Frappé: Functional Reactive Programming in Java[*]

Antony Courtney

Dept. of Computer Science
Yale University
New Haven, CT 06520
`antony@apocalypse.org`

Abstract. Functional Reactive Programming (FRP) is a declarative programming model for constructing interactive applications based on a continuous model of time. FRP programs are described in terms of *behaviors* (continuous, time-varying, reactive values), and *events* (conditions that occur at discrete points in time).
This paper presents Frappé, an implementation of FRP in the Java progamming language. The primary contribution of Frappé is its integration of the FRP event/behavior model with the Java Beans event/property model. At the interface level, any Java Beans component may be used as a source or sink for the FRP event and behavior combinators. This provides a mechanism for extending Frappé with new kinds of I/O connections and allows FRP to be used as a high-level declarative model for composing applications from Java Beans components. At the implementation level, the Java Beans event model is used internally by Frappé to propagate FRP events and changes to FRP behaviors. This allows Frappé applications to be packaged as Java Beans components for use in other applications, and yields an implementation of FRP well-suited to the requirements of event-driven applications (such as graphical user interfaces).

1 Introduction

Recent work in in the functional programming community has proposed Functional Reactive Programming (FRP) as a declarative programming model for constructing interactive applications. FRP programs are described in terms of *behaviors* (continuous, time-varying, reactive values), and *events* (conditions that occur at discrete points in time).

All previous implementations of FRP have been embedded in the Haskell programming language [15]. As discussed in [10], Haskell's lazy evaluation, rich type system, and higher-order functions make it an excellent basis for development of new domain-specific languages and new programming paradigms such as FRP.

In the Java community, recent work has produced the Java Beans component model [2]. The Java Beans component model prescribes a set of programming conventions for writing re-usable software components. A programmer writes a Java Beans component by

[*] This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

defining a Java class that specifies a set of *events* (“interesting” conditions which result in notifying other objects of their occurrence) and *properties* (named mutable attributes of the component that may be read or written with appropriate methods).

The FRP and Java Beans programming models have very different goals and appear, at first glance, to be completely unrelated. The goal of FRP is to enable the programmer to write concise descriptions of interactive applications in a declarative modeling style, whereas the goal of Java Beans is to provide a component framework for visual builder tools. However, the two models also have some alluring similarities: both have a notion of *events*, and both have a notion of values that change over time (*behaviors* in FRP, *properties* in Java Beans). Our primary motivation for developing Frappé was to to explore the relationship between the two models.

This paper presents Frappé, an implementation of FRP in Java. Our implementation is based on a correspondence between the FRP and Java Beans programming models, and our implementation integrates the two models very closely. There are two aspects to this integration: First, any Java Beans component may be used as a source or sink for the FRP event and behavior combinators. Second, the Java Beans event model is used internally by Frappé for propagation of FRP events and changes to FRP behaviors. Allowing any Java Beans component to be used as a source or sink for the FRP event and behavior combinators allows the Java programmer to use FRP as a high-level declarative model for composing interactive applications from Java Beans components, and allows Frappé to be extended with new kinds of I/O connections without modifying the Frappé implementation. Using the the Java Beans event model internally allows Java Beans components connected by FRP combinators to be packaged as larger Java Beans components for use by other Beans-aware Java tools, and yields a “push” model for propagation of behavior and event values that is well-suited to the requirements of graphical user interfaces.

The remainder of this paper is organized as follows. Section 2 gives a brief review of the FRP and Java Beans models. Section 3 describes the Frappé library interface and how the library is used to construct applications from Java Beans components and Frappé classes. Section 4 describes the implementation of FRP in Frappé. Section 5 summarizes the status of the implementation. Section 6 discusses some limitations of our implementation. Section 7 describes related work. Section 8 summarizes our contributions, and briefly discusses some open questions and plans for future work.

2 Preliminaries

2.1 Functional Reactive Programming

Functional Reactive Programming (FRP) is a high-level declarative programming model for constructing interactive applications. In this section, we give a very brief introduction to the aspects of FRP needed to understand the rest of the paper; see [6, 4, 11] for more details.

