# FrTime: Functional Reactive Programming in PLT Scheme

Gregory Cooper and Shriram Krishnamurthi

Department of Computer Science
Brown University
Providence, Rhode Island 02912

**CS-03-20**
April 2004

# FrTime: Functional Reactive Programming in PLT Scheme

Gregory Cooper and Shriram Krishnamurthi

Department of Computer Science
Brown University
P.O. Box 1910
Providence, RI 02912, USA
Fax: (401) 863-7657
{greg, sk}@cs.brown.edu
WWW home pages: http://www.cs.brown.edu/~{greg, sk}/

**Abstract.** Functional Reactive Programming (FRP) supports the declarative construction of reactive systems through *signals*, or time-varying values. In this paper, we present a new language called FrTime, which provides FRP-style signals atop a dialect of Scheme. We introduce the language with a few examples and discuss its implementation. FrTime uses impure features, such as state and asynchronous communication, to model time and to control evaluation. The use of such features yields a scalable, event-driven implementation with several important advantages. Specifically, it eases integration with other systems, supports distribution of signals across a network, and permits various benign impurities. To illustrate the language's expressive power, we present a concise implementation of a networked paddle-ball game in FrTime.

## 1 Introduction

Reactive software systems present several engineering challenges not found in conventional batch systems. In particular:

- They are naturally concurrent. For example, they need to respond to events that arrive asynchronously from multiple sources. Often, they also need to perform continuous processing, such as updating a display of changing data. Coordinating such concurrent computations is difficult and can involve complex synchronization patterns.
- Their state changes over time, and there are typically dependencies between data. Hence, when one thing changes, the programmer must update everything that depends on it. Done by hand, this is a tedious and error-prone task.
- Their control structure is inverted: instead of the application controlling itself, interaction with external entities determines what computations happen and when. For this reason, reactive systems are often structured around *callbacks*, application-supplied routines that perform imperative operations in response to events. Coordination within the resulting "callback soup" can be difficult, since numerous isolated code fragments end up manipulating the same data.

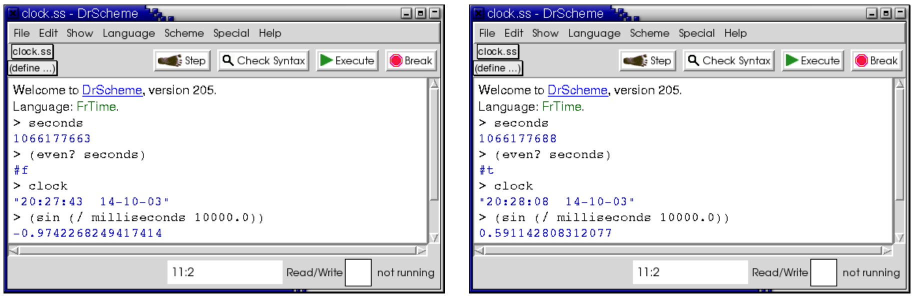


**Fig. 1.** Interacting with FrTime

Research on Functional Reactive Programming (FRP) [10, 20, 24] focuses on developing linguistic support for constructing reactive systems. In particular, it encourages the declarative specification of such systems by providing *signals*, or time-varying values. Signals come in two varieties, *behaviors* and *events*, which assist respectively with the handling of state and control. Specifically, behaviors offer a safe mechanism for managing state by automating the task of maintaining consistency. Events address control-structure inversion by allowing programmers to treat event sources as first-class values. Programmers construct FRP systems by connecting networks of signal processors, and the system computes values of all signals in parallel.

There have been a number of Haskell-based implementations of FRP, beginning with Elliott and Hudak's system Fran [10]. In this paper, we present a new, Scheme-based implementation of FRP called FatherTime (FrTime). In particular, it implements state directly through mutation, employs eager evaluation, and uses an asynchronous model of concurrency. After introducing FrTime and discussing its implementation, we describe several of the benefits that derive from this implementation strategy. In particular, we explain how it naturally supports the distribution of signals over asynchronous communication channels, eases integration with external systems, and permits various benign impurities. In addition, its event-driven computation model makes more efficient use of resources and provides better scalability for many systems. We take advantage of these features to integrate FrTime with the DrScheme programming environment and an interactive graphics library. To illustrate its expressive power, we present a concise implementation of a networked, multi-player paddle-ball game.

