Reactive Multimedia Documents in a Functional Framework*

Helen Cameron Peter King
Dept. of Computer Science, University of Manitoba
Winnipeg, Manitoba, Canada
{prking,hacamero}@cs.umanitoba.ca

Simon Thompson
Computing Lab, University of Kent
Canterbury, Kent, UK
S.J.Thompson@ukc.ac.uk

Abstract

This paper presents a novel approach to building highly reactive multimedia artifacts. The functional programming language Haskell is used as a basis for the Fran animation library, and this paper shows how Fran and Haskell are used to describe interactive multimedia systems. This is illustrated by two substantial case studies which are also used as examples in a review of related work.

1 Introduction

Reactivity is one of the major challenges for the designer of a multimedia authoring system. Whereas static presentations can be represented by placing objects on a timeline – either by assigning positions to them explicitly or by describing the various constraints which their timings must satisfy – it is by no means so clear how to describe the behaviour of a reactive multimedia artifact.

This paper presents one solution to the reactivity problem. A functional program gives a high-level description of a system; indeed, functional programs are often though of as 'executable specifications'. Here we use a functional language Haskell, [HP99, Tho99], to provide the foundation for constructing a toolkit for multimedia description, building on the Fran system, [PEL97, EH97], for reactive animation. The approach is demonstrated here by the development of two case studies which embody different degrees of interactivity or reactivity.

Background and context. A multimedia document is a structured document, comprising components known as media items. Such media items possess temporal attributes in addition to the spatial attributes of traditional static (paper) documents. A useful comparison of several authoring systems for multimedia documents is to be found in Buchanan and Zellweger [BZ93]. As their comparison indicates, existing authoring tools vary considerably in the type and complexity of the temporal attributes which they permit.

Our prime concern is the investigation of the various forms that reactivity can take in multimedia documents and how to provide authoring support for reactive documents. In using the term "reactivity", we include both *internal* reactivity, that is, reaction between components of the document, and also *external* (user- or system-driven) reactivity. Keramane and Duda [KD96] refer to internal reactivity as "causality", and Buchanan and Zellweger [BZ93] term it "content-based unpredictable behaviour", whereas they refer to external reactivity as "meta-directives".

Our interest in complex documents with rich sets of temporal attributes, constraints and reactions has led us in previous work to consider an authoring tool which is based on an appropriate formalism. Specifically, we have used an interval temporal logic – Mexitl, [BCKT97] – to examine ways in which an author can express complex temporal constraints; once described in the logic, the constraints can be executed and reasoned about. In the longer term we aim to investigate ways in which the toolkit for modelling reactive documents introduced in this paper can be combined with the constraint-oriented style of specification embodied in Mexitl and other authoring formalisms.

A functional programming approach. We have chosen to use an existing programming language as the base for our activities since it provides a variety of (basic and composite) types and structuring mechanisms such as modules. A functional language like Haskell provides in addition other facilities like type classes and functions as 'first-class' data. A functional program concentrates on describing **what** is to be done by a particular pro-

*The work of Peter King and Helen Cameron is supported by individual research grants from NSERC, Canada. Peter King acknowledges the hospitality of DI-LITH, EPFL, Lausanne, Switzerland. Simon Thompson acknowledges support from the British Council and the Royal Society.

gram rather than giving the details of **how** that is to be achieved. This is made abundantly clear by the Fran library, in which details of rendering and the like are hidden from a user's attention, allowing him or her to compose reactive artifacts at a high level of abstraction.

Examples. Although our interest lies in general multimedia documents, not specifically in animation, we nonetheless find it convenient to use the animation provided by Fran for modelling and prototyping, as will be seen in the two contrasting examples that we present. The examples are discussed in more detail in an extended version of the paper available together with the code from [CKT99].

The first of these examples, which we term the "Stage Manager Problem", is a scripting example which presents a theatrical scenario. It illustrates a progression from a strictly temporal specification to an increasingly reactive situation in which stage props may not work, or the actor may forget a line, and the stage manager must "react" to such events in an appropriate fashion.

The second example, a simulation of a user-controlled reactive tape player, illustrates external reactivity. The player allows a user to visualise the behaviour of a real-valued function through time, and to 'rewind', 'pause' and 'fast forward' the presentation. These operations are not built into the authoring tool, rather they are developed from the powerful abstractions which Fran provides. A system like this, in which a presentation exhibits powerful reactive behaviour, provides an interesting challenge for any multimedia authoring system.

The remainder of this paper is organised as follows. Section 2 expands on the notions of multimedia documents and on the various levels and classes of temporal relationship and reactivity which are of interest to us, and presents these notions in the context of our two examples. In order for this paper to stand alone, we provide brief introductions to both Haskell and Fran in Section 3. Sections 4 and 5 present our solutions to the two examples introduced above. Section 6 outlines our plans for further work in this area, and concludes.

2 An Overview of Multimedia Descriptions

In this section, we discuss the types of constraints which exist among media items. In particular we distinguish between temporal constraints and those which involve internal or external reaction. First, we mention several constraint classifications which have already appeared in the literature. Next, we describe three kinds of author-level constraints, and make the distinction between temporal constraints and reactive constraints. Finally, we introduce two representative examples to illustrate this distinction.

