavior in C++

Xiangtian Dai and Gregory Hager
Department of Computer Science
The Johns Hopkins University

John Peterson
Department of Computer Science
Yale University

Abstract

Most robot programming takes place in the "time domain." That is, the goal is to specify the behavior of a system that is acquiring a continual temporal stream of inputs, and is required to provide a continual, temporal stream of outputs. We present a reactive programming language, based on the Functional Reactive Programming paradigm, for specifying such behavior. The major attributes of this language are: 1) it provides for both synchronous and asynchronous definitions of behavior, 2) specification is equational in nature, 3) it is type safe, and 4) it is embedded in C++. In particular the latter makes it simple to "lift" existing C++ libraries into the language.

1 Introduction

The appropriate languages, language structures, and software architectures for developing complex robotic systems has been the topic of debate and discussion since the earliest, computer-controlled robots. The fact that no uniform consensus (at least in terms of a "universal" robot programming system) emerged is not surprising, given the range of issues such a language would need to address.

One set of issues arises from the need to integrate and react to sensing from a dynamic and, potentially unpredictable world. Thus, a language should make it simple to express continuous control as well as discrete reactivity. Methods such as prioritization of control [2] or monitors [12, 20] have been used in the past to deal with such issues. Dataflow languages [7, 4] have been employed for their expressive clarity. Charon [1] uses a hybrid system model as a language model for control. Whatever its form, a language should make it possible to use or create abstractions that are appropriate to the problem at hand.

A second set of issues arises from the nature of the tasks that autonomous robots would (ideally) perform. Many of these tasks are easily specified at an abstract level (e.g. deliver the mail, give a tour, or collect these objects), but their execution ultimately entails a complex interaction and sequencing of low-level behaviors. This has led to the development of layered systems [20, 19], and has also motivated the development of languages as a component of planners [6, 11]. Ideally, a robot programming language should span these layers, thereby providing a uniform means for creating abstractions and relating them to one another.

Finally, a third set of issues arises due to the practical problems of developing robot system software. Most robotic systems are built on pre-existing libraries of code, typically written in C or C++. Thus, it should be easy to use this code base for further development. At the same time, robot system programming is highly experimental in nature: we often do not know at the outset how best to use sensors, control actuators, set thresholds, and so forth. Thus, there is often a long cycle of development, testing, redevelopment, and refinement until a system reaches the point where it operates reliably. As a result, rapid, correct prototyping and software reuse are of paramount importance.

A more subtle practical problem is the fact that, at any given time, the computation needed to allow a robot to achieve its objectives is highly dynamic. More precisely, from one time interval to the next, there may be a large shift in the sensors employed, the algorithms used to process that information, the control algorithms currently operating, and the surrounding system monitoring that may be taking place. A good language should spare the programmer the details of explicitly managing the flow of computation, while keeping code execution to the minimum necessary to perform the currently active computations.

In the last few years, we have been investigating the *Functional Reactive Programming (FRP)* paradigm as a means for creating robot programs. Developed originally as a *Domain Specific Language (DSL)* for reactive animation [10], FRP has been applied to vision, robotics and other control systems applications among the programming community [5, 18, 17]. In terms of the previous discussion, we have shown that FRP is capable of expressing many of the previously cited language architectures [15], and is an elegant basis for expressing commonly used algorithms in vision and robotics [16, 8, 18]. FRP employs a lazy execution model, and therefore minimizes computation. It uses a declarative, equational style of expression that is compact and highly expressive, yet type-safe.

Until recently, however, all implementations of FRP have been based on Haskell, a pure functional language. In this paper we demonstrate that it is possi-

0-7803-7272-7/02/$17.00 © 2002 IEEE

ble to implement many of the essential aspects of FRP in C++, a widely used imperative programming language. As a result, we reap several benefits. First, it is much simpler to incorporate existing functionality into the language. Second, we can easily embed FRP constructs within other C++ programming abstractions. Finally, we are able to create a "leaner" execution environment targeted specifically to FRP (as opposed to general functional programming).