## 2 A Programmer's View of Father Time

Father Time (FrTime) is a new programming language that runs in the DrScheme [12] programming environment. It extends a purely functional subset of PLT Scheme (the language DrScheme implements) with *signals*, or time-varying values.

In order to define time-varying values, we need a way of referring to time itself. FrTime provides coarse- and fine-grained notions of time, which are called respectively *seconds* and *milliseconds*. Figure 1 shows screenshots of an interactive session with FrTime in which we evaluate several expressions, including *seconds*.

```
(define clock                                  ;; pads a number to two digits
  (match (seconds→date seconds)                (define (pad n)
    [($ date sec min hr day mon yr _ _ _ _)      (if (< n 10)
     (format "~a:~a:~a ~a-~a-~a"                    (format "0~a" n)
             hr (pad min) (pad sec)                 (number→string n)))
             day (pad mon) (pad (modulo yr 100)))]))
```

**Fig. 2.** A clock program in FrTime

*Seconds* is a signal, since it changes over time (every second, to be precise). More specifically, it is a signal that is defined at every moment in time, so we say it is *continuous* and call it a *behavior*. DrScheme displays behaviors just like ordinary Scheme values, except that the display automatically updates whenever a behavior changes. In Figure 1, we provide two screenshots of the same session, taken twenty-five seconds apart, to illustrate the dynamic display of behaviors.

We can imagine applying a function to a behavior at every moment in time. This is called *lifting* the function application, and the result is also a behavior. In FrTime, all applications of primitive functions are automatically lifted whenever necessary. For example, the second expression we evaluate in Figure 1 is (*even? seconds*). The value of this expression indicates, at every moment in time, whether the current value of *seconds* is even.

In general, the automatic lifting of function applications allows the programmer to treat behaviors like ordinary values. The programmer can also mix constants and behaviors within a single function application. For example, the following are all legal FrTime expressions:

```
(+ 2 3)
(modulo seconds 10)
(+ (* seconds 1000) (modulo milliseconds 1000))
```

It is possible to write interesting programs in FrTime using nothing more than *seconds* and bit of ordinary Scheme. For example, Figure 2 shows a brief implementation of a clock program, which presents a human-readable representation of the current time. Figure 1 shows the result of evaluating this program.

The DrScheme programming environment gives us the ability to interact with time-dependent values. To achieve a richer interactive vocabulary, we have implemented a signal-based interface to a graphics library, yeilding an interactive animation system in the spirit of Fran [10].

Our animation system creates a drawing window and provides a simple interface for interacting with it. The behavior *mouse-pos*, for instance, represents the current mouse position within the window. The library's primary interface procedure is *display-shapes*, which consumes a time-varying list of shape structures and displays them in the window. For example, the following simple program makes a blue ball of radius 20 that follows the mouse:

```
(display-shapes
  (list
```

```
    (make-ball mouse-pos 20 "blue")))
```

Modeling the mouse position as a behavior seems like a natural decision. However, we can also use signals to model other kinds of user input, such as mouse clicks and keystrokes. These signals are not continuous but instead consist of sequences of discrete occurrences. Following the FRP terminology, we call such signals *events*. instance, the graphics library provides an event called *keystrokes*. Unlike behaviors, events are not continuously valued, so evaluating an event does not produce any useful output:

```
> keystrokes
#<event>
```

To observe event-driven behaviors, we have a collection of combinators that create behaviors from events. For example, the function *hold* creates a behavior by "holding" onto the most recent event occurrence (it requires an initial value to use until the event occurs for the first time). If we evaluate (*hold* #\nul *keystrokes*), we get a behavior that corresponds to the most recent keystroke.

We can write more elaborate event processors, such as *count-occ*, defined below, which counts the occurrences of an event:

```
> (define (count-occ ev)
    (accum/e→b (map-e (λ (_) add1) ev) 0))
```

*Map-e* is analogous to the standard list-processing function *map*; that is, the expression (*map-e fn input*) creates a new event by applying *fn* to each occurrence of the input. Likewise, the expression (*accum/e→b trans init*) creates a behavior whose value starts as *init* but changes by cumulatively applying transformers carried by the event. For instance, (*count-occ keystrokes*) is a behavior that counts keystrokes.

## 3 A Peek Behind the Scenes

We have seen how a programmer can interact with FrTime. To give a deeper picture of what goes on behing these interactions, we briefly discuss the implementation now. In order to clarify the novel aspects of FrTime, we first give an overview of previous FRP implementation strategies.

