Modelling Reactive Multimedia: Events and Behaviours*

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

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

Abstract

This paper explores the idea of reactivity in multimedia, and proposes systems which can react to continuously-evolving behaviours as well as to more traditional discrete events. The idea is presented in a scenario as well as in a number of small programming examples.

The illustrative examples are written in the Fran system. Fran provides a high-level programming model for animations, built in the Haskell functional programming language. Whilst we use Fran for illustration – and indeed we argue that the functional paradigm is a natural choice of host for such a system – we should stress that the notion of external behaviours within multimedia is independent of the programming environment chosen and could be incorporated in other systems such as SMIL.

1 Introduction

Existing multimedia authoring tools, such as SMIL [SMI98], MacroMedia Director, the Amsterdam hypermedia system [HBv94] and Madeus [JLSI97], create multimedia artifacts which are largely event-driven. In such artifacts, media items may be initiated and terminated, and may react with each other according to events arising from the elapse of time, user interaction, and so forth.

In this paper we propose that multimedia artifacts can be enhanced by adding the notion of a continuously-evolving behaviour. Of course, multimedia artifacts currently contain sound, video and so forth: however, we propose to allow a multimedia system to react to the *content* of such a continuous behaviour. To clarify our ideas we first look at precisely how continuous behaviours such as sound are

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

incorporated into – say – a web page, and then we explain our proposal in more detail.

A piece of audio can be played, repeated and fast forwarded, for instance. These actions happen because the multimedia artifact causes certain events (or signals) to occur. Similarly, the multimedia artifact may react to the audio player signalling termination or some other event. The key to implementing such a scenario – and similar scenarios involving other media such as video – is an event model.

Events can be used to achieve temporal synchronisations of the sort seen in SMIL as well as to control various actions performed externally to the artifact itself. The only way an audio player can affect the multimedia artifact is to produce events. In particular, the content of the piece of audio played has no direct effect on the artifact.

The key to our proposal is to allow the content of an external behaviour to affect the behaviour of the artifact. In other words, we propose an enhanced model of reactivity in which a system can react to

- discrete *events*, such as an audio player reaching the end of a track; and
- continuously-evolving *behaviours* such as the volume of an audio track.

The body of the paper has three parts. First in Section 2 there is an informal discussion of the notion of reacting to an external behaviour, illustrated by a number of smaller examples. Section 3 presents a larger-scale scenario of an on-line encyclopedia entry. The discussions here also show that the model we propose allows us to express things which cannot be rendered in current event-based systems. In particular, we introduce the dynamic link, which is a hyperlink whose destination depends on one or more external behaviours, and which may thus change over time.

This discussion is followed in Section 4 by an introduction to the Fran model for multimedia, a functional approach, which is itself illustrated in Section 5 by the scenario of a short piece of a theatrical production. As well as illustrating our approach, the scenario shows that we are able to express the sorts of temporal synchronisation seen in current systems like SMIL. This part of the paper can also be seen as evidence of the power and elegance of the functional approach to multimedia description.

In Section 6 we return to the encyclopedia entry referred to earlier and present several components of this scenario implemented in Fran. The components give examples of various sorts of reaction to external behaviours. We draw some conclusions in Section 7.

2 Events and behaviours

The traditional view of a reactive system sees it as reacting to occurrences of discrete events: each time the 'left button press' event occurs, a certain action takes place, for example. The reactivity can be two-way, so that the system can also initiate event occurrences like 'start to play an audio presentation'. Events can also be used inside a system to achieve coordination between its various components. For example,

- components can be placed at positions on a timeline by defining events which happen at particular times;
- components can be sequenced (or 'chopped') by using the event that signals termination of the first component to start the second.

An example of this sort of coordination – which is to be seen in languages like SMIL – is given in Section 5. This serves also to give an introduction to a functional approach to multimedia authoring using the Fran system.

We propose in this paper that systems should be able to react to continuous behaviours, that is, behaviours which evolve continuously in time. Such behaviours can come in many forms, including audio and video feeds, as well as information from sensors or behaviours exported by other reactive systems.

Let us consider what sorts of reactions to behaviours are possible. The prototypical example is to generate an event (occurrence) on the basis of certain properties holding of a behaviour or behaviours. A number of examples follow.