There are two key polymorphic data types in FRP: `Behavior` and `Event`. Conceptually, a `Behavior` is a time-varying continuous value. One can think of type `Behavior a` as having the Haskell definition:

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

That is, a value of type `Behavior a` is a function from `Time` to `a`. Given this definition, we can think of sampling a behavior as simply applying the behavior to some sample time. The simplest examples of behaviors are *constant* behaviors: those that ignore their time argument and evaluate to some constant value. For example, `constB red` has type `Behavior Color`. It evaluates to `red` regardless of the sample time. An example of a time-varying behavior (taken from a binding of FRP for computer animation [6]) is `mouse` (of type `Behavior Point`). When sampled, `mouse` yields a representation of the mouse position at the given sample time. Sampling the mouse at different times may yield a different `Point` depending on whether the user has moved the mouse.

Conceptually, an `Event` is some condition that occurs at a discrete point in time. In Haskell, we write the type `Event a` for an *event source* capable of producing a sequence of *occurrences*, where each occurrence carries a value of type `a`. For example:

```
lbp :: Event ()
key :: Event Char
```

declare the types of two primitive event sources defined in the FRP binding for computer animation. The first event source, `lbp`, generates an event occurrence every time the left mouse button is pressed. Each occurrence carries a value of type `()` (read "unit"), meaning that there is no data carried with this event other than the fact that it occurred. The second event source, `key`, generates an event occurrence every time a key is pressed on the keyboard. Each occurrence carries a value of type `Char` representing the key that was pressed.

An implementation of FRP provides the programmer with a set of primitive behaviors and event sources, and a library of combinators for creating new behaviors and event sources from existing ones. For example, the expression:

```
lbp -=> red
```

uses the `-=>` combinator (of type `Event a -> b -> Event b`) to produce an event source of type `Event Color`. The event occurs whenever `lbp` occurs (i.e. every time the left mouse button is pressed), but each occurrence carries the value `red`. More complex event sources and behaviors are produced by nested applications of combinators. For example, the expression:

```
(lbp -=> red .|. rbp -=> blue)
```

uses the *merge* operator (`.|.`) to produce an an event source (of type `Event Color`) that occurs whenever the left or right mouse button is pressed. When the left button is pressed, an occurrence is generated carrying the value `red`; when the right button is pressed, an occurrence is generated carrying the value `blue`.

The FRP model defines a combinator, `switcher`, for converting an event source to a behavior. The type of `switcher` is given as:

```
switcher :: Behavior a -> Event (Behavior a) -> Behavior a
```

Informally, `switcher` produces a behavior that initially follows its first argument. However, every time an event occurs on the event source given as the second argument, `switcher` "switches" to follow the behavior carried in that event occurrence. For example:

```
c = switcher red (lbp -=> red .|. rbp -=> blue)
```

uses `switcher` to define `c` as a behavior with type `Behavior Color`[1]. When the application starts, `c` will initially be `red`. When the left mouse button is pressed, `c` changes to `red`, and when the right mouse button is pressed, `c` changes to blue. This is an example of a *reactive* behavior – it changes in response to a user input event.

A binding of FRP for a particular problem domain will usually define a type for a top-level behavior that represents the output of the application. A complete application is written by by using the FRP combinators to define an expression for a behavior of this type, and passing this value to a display function. For example, in computer animation, the output of an application is of type `Behavior Picture`. An example, then, of a complete FRP application for computer animation is:

```
exampleApp = withColor c circle
  where c = switcher red (lbp -=> red .|. rbp -=> blue)

main = animate exampleApp
```

this application renders a circle of unit size in an output window. The circle will be initially red, but the color will change between red and blue as the left and right mouse button are pressed.

2.2 Java Beans

This section gives a brief summary of the Java Beans programming model. For a more complete account, the reader is referred to the Java Beans Specification [2].