2.1 General Requirements

A classification of the temporal relationships of interest in multimedia document authorship is to be found in Erfle [Erf93]. Erfle's classification emphasises relations which are purely temporal in nature, and includes temporal attributes applied to a single media item, as well as several classes of temporal constraints specifying the relative placement and duration of two or more media items. This classification also allows for several types of user interaction with a displayed document, thus incorporating some reactivity.

In [BCKT97], we present a similar but somewhat more general and differently arranged constraint taxonomy, which we use to illustrate the applicability of interval temporal logic to the specification of such documents.

Buchanan and Zellweger [BZ93] also present such a taxonomy, but the scope of their work is broader. They examine all aspects of a multimedia authoring system, rather than simply the document-level temporal constraints.

The specification of temporal relations provides a basis for the development of mechanisms for authoring multimedia documents as well as for the automatic display of multimedia presentations. The expressive power of such systems is directly related to the complexity of inter-item relationships permitted. In [BZ93, Erf93] the authors examine several authoring systems in the light of their constraint taxonomies.

We find it convenient to refer to three levels of authoring tools, based on attributes and constraints of various types. We will refer to these as *timeline-based* (following [BH95]), *temporal relations* and *reactive descriptions*.

Timeline-based: This form of authorship consists of specifying the appropriate positions on a timeline for each media item in a document. In the Amsterdam Hypermedia model [HBv94], for example, items are placed onto a timeline by means of two operators which permit sequential or parallel composition. Clearly, such operators are of great use in presentations where the duration of items is statically known. This particular system includes other powerful features relating to its hypermedia usage.

Temporal relations: A number of authors have built systems based on the well-known Allen relations [All83] or on variations of them [SHT96, WR94]. These relations are binary, but may be composed to relate arbitrary sets of items. Such temporal relations encompass the timeline approach, but in addition provide for indeterminate specification, as will occur with the display of a media item, or with a delay, whose duration is *unpredictable* [BZ93]. Such relationships do not, however, provide for any notion of reactivity, nor for changing the rate of display of a media item, for example, by fast-forwarding.

Reactive descriptions: Reactivity refers to the dependency of some aspect of the behaviour of a media item, including perhaps its display, upon events external to that item. We distinguish between *internal* and *external* reactivity. The former is related to what is also known as *causality* [KD96], and is one instance of *content-based unpredictable behaviour* [BZ93]. Internal reactivity occurs when an event or set of events is based solely upon the internal state of the document during its display. A simple example would occur when the end of the display of one item, with dynamically determined duration, causes the start of the display of a second. More complex examples might arise when dealing with media items of finer *granularity* [BZ93].

The second form of reactivity, external reactivity, is analogous to Buchanan's and Zellweger's concept of *meta-directives*, which refers to reactivity events that are generated by an agent external to the document. Typically, this agent might be the user or reader of the document, who might *pause* a display or *rewind* a media item. This reactivity could also involve system-generated events or conditions.

In our work we are primarily interested in multimedia artifacts exhibiting a high degree of internal and external reactivity. We present a linguistic framework in which the author can express reactivity, and which permits rapid prototyping and proof of concept of a document. Thus, we present an example of what [BH95] refer to as *program-based authoring*. Note that determinate temporal constraints, that is those using media items and delays of predictable durations, may be regarded as a special case of internal reactivity, in which the timeline serves as a pseudo media item whose events trigger actions among the other media items. This observation will be of use in later sections.

2.2 Two Examples

These three classes of media item constraint will now be illustrated by two examples. Executable solutions to each of these will be presented in Fran in later sections. The first example is intended to explore internal reactivity; we have chosen to take up a challenge issued to us during a presentation of some earlier related work [KCBT98] and present the scenario of a stage manager, who must react to events occurring on the stage during the presentation of a play. The second example explores external reactivity, and presents the implementation of a tape player.

Example 1 – The Stage Manager Problem

In this example[1] an actor is speaking lines on the stage. Following the actor's dramatic utterance of line 105, the stage manager is to ring a telephone. At some subsequent point the actor picks up the telephone and delivers line 106, at which point the stage manager must stop ringing the telephone.

We now present two versions of this example, the first is purely temporal while the second illustrates reactivity. In the extended version of the paper we address a third level of the problem which exhibits more complex reactive behaviour.

Level One: As soon as the actor has finished line 105, the stage manager rings the bell five times. Each ring lasts for two seconds and is followed by a three second gap. The actor starts line 106 and picks up the phone exactly at the end of the fifth ring.

Level one corresponds to a timeline specification, and is fully deterministic since the start time and duration of every item, that is, each of the five rings and the intervening periods of silence, are known in advance. This level can equally well be specified using any of the sets of temporal relations referred to above, for example those in [WR94]:

```
line105 before(25) line106
line105 before(0) five_rings
five_rings before(0) line106
line106 cobegin(0) pick_up_phone
```

[1] We are grateful to Bob Morris for suggesting this to us.

Level Two: The actor is now given a measure of artistic licence. She may pick up the phone any time after line 105 has been spoken. However, she can be relied upon to pick up the phone in finite time, and also to start line 106 at the same time as she picks up the phone. The stage manager must stop all ringing activity (be it during a ring or an intervening gap) in response to the actor.