- We might detect the turning points in a real-valued behaviour: when a stock market average starts to go down, it might be time to sell stocks from a particular portfolio.
- The turning point might be detected in a behaviour derived from an external behaviour. For instance, from an audio feed – an external behaviour – we can derive the volume of the audio signal; to reiterate, this volume information is not an explicit part of the external behaviour. We might then detect *crescendi* or other sorts of musical features from this derived volume information.
- We can detect when a real-valued behaviour passes a certain threshold: when a temperature exceeds 25 Celsius, for instance. More significantly, we can detect when one of a pair of behaviours exceeds the other: when does the overall brightness of video feed A exceed that of feed B? At this point, we might wish to switch between feeds. The important point in this example is that the property does not belong to a single behaviour, but is derived from more than one behaviour.

These and other examples show how external behaviours can be used to control a multimedia artifact. Such control cannot, in general, be achieved by using external events to signal such properties. Such an 'external events' model would have two substantial drawbacks.

- This model requires the source of the behaviour to know all properties which might be recognised, and also to include events which signal each of these properties. This requirement is impractical and breaks all principles of modular design. The client of the service needs to recognise the particular behaviour, and so the client should be responsible for finding the behaviour.

- A property might not belong to a single behaviour, and so the event that signals the property cannot be associated with a single behaviour. The example of the two video feeds above illustrates this point.

In the light of these two observations, we conclude that the model of 'external behaviours' properly extends what can be done in a purely event-driven system. In the next section we look at the larger-scale case study of an entry on vulcanology in an on-line encyclopedia; this case study involves reactivity to continuous behaviours.

3 A scenario: on-line encyclopedia entry on vulcanonlogy

In this example, we consider an entry in an on-line encyclopedia on volcanoes. The entry explains the various types of volcano and how they are formed. It also incorporates current information from volcano observation stations around the world. This information is used to give current descriptions of volcanoes around the world and to describe how vulcanologists study volcanoes.

This example will illustrate several aspects of external behaviours, including

- the display of an external behaviour;
- events determined by one or more external behaviours;
- dynamic links in which the destination of the link depends on a continuous external behaviour; and
- behaviours constructed from several independent continuous external behaviours.

In the subsequent sections, we elaborate on these examples; a Fran model of these examples will be found in Section 6.

3.1 Displaying an External Behaviour

A description of how vulcanologists work contains explanations of the tools they use and how they use these tools. One important tool is the seismometer, which measures the vibration of the earth in its immediate area. The on-line encyclopedia entry for seismometers would explain their function and might also display the real-time output of a seismometer, a continuous external behaviour, at a particular volcano observation station of interest to the reader. The behaviour may be displayed as a stationary or a moving bar-graph. The user may pause such a behaviour in order to study the data more carefully.

3.2 Events

The user may wish to recognise maximum and minimum points in vibration values, or to recognise points at which the vibrations at Site A become less than that at Site B. In both cases these points are events defined from external behaviours.

3.3 Dynamic Links: A Simple Example

From a display of the output of a seismometer at a volcano observation station, there may be a dynamic hyperlink which brings up either

- a display from a video camera at the station, if the seismic activity of the magma chamber is great enough (there is something interesting to see above-ground such as clouds of steam), or
- another article explaining how vulcanologists can map the size and location of the magma chamber using a ring of seismometers around the volcano, if the seismic activity of the magma chamber is low (there is nothing interesting to see above-ground).

Thus the destination of this dynamic link – the camera display or the article – is computed from a continuous external behaviour. This notion is distinctly more powerful than the 'indirect' links described in [PBCH98].

3.4 Dynamic Links: A More Complex Example

In a section of the encyclopedia entry about the active volcanoes around the world, each volcano has its own "slide" with a picture of the volcano, a map showing its location and other information about the volcano. The slides are displayed, one at a time, from the most active to the least active (based on the seismic activity recorded at each site in the past 24 hours). When finished reading a slide, the user can choose "more" or "less", which take him or her to the slide corresponding to the next more-active volcano or the next less-active volcano, respectively. The links between slides are dynamic and where they point depends on many continuous external behaviours (feeds from seismometers at the volcanoes themselves). These links are on the slides and are, of course, distinct from the usual "back" and "forward" buttons found in the browser.

