# FRP in C++ and Its Application

Xiangtian Dai, Gregory Hager
Department of Computer Science
Johns Hopkins University
Baltimore, MD, USA
{xentar,hager}@cs.jhu.edu

Henrik Nilsson
Department of Computer Science
Yale University
New Haven, CT, USA
nilsson@cs.jhu.edu

## ABSTRACT

*Functional Reactive Programming (FRP)* is a framework for constructing interactive applications in a declarative manner. FRP augments a host programming language to express *time flow* in a simple, semantically uniform manner. FRP has been demonstrated in domains such as animation, vision, robotics and other control systems. In this paper we present an implementation of the FRP programming style using C++ as a host language. There are two primary technical challenges: the implementation of *signals* in a manner faithful to the FRP semantic model and the use of the C++ type system, the template mechanism and operator overloading in particular, to achieve the same degree of type safety operator re-use that the Haskell type system provides. The resulting FRP implementation thus retains the essential "look and feel" of previous FRP implementations while also offering seamless interoperation with C++ code.

We demonstrate FRP/C++ in a complete human-guided robot navigation system that integrates vision, control, and human-machine interfaces. This system shows that the high-level semantic model expressed by FRP can be captured conveniently in C++ and provide the same declarative programming style found in the Haskell based implementations of FRP without compromising type safety.

## 1. INTRODUCTION

*Functional Reactive Programming* (FRP) is a programming framework for declarative programming of reactive, hybrid systems; i.e., systems that exhibit both continuous and discrete behavior [19]. Originally a *Domain Specific Language* (DSL) [9] for reactive animation [5], FRP has also been used for vision, robotics, and control-system applications [18, 16].

While one could imagine a version of FRP in the form of a complete, self-contained language (with its own syntax, type system, etc.), most present FRP implementations are *embedded* in a host language. This means that the FRP implementations only need to provide the key abstractions for declarative reactive programming, while the host language provides the syntax, type system, as well as all the usual programming facilities which are not specific to the reactive domain.

Up until now, the only host-language for full-scale FRP has been Haskell. In many ways this is an excellent choice, since Haskell provides higher-order functions, lazy evaluation, a powerful polymorphic type system, and so on, yielding a very flexible and expressive compound language for reactive programming. However, the fact that FRP only has been realized as Haskell embeddings does raise the question to what extent FRP *in practice* is separable from Haskell-like languages. This is an important issue, since a Haskell embedding arguably implies a steep learning curve for people with little or no functional programming background. More importantly, much software development for complex interactive systems has taken place in C++. For example, the XVision library for visual tracking defines hundreds of C++ objects and methods for the programmer. While it is possible to move this functionality to a language such as Haskell, it is much easier to capture the FRP functionality in C++. This eliminates a complex and error-prone language interface problem.

The aim of the present paper is to show that FRP usefully can be embedded in C++, a language radically different from Haskell. We call this embedding FRP/C++. The advantages of declarative, reactive programming are thus brought to the C++ community, along with much of the syntactic flavor of the Haskell embedding: FRP expressions can be transliterated into the FRP/C++ almost verbatim, both "functional" and "reactive" aspects are preserved. Another key aspect of FRP, a polymorphic type system, is also retained. Additional advantages of the C++ embedding include easy interoperation with imperative software components with heavily object-oriented APIs (especially when they are fundamentally template based, like XVision [7]), more controllable resource consumption patterns, and not having to link the resulting application with a Haskell interpreter or run-time system which potentially could be a problem, for example in a multi-threaded setting. Finally, FRP/C++ has a pure, standard C++ implementation, completely compatible with most existing C++ programming environments. The paper explains the implementation, which has a number of interesting features in its own right, in considerable detail

## 2. A BRIEF INTRODUCTION TO FRP

Functional Reactive Programming (FRP) is a programming framework for constructing interactive applications in a declarative manner. Since FRP is originally developed from FRAN (Functional Reactive Animation) and FROB (Functional Robotics) as an Embedded Domain Specific Language on Haskell, it is hard to distinguish the core of FRP from the features of Haskell language itself. Nevertheless, here we present our description based on the preliminary FRP User's Manual[6], and all our discussions in this section are in (simplified) Haskell syntax.

