Modelling Reactive Multimedia: Design and Authoring

Simon Thompson[*]
Computing Laboratory
University of Kent
Canterbury, UK
sjt@ukc.ac.uk

Peter King[†]
Department of Computer Science
University of Manitoba
Winnipeg, Canada
prking@cs.umanitoba.ca

Helen Cameron[‡]
Department of Computer Science
University of Manitoba
Winnipeg, Canada
hacamero@cs.umanitoba.ca

10 May 2002

Abstract

Multimedia document authoring is a multifaceted activity, and authoring tools tend to concentrate on a restricted set of the activities involved in the creation of a multimedia artifact. In particular, a distinction may be drawn between the design and the implementation of a multimedia artifact.

This paper presents a comparison of three different authoring paradigms, based on the common case study of a simple interactive animation. We present details of its implementation using the three different authoring tools, MCF, Fran and SMIL 2.0, and we discuss the conclusions that may be drawn from our comparison of the three approaches.

1 Introduction

This paper presents the details and the results of a comparative study on aspects of the design and authoring of multimedia artifacts. The creation of a multimedia artifact is a multifaceted process, and may involve such diverse activities as design, implementation, prototyping, testing, verification, maintenance, and reuse. The term ‘authoring’ is used by different researchers within the multimedia community to cover quite different ranges of activity within the lifecycle of a multimedia document. For some researchers, a distinction is made between design and authoring, and the term ‘authoring’ refers more narrowly to the implementation of an existing design. For others, the term may include a broader range of activities, and

[*]Simon Thompson is grateful to the British Council for its funding of travel in support of the work reported here.

[†]Supported by an individual research grant from NSERC. This author’s research was performed, in part, at LIRMM, Montpellier, France.

[‡]Supported by an individual research grant from NSERC.

indeed the analogy between authoring in this broader sense and software engineering has led to the introduction of the term ‘document engineering’. Our intention is to discuss the extent to which different authoring paradigms may assist in the activities listed above.

We present an investigation in which we compare three approaches to authoring a multimedia artifact. The different approaches each use a different authoring tool, and in turn each of the three tools supports a different authoring or design paradigm. Indeed, the three approaches differ widely, both in their approach to authoring and design, and in their interface to the user. One approach supports a visual design tool, the second is based on a general purpose programming language, and the third approach is a special purpose notation. At the same time and despite their apparent differences, all three approaches share the goal of providing the author with an appropriate tool for the creation of relatively complex multimedia artifacts, and therefore one should expect to learn from a comparison of the three approaches. Two of the three approaches that we will discuss are the subject of work by the authors of this paper, and comprise on the one hand a graphical tool, MCF (Multimedia Construction Formalism), specifically created for the design of multimedia documents [14, 15] and on the other a powerful general purpose programming language, Haskell, to which has been added a special purpose library, Fran [5, 6, 16], which one can view as a domain-specific embedded language. The third approach we will discuss is SMIL 2.0 [23], the W3C framework for describing time-based multimedia documents.

Our intent in performing this study is threefold:

- to explore in greater detail the distinction — introduced above — between multimedia *design* and multimedia *authoring*;
- to compare these approaches from various points of view, including their relative expressive power, the level of approach, the user interface and so on, and thereby to use these three differing approaches as the basis of an exploration of authoring and design paradigms appropriate in the domain of multimedia; and
- to obtain a firm basis for continued development of the authors’ own approach to multimedia authoring and design.

We have deliberately chosen three quite different but complementary approaches; that is, the three approaches differ both in their intended level of use and in their user idiom, as well as in their expressive power. Thus, we naturally expect a number of conclusions to emerge regarding the suitability of each approach in a particular situation. Included in these are reflections on the general differences of scripting and general-purpose programming, of specification and implementation, of event-based and timeline temporal models and so forth.

Beyond this, much is to be learned about what has been termed ‘document engineering’, where in our case ‘document’ refers to a multi- or hyper-media document. Thus, we believe that comparisons of such differing approaches to the creation of multimedia artifacts — we by no means limit this comment to the three approaches that we have selected — will provide much insight into the design of authoring systems in the broadest sense of that term.