Conal Elliott [9] discusses several functional approaches to the implementation of behaviors. A simple approach is to use functions from time to values, but Elliott quickly rejects this on grounds of inefficiency. Since the value of a behavior often depends on cumulative effects of time, he offers an approach based on "residual behaviors". In this approach, a behavior consumes the current time and yields a value plus a new behavior which can typically compute future values more efficiently. A third approach, which is also discussed in the book by Hudak [18], model behaviors with stream processors—functions from infinite lists of time steps to infinite lists of values. The stream approach seems to have been the dominant approach for some time, although a recent system called AFRP [20] uses residual behaviors.

All of these implementations share two important features. First, they are purely functional, so they model time and state explicitly. Second, they are *synchronous*; that is, they simulate the passage of time by evaluating snapshots of the entire system at discrete moments in time.

From a theoretical standpoint, these features are beneficial. For one thing, it is often easier to reason about properties of a purely functional program than about an imperative one. Also, because they model time explicitly, the functional implementations can use analytical methods for event detection, and they can support general time-transformations on behaviors. By employing a synchronous evaluation strategy, they ensure that the values of all signals are consistent at each time step. Wan and Hudak [24] use this property to help formalize a semantics for a stream-based FRP implementation.

From a software engineering perspective, however, purity and synchrony are less useful. For example, a pure implementation hides the state of the system in the internals of the host language's evaluator, where it is not easily accessible. In Haskell specifically, lazy evaluation makes expressing the stream-based implementation easy and natural, but it complicates the interface to impure features, where evaluation order is crucial. External entities are typically autonomous, so the timing of their actions is unlikely to conform to an FRP system's clock. This is especially true in distributed systems, where communication crosses machine boundaries. To simulate synchronization with an outside system, a synchronous system uses polling, which sacrifices efficiency and temporal accuracy.

In FrTime, we follow a very different implementation strategy, and we obtain a different set of tradeoffs. In particular, we make no attempt to have a purely functional or a synchronous implementation. Instead, we model time and state directly with time and state, and we drive evaluation with asynchronous communication.

For each signal in a FrTime program, we construct a *signal* data structure. The data structure contains an update procedure, which computes the signal's value, and a mutable field that stores the signal's most recently computed value. Thus, there is an implicit notion of *real* time, which the system exhibits directly.

A special thread called the *signal manager* is responsible for keeping signals current. It maintains an explicit graph of the dependencies between signals. For example, when we evaluate (*even? seconds*), the resulting behavior depends on *seconds*, so the signal manager needs to recompute it whenever *seconds* changes.

The manager uses an asynchronous message queue to control all computation. For example, when *seconds* changes, the manager sends itself an *update* message for each dependent signal. When it dequeues such a message, it recomputes the corresponding signal. This in turn may demand recomputation of yet more signals, so evaluation proceeds recursively in a breadth-first, bottom-up manner.

Some signals require re-evaluation after intervals of time. For example, *seconds* needs to update once every second. The signal manager supports this capability by keeping a prioritized "alarm" queue, which maps signals to update times. On each iteration of its processing loop, the manager checks whether the current time exceeds the earliest alarm in the queue. If so, it executes the corresponding update; otherwise, it sleeps until either the next alarm or the arrival of a message. Our asynchronous message-passing library makes this easy by providing a **receive** construct with a fine (millisecond-granularity) timeout parameter.

Our evaluation algorithm employs two main optimizations. First, if recomputing a signal does not change its value, then the manager does not schedule updates for the dependents. Second, to avoid scheduling the same signal for multiple updates, we give each signal a flag to indicate whether it has already been scheduled. The manager sets this flag before sending an *update* message and clears it after updating the signal. (If the flag is already set, the manager does not send the message.)

#### A Note on Asynchrony

Since FrTime employs an asynchronous evaluation strategy, there are brief intervals during which behaviors are inconsistent with each other. This could be a problem if it prevented formal reasoning about programs written in FrTime. In fact, however, there are many tools, including model-checkers like SPIN [17], that are specifically designed to verify properties of asynchronous systems. We therefore intend to pursue the use of such tools for FrTime programs. In particular, we believe that the temporal nature of FrTime will yield programs better suited for analysis by tools such as model-checkers. In addition, work on reasoning about implicit invocation systems [13] is likely to offer more useful techniques for automated reasoning.