In the remainder of this paper, we develop the basic structures of FRP in C++, and illustrate its use on an example application that we have implemented in our laboratory.

2 Basic FRP in C++

The core data types in FRP are conceptually: 1) a continuous signal (referred to as a *continuous behavior*), 2) a synchronous, uniformly time-sampled signal (referred to as a *discrete behavior*), and 3) a discrete, asynchronous event (simply referred to as an *event*). We have chosen to implement only 2 and 3, as they are what most commonly occur in the robotics applications we are interested in.

Discrete behaviors and events are implemented as C++ templated data types `Behavior<T>` and `Event<T>`, respectively. Conceptually, these data types both represent "infinite" streams of data of type `T`. For example, an animation might be `Behavior<Image>`, the position vector of a visual tracker could be `Behavior<Pair<int,int>>`, and the location of the mouse on the keyboard when a button is hit might be `Event<Pair<int,int>>`. One important difference is that behaviors have a data value for every "clock tick[1]," whereas events do not. However, in both cases it is possible to access the time associated with any value the stream.

Computation in FRP is specified using equations that describe how behavior and event streams are related to one another. An FRP *program* is just a set of mutually recursive behaviors and events that has an identified "sink," and where no value mentioned in an equation is left undefined. The resulting program is executed by "pulling" on the sink, which in turn provokes exactly the computations required to produce a value for the sink within the program.

In order to illustrate these ideas, we describe the complete implementation of a video-based, interactive, event-driven interface for mobile robot navigation. The interface (shown in Figure 1) consists of a live video image, together with two buttons that are used to change robot control modes. At any time the system is always performing any or all of three

[1]In the implementation reported here, there is only a single clock. However, the extension to multiple clocks is straightforward, see [17]

Figure 1: A picture of the interface.

basic operations: goal-seeking control, obstacle avoidance, and monitoring for collisions (using a bumper). Goal-seeking control consists of moving in a direction supplied by the user by clicking in the image (manual control) or toward a visually tracked target (automatic mode). Goal-seeking control includes a wall-follower which takes over whenever the robot nears a wall (sensed using a range sensor). Obstacle avoidance and collision detection are constantly performed unless disabled through the on-screen icons. Thus, using this interface it is possible to drive toward an object and, once close, turn off obstacle avoidance to move up to it. Once in contact, collision detection can be disabled to push the object about.

2.1 Behaviors

In FRP, there are a variety of operators that produce or compose behaviors. The simplest behaviors are constant behaviors, which can be achieved simply by casting, e.g. `(Behavior<int>)1`. Behaviors can also be derived from from ordinary C++ functions using an operation called a "lift." For example,

```
Behavior<double> x;
Behavior<double> y = liftB(sin)(x);
```

`liftB(sin)` produces a new "lifted" version of `sin` that expects `Behavior<double>` and produces `Behavior<double>`. By extension, common operators, e.g. "+", "-", "*" and "/", can be lifted and, in our implementation, have been overloaded to operators on behaviors. Thus, for example, we can write

```
Behavior<double> a, b, c ;
BehaviorFn<double,double> sinB = lift(sin);
c = ( sinB(a) + sinB(b) ) / 2.0 ;
```

Note that we have now lifted `sin` as an independent function[2] that can be reused as many times as desired.

[2]Here and elsewhere we have taken the liberty of using type signatures that are somewhat simpler, but functionally equivalent, to those used in the actual system.

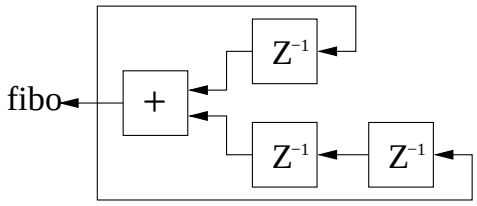

Figure 2: Dataflow view of the Fibonacci series.

As noted above, one important property of equations on behaviors is that values are computed *at most once* for each time step. That is, if we were to now write

```
d = c + c
```

the value of `c` would be computed only once for each value of `d`. Aside from minimizing the amount of computation needed to evaluate an expression, this also guarantees that all (active) behaviors are *time synchronized.* That is, after every computation, every active behavior contains a value that pertains to the most recently acquired input data.