Level Two is indeterminate in its specification, since the media item `phone` is of unpredictable duration, but in any particular performance it is entirely determinate. At first sight it appears that the temporal relationships that exist between the three components, `line105`, `phone_rings` , `line106` and `pick_up_phone` can be expressed in terms of, for example, the relations in [WR94]:

```
line105 before(0) phone
phone before(0) line106
line106 cobegin(0) pick_up_phone
```

However, these temporal relationships do not capture the fact that the utterance of `line106` *causes* the termination of the item `phone_rings`. Neither could we use these relations to express the fact that the action of starting line 106 causes the phone to be picked up. This point is made by Keramane and Duda [KD96], who introduce a set of temporal composition operators which also express causal relations between the intervals on which they operate. In their work binary relations are composed using functional composition, giving in this case:

```
seq(line105,
 follow(loop(seq(ring,3")),
    par-begin(line106,pick_up_phone)))
```

However, this functional composition is subtly different from the English specification. Using functional composition it is not possible to express that fact that `line106` causes *both* the termination of the telephone ringing *and* the pickup of the telephone.

Example 2 – The Tape Player

We also present a second example, in which external reactivity is of the essence. This example is a model of a tape player, complete with buttons which may be accessed by the user to achieve fast-forward, rewind and pause functions of a display. There is no notion of any temporal constraint in the specification of this example; it is entirely reactive in nature. It cannot be specified using temporal relations, nor by compositions of causal relations.

3 Haskell and Fran

Haskell is a standard lazy functional programming language [HP99], available on a variety of platforms. Fran [PEL97, EH97] is a Haskell library providing reactive animation facilities. The work reported here was developed using the Hugs Haskell interpreter [Hug98]. This section provides the briefest of introductions to functional programming, Haskell and Fran; further information about Haskell programming can be found in [Tho99] and introductions to Fran are provided by [Ell98, Tho98].

3.1 Functional Programming

A functional programming language supports **declarative modelling** in that it allows a user to write high-level descriptions of real-world and programming artifacts.

The main abstraction provided by such a language is the **function**, which returns a result dependent upon the values of its arguments. These functions can be higher-order – having functions as their arguments and results – and polymorphic – the same function can be used over many types rather than a single type – as well as permitting their names to be overloaded. The combination of these three features allow functions of great generality to be written.

A simple example of this is the `map` function over lists, which takes a function `f` and list `xs` as arguments and returns the result of applying `f` to each element of `xs`, so that examples include

```
map double [1,2,3] = [2,4,6]        (map.1)
map isEven [1,2,3]
  = [False,True,False]              (map.2)
```

From these it can be seen that function application in Haskell is given by juxtaposition, with the arguments (`double` and `[1,2,3]` in the example (`map.1`)) appearing after the function (here `map`). A single function definition encapsulates this 'mapping' operation for lists of every type, and for every type of function over those lists. The type of `map` is given by

```
map :: (a -> b) -> [a] -> [b]
```

This can be read as saying that `map` takes a function from `a` to `b` (the function type `(a -> b)`) and a list of elements of type `a` (the type `[a]`) and returns a list of `b`. In this type the `a` and `b` are variables, standing for arbitrary types, so that one particular type of `map` is

```
(Int -> Bool) -> [Int] -> [Bool]
```

In the example `(map.2)` above `map` is used at precisely this instance of its type.

Haskell has a general **overloading** mechanism which is exploited in Fran to make programs substantially more readable. To take an example, the symbol `+` is overloaded to mean addition over a variety of types, such as `Int`, `Double` and `Bool`; how is the type of `+` expressed?

```
Num a => a -> a -> a
```

which says that `+` has the type `a -> a -> a` for every type `a` which belongs to the type class `Num`. The types `Int`, `Double` and so forth are called the instances of the class `Num`. Users can make their own definitions of classes and instances, and there are examples in the exposition which follows.

The declarative nature of the language makes functional programs more amenable to formal specification and proof; moreover, and more importantly for multimedia modelling, it means that a high-level – yet executable – description of a multimedia presentation can be provided separately from the details of the way in which the presentation is achieved [EH97].

3.2 Fran

Fran provides two complementary modelling abstractions: **events** which occur at discrete points in time and **behaviours** which evolve in continuous time (and which we call 'continuous behaviours' in the remainder of the paper).

One of Fran's most important features – particularly in the case of multimedia modelling – is that it supports the direct manipulation of continuously evolving objects. Using this feature in combination with discrete events the user is able to effect changes in behaviours, and thus to control these behaviours in response to internal and external stimuli. The rest of this section surveys the foundations of Fran, while in later sections particular aspects of the system are introduced as and when they are used for modelling purposes.

`Behavior a` is the type of *time-dependent* values of type `a`. For example, a graphical animation is a time-dependent image of type `Behavior Image` (or `ImageB`) and a counter which evolves over time is implemented as a `Behavior Int` (or `IntB`). For the purposes of this paper one can think of the type `Behaviour a` as the function type

```
Time -> a
```

but in practice the implementation of behaviours is highly optimised.