What *is* a Bean? The Java Beans Specification defines a Java Bean informally as "a reusable software component that can be manipulated visually in a builder tool".[2] For example, all of the Swing user interface components (buttons, sliders, windows, etc.) are Beans. However, Beans need not have any visual appearance at run-time. For example, a software component that takes a ticker symbol as input and periodically delivers stock quotes for the ticker symbol could easily be packaged as a Bean.

More concretely, then, a Bean is a Java class that conforms to certain programming conventions prescribed by the Java Beans specification. These conventions specify that a Bean should make its functionality available to clients through:

– *Properties*–mutable named attributes of the object that can be read or written by calling appropriate *accessor* methods.
– *Events*–A named set of "interesting" things that may happen to the Bean instance. Clients may register to be notified when an event occurs by implementing a *listener* interface. When the event occurs, the component notifies the client by invoking a method defined in the listener interface.

[1] Here we use *implicit lifting* of constants to Behaviors. Strictly speaking, we should have written `constB red` instead of just `red`, but the Haskell implementations of FRP use `instance` declarations to perform this translation automatically.

[2] In the remainder this paper, we use the terms "Java Beans component" and "Bean" interchangeably.

– *Methods*–Ordinary Java methods, invoked for their side-effects on the Bean instance or its environment.

The Beans model does not require a separate interface definition language for specifying the interface to a Java Beans component. Instead, the Beans model prescribes that a Java Bean can be written as an ordinary Java class in the Java language, and can be packaged for use by a builder tool simply by compiling the source file to the standard Java class file format. The "builder tools" use reflection [13] on the class file to recover information about the features exported by the particular Bean, and the standard Java library provides a set of helper classes in the `java.beans` package for use by builder tools. These helper classes perform the low-level reflection operations to recover information about events, properties and methods supported by a Bean.

A full discussion of the programming conventions used to define Java Beans is outside the scope of this paper. However, to give a basic feel for the programming conventions, we will show how a Java class is defined to export a set of *properties*, thus making the class a Java Bean.

Properties are accessed by means of *accessor methods*, which are used to read and write values of the property. A property may be *readable* or *writable* (or both), which determines whether the property supports *get* or *set* accessor methods. The convention for *get* accessor methods is:

```
public PropertyType getPropertyName();
```

and the convention for *set* accessor methods is:

```
public void setPropertyName(PropertyType arg);
```

where *PropertyName* is the (appropriately capitalized) name of the property, and *PropertyType* is the Java type of the property.

For example, the class `JComponent` of Java Swing defines get and set accessors for its width and height properties as:

```
public class JComponent {
  ...
  public int getWidth();
  public void setWidth(int arg);
  public int getHeight();
  public void setHeight(int arg);
}
```

Bound Properties A particularly important aspect of the Java Beans specification is its provision for *bound* properties. A component author may specify that a property is a *bound* property, meaning that interested clients can register to be notified whenever the property's value changes. A property's value might change either as a direct result of the application program invoking a *set* accessor method, or as an indirect result of some user action. For example, a text entry widget might have a *text* property representing the text entered into the field by the user. If implemented as a bound property, an application could register to be notified whenever the user changed the contents of the text entry component. Bound properties play a critical role in the implementation of FRP Behaviors in Frappé.

3 Frappé - A User's Perspective

Frappé is implemented as a Java library organized around two Java interfaces: `Behavior` and `FRPEventSource`. These interfaces correspond to the parameterized types `Behavior` and `Event` in Haskell implementations of FRP.

The combinators in core FRP are implemented as concrete classes in Frappé. Each such class provides a constructor that takes the same number and type of arguments as the combinator's Haskell counterpart, and the class implements either the `Behavior` or `FRPEventSource` interface in accordance with the result type of the particular combinator. For example, the `switcher` combinator introduced in section 2 is realized as the following Java class:

```
public class Switcher implements Behavior {
  public Switcher(Scheduler sched,
                  Behavior bv,
                  FRPEventSource evSource)
  ...
}
```

The first argument to the constructor is a global *scheduling context* used by the implementation. The programmer obtains such a context once during initialization, and simply passes it to all constructors of Frappé classes. The next two arguments correspond to the arguments of the `switcher` combinator: the behavior to follow initially, and an event source whose occurences carry a behavior to follow after the occurence.