3.5 Constructed Behaviour

For a currently active volcano, the entry displays an animation which shows what is going on both above and below ground. Pressure build-up in the magma chamber may be displayed by changing and intensifying the colours used to represent the chamber in the animation. The pressure cannot be measured directly, but it can be computed from the output of various independent measuring devices, such as seismometers, tiltmeters and thermometers.

We now move to an examination of how some of these ideas are implemented. First we introduce Haskell and Fran, and then go through a sequence of examples of multimedia artifacts, the later ones incorporating external behaviours. Then, in Section 6, we show how to model aspects of the vulcanology scenario in Fran.

4 An Introduction to 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 on Windows platforms. The work reported here was developed using the Hugs interpreter for Haskell [Hug99]. This section provides the briefest of introductions to functional programming, Haskell and Fran; further information about Haskell programming can be found in [Tho99b] and introductions to Fran are provided by [Ell98a, Tho99a].

4.1 Functional Programming and Haskell

A functional programming language supports high-level declarative modelling by allowing a user to write high-level descriptions of real-world and programming artifacts. A functional program consists of a collection of definitions, so that in describing a theatrical scenario, for instance, one would write descriptions of the various players and props. These descriptions can depend on each other, since in general these components will interact with each other.

The main abstraction provided by such a language is the function, which returns a result dependent upon the values of its arguments. Functions are 'first class citizens' and can be handled just as other sorts of data. Functions can therefore be used to model real-world artifacts and this is of great value in modelling multimedia. For instance, the motion of an animated component or any other kind of continuous behaviour is natuarally represented as a function; the 'first class' nature of functions makes it possible to write functions which combine behaviours together. One example is a temporal 'chop' which sequences one behaviour after another. A second example is the comparison of two video feeds mentioned in Section 2.

Functions which have functions as their arguments and results are called higher-order. Functions can also be polymorphic – with the same function definition being used over many types – and 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 and a list as arguments and returns the result of applying the function to each element of the list. Examples include

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

From these examples 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 the type declaration

```
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. In Section 6, `map` will be used in the construction of a moving display of seismometer output.

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`. In our multimedia applications, overloading provides a mechanism for re-using components in other contexts.

A binary function such as `map` can be made infix by surrounding it with back-quotes, as in the application

```
double ‘map‘ [1,2,3]
```

To summarise, observe that the language is declarative in the sense that a definition of a function describes what the behaviour of the function is rather than how that behaviour is achieved. First of all, this makes functional programs more amenable to formal specification and proof; secondly, and more importantly for multimedia modelling, a high-level – yet executable – description of a multimedia presentation can be provided separately from the details of how the presentation is achieved [Ell98b].

4.2 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 [Ell98b].

In order to react to internal or external events of various sorts, 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`. Events can be thought of as 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 [Ell98b]. 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].

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 in our study of the foundations of reactive multimedia.

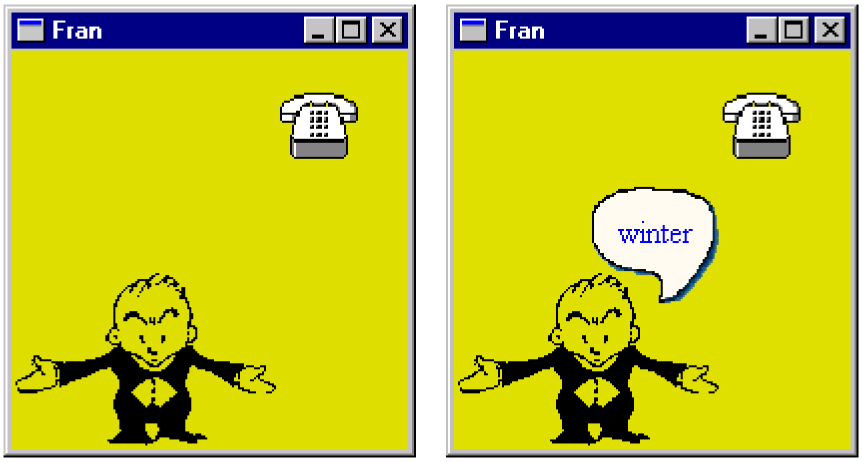

Figure 1: The scene begins ... and the actor speaks the line *'Now is the winter of our discontent'*.

5 Case study: a theatrical scenario

This section of the paper covers the example of making a presentation of a short theatrical scene:

> An actor is on a stage which is bare apart from a telephone.The actor speaks a line (Figure 1) and then the phone starts to ring, indicated by it flashing red and white. The actor moves upstage, picks up the phone and answers it (Figure 2).

The example is used to introduce the Fran approach to modelling and also to show how we handle temporal placement, sequencing and other authoring constructs found in systems such as SMIL, Macromedia Director and others. Moreover, the example contains some animation, which goes beyond what is currently possible in SMIL and others.

In the remainder of this section we explain the main features of the scenario, explaining *inter alia* features of Fran and of our modelling approach. We also comment on different possible design choices where that is appropriate.

The top level

At the top level the system is given by