In order to react to internal or external events, Fran provides `Event` types, which can model, for instance, user input, timers, and interaction between components of an animation. An `Event a` is a stream of event occurrences, each of which is associated with a time and a value of type `a`. In this paper we take the simplified view that members of `Event a` are lists of the form

$$[\ (a_1,t_1),\ \ldots\ ,\ (a_i,t_i),\ \ldots\ ]$$

where each pair (a_i,t_i) represents an event occurrence and consists of a value from `a`, a_i, and the time t_i at which the occurrence occurs; details of the actual implementation are given in [EH97]. Fran implements the notion of an event handler which gives access to these values, and which also allows the subsequent evolution of a behaviour to depend upon those values, thus giving reactivity.

A third abstraction provided by Fran is the type `User`. Values of this type provide timing information as well as events such as button presses and details of mouse movements and position. An interactive image animation would have the type

```
User -> ImageB
```

indicating that the image produced is a function of the information provided by a particular `User` value, that is by the user interaction which takes place.

Fran provides the modelling abstraction of continuous behaviours; it is implemented by a sampling engine which produces graphical images and so forth in a standard way; more details are to be found in [PEL97].

As can be seen from the discussion here, functions are used to represent behaviour and user-interactivity, and so many of the Fran operations work over functions. An example is

```
over :: ImageB -> ImageB -> ImageB
```

which is used to build a composite (moving) image by placing one such over another. Event-handlers and other composition operations are also of this kind, as we shall see below. Fran contains many features which we do not use in this paper, and it should be stressed that we only address in this introduction some of the features which we use.

4 The Stage Manager Problem

In this section we look at the solution of the stage manager problem introduced in Section 2.2.

Our approach, described in this paper, is to animate a prototype behaviour using moving images, sound and colour as appropriate. For instance, we use colour changes – from green to yellow – to indicate a ringing telephone, which is itself represented by a circle, as indeed is the actor. The act of being picked up, or of embarking on line 106, is indicated by moving the phone or the actor images.

Clearly this is not a fully-fledged presentation of a part of a stage play, but all the important programming aspects of the problem are fully reflected in the animations which we present. Section 6 discusses aspects of real media, and how they fit into an extended Fran model.

4.1 First Level

The Whole Scenario. The scenario here involves an actor and a phone, so at the top level we define

```
scenario :: ImageB
scenario = actor `over` phone
```

Note that a Haskell function such as `over` can be written in infix form by enclosing its name in backquotes. The full code, which is discussed in the following pages, is presented in Figure 1. The `actor` and the `phone` have similar definitions; for now we concentrate on describing the `actor`, and in the conclusion to the section we discuss the `phone` code.

All the implementations contained in this paper can be downloaded from

`http://www.cs.ukc.ac.uk/people/staff/sjt/Fran/`

The Moving Actor. The actor is also an `ImageB`, and is given by

```
actor = moveXY xPos yPos actorGraphic
```

where the function

```
moveXY :: RealB -> RealB -> ImageB -> ImageB
```

```
scenario :: ImageB
scenario = actor `over` phone

--  The system events

startUp, ringOn, ringOff, liftUp :: Event ()
startUp = makeEventRel 0 [2]
ringOn  = makeEventRel 2 [0,5 .. 25]
ringOff = makeEventRel 2 [2,7 .. 27]
liftUp  = makeEventRel 2 [23]

-- The actor

actor :: ImageB
actor = moveXY xPos yPos actorGraphic
        where
        xPos = -0.6
        yPos = stepper (-0.6) actorPos

actorPos :: Event Double
actorPos = startUp  -=> 0
              .|.
              liftUp   -=> 0.6

-- The phone

changeColor :: Event ColorB
changeColor = ringOn  -=> green
                 .|.
                 ringOff -=> yellow

changePos :: Event Double
changePos = liftUp -=> 0.6

phone :: ImageB
phone = moveXY xPos yPos phoneIm
        where
        phoneIm = withColor col phoneGraphic
        xPos = 0.6
        yPos = stepper 0 changePos
        col  = switcher yellow changeColor
```

Figure 1: The first level: the full scenario

is used to position an `ImageB` according to two real-valued behaviours which give its x- and y-coordinates. Since these are time-dependent positions, the effect is to move the `ImageB`, but observe also that this might itself be a moving image, so that motions can be composed in subtle ways.

In our model the actor is represented by a graphic, `actorGraphic`, which is a constant image. Because it is a `ImageB`, the programs we give here would work just as well if it were to be animated. The value `xPos` is a constant behaviour of type `RealB`, defined by

```
xPos = -0.6
```

This is one of the ways that Fran uses overloading, so that the numeric literal `-0.6` is overloaded to mean 'the constant behaviour with value `-0.6`' at type `RealB`; as was noted earlier, this makes programs substantially easier to read, and also mirrors informal mathematical usage which would do exactly this.

Now we need to look at how `yPos` is defined; before we do that, however, we look at the events which initiate the various parts of the presentation.