## 4 Benefits of FrTime's Implementation Strategy

In this section, we explain several of the benefits that derive from FrTime's implementation strategy.

### 4.1 Distributed Signals

Events in FrTime map naturally to messages, and therefore they easily generalize to a distributed setting. To take advantage of this property, we provide a simple interface for distributing events. The expression (*bind event name*) binds *name* to *event* in a local registry, while (*remote-event machine name*) creates a local proxy for the event bound to *name* on the remote *machine*. Whenever the remote event occurs, the local proxy emits an identical occurrence.

In Section 5, we discuss an example in which we use distributed events to implement distributed behaviors.

### 4.2 Integration with Other Systems

In Section 3, we explained how FrTime's signal manager performs computation in response to asynchronous messages. In most cases, these messages come from the signal manager itself, indicating internal changes to parts of the system. However, the existence of a general-purpose message queue also offers an easy mechanism for integration with other systems.

For example, to implement the animation library presented in Section 2, we built a signal-based interface to DrScheme's Viewport Graphics Library. Part of this task involved translating the library's keyboard and mouse events into FrTime events. Since

```
(define keystrokes                    (define (key-callback key-event)
  (make-event-receiver))                (! signal-manager
                                          (make-event
                                            (get-key-code key-event)
                                            keystrokes)))
```

**Fig. 3.** An external event source for key strokes

```
(define (draw-shapes list-of-shapes)
  (begin
    ((clear-viewport offscreen-port))
    (for-each
      (λ (shape)
        ((draw-shape offscreen-port) shape))
      list-of-shapes)
    ((copy-viewport offscreen-port) canvas)))

(define (display-shapes list-of-shapes)
  (begin
    (draw-shapes (cur-val list-of-shapes))
    (map-e (changes list-of-shapes) draw-shapes)))
```

**Fig. 4.** Animation routines

the library handles these events in a separate thread, the conversion is not necessarily easy. In a synchronous system, for instance, we would need to poll the library at each time step. In contrast, message queues allow us to build a general mechanism for adding external event sources. For example, the graphics library uses the code in Figure 3 to define an event for keystrokes. A callback[1] in the library (*key-callback*) sends the signal manager a message each time the user presses a key ("*!*" is the message-sending primitive). The manager emits the value as an occurrence of *keystrokes*. *Keystrokes* behaves like any other event, and FrTime is not even aware that it is interacting with a graphics library.

We exploit FrTime's support for imperative commands to implement an animation system with this library. The main interface routine is called *display-shapes*; it takes a time-varying list of shapes and animates them on a canvas. We show an implementation for *display-shapes* in Figure 4. The procedure *draw-shapes* takes a (constant) list of shapes and draws them to a canvas, using an offscreen buffer to prevent flicker.

*Display-shapes* takes a time-varying list of shapes, draws an initial snapshot to the canvas, and then redraws whenever a change occurs. The expression (*changes list-of-shapes*) creates an event that delivers a new snapshot of the shape list each time it changes. We use *map-e* to apply *draw-shapes* to the resulting sequence.

[1] The use of a callback here is mandated by the graphics library's interface, not by FrTime. FrTime itself does not use callbacks.

User interface toolkits are another natural target for interaction with FrTime, since many types of widgets have properties that naturally map to signals. For example, we can use a numeric behavior to indicate the current position of the thumb in a slider. In this case, the behavior serves as an input signal. We can also use behaviors for output, for example by creating a label whose contents reflect the value of a string behavior. Still other widgets, such as buttons, associate more sensibly with events than with behaviors.

Integrating FrTime with an imperative user interface toolkit involves essentially the same techniques as described above for the graphics library. To receive input from a widget, we use *make-event-receiver* and send messages to the signal manager. To provide output through a widget, we use *map-e* to update the widget's properties each time a behavior changes. This is also how we implement the dynamic display of behaviors in DrScheme's Interactions Window—the textual representation of the value is essentially a widget.

FrTime's use of state also permits a more meaningful notion of interaction. In particular, since behaviors always exhibit their current values, we can easily implement *cur-val*, which permits an external entity (such as a programming environment) to query the current state of a behavior. This capability is valuable for learning about and interactively developing signal-based systems. In the pure Haskell-based FRP systems, state is implemented indirectly, so there is no way to read a signal's value from outside.