```
actor `over` phone
```

where both `actor` and `phone` are animated images, of type `ImageB`. The function

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

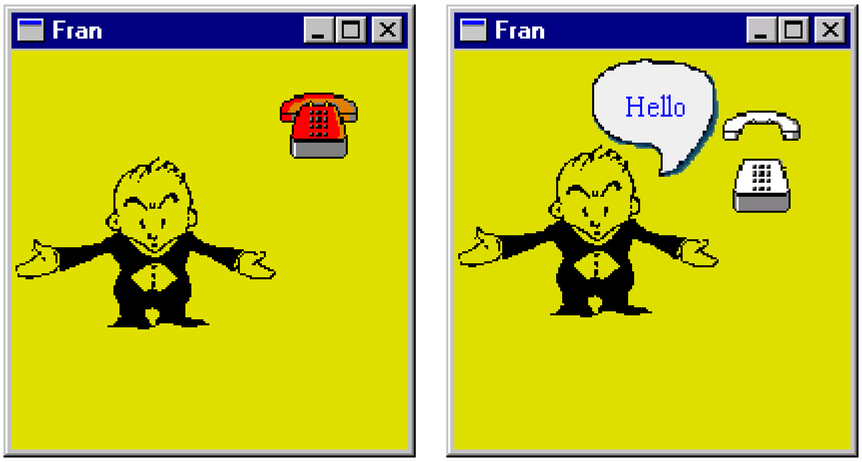

Figure 2: The actor moves to the ringing phone . . . and answers it.

is used to place one moving image over another. These two behaviours, `actor` and `phone`, are described in the same program; they will both depend upon other objects. Thus the collection of objects in a presentation will be interdependent. From a different point of view, the components evolve concurrently, but this concurrency is *implicit*; the components simply depend upon each other as they evolve in a common frame.

The discussion now will concentrate on the `actor`; full details of the `phone`, which is programmed in a similar way, are included in the full code for the example which can be found at

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

The actor

The behaviour of the actor is given by

```
actor = move actorPath actorImage
```

The motion of the actor is given by

```
actorPath :: Vector2B
```

a vector behaviour which describes the position of the actor throughout the scene. The function `move`,

```
move :: Vector2B -> ImageB -> ImageB
```

produces a moving version of the actor from separate descriptions of motion and the animated image. This separation of concerns allows the modeller to work at an appropriate level of abstraction, rather than having to think ‘frame by frame’ about positioning static images. The appearance of the actor is described by

```
actorImage :: ImageB

actorImage = man
              `over`
             wordsImage
              `over`
             bubbleImage
```

where the `man` is a bitmap, and the actor's words are shown one at a time by `wordsImage` which plays in front of a speech bubble, `bubbleImage`, which is only visible while the actor is speaking.

The actor speaks a second line after moving upstage. We therefore group the images which represent his spoken words – the `wordsImage` and the `bubbleImage` – with the image of the actor itself. When the actor is silent the word and bubble images are null.

Events

Thus far we have explained the top level of the system, and its various components. Each of the components evolves in time, in coordination with other components. Events are used to mediate this timing and coordination. Primitives of Fran are used in defining the function