Events. Fran provides a primitive `Event`-formation operator; on top of this we have built the function `makeEventRel` of type

```
Double -> [Double] -> Event ()
```

which builds an `Event` whose occurrences take place at specific times, given by (double-precision) real numbers. For example, the event `makeEventRel t [`t_1`, ... ,`t_n`]` will occur at times `t+`t_1, ..., `t+`t_n, so that the effect of the function is to make an `Event` whose occurrences happen at specific times – in seconds – relative to some start time.

The return type of this event is `()`, which is a one-element type whose single value is also written `()`; this type is used as a 'place holder' when some value needs to be returned, yet the specific value is unimportant, as is the case here.

Now we are in a position to define the events which govern the scenario – `startUp`, `ringOn`, `ringOff`, `liftUp` each of type `Event ()`. These are given in Figure 1, where we use the Haskell notation for arithmetic lists, so that `[0,5 .. 20]` means `[0,5,10,15,20]`.

Events into Behaviours. We now turn to the question of how occurrences of the events affect the behaviours representing the actor and the phone.

Fran provides a powerful and general mechanism from controlling behaviours by means of events.

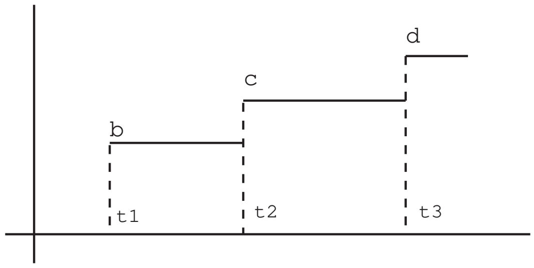


Figure 2: Event into behaviour: building a step function

The details of handlers and so forth are covered in Section 5, but Fran also provides a number of simplifications of the general mechanism, and it is one of these which we look at here.

The position of the `actorGraphic` is determined by a series of constants, so that it is given by a step function, as illustrated in Figure 2. We can describe the step function by the values of the steps and the times at which they start. This is precisely the information contained in the occurrences of an `Event Double`,

$$[\ (b,t_1),(c,t_2),(d,t_3),\ \dots\ ]$$

These values are converted to a real behaviour by the `stepper` function,

```
stepper :: Double -> Event Double -> RealB
```

whose first argument is the starting value of the behaviour.

In the case of the `actorGraphic`, we want to move its `yPos` from -0.6 to 0 at `startUp`, signifying here ending line 105, and from there to 0.6 when the phone is lifted up, `liftUp`.

The event whose values will determine the position of the graphic is defined by

```
actorPos :: Event Double
actorPos
  = startUp  -=> 0
    .|.
    liftUp   -=> 0.6
```

Two things are going on here. The operator `-=>` defines a new event whose occurrences are at the same times, yet which return the value given. The operator `.|.` merges two event streams into one. The overall effect of this is that `actorPos` will have an event occurrence after two seconds with value 0, and twenty three seconds after that it will occur again with value 0.6, and we obtain exactly the behaviour required if we define

```
yPos = stepper (-0.6) actorPos
```

Displaying Behaviours. To display a Fran moving image, such as `scenario`, which has type `ImageB`, the function

```
display :: ImageB -> IO ()
```

is used. This function uses the IO facilities of Haskell to open a window and display the scenario therein.

Conclusion. The account here has covered the `actor` part of the scenario, which is summarised in Figure 1; the phone part is defined in exactly the same way, with changes in behaviour triggered by the events used here. One new aspect introduced in the `phone` part is the function `switcher` which switches between non-constant behaviours in exactly the same way that `stepper` moves between constant behaviours.

The event model will adapt to reactive behaviours as we shall see in the next section. It is also worth commenting that the model is implicitly parallel: in placing one behaviour `over` another the two evolve in parallel, with the events providing the necessary synchronisation between the behaviours.

4.2 Second Level

At the second modelling level, instead of the phone being lifted after exactly five rings, lifting the phone is signalled by an event external to the system, here modelled by a left button press. The model has to be modified as follows.

- Reactivity is added to the scenario by redefining the 'lift up' event,

  ```
  liftUp :: User -> Event ()
  liftUp u = lbp u
  ```

  `lbp` is a reactive event defined by Fran which extracts the left button presses from a given `User` argument `u`.

- User interaction is added. In particular, instead of building an `ImageB` we will develop a function

  ```
  User -> ImageB
  ```

  which we call a **reactive** `ImageB`. Such a function is displayed using the function `displayU` which couples the `User` input to such a reactive image during display.

- The program is modified to pass the `User` argument to every component which is dependent upon the user interaction. The modification for the actor part of the scenario is given in Figure 3. It can be seen that all elements of the program become `User`-dependent, except for the timing events, `startUp`, `ringOn` and `ringOff`, which are obviously independent of any user interaction.

```
scenario :: User -> ImageB

scenario u = (actor u) `over` (phone u)

-- The system events

startUp, ringOn, ringOff :: Event ()
startUp = makeEventRel 0 [2]
ringOn  = makeEventRel 2 [0,5 .. ]
ringOff = makeEventRel 2 [2,7 .. ]

liftUp :: User -> Event ()
liftUp u = lbp u

-- The actor

actor :: User -> ImageB
actor u = moveXY xPos yPos actorGraphic
          where
          xPos = -0.6
          yPos = stepper (-0.6) (actorPos u)

actorPos :: User -> Event Double
actorPos u = startUp  -=> 0
             .|.
             liftUp u   -=> 0.6
```

Figure 3: The `actor` at level 2

- The ringing behaviour is now made unbounded,

  ```
  ringOn  = makeEventRel 2 [0,5 ..]
  ringOff = makeEventRel 2 [2,7 ..]
  ```

  where the list `[0,5 ..]` is infinite but evaluated by need.