Sometimes the value of one behavior at the current time relies on values of this and/or other behaviors at previous times. In this case, we need to be able to write "recursive" equations: groups of equations where the same value appears on both the left and right sides. Based on the discussion above, this should be impossible, as it would imply that such a variable is non-causal. In order to allow this, we introduce the expression `delayB` to delay a behavior for one sampling step, given an initial value. A variation, `delay1B`, simply replicates the first value that occurs in a behavior, thus avoiding the need to separately specify an initial value.

We can now describe a wide class of interesting computations. For example, a linear predictor can be expressed in mathematical terms as:

$$\hat{x}_{i+1} = x_i + k(x_i - x_{i-1})$$

It is now straightforward to write its equivalent behavior:

```
Behavior<T> x_pred = x + k*(x-delay1B(x)) ;
```

A natural abstraction for such systems of behaviors is a dataflow diagram. For example, Figure 2 shows a dataflow diagram for the familiar Fibonacci series. The corresponding equation could be written in either of the following ways:

```
Behavior<int> fibo1, fibo2 ;
fibo1 = delayB(0)( fibo1 )
      + delayB(0)( delayB(1)( fibo1 ) );
fibo2 = (delayB(0) <<= fibo1 )
      + (delayB(0) <<= delayB(1) <<= fibo2 );
```

Both `fibo1` and `fibo2` are functionally equivalent; the latter, however, emphasizes the dataflow aspect of the language. Both `a <<= b` and `a(b)` means "to apply (function) `a` to `b`".

Turning to our robot example, consider a differential drive system controlled using a speed value and a rotational velocity. We introduce variables for controlling this robot by writing

```
Behavior<double> speedB ;
Behavior<double> rotationB ;
Behavior<int> robotexampleB
  = liftB(setvelocity)(speedB,rotationB) ;
```

where `setvelocity` is some predefined C or C++ library function. We note that handling speed and rotation as separate behaviors is a matter of convenience for subsequent discussion. They could have just as easily been combined into a vector or similar data structure.

In what follows, we'll be generating control errors and closing control loops. Thus, it makes sense to define the following function:

```
Behavior<double>
PIDcontrollerB(double ig,double dg,double pg,
               Behavior<double> errorB)
{
   Behavior<double> integralB
      = delayB(0.)(integralB) + errorB;
   Behavior<double> derivativeB
      = errorB - delayB(0.)(errorB);
   return pg * errorB + ig * integralB
      + dg * derivativeB;
}
```

Note that this is the first time that we have created a program with *hidden state.* That is, once created, it is no longer possible to access the values associated with the integral or derivative terms of the controller.

It is worth noting that the definition of `PIDControllerB` could also be done inline using the C++ equivalent of a "lambda expression," similar to lambda expressions in functional languages [3, 14]. Furthermore, in some cases it is useful to be able to perform two operations without creating a named function: function argument binding, and function composition. FRP/C++ allows this using the notion of a "Functoid" [13]. For example, we can also write

```
Fun2<Behavior<double>,double,Behavior<double> >
  PController = liftF(PIDcontrollerB)(0,0)
```

Here, `liftF` creates a functoid object that acts like a normal function, and can form a new functoid by binding its first two arguments to zero. More generally, it is possible to bind any subset of the arguments of any function, and to compose two functions without evaluating them. Here is an example of function composition:

```
Fun1<int,int> add1=lambda(x,x+1), add2=add1(add1);
```

Returning to our robot example, we assume that there is a library function in C++ that acquires range

data, and likewise a function that analyzes that data to return the distance and orientation to a wall as well as a binary value indicating whether a valid "wall" object was found. For simplicity, we'll assume the latter values are returned as a record with three components, and that we've lifted functions to pull out those components. For now, forward speed will be set at a constant and we will only control rotation. Putting this together, we have[3]

```
normalspeedB = 10.;
rangedataB = liftB(rangedata)();
lineB = liftB(fitline)(rangedataB);
followallB = rotationvalB(lineB)
  + PController(GAIN1,(distanceB(lineB)-SETPOINT));
```

Note that we are ignoring the boolean value returned by `line`. One option for using it is to employ a "continuous if" as follows:

```
speedB = ifB( foundB(lineB), normalspeedB, 0. );
rotationB = ifB( foundB(lineB), followallB, 0. );
```

Now, executing `robotexampleB.run()` would the robot move along a wall whenever one was present, and sit still otherwise. In order to make it "smarter" we need to use Events.