```
makeEvent :: [Time] -> Event ()
```

In Fran, temporal placement, be it absolute or relative, is achieved by creating an event corresponding to the desired time(s), as in

```
startMove = makeEvent [10]
```

This event, which occurs (uniquely) at time 10 from the start of the scene, is the absolute time signalling the start of the actor's motion. This corresponds to placing objects at a particular place on a timeline.

In controlling the ringing phone we can build an event whose timing is relative to `startMove`:

```
ringOn = makeEventRelEvent startMove [2,4 .. 8]
```

using the function

```
makeEventRelEvent :: Event () -> [Time] -> Event ()
```

which is analogous to `makeEvent` except that the event placement is tied to another event. From an author's point of view this relative placement is advantageous if the scenario is likely to be modified; if, for instance, the actor is to start moving earlier or later, synchronisation with the phone will remain.

The event causing motion to stop could be defined by supplying an explicit value to `makeEvent`; it is instead defined from a behaviour. We now look at the general mechanism which makes the definition of such causal events possible.

Behaviours into events

Behaviours and events are linked in two ways by the Fran model. We shall see in the next section how events can be used in defining behaviours; here we do the opposite and show how events can be defined from behaviours.

The events are intended to signal that something has happened in a behaviour: maybe a real-valued behaviour has become positive, or one behaviour has become greater than another, say.

A property will be represented by a Boolean-valued behaviour, whose type is `BoolB`. Fran provides the primitive `predicate` function,

```
predicate :: BoolB -> User -> Event ()
```

so that an occurrence of the event `predicate boolB u` is generated each time that `boolB` becomes `True`.[1]

Boolean operators such as `>=` can be *lifted* to work over real-valued behaviours, and are written with a suffix, thus: `>=*`. The effect of applying such an operator to two `RealB`s is to produce a `BoolB`.

For instance, if `yPos` is a real-valued behaviour, then

```
predicate (yPos >=* yEnd) u
```

will generate an event when the value of the behaviour `yPos` becomes greater than or equal to `yEnd`. This definition is of the event which stops the actor's motion:

```
endMove :: Event ()
endMove = predicate (yPos >=* yEnd) u
```

Timeline-based formalisms do not provide a construction corresponding to `predicate`, as such formalisms do not contain continuous behaviours. Using `predicate` is advantageous from the programmer's point of view: it gives flexibility in the design of the system. `endMove` permits the actor to move faster or positions to be changed without being redefined.

Events into behaviours

We now turn to the question of how occurrences of events can affect the behaviour of the actor and the phone.

Fran provides a powerful and general mechanism for controlling behaviours by means of events. From the point of view of a multimedia author the central construct is a function to switch from one behaviour to another. We use

```
untilEv :: GBehavior b => b -> Event () -> b -> b
```

which can be used over any 'generalized behaviour' type, represented by the type class `GBehavior`. The effect of the behaviour

[1] The `User` argument is provided to supply timing information and can be ignored for the purposes of this exposition.

```
beh1 ‘untilE‘ ev $
beh2
```

is to behave like `beh1` until the event `ev` first occurs; after that the behaviour is `beh2`.[2]

The position of the actor can now be defined. The actor is in the starting position until he starts to move; he then moves until he reaches his final position, where he remains indefinitely. This description is almost a transliteration of the definition which is given now.

```
actorPath :: Vector2B
actorPath
   = startPos ‘untilEv‘ startMove $
     motion   ‘untilEv‘ endMove   $
     endPos
```

The particular motion is upwards, and so only affects the y-coordinate of the actor. Once motion starts, the y-position increases at a constant rate.