- In our model, the phone is coloured green while it is ringing. In the level 1 model, this was controlled by the ringOn event, which was scheduled to happen exactly five times. In the model here, ringOn will happen indefinitely often, and so a mechanism is needed to register when the phone is down, so that ringing will only happen while this is the case. This is done with a **state variable**, which is the Boolean-valued behaviour given by

```
phoneDown :: User -> BoolB
phoneDown u
  = stepper True (liftUp u -=> False)
```

phoneDown is a stepper behaviour which is True initially and is set to False by the phone being lifted. Now it is possible to describe the Event which changes the colour of the phone by means of the switcher.

```
changeColor :: User -> Event ColorB
changeColor u
  = ringOn   -=> green  `whenE` phoneDown u
    .|.
    ringOff  -=> yellow
    .|.
    liftUp u -=> yellow
```

The phone is turned green if there is an occurrence of ringOn at a point when the (reactive) state variable phoneDown has the value True; it is precisely the effect of whenE to extract occurrences in this way.

A ringOff or the phone being lifted cause the phone to turn yellow.

This concludes the changes that need to be made to the scenario to make it fully reactive. The main change is the introduction of the User argument which carries information about the interaction environment to those parts of the system which depend upon it. Beyond this, the system responds to internal events much as it did previously.

4.3 A third level

In the extended version of the paper we look at a third level of the problem, including timeouts and more complex internal reactivity, which involves the generation of events from continuous behaviours. For instance, an event occurrence is generated by the theatre curtains reaching the fully open position. Other examples of this are given in [Tho98].

5 'Playing' Real Behaviours

Our second example is an application which allows a user to see an animated view of a real-valued behaviour, that is a Behavior Double or RealB.

In its simple form, a graph of the behaviour over a sliding two-second window is displayed. In its full implementation, the player is supplemented with controls which allow a user to interact with the playing of a behaviour, and to pause (||), fast forward, (>>) and rewind (<<) the behaviour which is being played. The choice of functions is made by selecting on-screen buttons, as shown in Figure 4, where a sawtooth function is displayed. We have

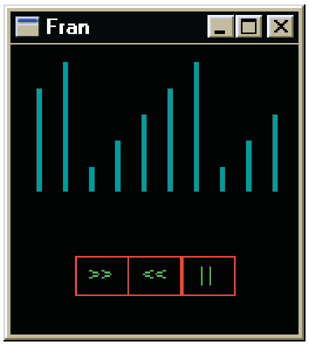


Figure 4: Displaying a sawtooth function

chosen this example for a variety of reasons.

The player is a very useful tool for the developer of Fran systems, since it allows a user to view directly how a behaviour evolves. It can also be seen as an example of how a multimedia system might exhibit a stream such as sound, giving a dynamic picture of the volume of a piece of music, for instance.

The example provides examples of the most common sorts of user interventions into multimedia presentations, namely pausing the presentation or choosing to move forwards or backwards in time. It therefore provides a test of the expressivity of the Fran system.

Building an interface between an object like this and Fran can also provide a prototype for an interface between Fran and an external media player, such as a tape player.

More speculatively, it allows us to experiment with the notion of an 'external behaviour', such as the volume of music, as might be provided by an external device.

In the remainder of this section we look first at the non-reactive player and then at the full device.

5.1 The Non-reactive Player

The non-reactive player – that is the player without controls to pause and so forth – is built in two stages. First, a single varying bar is implemented, and then this is replicated in space and time to give the dynamic appearance of the bars as illustrated in Figure 4.

A Single Varying Bar. The bar of varying size is implemented thus

```
showBar :: RealB -> ImageB
showBar beh
  = polygon [ (point2XY 0    0) ,
              (point2XY 0    beh) ,
              (point2XY 0.02 beh) ,
              (point2XY 0.02 0) ]
```

which uses the standard Fran function `polygon` to build a polygon whose coordinates vary in time. As in the case of the `actor`, we use overloaded numeric literals `0` and `0.02` here; the argument `beh` is a behaviour which will vary, in general.

Of more novelty is the fact that we can then transform these bars and replicate the results in time and space. Given a real parameter `r` we can use that to translate the bar both in space using `moveXY` and in time using `later`, which delays a behavior by a given amount, and is an example of a Fran **time transform**.

```
moveIt :: ImageB -> Double -> ImageB
moveIt image r
  = later r (moveXY (0.9 - r) 0 image)
```

When the parameter `r` has the value `0` then the bar is moved by `0.9` to the right, and not shifted in time; with value `0.6`, the bar is not moved so far to the right and is shifted later in time, so that the larger the parameter the further to the left and the older the bar is.