### 2.1 Behaviors

The key abstractions in FRP are *signals*. There are two types of signal: `Behavior` and `Event`. Conceptually, a `Behavior t` is a value of type `t` that varies over continuous time. The simplest behaviors are constant behaviors, which always has the same value over time, such as `1 :: Behavior Int`, or `red :: Behavior Color`. More complex and interesting examples including animations as `Behavior Picture`, or position vectors of a visual tracker as `Behavior (Int,Int)`.

Sometimes behaviors defined by pointwise application of an ordinary function to existing behaviors. This kind of operation is called a "lift", for example,

```
lift0 :: a -> Behavior a
lift1 :: (a -> b) -> (Behavior a -> Behavior b)
lift2 :: (a -> b -> c)
      -> (Behavior a -> Behavior b -> Behavior c)
```

Since functions that take no argument always return the same values, `constB`, which makes a constant behavior, is same as `lift0` (on lifting constants to behaviors). `lift1` returns a function which applies the function argument of `lift1` to all values of a behavior to create a new behavior; it works like `map` in functional languages.

Most arithmetic operators can be overloaded in Haskell so that same "lifted" form can be used conveniently, for example,

```
a, b, c :: Behavior Real
c = ( a + b ) / 2
```

Here "+" and "/" are both functions lifted to behaviors level, and "2" is used as a constant behavior.

Sometimes value of one behavior at current time relies on values of this and/or other behaviors at previous times. We can form a "delayed" behavior of existing one to allow this. `delayB` delays a behavior for one sampling step, given an initial value.

```
delayB :: a -> Behavior a -> Behavior a
```

The `delayB` is intended for direct use by the user - unrestricted use can lead to behaviors whose values do converge as sampling rates increase. However, this function is an essential part of the FRP implementation.

### 2.2 Events

Conceptually, an `Event t` is a time-ordered sequence of event occurrences, each carrying a value of type `t`. For example, a left button press `lbp :: Event ()` and a keyboard press `key :: Event Char`.

```
constE :: a -> Event a
neverE :: Event a
```

`constE` lifts a constant to an event that is always happening and `neverE` constructs an event that never happens. Lifted functions can also apply on events.

```
whileE :: Behavior Bool -> Event ()
whenE :: Behavior Bool -> Event ()
whileByE :: (a -> Maybe b) -> Behavior a-> Event b
```

Events can be derived from behaviors and vice versa. `whileE` turns boolean behaviors to events using the rule that the event happens if and only if the value of the behavior is true, and `whileByE` turns behaviors to events by specified functions. `whenE` is same as `whileE` except that it discards repeating events.

```
stepB  :: a -> Event a -> Behavior a
```

Given an initial value, `stepB` holds the value of the most recent event as the current value of the derived behavior:

```
snapshotE_ :: Event a -> Behavior b -> Event b
timeOfE    :: Event a -> Event Time
```

`snapshotE_` captures the value of a continuous behavior at the time of an event occurrence, and `timeOfE` captures the current time.

An FRP *program* is a set of mutually recursive behavior and event definitions. The ordering of these definitions should not change the behavior of the program in any way.

### 2.3 Switches

A rich set of operators is provided for users to compose new behaviors and events from existing ones. Some of the operators describe the way they react. Event mapping operators, `==>` and `-=>`, can be used to create an event of behaviors, and the `switchB` and `tillB` operators switch active behaviors according to such event of behaviors.

```
(==>) :: Event a -> (a -> b) -> Event b
(-=>) :: Event a -> b -> Event b
switchB :: Behavior a -> Event (Behavior a)
                      -> Behavior a
tillB   :: Behavior a -> Event (Behavior a)
                      -> Behavior a
```

`switchB` switches to the behavior carried by the event each time it occurs; `tillB` switches on the first occurrence once and for all.

A more complicated example follows:

```
color :: Behavior Color
color = red ‘tillB‘ (lbp -=> blue .|. rbp -=> green)
```

This reads as “initially behave as red, after the left button is pressed change to blue, or after the right button is pressed change to green”.

## 2.4 Tasks

Another way to express reactivity is using the concept of `Task`. There is actually a family of different types of tasks. Here we only focus on their common aspects. A task combines a behavior and a terminating event, and can be composed either sequentially or in parallel.

```
(>>)      :: Task a b -> Task a c -> Task a c
(|||)     :: Task a x -> Task b y
                      -> Task (a,b) (Either x y)
```

Tasks can be sequenced using `>>`, or put in parallel using `||`. Sequenced tasks are executed in order: the second one begins to be evaluated when the first one ends (that is its terminating event occurs). Parallel tasks are executed simultaneously: they begins at the same time, and the end of either of them also terminates the other.

```
mkTask :: Behavior b -> Event x -> Task b x
```

The `mkTask` function constructs a task from its behavior and terminating event:

```
tillT_   :: Task b x -> Event x -> Task b x
```

`tillT` terminates a task by an extra event. The occurrence of this event terminates the associated task if it is not already terminated.