### 4.3 Resource Utilization and Scalability

A fundamental goal for our work has been to develop a completely event-driven implementation, and one motivation for this goal is more efficient use of resources. Specifically, in a synchronous system, computation proceeds in a top-down fashion, recursively evaluating each subexpression at each time step. This means that the amount of computation performed at each step is proportional to the total number of signals in the system. Therefore, the total "processing power" required is proportional to the product of the system size and the sample rate. This processing requirement is independent of the frequency with which signals actually change.

FrTime only performs computation in response to events, so the amount of computation is proportional to the rate at which signals require recomputation. This property is advantageous for large systems, especially if the majority of signals change rarely, such as in response to human interaction. In this case, the size of the system itself does not strain processing resources. Because FrTime's evaluation strategy is asynchronous, it also has an advantage when small parts of the system need to update rapidly, since there is no central clock forcing recomputation of everything at the same rate.

In Figure 5, we show performance comparison plots for FrTime versus a synchronous stream-based implementation of behaviors. The stream-based implementation is a direct translation of Elliott's [9] into PLT Scheme. We have made measurements to verify that Haskell systems (Hugs, GHC) exhibit similar performance trends (within constant factors), but for consistency we run everything in exactly the same environment (virtual and physical machines).

The graph on the left shows processing requirements for systems consisting of 2000 signals and running at various update rates. The stream-based system recomputes every signal, in lockstep, at the reported update frequency. The actual rate at which signals

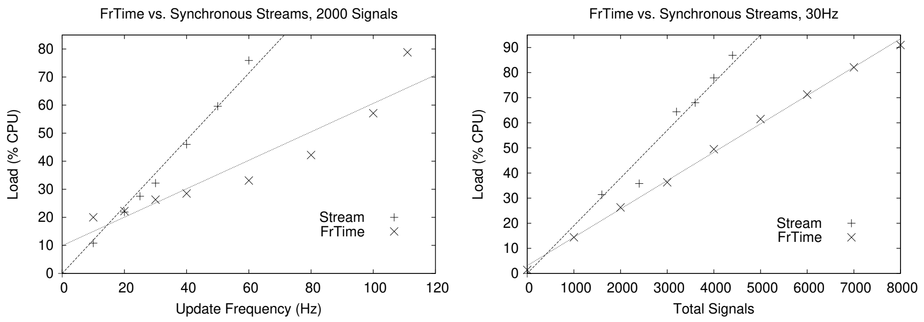


**Fig. 5.** Performance Graphs

change has no effect on the processing requirements, and the system is not sensitive to any change that occurs more quickly than the sample rate. In FrTime, we have 1900 signals that update at a slow rate (about 1Hz), and 100 signals that update at the reported variable rate. Note that, since there is no central clock, all of the signals update out of phase with each other on a conceptually continuous time scale. As a result, even though no single signal updates at more than about 100Hz, when we view the system as a whole, we can perceive updates occurring at an extremely high frequency (approximately 2KHz).

The graph on the right shows processing requirements for systems of various sizes in which no single signal updates faster than 30Hz. Again, in the stream-based system, all signals update in lockstep at the specified sample rate. In FrTime, we make 100 signals update at 30Hz, while the rest update at 1Hz. As before, the FrTime systems exhibit changes continuously, since there is no synchronization between signals.

The graphs show that using FrTime is advantageous for large systems in which signals update at variable rates. Specifically, FrTime allocates processing to the subsystems requiring the most frequent updates, which allows it to support more signals, and signals with more frequent updates, with a lower overall demand on system resources. In addition, since we have not yet optimized our current implementation, we expect that we can further improve performance in the future.

### 4.4 Support for Benign Impurities

Because FrTime is impure, it can take advantage of imperative features to improve interaction. As we saw earlier in this section, we can use imperative drawing commands within procedures like *map-e* to implement animations. FrTime also naturally supports input and output, which can be useful for understanding and debugging programs. For example, to observe events, a programmer can insert functions like *printf* in event-processing combinators.