```
motion = vector2XY xPos yPos
yPos   = yStart + atRateU 0.05
```

Observe here that the operator `+` and indeed the numeric literal `0.05` are overloaded. Addition is in a lifted form, adding the values of two behaviours pointwise; the literal represents a constant behaviour.

Motion can be programmed in various ways. We could, for example, give an explict formula for the position. However, the rate-based animation here has the advantage of simplicity: constant motion is represented by a constant in the program.

The other components

The remaining components of the scenario – namely the phone, the words spoken and the speech bubble – are defined using exactly the same means as the actor. Behaviours are built by switching between simple behaviours using `untilEv`, and the main control events for these components are `startMove` and `endMove`. Using these events for timing ensures coordination between components.

6 Displaying external behaviours

After looking at the generalities of the functional multimedia model, we now use it to illustrate our ideas about reacting to external behaviours in a multimedia setting. This section develops a sequence of examples corresponding to scenarios

[2] It is a common Haskell idiom to use the infix function application operator `$` as an alternative to parethesisation, particularly when parentheses would be deeply nested.

introduced in Sections 2 and 3, to do with the display of external behaviours and how their properties can be detected. We include displays of a single behaviour, of its evolution over time, and how properties of a behaviour can be detected. We then look at properties of multiple behaviours, before explaining how a powerful 'behaviour player' can be written.

The examples are included for two reasons. First, the methods used are applicable in general: the examples given here could not be written in traditional multimedia authoring systems yet they are both simple to write and natural to include in a multimedia setting. Second, they show that the Fran model of functional reactive animation is powerful and natural; the examples exploit polymorphism, functions-as-data and overloading.

6.1 Displaying a numerical behaviour, part 1

The simplest view of a behaviour is to show it by an animated 'bar', whose height varies with the magnitude of the behaviour. For example, the output of a seismometer could be displayed in this way. The bar 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.

6.2 Displaying a numerical behaviour, part 2

The display of seismometer output, for example, is much more useful if it gives some of the history of the behaviour over a period, as shown in Figure 3. This we do by transforming the bars defined above and replicating them 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)
    where
    r' = constantB r
```

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

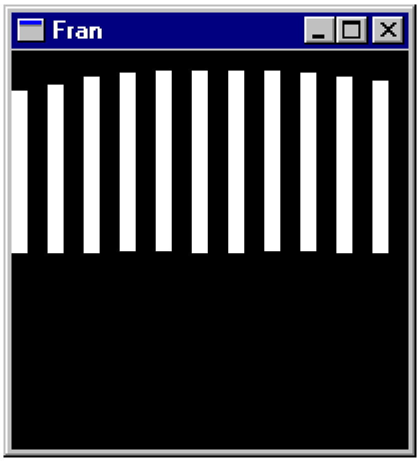

Figure 3: Displaying the history of a behaviour.

is shifted later in time, so that the larger the parameter the further to the left and the older the bar is. (Note that `constantB` is used here to turn a value into a constant behaviour with that value.)

Informally, the behaviour moves from right to left, or alternatively the display moves as it it were a window on the behaviour moving from left to right.

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) [0,0.2 .. 2.0])
    where
    bar = showBar beh
```

To understand this code one should 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 library function `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 (+)`. The effect of the fold in `historyDisplay` is to place all the layers of the display on top of each other, giving an animated histogram as illustrated.

6.3 Recognising maxima and minima

The player in Section 6.2 displays a real-valued behaviour over a time window. Here we show how to augment the display by colouring the graph to show when

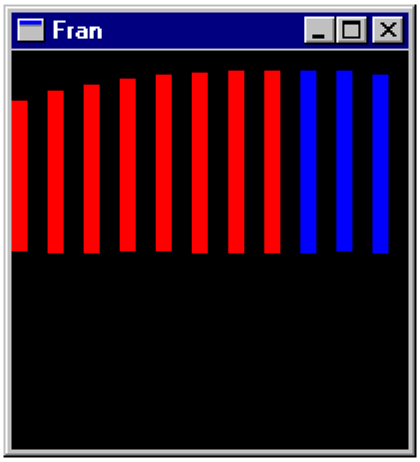

Figure 4: Indicating increasing and decreasing behaviour.

it has passed a maximum or minimum; this is shown in Figure 4. The example illustrates two things:

- an `Event` being generated from a `Behavior`, and
- the modularity encouraged by a functional approach.

In the animated histogram given in Section 6.2 the columns (or 'bars') are uncoloured; in this section they are coloured to indicate the behaviour of the function: they are red after passing a minimum of the function, and blue after passing a maximum. In this fashion, the maximum and minimum values detected by a seismometer can be displayed in an encyclopedia entry.

Renaming the bars of Section 6.2 as `showBar'` the new definition of coloured bars given by