## 2.5 The Streams Implementation of FRP in Haskell

Practically, an implementation of FRP has to sample continuous behaviors. In other words, behaviors are presented by infinite streams of values sampled at an infinite stream of times [4]:

```
type Behavior a = Stream Time -> Stream a
```

Events are defined similarly, as infinite streams of value of type `Maybe`, each indicates whether or not the event occurs (`Just x` or `Nothing`) and the value it is carrying if it does, sampled at an infinite stream of times:

```
data Maybe a = Nothing | Just a
type Event a = Stream Time -> Stream (Maybe a)
```

Then it is easy to define behavior and event operations on the base of stream and functional operations, such as `constB` is defined as a function for any input of stream of time, return the stream of repeating some same value. The complexity of manipulating infinite streams is hidden by lazy evaluation feature of Haskell, as the value of an element in a stream will not be computed until it is needed immediately.

## 3. A USER’S VIEW OF FRP IN C++

In this section, we turn to a description of the basic software structures and programming techniques we have used to implement FRP/C++. In particular, the dataflow view of FRP plays an important role in our understanding and implementation of FRP in C++. We begin with that view of the system, and then expand to explain how other elements of the system have been implemented.

## 3.1 Behavior and Event

There are two basic data types in FRP in C++: `Behavior<T>` and `Event<T>`, which mirror their Haskell counterparts. `Behavior<T>` is the type to declare behavior of type `T`, and `Event<T>` is to declare event of `T`. Constants and functions can be lifted to behavior level by `constB` and `constE` or overloaded `liftB` and `liftE` respectively. Common operators such as “+”, “-”, “*” and “/” already have their lifted version overloaded as well. Thus, for example, we can write

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

Behaviors and events can be used (in definitions of other signals) before their definition as long as their type have been declared. This allows recursive definition and other equation-like expression. Here is an example of Fibonacci series, which is defined as each succeeding term is the sum of the two immediately preceding. Here, FRP provides a definitional style nearly identical to the mathematical notation.

$$x_n = x_{n-1} + x_{n-2}$$

```
Behavior<int> x ;
x = delayB(0)( x ) + delayB(0)( delayB(1)( x ) );
```

## 3.2 Switches and Tasks

Since C++ does not allow definition of new operators, some of the infix combinators in FRP have to use slightly different notation, and some others have to use literal macros which translate to same functions. The same expression we have showed in the last section turns out to be:

```
Behavior<Color> color, red, blue, green ;
Event<void> lpb, rbp ;
color = red TillB ( lpb ThenConstB blue
                  || rpb ThenConstB green );
```

`Task<T,Y>` is the type of task composed by a behavior of type `T` and a terminating event of type `Y`. Same as behaviors and events, tasks can be defined recursively. Here is an example of task:

```
Task<Color,void> display ;
display = mkTask(blue, lbp) >> mkTask(green, rbp)
                            >> display;
```

The above expression reads as "display blue until left button is pressed, then display green until right button is pressed, then repeat".

## 3.3 Execution

To run a FRP program means to repeatedly sample a signal or a task by member function `run()`. `run()` can take several parameters, one of them is the time interval for sampling. In practice, however, computation itself is the main time concern so we just want to sample as fast as possible by simply call `run()` with default parameters like this:

```
display.run() ;
```

## 4. THE C++ IMPLEMENTATION

In this section, we turn to a description of the basic software structures and programming techniques we have used to implement FRP/C++. In particular, the dataflow view of FRP plays an important role in our understanding and implementation of FRP in C++. We begin with that view of the system, and then expand to explain how other elements of the system have been implemented.

### 4.1 Dataflow View of FRP

One way to look at the FRP model is to view it in terms of dataflow graphs. Each graph vertex, which represents a behavior or event, provides a typed time-varying value; each directed edge is a data dependency. As time advances, data flows along the directed edges and is processed by some functions at vertices. For example, at any specific time, a specific use of a lifted `(+)` operator is a vertex whose value is the sum of current values of the two vertices that are attached to the vertex's two incoming edges. In this sense, FRP shares common merits with graph or dataflow languages, except that it is non-visual.