2.2 Events

As noted earlier, `Event<T>` is a time-ordered sequence of event occurrences, each carrying a value of type `T`. For example, a left button press `Event<Position> lpb` and a keyboard press `Event<char> key` are two events that can be captured from the underlying window system. One special event worth noting is

```
Event<T> neverE () ;
```

which constructs an event that never happens.

An important operator on events is the *merge* operation:

```
Event<T> operator || ( Event<T>, Event<T> );
```

It "merges" two event sequences of the same type into one. In some cases, the type restriction on events is unnecessary — e.g. we may just want to know if one of several events has happened without worrying about the value carried by the event. We can use `castE` to cast an event to another type for which a valid C++ cast operator is known. For example, using `castE` we can write

```
Event<void> anyEvent = castE<void,char>(key) ||
         castE<void,Position>(lbp || rbp || mbp);
```

It is also possible to use a function to transform the value of an event stream using `liftE`.

Events can be derived from behaviors and vice versa. `whileE` turns boolean behaviors to events using the rule that the event occurs as long as the value of the behavior is true; `whenE` is same as `whileE` except that it only occurs when the variable *becomes* true. Returning again to our example, we can now define events that are triggered if wall following succeeds or fails:

```
foundwallE = whenE(foundB(lineB));
lostwallE = whenE(!foundB(lineB));
```

A few other useful operators are `snapshotE`, which captures the value of a continuous behavior at the time of an event occurrence, and `stepB` which creates a behavior capturing and holding the most recent value in an event stream. An event stream can be filtered by a function using `FilterE` to only pass the events for which the function returns true. Finally, `timeOfE` captures the time of an event. For example, the following function returns a boolean behavior which switches whenever an event occurs:

```
template<class T>
Behavior<bool> flip_flop( Event<T> e )
{
  Behavior<bool> b ;
  b = stepB(true)(snapshotE(e,!(delayB(true)(b))));
  return b ;
}
```

Recall that we can capture button events from the underlying window system. Using the functionality defined above, we can now filter button events to create a new event stream related to the icons on the screen. We can then resample those icons into two "flip-flops" defined above that provide the current high-level state of the interface:

```
buttonE = display.lbp() ;
icon1hitE = filterE(liftF(contains,area1))(buttonE);
icon2hitE = filterE(liftF(contains,area2))(buttonE);
icon1stateB = flip_flop( icon1hitE );
icon2stateB = flip_flop( icon2hitE );
```

Given this, we can now create a bumper event that only occurs when the bumper is depressed *and* reaction to the bumper is enabled:

```
bumperE = whenE(liftB(frontbumper)()&&icon2stateB);
```

Finally, we define how the robot turns to the direction given by a left button press when in manual guidance mode. Here, we need to do the following: capture the mouse position on the screen and transform it to an angle; record the orientation of the robot when the button was pressed; and finally create an error term that is the difference between the desired and current orientation. We assume the existence of `pos2angle` which converts a screen position to a pan angle. The equations are:

```
directionE = liftE(pos2ang)(display.lpb());
desireddirectionB =
   stepB(0.)(snapshotE(directionE,orientationB))
 + stepB(0.)(directionE);
directionerrB = desireddirectionB - orientationB;
```

[3]Henceforth, we drop declaration of variables with the understanding that anything ending in "B" is a behavior of the appropriate type.

2.3 Switches

Thus far, we have described how how to produce and operate on behaviors and events. Arguably, similar notions exist in a wide variety of stream-processing models, e.g. [7]. However, what we cannot do (at least not explicitly), is to condition the data produced in a behavior on the occurrence of an event.