3.1 A Simple Example

To write programs in Frappé, the programmer simply instantiates a number of Java Beans components and connects those components together using the Frappé classes corresponding to FRP combinators. The program then relinquishes control to the Java runtime library's main event loop.

As a concrete example, consider writing a a program to display a red circle on the screen that tracks the current mouse position. In SOE FRP, this would be written:

```
ball = stretch 0.3 (withColor red circle)
anim = (lift2 move) (p2v mouseB) (constB ball)

main = animate anim
```

The first line here defines `ball` as a static picture of a red circle located at the origin scaled by a factor of 0.3. The second line applies the *lifting* combinator `lift2` to the function `move`. The lifting combinators convert functions over static values to functions over Behaviors, so if we have a function

```
f :: a -> b -> c
```

then the lifted version of this function is:

```
(lift2 f) :: Behavior a -> Behavior b -> Behavior c
```

Here is an equivalent code fragment that implements this same example in Frappé[3]:

```
Drawable circle = new ShapeDrawable(new Ellipse2D.Double(-1,-1,2,2));

Drawable ball = circle.withColor(Color.red).stretch(0.3);

Behavior mouse = FRPUtilities.makeBehavior(sched, frame, "mouse");

Behavior anim =  FRPUtilities.liftMethod(sched, new ConstB(ball),
                                  "move", new Behavior[] { mouse });

franimator.setImageB(anim);
```

First, we define `circle` as a `ShapeDrawable` constructed from a Java2D Ellipse. `Drawable` is an abstract class provided by Frappé to simplify animation programming. The concrete class `ShapeDrawable` is a `Drawable` capable of rendering any `Shape` from the Java2D API [9]. The methods provided by `Drawable` are based on the low-level graphics model of Fran [6] and provide support for scaling, translating or changing the color of a `Drawable` as well as overlaying two Drawables to form a composite Drawable. We use two of these methods of `circle` to define `ball` as a scaled, red circle.

Next we define `mouse` using `FRPUtilities.makeBehavior`. This static method creates a `Behavior` from a bound property of a Java Bean instance. In this case, the variable `frame` is an instance of the class `FranFrame`. A `FranFrame` is a top-level window (specifically, a specialization of Swing's `JFrame` class) for displaying an animation. `FranFrame` is a Java Bean that provides a bound property "mouse". At any point in time, the value of this property is the current mouse position within the window. This definition binds the variable `mouse` to a Behavior that, when sampled, will return the value of `frame.getMouse()` at the time of sampling. The property name is passed as a string to `makeBehavior()` because the implementation uses Java reflection [13] on the class instance to look up the appropriate accessor method and to register for notification when the property changes.

Next we use `FRPUtilities.liftMethod` to construct the animation. This static method is a *lifting combinator* similar to the `lift`N combinators provided in Haskell implementations of FRP. The second argument to `liftMethod()` is a Behavior whose sample values implement the given method. In this example, we pass `new ConstB(ball)` as the second argument. This is a new constant Behavior whose value at every sample point is `ball`. Since `ball` is an instance of `Drawable`, it supports `Drawable`'s `move()` method, defined in `Drawable` as:

```
public abstract class Drawable {
  /** return a new Drawable that is a translated version of this */
  public Drawable move(Point2D pos);
  ...
}
```

[3] There is also a small amount of standard "boilerplate" code required to wrap this code in a Java class, instantiate the top-level window in which the animation is displayed, and initialize Frappé. We have elided this extraneous code due to space limitations, but a complete version of this example (and many others) is available in the Frappé distribution from the Frappé web site.

By using `liftMethod`, this method is lifted to operate on *Behaviors* rather than static values: the method is applied pointwise to the values of the target instance Behavior and argument Behaviors at every sample time. In this case, the target instance is a constant Behavior, and the argument is the Behavior variable `mouse` that yields the current mouse position as a `Point2D` at every sample time. The result will be a `Behavior` whose value at every sample time is a `Drawable` that renders a scaled red circle moved to the current mouse position.