There is one important difference in the FRP model. In most dataflow languages, the graph structure is statically defined. In FRP, Behaviors and Events, which represent the graph structure, are first class entities and can thus be carried around as data in a running dataflow graph. In fact, switching in FRP is accomplished by a graph node which replaces a subordinate graph, connected to ordinary data inputs and outputs, by a new graph whenever such a graph is received on a separate control input. Thus, in FRP, the dataflow graph can have a dynamic structure which can evolve, grow, and shrink over time in response to outside stimuli.

```
Graph G: (V,E)
Set S ;

synchronize(sink) :
  clear(S)
  update(sink)
  for all v in S do
    update(v)

update(v) :
  if v is delayB then
    let (u,v) in E
    if not updated(u) and not updating(u) then
      if update(u) == fail then
        add(S,u)
    compute(v)
    return success
  else
    for all u that (u,v) in E do
      if not updated(u) and not updating(u) then
        if update(u) == fail then
          return fail
    compute(v)
    return success
```

**Figure 1: The dataflow graph two-phase synchronization algorithm**

### 4.2 FRP/C++ Basic Engine

If we examine the dataflow view of FRP more closely, there are two immediate implications for the underlying computational engine:

1. Only one value – the current value of a behavior – needs to be held in each graph node.
2. Any recursively defined FRP signal must have at least one `delayB` or its equivalent somewhere in the definition.

Based on the first observation, we implement the core behavior architecture using a generic, templated class. This class stores one time-stamped[1] value type as well as links to all behaviors it depends on. In the terms of the dataflow graph, this defines the vertex and all its incoming edges.

At each time step, the graph is updated by a two-phase synchronization process (outlined in figure 4.2) that results in updated data at the graph sink (output). The first phase traverses the graph recursively from the sink. On visiting each vertex, any vertex which has a directed edge pointing to it (i.e. behaviors it depends on) is visited if it has not been visited before and it is not being visited currently. A node with no outgoing edges (a source) is updated and control returns to the calling parent. When all children return successfully, then the local computation of the vertex (i.e. the function of the behavior) is performed using the value of its children to update the value it holds. If one of the

[1]The time-stamp uses an internal, global clock-tick instead of the real time.

input vertices of a node is being visited concurrent with the node itself (indicating a loop in the graph), the update fails and no computation is done. No vertex is visited more than once, thus no computation is repeated.

Based on the second observation, we expect that any properly formed graph will have a delay operator in any loop. Thus, In the first phase we allow a "delay" vertex to update its value from a pre-fetched buffer before trying to update the vertex it depends on. If the vertex it depends on fails according to the previous rules, it will be added to a list. Everything in the list will be updated in the second phase of recursive synchronization. This time the update will always success because all "delay" vertices have already been updated in the first phase, and from our second observation above, the dependency link cannot loop back to itself without going through a "delay" vertex, which stops the updating propagation.

It would be interesting to compare the above algorithm with the lazy stream implementation in Haskell. A single, time-stamped storage location replaces the cons-cells of the stream, and synchronization is done in an explicit process instead of by letting all streams be consumed at the same rate.

## 4.3 Construction of Graph

Many behaviors and events are pre-defined operations defined from the generic templated classes. Examples include `timeB`, which represents the time elapsed from the start of the program in second, and `Display::lpb()`, which returns the left button press event on given display. Other signals are obtained from applying combinators to existing ones. Combinators are either templated functions (e.g. `integral`) or templated functional objects which can be generated by functions (e.g. `liftB`) to simulate higher-order functions in C++. In either case, the combinator creates a typed graph vertex using the supplied signal arguments as its incoming edges.

Recursive definition is common in FRP expression. For example, a counter is defined as

```
a = delayB(0)(a) + 1 ;
```

Again, note that an explicit delay is required to make this a valid graph.

Another issue in defining recursive behaviors is that the same behavior `a` appears on both side of the equation. We achieved this by employing the envelope-letter idiom: `Behavior` is really the envelope class to declare variables for behaviors. When it is assigned to, it is to contain the letter class created by the right hand side of the assignment. Whether the envelope contains a valid letter yet or not, it has a well-define interface that can be used in graph construction. In this way, as long as an object of `Behavior<T>` has been declared, it can be used before its definition (appearance on the left side of an equation). Of course, everything must be defined before the execution of that part of the graph.