Switching is the FRP means for doing just that. Intuitively, imagine creating a data flow diagram where, when an event occurs, a new portion of the data flow graph is "switched" into the data flow. This picture is a little misleading, as it suggests that the new portion of the graph previously existed, and is merely activated. In fact, FRP switching is accomplished by *creating* a new behavior and activating it within the running computation.

In order to talk about switching, it is essential to realize that a behavior is a datatype like any other. Hence, it is possible to define the data type `Event<Behavior<T>>`, that is an event which, when it occurs, supplies a value of `Behavior<T>`. Using this data type, various event mapping operators can be defined. Examples include `ThenB`, which can be used to create an event of behaviors, and the `switchB` and `tillB` operators, which switch new behaviors into the data stream.

```
Behavior<T> switchB (Behavior<T>, Event<Behavior<T>>);
Behavior<T> tillB (Behavior<T>, Event<Behavior<T>>);
Event<B> thenConstB ( Event<X>, B );
Event<B> thenB ( Event<X>, B (*)(X)  );
```

From the type signatures, we see that `thenB` executes a function (a constant function in the case of `thenConstB`) that returns a value that is "packaged" as an event. The type `B` should be a behavior, i.e `Behavior<T>`. Although `switchB` and `tillB` have the same type signature, they operate differently. Both initially produce the value of their first (behavior) argument, but on an event `switchB` switches to the behavior carried by the event each time it occurs; `tillB` switches on the first occurrence and ignores subsequent events.

Now we can rewrite[4] the equation of `speedB` and `rotationB` to produce a more complete wall follower

```
stopkeyE = filterE(lambda(x,x==' ')) <<= keypress;
stopE = stopkeyE || lostwallE ;
speedB = switchB( normalspeedB,
    ||(foundwallE ThenConstB normalspeedB)
    ||(stopE ThenConstB constB(0.)) );

rotationB = switchB( constB(0.),
      ||(foundwallE ThenConstB followallB)
      ||(stopE ThenConstB constB(0.)) );
```

This code can be read as follows: start by moving straight ahead; anytime a wall is found, follow it; anytime the space bar is pressed or the wall is lost, stop.

[4]In practice, the value of a variable can only be defined once in an FRP/C++ program. For the purposes of exposition, we are allowing redefinition.

2.4 Tasks

While switching is the basic source of reactivity in the FRP framework, it is often too low-level for our purposes. One useful abstraction is the concept of a `Task`. A task combines a behavior and a terminating event. It can be constructed using `mkTask`. For example, here are three basic navigation tasks:

```
gotoT = mkTask(directionerrB,foundwallE);
followT = mkTask(followallB,lostwallE);
stopT = mkTask(constB(0.),neverE<void>());
```

Tasks can be sequenced using `>>`, or put in parallel using `||`. Sequenced tasks are executed in order: the second one is be evaluated when the first one ends (that is when its terminating event occurs). Optionally, the value of the terminating event is passed to the subsequent task. Parallel tasks are executed simultaneously: they begin at the same time, and the end of either one also terminates the other. It is possible to make the behavior of one task depend on the other. For details, we refer to [17].

One of the interesting properties of tasks is that they can be *transformed* by modifying their behavior or event terms [8]. For example, it is possible modify the execution of a task by adding extra terminating events using the `tillT` operator. With these additions, we can now construct a set of definitions for wall following equivalent to the previous switched version, but with a clearer sequence of execution:

```
actionT = ( ( gotoT >> followT >> stopT )
             TillT stopkeyE ) >> stopT ;
loopT = ( actionT TillT directionE ) >> loopT ;
```

The definition of `safeT` says that we go in the direction the user has indicated with the mouse until the wall is found. We follow the wall until it ends, then we stop. Hitting space (`stopkeyE`) also halts execution, no matter what is happening. The definition of `loopT` restarts execution every time a new direction is supplied.

In order to execute this, we would need to couple the rotation control to the robot. We could do this explicitly with the line `rotationB = loopT.behavior()`. A more elegant approach is to lift the entire execution architecture to tasks. In the interests of space, we refer the reader to [8].