The bars can be replicated and combined using standard functions from Haskell, underlying the utility of embedding a system like Fran into a programming language, and particularly one like Haskell which has higher-order features.

```
historyDisplay :: RealB -> ImageB
historyDisplay beh
  = foldr1 over (map (moveIt bar) offsets)
    where
    offsets = [0,0.2 .. 2.0]
    bar = showBar beh
```

Recall that the effect of `map` is to apply a function to every member of a list, so that `moveIt bar` is applied to the numbers in the list, giving a list of `ImageB`s. These values are then combined together using the operation `over`; the effect of `foldr1` is to combine (or 'fold together') the values in a list using the function supplied, so that the sum of a list of numbers is given by the function `foldr1 (+)`, for instance.

5.2 The Full Player

In considering the full player we first focus on one particular – but central – aspect of the system: we look at how a pause operation is implemented. Given the details of this, we then explain how the full player is built by a series of abstractions and generalisations based on this single function.

Implementing a Pause Operation. We can explain the pause operation informally, thus. Given a behaviour `beh` we seek to deliver a new behaviour which behaves as follows.

- Until the first `pauseE` event, it behaves like `beh`;
- if that event happens at time `t`, it then has the constant behaviour '`beh` at `t`' until the next `pauseE` event;
- if that event happens at time `t'`, it then behaves as `beh`, but `t' - t` time units `later`, until the next pause ... .

The Fran function `untilB` is used to build a composite behaviour which works like `beh` until an event occurrence. At that point, we construct a new behaviour using

- the time of the event occurrence, `t`, and
- what remains of the original user argument, `u`, after time `t`; this is called `u'`, and is produced by the function `nextUser_`.

We use the mechanism of event handling, `handleE`, to build these values into the ensuing behaviour, which is a time-transformed version of `beh`,

```
beh `timeTransform` (constantB t)
```

where the function `timeTransform` does as expected, and builds a constant behaviour of '`beh` at `t`'.

A similar procedure builds the delayed behaviour after the pause button is pressed again at time `t'`,

```
later (constantB (t' - t)) beh
```

```
behPause :: GBehavior a => User -> a -> a

behPause u beh
  = beh                                                     -- original Behavior beh
    `untilB`                                                -- until
    (nextUser_ pauseE u                                     -- pauseE happens at time t
       `handleE`                                            -- then,
        (\t u' _ ->                                         -- with aged User u',
             (beh `timeTransform` (constantB t))            -- constant beh at t
             `untilB`                                       -- until
             (nextUser_ pauseE u'                           -- pauseE happens again at t'
                `handleE`                                   -- then,
                (\t' u'' _ ->                               -- resume behPause beh from stop
                                                            -- point with aged User u''
                    behPause u'' (later (constantB (t' - t)) beh))))
```

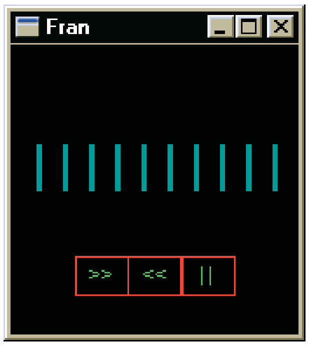
Figure 5: The full program for `behPause`

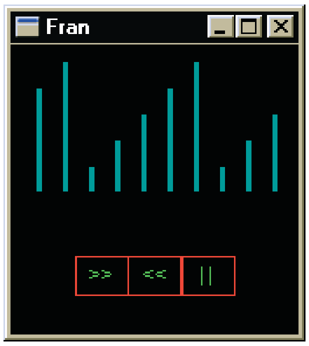
The full code for `behPause` is presented in Figure 5.

Overloading and Polymorphism. Our intention in writing the definition of `behPause` was to give a function which would modify a `RealB`, producing a behaviour which is paused at various points. Many Haskell and Fran functions are overloaded, and indeed we find for the function `behPause` that its type is more general than we had expected, namely,

```
behPause :: GBehavior a => User -> a -> a
```

which can be read as saying that `behPause` can pause many behaviour types, including `ImageB`. We can then implement pause in two ways, giving different graphical results as illustrated here.





If the **behaviour** itself is paused and displayed, then a constant graph results; if the **display** itself is paused, a section of the behaviour is shown.

Building the Full Functionality. It is possible to generalise the function `behPause` in various ways.

- Different events can 'toggle' the change in behaviour, by making this event a parameter of the function.
- The time transforms which are applied after the first and second button presses can themselves be parameters. After the first press we need to describe the transform in terms of `t`; after the second it is described in terms of `t` and `t'`. These descriptions will be **functions** which describe the behaviour in terms of the parameters `t` (and `t'`) and this shows an advantage of the functional approach here.

Using the appropriate pairs of transforms we can implement fast forward,

```
t + 3*(time-t)      time + 2*(t' - t)
```

and rewind,

```
t       time - 3*(t' - t)
```

where `time` is the untransformed time, and these implementations are simply applications of the generalised function.

In a similar way we can combine the three operations into a single function, and also we add the button bar by placing it `over` the player, and using the events that the presses generate to control the player. The details of this – which are somewhat technical – can be found on the Web page supporting this paper.