Returning to the counter example. `delayB(0)` returns a behavior combinator, which takes the empty envelope `a` and generates a new behavior. "+" is another combinator which takes the behavior just generated and the const behavior `1` and generates the resultant behavior of the right hand side. This is assigned back to `a` through "=" as its letter. So the resulting graph is indeed circular.

As a result of such design, another type, `BehaviorRef<T>` is used for behaviors in FRP/C++ function signatures to avoid the overhead of constructing an envelope. This is mostly transparent to users as `Behavior<T>` and `BehaviorRef<T>` can be automatically converted to each other.

## 4.4 Events and Switching

It is natural to define `Event<T>` as `Behavior<Maybe<T> >`, except we have to define a efficient `Maybe` class in C++:

```
template<class T>
class Maybe {
  T * value ;
  char data[sizeof(T)] ;
 public:
  Maybe(/*Nothing*/) : value(0) {}
  Maybe(/*Just*/ const T& x ) :
   value(new((void*)data) T(x)) {}
  ~Maybe()
   { if(value) value->~T() ; }
};
```

where `value` is either `data` or `0`. The memory for `T` is reserved in `data` to avoid additional heap allocation. `data` is just the raw memory so as not to impose a default constructor requirement on the type `T`.

In the interest of efficiency, we specialize `Maybe<void>`:

```
template<>
class Maybe<void> {
  bool value ;
};
```

Now we can derive `Event<T>` from `Behavior<Maybe<T> >` and build switches upon it. The switching vertex itself is quite trivial: it stores both a reference to a behavior of `T`, which is its current behavior, and a reference of an event of behavior of `T`, which it observes. When the switching vertex being updated, it first tries to see whether the event had occurred. If it is, the new behavior carried in the event is taken out to replace the current behavior. Either case, the value of its current behavior is used as the value of itself.

## 4.5 Tasks

Since a task is composed of a behavior and an event, many task operations can be reduced to behavior and event operations. For example, putting two tasks in parallel will result in a new task whose behavior is a pairing of the behaviors of the two and whose event is a merging of the events of the two. Sequential tasks are a little bit more complex: both the resultant behavior and event are initially those of the first task, and switched to those of the second task upon the occurrence of the event of the first task. More sophisticated sequencing involves a task and a function that takes the terminating event of the first task to generate its successor. This can be implemented in the similar way as in behavior switching.

As with `Behavior`, `Task` is the envelope class for tasks to allow recursive task definition.

### 4.6 Garbage Collection

Although garbage collection is not a functional feature by itself, almost all functional languages have built-in garbage collectors. On the other hand, C++ programs are responsible to delete object that are no longer used. Due to the nature of the language, it is a daunting task for a general portable garbage collector for C++. Furthermore, not only used memory should be freed, the destructors of collected objects must be called to ensure that external devices have been closed, etc. "Smart pointers", pointer objects with reference counting, can help collecting unreferenced objects automatically, but they cannot deal with looped data structures , which is exactly what recursive FRP expressions translate to.

Fortunately, we needed only to implement a garbage collector for FRP expressions themselves, and leave the users free to choose memory management for their C++ style code. Our garbage collector is based on the same basic "mark-and-sweep" method as some existing generic garbage collector such as the Boehm-Demers-Weiser conservative garbage collector[2] However, since we don't have to scan through objects for possible pointers, it is more efficient then most generic collection schemes. The garbage collection algorithm can be briefly described like this: the "mark" step begins with "active" objects (objects in the stack or statically allocated) and marks all objects they directly or indirectly refer to; the "sweep" step frees everything else that is not marked. It traverses the graph, but since synchronization process traverses the graph anyway, so the cost is still proportional.

## 5. FUNCTIONAL SUPPORT

We found it important to allow functional expressions to fully take advantage of FRP. However, we have constructed this facility mainly to facilitate FRP expression, no to create a complete functional language within C++. Nevertheless, our functional library has its own merit and can be used independently.

### 5.1 Functoid

*Functoid* is a term for functional object in C++ [14]. Although address of a C++ function can be taken, passed around, and called to invoke the function, such native form cannot be used to write a higher-order function because no function can be created on the fly. On the other hand, a C++ object can be manipulated more freely while overloading of the `()` operator in C++ allows it to be used as if it is a real function. This is how a Functoid works. Some of often used manipulations, such argument binding (to create closures) and function composition, are a part of functoid's functionality.