```
showBar beh u
  = withColor (changeCol beh u) (showBar' beh)
```

With this redefinition of the bars, the original definition of `historyDisplay` can still be used.

It remains to explain the new definition of `showBar`. It uses the primitive function

```
withColor :: ColorB -> ImageB -> ImageB
```

which changes the colour of an image. The arguments to this function are behaviours: an evolving colour and a moving image; the result is to colour the moving image with the changing colour. The changing colour is given by the function `changeCol` which we examine now.

As we mentioned in Section 5, the `predicate` mechanism is used to detect properties of behaviours, and we define

```
spotMax :: RealB -> BoolB

spotMax beh
  = (beh <* beh1) &&* (beh2 <* beh1)
    where
    beh1 = later 0.2 beh
    beh2 = later 0.4 beh

maxEvent :: RealB -> User -> Event ()
maxEvent beh u = predicate (spotMax beh) u
```

which samples the behaviour at the same frequency as the display, and detects a situation where the function reaches a maximum value, `beh1`. Minima are spotted in a similar way.

These events signalling maxima and minima now have to be turned into behaviours. On the occurrence of `maxEvent` the colour should be turned to `red`; on `minEvent` it is turned to `blue`. This colour change is effected by the function `switcher`, which works in a similar way to `untilEv`:

```
changeCol :: RealB -> User -> ColorB
changeCol beh u
  = white `switcher`
      (minEvent beh u -=> red
       .|.
       maxEvent beh u -=> blue)
```

The colour will initially be white; any occurrence of the minimum (maximum) event causes the colour to become red (blue).

An illustration of the augmented display is given in Figure 4.

6.4 Interreacting behaviours

In a similar way to the last example, we may detect properties of more than one behaviour. In particular, one can detect the situation where one behaviour becomes larger than another, such as when one seismometer reading exceeds a second, and change colour accordingly:

```
compareColour :: RealB -> RealB -> User -> ColorB

compareColour f g u
  = switcher white changer
    where
  = white `switcher`
      (changeEv f g u -=> red
       .|.
       changeEv g f u -=> blue)

    changeEv f g u = predicate (f >* g) u
```

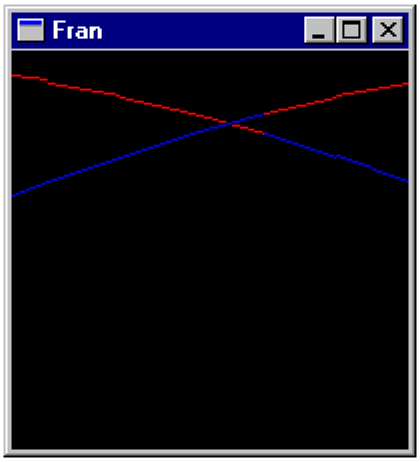

Figure 5: Displaying two behaviours and indicating the greater.

The definition is similar to `changeCol`: all that changes is the property recognised, `(f >* g)`, and the fact that the comparision is parameterised over the two functions `f` and `g`.

This example is illustrated in Figure 5.

6.5 Dynamic Links

We now outline how we model dynamically-computed hyperlinks, which were illustrated in Sections 3.3 and 3.4.

In the first of our two examples, the link destination depends upon a single value computed from an external behaviour. In our model, this value may simply be computed when the reader presses the link button on the page. Alternatively, the link button may be made to correspond to a behaviour defined using the `switcher` function, much as in the illustration of maxima and minima. In this case, the value of the link would alternate in time among its various possible destinations according to changes in the value in question. In particular, the link would already correspond to the appropriate destination at the time when the reader activates it.

The second example requires the destination to be computed from several external behaviours. Consider the hyperlink "less" which navigates to the next less-active volcano. This link corresponds to a behaviour, defined as a function over all seismometer behaviours, which performs a computation to compute the greatest seismometer reading which is not greater than the reading from the current seismometer. Since functions are first-class objects, this behaviour is no more complex to write than the similar computation over, say, a set of integers.

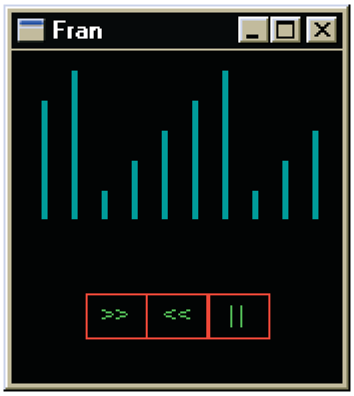

Figure 6: A player for behaviours

6.6 A behaviour player

This section introduces a 'player' for behaviours, built by analogy with a tape or audio player and illustrated in Figure 6. The user 'plays' a behaviour using the display of Section 6.2 with the added functionality of being able to pause, rewind or fast forward the display. We should emphasise that this player is not a plug-in; it is written entirely within the Fran system.

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.

6.7 Implementing a Pause Operation

We can explain the pause operation informally, thus. The event `pauseE` is used to toggle the pause on and off. Given a behaviour `beh` we seek to deliver a new behaviour which behaves as follows.

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

The remainder of this section is somewhat more technical and can be omitted on first reading.

The 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