3 Adding Vision

To this point, we have created a system of equations that implement manual navigation using the mouse. In this section, we use the concepts developed above to integrate vision, extend the user interface, and ultimately complete the system.

3.1 Visual Tracking

In [18], we created a prototype FRP design for the XVision tracking system [9]. In this design, a visual tracker can be viewed as a function from a behavior of image to a behavior of state. The tracker is composed of a stepper, a state predictor and an image warper.

A predictor is a function on a behavior of current state to a behavior of predicted future state. The simplest one is a null predictor:

```
template<class State>
Behavior<State>
null_predictor( Behavior<State> x )
  {  return x ; }
```

More complex predictors may have adjustable parameters, such as the linear predictor we mentioned in the previous section or the Kalman filter.

A stepper is a function from a predicted state and an image to an increment to the supplied state. The state value also contains a boolean that indicates whether the tracking was successful. A warper is a function on a behavior of image to a behavior of warped image which the stepper can consume. For simplicity, we will "absorb" the warper into the stepper and, assuming all steppers take the same form of initializer and produce identical states[5], we can write:

```
template<class Stepper>
Behavior<State>
tracker( Predictor predictor,Image initimage,
         Behavior<Image> imageB,Position point)
{
  Behavior<State> stateB, predstateB ;
  Stepper stepper(subimage(initimage,point));

  stateB = predstateB + stepper(imageB,predstateB);
  predstateB = delayB(point)(predictor(stateB));
  return stateB ;
}
```

Now assume our task is to track any object the user selects on the screen: a left mouse click indicates template tracking and a right mouse click indicates color-blob tracking. Both will use the null predictor. We can define our tracking behavior as

```
ssdE = splitE( display.lpb(),
               button1hit||button2hit );
blobE = display.rpb() ;

ssd_trackerF = liftF(&tracker<XVSSDStepper>)
 ( &null_predictor<State>, snapshotE(ssdE,sourceB),
   sourceB );
blobTrackerF = liftF(&tracker<XVBlobStepper>)
 ( &null_predictor<State>, snapshotE(blobE,sourceB),
   sourceB );

targetB = switchB( constB<State>(0),
   ( ssdE ThenB ssd_trackerF )
 ||( blobE ThenB blob_trackerF );
```

[5]The real stepper state types differ in XVision, however, it is easy to convert one form to another.

So each time the user clicks, a new tracker is generated. If we wanted to test this program, we would supply a video source and display the current tracking state as follows:

```
(display << drawstateB(targetB) <<= source).run();
```

where "<<" is an overloaded opeartor to draw graphics on the display.

For later use, we will create a task that does just this:

```
displayT = mkTask(display << drawstateB(targetB)
                          <<= source,neverE<void>());
```

3.2 Vision-Based Control

In order to integrate vision-based control, we will again assume that a variety of details have been accomplished for us through external libraries. We will also assume that we have a pan mechanism that must be controlled using visual information. We accomplish this as follows:

```
targanglecamB = pos2angB(targetB);
cntlsignalB = PController(PGAIN,targanglecamB);
pancntlB = liftB(pancntl) <<= cntlsignalB;
```

It is worth noting that we might want to now use the pan control signal as *feedforward* to the visual trackers. We could obviously incorporate this using the `predictor` argument of those functions.

For the purposes of later discussion, we will create a task for controlling the camera pan angle:

```
pantaskT = mkTask(pancntlB,neverE<void>());
```

Through all of this, we want to compute the robot-relative angle to a visual target. This is now just:

```
targangleB = panangleB + targanglecamB;
```

Here `panangleB` is a lifted function that returns the current pan angle.