Another difficulty on applying the functional paradigm on C++ is that there are too many forms of functions. Take the simplest example:

```
// direct form
int inc1( int x ) { return x+1 ; }
// direct form too, but different signature
int inc2( const int& x } { return x+1 ; }
// indirect form
void inc3( int& r, int x ) { r = x+1 ; }
// implicit form
void inc4( int& x ) { x++ ; }
// class member form
struct A {
  int inc5( int x ) { return x+1 ; }
};
```

The above list is far from complete, and the number of different signatures goes exponentially with number of arguments. Many FRP functions (e.g. Lift) require a functional argument so it is undesirable to overload every such function on every possible functional signature. So in FRP/C++, functoids are also used to unify many of those signatures. `liftF` is used to 'lift' a function to functoid. An example to demonstrate this is:

```
// function composition
Fun1<int,int> add_two = liftF(inc1)(liftF(inc2)) ;
int x = add_two(3) ; // x == 5
```

where `Fun1<A,B>` is the type of functoid which takes a argument of type `B` and gives a result of type `A`.

### 5.2 Lambda Expressions

Another limitation of C++ is that all (non-member) functions must be defined in global scope[2]. Although argument binding can be used to create equivalent of local functions, sometimes the code is is trivial and doesn't really warrant writing an out-of-context global function. In our functional library, `lambda` is to create a functoid out of an expression:

```
LambdaVar<int>::X x ;
Fun<int,int> inc6 = lambda(x,x+1) ;
```

We call this a "typed" lambda expression because the type of the lambda variable is fixed so that the result functoid has a fixed (non-polymorphic) signature. Although it is also possible to implement "un-typed" lambda expressions in C++[11], we have opted not to do so because we see it not worth the effect to emulate the type inference mechanism in C++.

## 6. ROBOTIC CONTROL SYSTEM

Here an interactive vision guided robotic control system is presented as an example of FRP/C++ applications. The

[2]Technically, non-member functions can be only defined in namespace scope, including the global namespace scope. However, this discrimination is irrelevant to our discussion here.

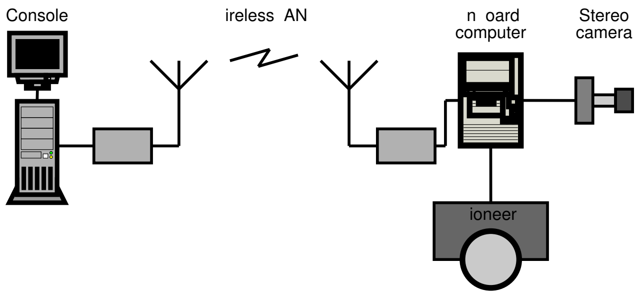

**Figure 2: Overview of the hardware setup.**

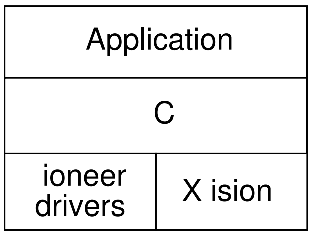

**Figure 3: Software architecture.**

system consists of an ActivMedia Robotics Pioneer 2 robot, a stereo camera, and an on-board computer which hosts our software system. Figure 6 shows the hardware organization.

The Pioneer 2 is a two-wheel drive mobile robot. It communicates with the on-board computer over serial link. In the commmand-driven mode we use, the robot simply accepts movement commands from the computer and provides feedback about its status and position information through dead-reckoning. The stereo camera communicates with the on-board computer over IEEE 1394 firewire interface. A stereo camera provides depth information and color images. In our system, the depth information is used both in depth disparity tracking and obstacle detection.

The on-board computer is responsible for high-level operation of the robot as well as vision processing. It communicates with outside world over a wireless LAN, which allows the robot to move about without any attached cables. The robot usually operates semi-autonomously under the supervision of an operator at a console. The on-board computer feeds live video to the console, where the operator can ask the robot to carry out certain tasks, such as to track and follow an object while avoiding obstacles, or moving to some particular direction. Of course, if necessary, the operator could take direct control and tele-operate the robot.