Finally, we pass this Behavior (`anim`) to the `franimator` component's `setImageB()` method to actually display the animation on the screen. The variable `franimator` is an instance of the `Franimator` class obtained from `frame`, and is a specialization of the Swing `JPanel` component for displaying animations.

It is interesting to compare the Haskell version of this example, the Frappé version, and to consider what a corresponding version of this example would look like in pure Java / Swing without Frappé. The Frappé version introduces considerable notational overhead relative to the original Haskell version, but much of this stems from Java's explicit type declarations, verbose method / class names, and lack of name overloading (a la Haskell type classes).

Space considerations prevent us from including complete source for a pure Java / Swing version of this example, but we can outline the basic approach. A pure Java / Swing version would have to implement a `MouseMotionListener` to handle `mouseMoved` events, and register this listener instance with the `JPanel` in which the circle is displayed. The programmer would then have to implement the handler to extract the (x, y) position from the `MouseMotionEvent`, and use this value to set the (x, y) position of the circle on every event occurence.

We have found Frappé to be more concise than Java alone for many of our example programs. In this example, the primary savings comes from using combinator classes instead of trivial listener instances to do simple forms of event propagation. Perhaps more importantly than the savings in code size, we feel the Frappé version is conceptually cleaner than a corresponding pure Java/Swing version. Instead of writing a plethora of event handlers that perform imperative actions to change the program state, the Frappé programmer simply writes a set of *time-invariant, side-effect free definitions* describing how the different components of the program are connected together.

3.2 Using Java Beans Events

Just as Java Beans properties may be converted to Behaviors by a call to `FRPUtilities.makeBehavior()`, Java Beans events may be converted to FRP Event Sources by a call to `FRPUtilities.makeFRPEvent()`. For example, the following sets the variable `lbpEventSource` to an `FRPEventSource` that has an event occurrence every time the "lbp" event occurs on `frame`:

```
FRPEventSource lbpEventSource = FRPUtilities.makeFRPEvent(sched, frame,
                                                          "franMouse","lbp");
```

The second argument is the Bean instance whose event occurrences are being observed. The third argument is the name of the "event set" of interest. This determines the *listener type* that is used to register for event notifications. The fourth argument is the name of the specific notification method of interest within the listener type. In this example, `franMouse` is an event set that identifies the `FranMouseListener` event listener class, and `lbp` is the name of the particular method invoked on a `FranMouseListener` instance when the left mouse button is pressed.

3.3 Using Java Beans Properties as Output Sinks

We have seen that Java Beans properties can be used as inputs to Frappé's combinator classes. On the output side, we can also connect any Behavior to a writable property of some Bean, using a `BehaviorConnector`. For example, the following creates a `BehaviorConnector` that will connect `strB` (a String-valued Behavior) to a JLabel's "text" property:

```
JLabel label = ...
Behavior strB = ...

new BehaviorConnector(sched, strB, label, "text");
```

This `BehaviorConnector` will invoke `label.setText()` every time the value of `strB` changes value.

3.4 Encoding Recursive Definitions

In the Haskell implementations of FRP, many programs rely on Haskell's lazy evaluation to write mutually recursive behaviors and events. For example:

```
sharp2 = when (time >* 1)
sharp3 = when spike
spike = (constB False) `switcher` ((sharp2 -=> (constB True)) .|.
                                   (sharp3 -=> (constB False)))
```

The above defines `spike` as a `Behavior Bool` that is momentarily `True` at some point at time $t = 1 + \epsilon$ (for some ϵ) and `False` everywhere else.