While vision is operating, we also want to ensure that the robot does not collide with anything. To this end, we assume that we have another function that analyzes range data and returns a vector representing distance and range to the closest object. We reverse this vector and rescale it. We then create a simple obstacle avoidance mechanism that "blends" the result with a vector toward the target. Recall that the current state of one of our icons controls whether obstacle avoidance is in effect or not.

```
obsdirB = liftB(nearestobj)(rangedataB);
obsmagB = liftB(magnitude)(obsdirB);
avoidvecB = ifB(icon1stateB,
    -(1./(obsmagB*obsmagB))*obsdirB,
    0);
blendB = vec2angB(ang2vecB(targangleB)*normalspeedB
                  +avoidvecB);
// Now, redefine the robot control values
rotationB = PController(PGAIN,blendB);
```

To facilitate the subsequent discussion, let us put all of the definitions above into a function named `controlB` which accepts `targetB` and returns `rotationB`.

3.3 The Complete System

Finally, we want to integrate vision-based control into the existing wall-following control system. Obviously, this amounts to adding more control tasks to the previous definitions. However, a useful abstraction is to realize that what we really want to do is to *dispatch* a new control task with each defining event, and to follow that task with some sort of "safe" mode should it terminate before the next task takes over. This is really just the switching of tasks (similar with switching of behaviors as we discussed earlier). One minor problem is that not all task terminating events are necessarily the same. Since we don't care about the actual data associated with these terminating events, the problem is handled by simply casting all event types to `void`. Furthermore, we add the "safe" task to the end of each:

```
safeT = mkTask(
  liftB(setvelocity)(constB(0.),constB(0.)),
  neverE<void>());

template<class T,class E>
Task<T,void>
mksafeT(Task<T,E> t)
  { return castTE<T,void,E>(t) >> safeT; }
```

We then create a "driving" task

```
Task<int,void>
driveT( Behavior<State> vistargetB )
{
  lostE = whileE(liftB(&State::losttrack)
                 (vistargetB));
  driveB = liftB(setvelocity)
         (normalspeedB,controlB(vistargetB));
  return mksafeT(mkTask(driveB,lostE));
}
```

We can define a "goto" task analogously, based on slight modifications to the definitions of the previous section. We put it all together using `switchT`:

```
gotoE = display.mpb() ;

  dispatchE =
     (gotoE ThenConstT (gotoT >> followT))
   ||(ssdE thenT
      liftF(driveT)(ssd_trackerF)
   ||(blobE ThenT
      liftF(driveT)(blob_trackerF)
   ||((stopkeyE || logicbumperE) ThenConstT
      safeT);
```

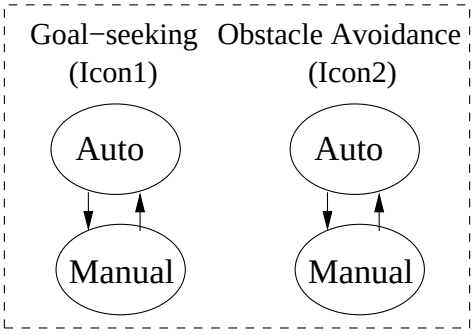

Figure 3: Control state diagram.

```
  startT = safeT ;
  actionT = switchT(startT,dispatchE);
  foreverT = actionT >> foreverT ;
  systemT = foreverT || displayT || pttaskT;
  systemT.run();
}
```

The final system performs tracking, control, display, and pan management. Its controller has four states as originally described, that are controlled through the on-screen icons.

4 Conclusions

The current implementation of FRP/C++ is being actively used within our laboratory as a way of expressing programs for visual tracking, robot control, and human-computer interfaces. In our experience, the main advantage of the FRP style of expression is the compactness and simplicity of the resulting code. It is straightforward to prototype system components, transform the code until it is correct, then to integrate separate components into a working application. The strongly typed nature of the specification minimizes coding errors while the lazy execution model maximizes computational efficiency. The embedding of FRP in an imperative language also means that it is possible, within a single program, to move between traditional imperative execution and FRP style execution as appropriate.

The principal disadvantage of the C++ embedding of FRP has to do with the somewhat "brittle" nature of the language constructs used to create it. Small syntactic or type mistakes in the code can generate complex compile-time errors which are difficult to locate due to the extensive use of templates and macros. Furthermore, using the FRP mode of expression in C++ requires a detailed and extensive knowledge of C++ itself.