Figure 6 shows the software architecture of the on-board system. At the lowest level, there are routines for communicating with the microcontroller. XVision2 [7] handles the camera interfacing and all computationally demanding vision processing. It is also responsible for the console. Interfaces to these two subsystems are lifted into FRP/C++ and the entire application itself is written in FRP/C++. Since everything from top to bottom are is C++ or C, interoperation is seamless.

## 6.1 Visual Tracking

A visual tracker can be described as the loop of a stepper and a predictor. The predictor guesses the current state from the previous state, and the stepper searches for the current state in an image directed by the guess. In FRP/C++, we can write:

```
Behavior<TargetState> target ;
Fun1<BehaviorRef<TargetState>,
     BehaviorRef<TargetState> > predictor ;

// other assignments here
target = stepper( predictor( delayB(initGuess)
                                 (target) ),
                  video );
```

where `TargetState` is the type for our tracker's state, in our case, a rectanglar blob contains the target in the image.

The location of the target is predicted by combining a simple linear predictor with compensation for rotation of the robot base:

```
BehaviorRef<TargetState>
linearPredictor( BehaviorRef<Orientation> r,
                 BehaviorRef<TargetState> x ) {
  return x + K1*(x-delay1B(x))
           - K2*(r-delay1B(r)) ;
}
```

The `predictor` appears in the tracker loop using the following partially bound definition:

```
predictor = liftF(linearPredictor)(orientation) ;
```

All code above (except the `linearPredictor` function) is then packaged into a (C++) function called `trackTarget`:

```
BehaviorRef<TargetState>
trackTarget( Stepper stepper, BehaviorRef<Image> video,
             BehaviorRef<Orientation> orientation,
             Position button ) {
  TargetState initGuess = segment( video, button );
  // ...
  return target ;
}
```

The initial guess of the target state is obtained by segmenting the image region around the given button position.

## 6.2 Driving the Robot

The robot is driven by providing a behavior of vector of velocities of its two wheels. Here we focus on the primary task of the robot: to follow the tracked object while avoiding obstacles. The former can be done directly in a C++ function using current tracker state to compute requested velocity vector. The latter needs two steps: first a set of "obstacle lines", the range map of visible obstacles, is computed from stereo images, then an algorithm is executed over the obstacle lines to find the nearest "gap" which fits in the robot. The robot is steered toward that gap. These two velocity vectors are blended by the following formula:

$$V_r = V_f \cdot \min(1, k * (d - d_s)) + V_o$$

where $V_r$ is the result velocity vector, $V_f$ is the velocity vector returned by the target follower and $V_o$ is the one returned by the obstacle avoider; $d$ is the distance between the robot and the nearest obstacle in its direct front, which can also computed from the obstacle lines, and $d_s$ is a safety distance.

Here is the code that implements the above:

```
BehaviorRef<RobotVel>
follower( BehaviorRef<Image> video,
          BehaviorRef<TargetState> target )
{
  Behavior<RobotVel> Vr, Vf, Vo ;
  Behavior<ObstacleLines> obstacles ;
  Behavior<float> d ;

  obstacles = getObstacles(video) ;
  Vf = follow( target );
  Vo = avoid( obstacles );
  d = frontDistance( obstacles );
  Vr = Vf * (smaller(1.0f,d-dSafe)) + Vo ;
  return Vr ;
}
```

## 6.3 Interactive Control

Now we have the basic functions; what remains is to put them together to form an interactive system. The left mouse button is used to select an object on the screen of the console to let the robot follow it while avoiding obstacles; the right mouse button is to manually set a direction for the robot to turn to and move along; the space key stops the robot immediately.

```
int main()
{
  // declarations omitted for the interest of brevity

  disp = display( video ) ;
  robot = drive( vel ) ;
  vel = switchB( stop, events );
  events = followE || moveE || stopE ;
  followE = display.lpb() ThenB
            liftF(follower)(video)
             (liftF(track)(stepper,video,
                           orientation));
  moveE = display.rbp() ThenB move ;
  stopE = filterE(lambda(c,c==' '))(display.key())
          ThenConstB stop ;
  orientation = getOrientation(robot) ;

  (disp,robot).run() ;
}
```

Everything is quite straightforward except the expression which defines `followE`. It works like this: the function `follower` we defined before has its first argument bound, and `track` has its all arguments other than the last bound, then they are composed to a single function, which takes a `Position` and gives a `BehaviorRef<RobotVel>`. Whenever `display.lpb()` occurs, the position of mouse button is used to call that composed function, and switch to the new behavior it generated.