FrTime also provides mutable references, which are created with *new-cell* and mutated with *set-cell!*. Cells give the programmer fine control over which values can be mutated imperatively, like references in ML. However, cells in FrTime are behaviors, so we can define other "pure" behaviors in terms of them, and FrTime automatically maintains consistency between them. For example, we have found that cells are useful for interactive experimentation in the DrScheme read-eval-print loop. While constructing an animation, we can use cells to help tune parameters, such as the size and color of various objects:

```
> (define ball-radius
    (new-cell 10))
> (define ball-color
    (new-cell "green"))
> (display-shapes
    (list (make-ball (make-posn 50 50) ball-radius ball-color)))
```

graphics window now shows a green ball of radius 10

```
> (set-cell! ball-radius 15)
```

ball radius increases (in graphics window)

```
> (set-cell! ball-color "blue")
```

ball changes color

The combination of cells and signals provides a powerful notion of "transparency" for FrTime programs. That is, when the programmer changes a value, the effect of the change propagates throughout the system, but the state associated with the interactive session is otherwise preserved. In contrast, changing values in any other DrScheme language generally results in inconsistency and requires a complete re-execution of the program, which destroys the state of the interactive session.

## 5 Extended Example

To tie together everything we have discussed so far, we present a lengthy example illustrating the use of FrTime in a distributed, graphical application. The application is a networked, multi-player paddle-ball simulation inspired by an example in the Haskell School of Expression [18] (the original example is neither networked nor multi-player).

We start by defining the simple motion of a ball bouncing inside a window. We simplify this task by concerning ourselves with the ball's *velocity* rather than its position. As a prime example of FRP's expressive power, we can then define the ball's position by taking the *integral* of the velocity.

Our simulation assumes the absence of wind and friction, so the ball's speed is always the same, and its velocity only changes if it collides with an obstacle. Initially, we can ignore the paddles and only treat collisions with walls. In this case, when the ball hits a side wall, we negate the horizontal component of its velocity. Likewise, if it hits the top or bottom wall, we negate the vertical component. We can express this definition as follows:

```
(define ball-velocity
  (accum/e→b
    (merge-e (map-e
               (λ (_) negate-x)
```

```
        (when-e (or (< (posn-x ball-pos) ball-radius)
                    (> (posn-x ball-pos) (− window-width ball-radius)))))
    (map-e
      (λ (_) negate-y)
        (when-e (or (< (posn-y ball-pos) ball-radius)
                    (> (posn-y ball-pos) (− window-height ball-radius))))))
  initial-velocity))
```

Next, we define the positions of the paddles. To make things easy, we have each paddle follow the mouse on its respective machine. We only need to ensure that the paddle stays on its half of the screen, a condition we can enforce with the function *clip*:

```
(define (clip n lo hi)
  (cond
    [(< n lo) lo]
    [(> n hi) hi]
    [else     n]))

(define paddle1-pos
  (make-posn
    (clip (posn-x mouse-pos) 0 (/ window-width 2))
    (clip (posn-y mouse-pos) 0 (/ window-height 2))))
```

The paddles in our game are round, so detecting collisions is easy: when the distance between the centers of the ball and paddle is less than the sum of their radii, they have collided. The effect of a collision is somewhat complicated, however, since the ball may be deflected in different directions depending upon the angle at which it strikes. We provide code for the velocity transformer without attempting to explain the mathematics:

```
(define (collision paddle-pos ball-pos ball-v)
  (let ([u (normalize (posn− paddle-pos ball-pos))])
    (posn− ball-v (posn∗ u (∗ 2 (posn-dot ball-v u))))))
```

To make the game interesting, we put a goal on each side of the screen and keep a count of how many times the ball hits each player's goal. We check this with a simple additional constraint on the the wall collision test, and we accumulate the score with the function *count-occ* (defined in section 2). The *filter-e* expression takes all collisions with the left wall and filters out the ones that fall outside the goal area. Note that the predicate uses *cur-val* to sample the current position of the ball.

```
(define player1-score
  (count-e
    (filter-e
      (λ (_) (and (> (cur-val (posn-y ball-pos)) goal-top)
                  (< (cur-val (posn-y ball-pos)) goal-bottom)))
      (when-e (< (posn-x ball-pos) ball-radius)))))
```

Finally, we need a way of making the simulation work for two players over a network. For simplicity, we make the first player's process ($P_1$) administer the simulation and