6 Further Work

In the foregoing sections, we have presented complete working solutions to two contrasting examples of multimedia documents. Our intention in selecting these two examples (beyond simply responding to a challenge) was to explore the various ways in which both internal and external reactivity can affect the display of media items in multimedia documents, and to explore how such situations may be modelled. Most existing authoring systems concentrate on features for timeline specifications, and while some provide features for relationships among media items, for the most part these are limited to temporal relationships, as our discussion in Section 2 suggests.

While we certainly do not propose Fran as a complete authoring system, we do believe that the use of a programming language as the basis of an authoring system has a number of advantages, arising from the expressive power of a general purpose language. At the same time, for the reasons which we stated at the outset, we see equal advantage in the use of a formal system. Functional languages may be regarded as formal systems in this sense, as they readily admit of proof systems. Thus, we regard Fran in this sense as a specification, rather than a programming, language because

- Fran consists entirely of code in a functional language, and
- the features provided by Fran allow us to express the types of reactivity of interest to us.

As suggested earlier, because Fran is designed for animation rather than more general multimedia, Fran affords us a convenient prototyping mechanism. Fran's design also suggests useful lines of further research as we develop our current system towards a fuller authoring tool.

We see a number of directions in which this future work will go. All share the common theme of using Fran and Haskell to construct an authoring tool. One question we are currently investigating concerns multimedia items themselves. They do not appear to have a single, obvious counterpart in Fran. We are considering, therefore, questions of how media items are to be represented, and how a hierarchy of object classes for such items, with its obvious attractions in an authoring system, might be represented.

Two important questions, both related to this class issue, concern first, the composition of items to form more complex items, and second, the expression in Fran of other forms of synchronisation. In [KCBT98], we present a complete synchronisation taxonomy for multimedia documents; we are actively considering what the counterpart is in the reactive world and how it can be expressed in our functional paradigm. A more general related issue here is the interaction between non-determinism and reactivity.

Finally, we are keenly interested in seeing how Fran may be used to monitor and control external behaviours, just as we have used it above for internal behaviours. This research involves adding appropriate events for controlling and signalling such external objects. As a specific instance of this, we intend to investigate how one might annotate MIDI sound files with control information, which might trigger or synchronise with other information in the Fran system.

References

[All83] J. F. Allen. Maintaining knowledge about temporal intervals. *Comm. ACM*, 26(11), 1983.

[BCKT97] H. Bowman, H. Cameron, P. King, and S. Thompson. Specification and prototyping of structured multimedia documents using interval temporal logic. In *Internat. Conf. on Temporal Logic*, 1997.

[BH95] D. C. A. Bulterman and L. Hardman. Multimedia authoring tools. In J. van Leeuwen, editor, *Computer Science Today; Recent Trends and De-*

velopments, volume 1000 of *LNCS*. Springer-Verlag, 1995.

[BZ93] M. C. Buchanan and P. T. Zellweger. Automatic temporal layout mechanisms. In *ACM Multimedia 93*. ACM Press, 1993.

[CKT99] H. Cameron, P. King, and S. Thompson. Reactive multimedia documents in a functional framework (extended version). Technical report, Computing Lab, University of Kent, 1999. Available via: `www.cs.ukc.ac.uk/people/-staff/sjt/Fran`.

[EH97] C. Elliott and P. Hudak. Functional Reactive Animation. In *ICFP97*. ACM Press, 1997.

[Ell98] C. Elliott. Composing reactive animations. *Dr Dobb's Journal*, July 1998.

[Erf93] R. Erfle. Specification of temporal constraints in multimedia documents using HyTime. *Elec. Pub.*, 6(4), 1993.

[HBv94] L. Hardman, D. C. A. Bulterman, and G. van Rossum. The Amsterdam hypermedia model: Adding time and context to the Dexter model. *Comm. ACM*, 37(2), 1994.

[HP99] J. Hughes and S. Peyton Jones. *Report on the Programming Language Haskell 98*. `www.haskell.org/report`, 1999.

[Hug98] The Hugs System. Available from `www.haskell.org/hugs`, 1998.

[KCBT98] P. King, H. Cameron, H. Bowman, and S. Thompson. Synchronization in multimedia. In R. D. Hersch, J. André, and H. Brown, editors, *EP 98*, number 1375 in LNCS. Springer, 1998.

[KD96] C. Keramane and A. Duda. *Interval Expressions* – a functional model for interactive dynamic multimedia presentations. In *MULTIMEDIA '96*. IEEE, 1996.

[PEL97] J. Peterson, C. Elliott, and G. S. Ling. Fran Users' Manual. Available from `www.haskell.org/fran`, 1997.

[SHT96] T. K. Shih, L.-J. Hwang, and J.-Y. Tsai. Formal model of temporal properties underlying multimedia presentations. In J.P. Courtiat et al., editors, *Multimedia Modeling: Towards the Information Superhighway*. World Scientific Publishing, 1996.

[Tho98] S. Thompson. A functional reactive animation of a lift in Fran. Technical report, Computing Lab. University of Kent, 1998. Available via: `www.cs.ukc.ac.uk/people/-staff/sjt/pubs.html`.

[Tho99] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, second edition, 1999.

[WR94] T. Wahl and K. Rothermel. Representing time in multimedia systems. In *Proc. of the Int. Conf. on Multimedia Computing and Systems*, 1994.