To encode this example in Frappé, we must perform the cyclic wiring explicitly. To support this, Frappé defines an extra constructor for the `Switcher` class that takes only a scheduling context. A call to this constructor returns an *uninitialized* `Switcher` instance. Recall that the `switcher` combinator takes two arguments (a target behavior and an event source) that are usually passed in explicitly to the `Switcher` constructor. If this alternative constructor is used, then the programmer must make explicit calls to the `bindTarget()` and `bindEventSource()` methods before running the resulting Behavior. The above example is coded in Frappé as:

```
Switcher spike = new Switcher(sched); // uninitialized instance!

Behavior gtOneB = // Frappe encoding of gtOneB = time >* (constB 1)
  FRPUtilities.lift(sched, this.getClass(),
                    "gtOne", new Behavior[] { time });

FRPEventSource sharp2 = new When(sched, gtOneB);

FRPEventSource sharp3 = new When(sched, spike);

spike.bindTarget(new ConstB(Boolean.FALSE));
spike.bindEventSource(new EventMerge(sched, ...));
```

Note that the reference to the uninitialized instance `spike` is used in the definition of `sharp3`, but the appropriate calls are made to `spike.bindTarget()` and `spike.bindEventSource()` during initialization.

4 Implementing FRP in Java

4.1 Behaviors

Like other Haskell implementations of FRP, Frappé represents the FRP program as a graph structure at runtime. We achieve this by defining a Java class for each FRP combinator. Each node in the combinator graph is represented by an object instance at runtime, and each edge is represented by a field with a reference to another instance of a combinator class.

What operations must each Behavior node in the runtime graph support? A detailed study of one particular FRP implementation (SOE FRP, described in [11].[4]) revealed that, in the absence of generalized time transformation, each node in the graph essentially needs to support only one operation: *get the value of the Behavior at the current sample time.*

Interestingly, we can model this operation by defining a Behavior as a Java Bean with a single bound property. This leads to the Java encoding illustrated in figure 1. While the syntax is somewhat verbose, this can be read simply as "Every Behavior is a Bean that provides a bound property named *value*."

Individual Behavior objects might be connected as inputs to other nodes in the combinator graph, and those nodes will need to be informed when a Behavior's value has changed. Hence, we make *value* a *bound* property, so that other nodes can register for a *PropertyChangeEvent* when the value of the Behavior changes. Our implementation uses such events to propagate behavior values through the system.

An implementation of the `Behavior` interface supports registration of *listeners*, as required of bound properties. All output connections for a node are stored in this listener list. If an FRP combinator class uses a `Behavior` as one of its inputs, the combinator class must implement the `PropertyChangeListener` interface in order to be notified of changes in its input behavior.

[4] So-named because the implementation is described in the textbook "The Haskell School Of Expression".

```
public interface Behavior {
    /** Accessor to read the current value of this Behavior */
    public Object getValue();

    /** register a PropertyChangeListener */
    public void addPropertyChangeListener(PropertyChangeListener l);

    /** Remove a PropertyChangeListener from the list of listeners. */
    public void removePropertyChangeListener(PropertyChangeListener l);
}
```

Fig. 1. Java encoding of Behaviors

Since the Haskell definition of Behavior is a polymorphic type, we declare the return type of `getValue()` as `Object`. The value returned must be converted to an instance of the appropriate type using a cast, and the cast is checked at runtime.

4.2 Events

We implement FRP Events by mapping the FRP notion of "event" directly to a Bean Event named `FRPEvent`.

The class `FRPEvent` represents a single event occurrence. It extends `java.util.EventObject`, as required by the Java Beans specification. The `FRPEventSource` interface is implemented by every class that generates FRP Events. This interface corresponds directly with the Haskell type `Event a` that identifies an event source in the SOE implementation. The methods defined in `FRPEventSource` are those prescribed by the Java Beans conventions for registering event listeners. This interface declaration can be read as stating that "Every FRP Event Source is a Bean event source for the event named *FRPEvent*."

The `FRPEventListener` interface is implemented by any class that wishes to be notified when an `FRPEvent` occurs on some source. A listener is registered with the event source by passing a reference to the listener to the source's `addFRPEventListener()` method. Then, at some point later when the event occurs, the event source will notify all registered listeners by invoking each listener's `eventOccurred()` method, passing it an `FRPEvent` instance representing the event occurrence.