```
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 7: The full program for behPause

- 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
```

The full code for behPause is presented in Figure 7.

6.8 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
```

Figure 8: Pausing a sawtooth behaviour and its display

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 in Figure 8. If the behaviour itself is paused and displayed, then a constant graph results; if the display is paused, a section of the behaviour is shown.

6.9 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 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:

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

7 Conclusions

This paper has introduced the idea of behaviours external to a multimedia artifact, and how these continuously-evolving behaviours can affect the behaviour of the artifact itself. This notion is new to multimedia development, and introduces a new paradigm for designing multimedia systems which embraces dynamic linking, interrelating different information sources and so forth.

After giving an informal description of various kinds of external reactivity, we presented the Fran system, which is a library extending the Haskell functional programming language. We then used Fran to model some of the scenarios for reactivity from the example of an on-line encyclopedia.

We also pointed out that the functional approach promotes a high-level declarative model for reactive multimedia. The description of an artifact is thus close to the level at which an author would describe it informally whilst at the same time remaining an executable program.

Acknowledgement

We are grateful to Lynne Romuld of the University of Manitoba for her help in converting Windows bitmaps to a printable form.

References

[EH97] Conal Elliott and Paul Hudak. Functional Reactive Animation. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP97)*. ACM Press, 1997.

[Ell98a] Conal Elliott. Composing reactive animations. *Dr Dobb's Journal*, July 1998.

[Ell98b] Conal Elliott. Functional implementations of continuous modeled animation. In *PLILP/ALP98*. Springer Verlag, 1998.

[HBv94] L. Hardman, D.C.A. Bulterman, and G. van Rossum. The Amsterdam Hypermedia Model: Adding time and context to the Dexter Model. *Communications of the ACM*, 37, 1994.

[HP99] John Hughes and Simon Peyton Jones, editors. *Report on the Programming Language Haskell 98*. `http://www.haskell.org/report/`, 1999.

[Hug99] The Hugs 98 System. `http://www.haskell.org/hugs/`, 1999.

[JLSI97] M. Jourdan, N. Layaïda, and L. Sabry-Ismail. Time Representation and Management in MADEUS: an authoring environment for multimedia documents. In M. Freeman, P. Jardetzki, and H.M. Vin, editors, *Multimedia Computing and Networking 1997*, 1997.

[PBCH98] S. Proberts, D. Brailsford, L. Carr, and W. Hall. Dynamic link inclusion in online PDF journals. In R. Hersch, J. André, and H. Brown, editors, *EP-RIDT'98*. LNCS 1375, Springer-Verlag, 1998.

[PEL97] John Peterson, Conal Elliott, and Gary Shu Ling. Fran Users' Manual. Available from `http://www.haskell.org/fran`, 1997.

[SMI98] Synchronized Multimedia Integration Language (SMIL) 1.0 Specification. `http://www.w3.org/TR/REC-smil/`, 1998.

[Tho99a] Simon Thompson. A functional reactive animation of a lift using Fran. To appear in J. Functional Programming; available from `http://www.cs.ukc.ac.uk/people/staff/sjt/Fran`, 1999.

[Tho99b] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison Wesley, second edition, 1999.