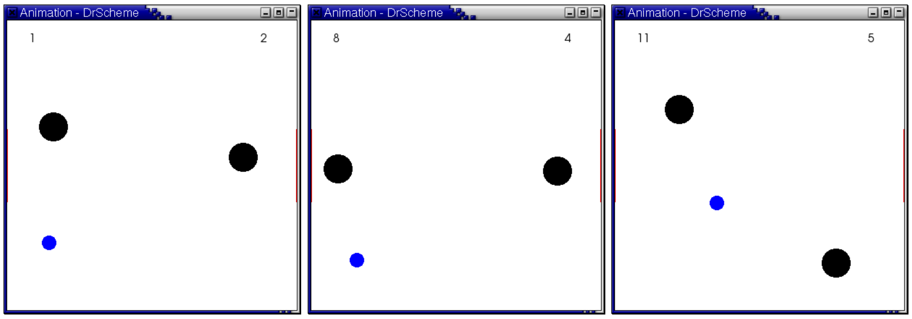


**Fig. 6.** Screenshots of a networked paddle-ball game

the second player's process ($P_2$) simply echo it to a display. Although this establishes a hierarchical relationship between the processes, they still require mutual communication. Specifically, $P_2$ provides the position of the second player's paddle to $P_1$, and $P_1$ provides everything else (scores, ball, first player's paddle) to $P_2$.

We implement the communication through remote events that indicate changes in the corresponding values. For example, $P_2$ uses *bind* to publish the position of its paddle:

```
(bind 'paddle2-pos-changes (changes paddle2-pos))
```

$P_1$ uses *hold* to construct a behavior from this event.

```
(define paddle2-pos
  (hold (make-posn 100 100) ; arbitrary initial position
        (remote-event 'machine2 'paddle2-pos-changes)))
```

Communication in the other direction works in the same manner, except that there is more information to transmit.

The use of signals makes the expression of the paddle-ball simulation relatively straightforward. In particular, the ability to distribute signals across machines eases the implementation of a networked simulation. The resulting descriptions are very concise: we need less than a hundred lines of FrTime code (total) to define the two processes. In Figure 6, we show a screenshot of the running simulation.

## 6 Related Work

Fudgets [5] is a library for building reactive systems in a purely functional language. Originally designed for programming graphical user interfaces, it models interactive components as demand-driven stream processors, which support general forms of communication and interaction. The approach conceptually resembles our model of event processors. However, our use of multiplexed asynchronous message-passing offers a natural implementation of asynchronous stream merging, which the sequential Haskell implementation of Fudgets cannot in general achieve.

Elliott and Hudak present Fran [10], a Haskell library for reactive animations based on a continuous model of time, and Elliott [9] discusses various functional implementation strategies for Fran-like systems, out of which more general FRP systems evolved. Wan and Hudak [24] develop a formal semantic model for FRP, studying in particular the properties of a stream-based implementation.

FranTk [21] extends Fran with a library of user interface combinators. Courtney and Elliott [8] present a highly declarative user interface system, and Nilsson, Courtney, and Peterson [20] discuss the associated implementation, which is based on arrows [19]. These FRP implementations adopt a synchronous model of concurrency, which is also used in a number of dataflow languages, such as Lucid [23], Lustre [6], Esterel [4], and Signal [14].

We have generally tried to follow the spirit of FRP when developing our notions of behaviors and events, though not the implementation strategy. In contrast to these languages, signals in our system change in response to asynchronous events. The event-driven model more closely resembles the architecture of implicit-invocation (II) systems [22]. II systems are generally modeled in terms of communicating sequential processes [16] that register interest in particular classes of events. Importantly, the runtime system manages much of the actual communication, including dispatching events to appropriate processes. Our signal manager is in some ways analogous to an II dispatcher, since it tracks dependencies between signals and uses the information to scheduling computation.

Various languages, such as Kali Scheme [7] and Erlang [2], provide mechanisms for concurrency and communication that are appropriate for building implicit-invocation systems. Since we needed other features specific to PLT Scheme, we built a custom message-passing system in the style of Erlang.

Adaptive Functional Programming (AFP) [1] is concerned with efficiently recomputing a function application when the argument changes. The ML implementation employs an update propagation algorithm that closely resembles the approach taken in FrTime. However, AFP is not intended for interaction and has no notion of behaviors or events. Still, with slight modifications, it might be possible to improve FrTime's update algorithm with ideas from AFP, or to build an FRP implementation in ML atop the AFP system.

Scalable Vector Graphics (SVG) [11] is a declarative language for vector-based images with some support for animation and interaction. Slithy [25] has similar capabilities but is more heavily focused on developing animated presentations and has richer mechanisms for modeling and hierarchical composition. Though these languages follow a more imperative approach to interaction, they also provide interesting abstractions that may be useful in a system like FrTime.