## 7. RELATED WORK

Implementations of Functional Reactive Programming (FRP) on Haskell have been pioneered by Elliott[4] and Hudak[10]. Our implementation in C++ share some of working mechanism with their work. Some FRP applications, such as FVision [17] and [15], combine Haskell code of FRP and C++ code of libraries through special wrapping.

Recently, Courtney[3] implemented (a sub-set of) FRP in Java, focused on the relationship between the FRP event-behavior model and the Java Beans event-property model. However, the lack of a strictly typed polymorphic system in Java becomes a limitation. Furthermore, that implementation of behaviors heavily depends on an event propagation ("push" mode), somehow different from existing Haskell ones. The latter mainly make use of the lazy evaluation feature ("pull" mode). This makes significant differences when recursive or mutually recursive behaviors being defined.

More Recently, there are some interesting development on Real Time FRP (RT-FRP)[20] and Event-Driven FRP (E-FRP)[21]. A subset of FRP can be identified to have bounded resource consumption, and programs of a variant of it can be compiled into efficient C code.

On the other hand, the idea of dataflow-based (graph) programming has existed for a long time. Lucid[1], ECOS[8] and Signal[12] are such languages. In our work, we explore the connection between dataflow and functional programming and build our implementation upon this connection.

C++ Programming in functional style, including higher-order functions, is already well-known to the community[13] [14] [11]. [14] has even demonstrated that it is possible provide a functional library for C++ to implement Haskell's `prelude`. However, our implementation is not directly based on existing works: It is neither our purpose to re-invent a functional programming language over C++, nor to insert an additional functional layer between FRP and hosting C++ environment.

## 8. CONCLUSION

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 errors in the code can be difficult to trace 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.

## 9. ACKNOWLEDGEMENTS

## 10. REFERENCES

[1] Edward Ashcroft and William Wadge. *Lucid, the Data-Flow Programming Language.* Acadamic Press, 1985.

[2] Hans-J Boehm and Mark Weiser. Garbage collection in an uncooperative environment. In *Software Practice and Experience*, pages 802–820, September 1988.

[3] Antony Courtney. Functional reactive programming in java. In *Proceedings of Third International Symposium on Practical Applications of Declarative Languages (PADL '01)*, March 2001.

[4] Conal Elliott. Functional implementations of continuous modeled animation. *Lecture Notes in Computer Science*, 1490:284–??, 1998.

[5] Conal Elliott and Paul 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] The Yale Haskell Group. The Yale FRP user's manual. **http://haskell.cs.yale.edu/frp/manual.html**.

[7] Gregory Hager and Kentaro Toyama. The XVision system: A general-purpose substrate for portable real-time vision applications. *Computer Vision and Image Understanding*, 1998.

[8] J. C. Huang, Jos Muoz, Hal Watt, and George Zvara. ECOS graphs: a dataflow programming language. In *Proceedings of the 1992 ACM/SIGAPP symposium on Applied computing*, pages 911–918, March 1992.

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

[10] Paul Hudak. *The Haskell School of Expression - Learning Functional Programming through Multimedia.* Combridge University Press, Cambridge, UK, 2000.

[11] Forschungszentrum Juelich. FACT! – functional additions to C++ through templates and classes. **http://www.kfa-juelich.de/zam/FACT/**.

[12] Richard Kieburtz. Real-time reactive programming for embedded controllers. In *ACM International Conference on Functional Programming (submitted)*', Sept. 2001.

[13] Konstantin 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] Brain McNamara and Yannis 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] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. In *Practical Aspects of Declarative Languages*, pages 91–105, 1999.

[17] John Peterson, Paul Hudak, Alastair Reid, and Gregory Hager. FVision: A declarative language for visual tracking. In *Proceedings of Third International Symposium on Practical Applications of Declarative Languages (PADL'01)*, March 2001.

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

[19] Zhanyong Wan and Paul Hudak. Functional Reactive Programming from first principles. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–252, Vancouver, British Columbia, Canada, June 2000.

[20] Zhanyong Wan, Walid Taha, and Paul Hudak. Real-Time FRP. In *Proceedings of Six ACM SIGPLAN International Conference on Function Programming*, Florence, Italy, September 2001.

[21] Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven FRP. In *Proceedings of Third International Symposium on Practical Applications of Declarative Languages (PADL '02)*, March 2002.