There are a variety of obvious issues and extensions that we are considering. Two obvious extensions are to create a multi-threaded version of FRP/C++, and

to create a mechanisms whereby the system can operate in "push" mode (interrupt driven) rather than "pull" mode (polling). In the case of the former, the main issue is providing a way for different threads to communicate in a reasonable manner. Previous work in FRP on message passing suggests that this is a reasonable route to follow. Operating in a "push" mode would have the advantage of tying the system more closely to a fundamental clock. However, the danger is that the system could become saturated with input and no longer respond in a timely manner. Again, there are a variety of obvious solutions to this problem that could be easily added to the FRP/C++ model.

In summary, FRP/C++ appears to be a viable and reasonable approach to using functional reactive programming concepts without recourse to Haskell. Further work in developing larger systems will determine whether this approach scales effectively, and also how best to combine the FRP style of programming with more traditional programming structures.

Acknowledgments The authors wish to thank Izzet Pembeci for many useful discussions on FRP and its applications to robotics. We also gratefully acknowledge the support of the NSF under grant EIA-9996430 and the DARPA MARS program.

References

[1] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. Modular specification of hybrid systems in CHARON. In *HSCC*, pages 6–19, 2000.

[2] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, April 1986.

[3] X. Dai. FRP in C++. CIRL lab technical report, Dept. of Computer Science, Johns Hopkins University.

[4] H. M. E. Marchand, E. Rutten and F.Chaumette. Specifying and verifying active vision-based robotic systems with the signal environment. *The International Journal of Robotics Research*, 17(4), April 1998.

[5] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, volume 32(8), pages 263–273, June 1997.

[6] R. J. Firby. Task networks for controlling continuous processes. In *Proc. of the Second Int. Conf. on AI Planning Systems*, Chicago, IL, 1994.

[7] P. L. Guernic, T. Gauthier, M. L. Borgne, and C. L. Maire. Programming real time applications with signal. In *Proceedings of the IEEE*, volume 79, 1991.

[8] G. Hager and J. Peterson. A transformational approach to the design of robot software. In *Robotics Research: The Ninth International Symposium*, pages 257–264. Springer Verlag, 2000.

[9] G. D. Hager and K. Toyama. The "XVision" system: A general purpose substrate for real-time vision applications. *Computer Vision and Image Understanding*, 69(1):23–27, Jan. 1998.

[10] P. Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142, June 1998.

[11] F. Ingrand, M. Georgeff, and A. Rao. An architecture for real-time reasoning and system control. *IEEE Expert*, 7(6), 1992.

[12] K. Konolige. Colbert : A language for reactive control in sapphira. Technical report, SRI International, June 1997.

[13] K. Läufer. A framework for higher-order functions in C++. In *Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 103–116, June 1995.

[14] B. McNamara and Y. Smaragdakis. FC++: Functional programming in C++. In *Proceedings of International Conference on Functional Programming (ICFP)*, Montreal, Canada, September 2000.

[15] I. Pembeci and G. Hager. A comparative review of robot programming languages. CIRL lab technical report, Dept. of Computer Science, Johns Hopkins University.

[16] J. Peterson, G. Hager, and P. Hudak. A language for declarative robotic programming. In *Proc. IEEE Int. Conference Rob. Automat.*, pages 1144–1151, May 1999.

[17] J. Peterson, P. Hudak, and C. Elliott. Lambda in motion: Controlling robots with Haskell. In *Practical Aspects of Declarative Languages*, pages 91–105, 1999.

[18] A. Reid, J. Peterson, G. Hager, and P. Hudak. Prototyping real-time vision systems: An experiment in DSL design. In *International Conference on Software Engineering*, pages 484–493, 1999.

[19] V. M. Santos, J. P. Castro, and M. I. Ribeiro. A nested-loop architecture for mobile robot navigation. *The International Journal of Robotics Research*, 19(12), December 2000.

[20] R. Simmons and D. Apfelbaum. A task description language for robot control. In *Proceedings Conference on Intelligent Robotics and Systems*, 1998.