## 7 Conclusions and Future Work

We have presented FrTime, a Scheme-based implementation of Functional Reactive Programming, and we have discussed its distinct implementation strategy. We have also seen that several advantages derive from this implementation strategy, specifically easier integration with external systems, distribution of signals across networks, allowance of benign impurities, and efficient use of processing resources. We have presented a non-trivial FrTime program that takes advantage of some of these capabilities.

There are several possible future directions for this work. We have discussed one drawback of our asynchronous notion of concurrency—the existence of momentary inconsistencies between behaviors. Fortunately, in many cases there is little or no harm in allowing a value to be slightly and briefly out of date. However, to make our system more robust, we are interested in exploring the introduction of limited forms of synchrony. For example, it might be desirable to provide a generalized *cur-vals* procedure, which would consume a list of signals and return a consistent snapshot of all of them.

We are also interested in applying FrTime to new applications, especially ones in which our implementation would be advantageous. For example, as mentioned in the introduction, streaming databases [3] and sensor networks [15] rely heavily on asynchronous communication and other forms of interaction. They are also well suited for the declarative programming style afforded by dataflow languages and FRP. Especially in sensor networks, where communication is expensive and power consumption is critical, an asynchronous, demand-driven implementation is valuable. For these reasons, FrTime seems like a natural match for programming such systems. We are also curious to see what sorts of new abstractions might prove useful in these novel applications.

## References


[1] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 247–259, 2002.

[2] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 1996.

[3] S. Babu and J. Widom. Continuous queries over data streams. *ACM SIGMOD Record*, 30(3):109–120, 2001.

[4] G. Berry. *The Foundations of Esterel*. MIT Press, 1998.

[5] M. Carlsson and T. Hallgren. FUDGETS: a graphical user interface in a lazy functional language. In *Conference on Functional Programming Languages and Computer Architecture*, pages 321–330, 1993.

[6] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178–188, 1987.

[7] H. Cejtin, S. Jagannathan, and R. Kelsey. Higher-order distributed objects. *ACM Transactions on Programming Languages and Systems*, 17(5):704–739, September 1995.

[8] A. Courtney and C. Elliott. Genuinely functional user interfaces. In *Haskell Workshop*, 2001.

[9] C. Elliott. Functional implementations of continuous modeled animation. In *International Symposium on Programming Languages: Implementations, Logics, and Programs*. Springer-Verlag, 1998.

[10] C. Elliott and P. Hudak. Functional reactive animation. In *ACM SIGPLAN International Conference on Functional Programming*, pages 263–277, 1997.

[11] J. Ferraiolo. Scalable vector graphics (SVG) 1.0 specification, December 1999. http://www.w3.org/TR/1999/WD-SVG-19991203/.

[12] R. B. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi, and M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *International Symposium on Programming Languages: Implementations, Logics, and Programs*, number 1292 in Lecture Notes in Computer Science, pages 369–388, 1997.

[13] D. Garlan, S. Jha, D. Notkin, and J. Dingel. Reasoning about implicit invocation. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 209–221, 1998.

[14] T. Gautier, P. Le Guernic, and L. Besnard. Signal: A declarative language for synchronous programming of real-time systems. In G. Goos and J. Hartmanis, editors, *Functional Programming Languages and Computer Architecture*, pages 257–277. Springer-Verlag, 1987. LNCS 274.

[15] J. Hill, R. Szewczyk, A. Woo, S. Hollar, and D. C. K. Pister. System architecture directions for networked sensors. In *ASPLOS*, November 2000.

[16] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[17] G. J. Holzmann and D. Peled. The state of SPIN. In *Conference on Computer-Aided Verification*, 1996.

[18] P. Hudak. *The Haskell school of expression: learning functional programming through multimedia*. Cambridge, 2000.

[19] J. Hughes. Generalizing monads to arrows. *Science of Computer Programming*, 37(1-3), May 2000.

[20] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *ACM SIGPLAN workshop on Haskell*, pages 51–64, 2002.

[21] M. Sage. FranTk: A declarative GUI language for Haskell. In *ACM SIGPLAN International Conference on Functional Programming*, 2000.

[22] K. J. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–268, July 1992.

[23] W. W. Wadge and E. A. Ashcroft. *Lucid, the dataflow programming language*. Academic Press U.K., 1985.

[24] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–252, 2000.

[25] D. E. Zongker and D. H. Salesin. On creating animated presentations. In *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 298–308, 2003.