4.3 Propagation of Event and Behavior Values

Propagation of event and behavior values in Frappé is purely event-driven. To execute an FRP specification, a user program simply constructs an explicit graph of FRP combinators (*initialization*), and relinquishes control to the main event loop in the Java runtime library. When there is input to the application (for example, when the user presses a mouse button), the Java runtime will invoke an event handler of some object in the Frappé implementation that implements a primitive FRP event source or behavior. This primitive event handler, in turn, will invoke the appropriate event handler of each registered listener:

– For an event source, each event listener implements the `FRPEventListener` interface. The listener's `eventOccured()` method is invoked to propagate the event occurrence.
– For a behavior, each event listener implements the `PropertyChangeListener` interface. The listener's `propertyChanged()` method is invoked to propagate the change in the behavior's value.

Each registered listener for a primitive event or behavior is an FRP combinator. The combinator's event handler will compute any changes to its output and invoke an event handler method on each of *its* registered listeners. Propagation of events continues in this way until some "output" listener is reached.

5 Status and Availability

The complete Frappé distribution is available from the Frappé web site at `http://www.haskell.org/frappe` under the terms of the GNU General Public License. We have working implementations of all of the core combinators and examples given in [14], using the encoding of behaviors and events presented here. For the most part, the implementation of these combinators is a straightforward translation of the formal definition into the Java language using the types and propagation model presented here.

6 Limitations

Because Java lacks a polymorphic type system, and because our implementation makes extensive use of Java reflection, our implementation of FRP is not statically type-safe. In this respect, Frappé is no better and no worse than many other Java libraries, such as the Java collection classes. Nevertheless, it would be an interesting exercise to rewrite Frappé using GJ [1]. An alternative approach (that we are pursuing) is to use Frappé as a compilation target for some other high-level Haskell-like FRP notation. In this case, the front-end translator can perform polymorphic type-checking statically, and generate Frappé code that is guaranteed not to have type errors at runtime.

Frappé assumes that event processing is single-threaded and synchronous. That is, all primitive Java Beans events used as event or behavior sources for Frappé must be fired from the system's event dispatching thread, and each event must completely propagate through the FRP combinator graph before the next event is handled. This single-threaded, synchronous event processing model is also required by Java Swing, and Frappé does not impose any further restrictions than those already required for event handling in Swing.

Like the stream-based implementation from which it derives, our implementation of FRP is unable to detect instantaneous predicate events. An instantaneous predicate event is one that happens only at some specific instantaneous point in time. For example:

```
sharp :: Event ()
sharp = when (time==*1)
```

is only true instantaneously at time=1. An event such as `sharp` can not be detected simply by monotonic sampling; accurate detection of predicate events requires *interval analysis*, as discussed in [6, 3]. In many ways, the inability to detect instantaneous predicate events is similar to the problem of comparing two floating point numbers for equality using `==`, lifted to the time domain.

Finally, Frappé does not support generalized time transformations.

7 Related Work

Elliott [3] has done much of the pioneering work on implementations of the FRP model in Haskell, and reported on the design tradeoffs of various implementation strategies. Hudak [11] provides a completely annotated description of a stream-based implementation of FRP from which our implementation is derived.

Recent work in the functional programming community has produced ways to make component objects and library code written in imperative languages available from Haskell [8, 7, 12]. Our work and this previous work share the common goal of providing programmers with a declarative model for connecting component objects written in imperative languages. However, our approach can be viewed as the "inverse" of these efforts: instead of embedding calls to component objects written in an imperative language into a declarative programming model, Frappé takes a declarative programming model and embeds it in an imperative language that supports component objects.

Elliott's work on *declarative event-oriented programming* [5] showed that FRP's event model (implemented in Fran) could be used to compose interactive event-driven user interfaces in a declarative style, and compared this to the conventional imperative approaches for programming user interfaces. FranTk [16] is a complete binding of the FRP programming model to the Tk user interface toolkit. FranTk demonstrates the viability of using FRP for user interfaces, and inspired us to explore how we might adapt the FRP model for use with the Java Swing toolkit.

8 Conclusions and Future Work

We have presented an implementation of FRP in the Java programming language. The most significant aspect of our implementation is that it is based on a close correspondence between the FRP event/behavior model and the Java Beans event/property model.

One of the unique aspects of Frappé is its ability to use Java Beans components as sources or sinks for FRP combinators. In principle there is no reason why this features needs to be limited to our Java implementation of FRP. It would be interesting to explore adding a similar feature to one of the Haskell-based implementations of FRP using COM objects as components.

Our experience with Frappé to date is limited, but promising. As discussed in section 3, Frappé programs compare favorably with similar programs written in pure Java / Swing, both in terms of program size and conceptual clarity. Nevertheless, Frappé is still exceedingly verbose when compared with Haskell-based implementations of FRP. To correct this deficiency, and to provide for static type checking of Frappé programs, we are currently developing a translator that compiles a Haskell-like FRP notation to the corresponding Frappé code. A prototype of this translator already works for some small examples, and the results appear promising.

9 Acknowledgements

Conal Elliott, John Peterson and other anonymous reviewers provided invaluable feedback on early drafts of this paper. Paul Hudak has provided encouragement, support and suggestions as this work has progressed. We are grateful to Valery Trifonov and Zhanyong Wan for discussion and debate on the limits of the implementation presented here.

References

[1] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In C. Chambers, editor, *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices volume 33 number 10, pages 183–200, Vancouver, BC, Oct. 1998.

[2] G. H. (editor). *Java Beans API Specification 1.01.* Sun Microsystems, 1997. Available online at `http://java.sun.com/beans/`.

[3] C. Elliott. Functional Implementations of Continuous Modelled Animation. In *Proceedings of PLILP/ALP '98*. Springer-Verlag, 1998.

[4] C. Elliott. An embedded modeling language approach to interactive 3D and multimedia animation. *IEEE Transactions on Software Engineering*, 25(3):291–308, May/June 1999. Special Section: Domain-Specific Languages (DSL).

[5] C. Elliott. Declarative event-oriented programming. In *Proceedings of the 2nd International Conference on Principles and Practice of Declarative Programming (PPDP 2000)*, September 2000.

[6] C. Elliott and P. Hudak. Functional Reactive Animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.

[7] S. Finne, D. Leijen, E. Meijer, and S. P. Jones. Calling hell from heaven and heaven from hell. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, volume 34.9 of *ACM Sigplan Notices*, pages 114–125, N.Y., Sept. 27–29 1999. ACM Press.

[8] S. Finne, D. Leijen, E. Meijer, and S. P. Jones. *H/Direct*: A binary foreign language interface for haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 153–162. ACM, June 1999.

[9] V. J. Hardy. *Java 2D API Graphics.* P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1999.

[10] P. Hudak. Modular Domain Specific Languages and Tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142, June 1998.

[11] P. Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia.* Cambridge University Press, Cambridge, UK, 2000.

[12] S. P. Jones, E. Meijer, and D. Leijen. Scripting COM components in haskell. In *Fifth International Conference on Software Reuse*, Victoria, British Columbia, June 1998.

[13] S. Microsystems. *Java Core Reflection API and Specification.* Sun Microsystems, 1997. Available online at `http://java.sun.com/products/jdk/1.3/docs/guide/reflection/index.html`.

[14] J. Peterson, C. Elliott, and G. S. Ling. *Fran 1.1 Users Manual.* Dept. of Computer Science, Yale University, June 1998. Included in Fran distribution available at `http://www.research.microsoft.com/ conal/Fran/`.

[15] S. Peyton-Jones and J. H. (eds.). Report on the Programming Language Haskell 98: A non-strict, purely functional language. Technical Report YaleU/DCS/RR-1106, Dept. of Computer Science, Yale University, 1999.

[16] M. Sage. *FranTk: A Declarative GUI System for Haskell*. Dept. of Computer Science, University of Glasgow, 1999. Available at `http://www.haskell.org/FranTk/userman.pdf`.