The remainder of this paper is organized as follows. In Section 2 we present details of the manner in which our study was undertaken. We introduce the three systems, the multimedia artifact that we wish to describe in each, and we also provide details of the criteria that we use in discussing the three approaches. In Sections 3, 4 and 5 we present the details of each of the three models in turn, complete with the details of how the example was approached in each. We also provide some individual discussion of each approach in terms of the criteria developed in Section 2. Section 6 provides reflection on the experience of implementing the case study and Section 7 contains further discussion, draws some general conclusion and outlines our plans for future work.

2 The Study

In this section we give details of the study that we are undertaking, and motivate its various components. As indicated in the introduction, we have selected three differing multimedia authoring and design systems, and use each of these to develop a specific multimedia artifact. Each system is described in detail in a separate subsequent section. Here we wish to introduce each of the three systems, and provide some motivation for each choice. Additionally, we describe in detail the multimedia artifact that we wish to develop, and explain, in terms of its relationship with both the study and the systems, why we have chosen this particular case study.

2.1 The Three Systems

In order to provide a basis for comparison, we have deliberately selected systems in which the user interacts with the system both in a different manner and — perhaps more significantly — at a different stage in the development process. Thus one system comprises a tool for high-level multimedia design and is graphical in nature, while another of the selected tools is intended for lower-level authoring and is textual in nature. One of the tools is based on a general purpose language, while the others are specifically oriented toward this particular domain.

The first tool is **MCF**, Media Construction Formalism [14, 15]. MCF is a tool for multimedia artifact design and therefore targets multimedia design issues, rather than issues directly associated with implementation or with rendering. MCF provides the designer with an interactive graphical language in which to specify and manipulate multimedia behaviours and their interrelationships.

Fran [5, 6, 16], the second system we have selected, is based on a general-purpose programming language — Haskell [10] — which hosts a domain-specific embedded language for building reactive animations, using simple vector graphics, text, (animated) bitmaps and sound. Underpinning this system is what might be called the **Functional Reactive Programming** (FRP) paradigm in which continuously evolving behaviours and discrete events are used to model systems. FRP has been used in applications as diverse as robotics [24], animation and real-time programming. Moreover, whilst a functional language like Haskell is the natural host for an FRP approach, other languages such as Java can also be used (see [4]). In this paper 'Fran' will be used to refer to the particular Fran system and 'FRP' to the general approach.

Haskell embodies a *declarative* paradigm in which users describe 'what' it is that functions should do, rather than 'how' the effect should be achieved, and indeed Haskell programs can be read as *executable specifications* of behaviours.

The third system is **SMIL** (Synchronized Multimedia Integration Language) [23]. SMIL, which became a World Wide Web Consortium recommendation in August 2001, is a free-standing modelling framework that allows an author to specify the spatial and temporal layout of a multimedia artifact, as well as the internal and external reactivity of the artifact. The World Wide Web Consortium has added an animation module to SMIL 2.0, which became a W3C Recommendation in September 2001, and which we will use in the example.

2.2 The Case Study

In order to provide as wide a basis as possible for comparison, we have selected as our example an artifact that illustrates a number of design issues and reflects a wide variety of multimedia issues. In the subsequent subsection we will elaborate on the points of comparison that our example affords; for the present our intention is simply to present the problem we have tackled in these three systems.

Our case study is a simple prototype video game. It is comprised of five balls, placed in a presentation window at equal horizontal distances from each other. The balls move up and down with different speeds. Using the mouse, the player is expected to hit any of the balls. When a ball is hit, it falls to the bottom of the presentation window and ceases thereafter to move. When three balls have been hit, the game ends.

The game discussed here is a simplified version of the case study of Nanard et al. [14] from which some inessential complications have been removed.

2.3 Bases of Comparison

Despite its apparent simplicity, this game contains a number of significant elements:

- Five *media elements*, the balls, are *animated* in *parallel*.
- Each animation consists of two *sequential* segments: first, repeated up and down motion, and then the ball falling to a rest.
- The animation must *react* to the *external event* of the mouse click.
- The termination *condition* is based on a *computation* of the third ball being hit.
- The overall artifact must react to the *internal event* corresponding to the occurrence of this condition.

These and other related concerns are matters of significant concern to the multimedia author, and therefore support for such items should be provided in the authoring tool. In this subsection we provide a more precise account of these points of comparison for our three systems. We consider these points under three headings: multimedia features, document engineering aspects, and methodological aspects.

2.3.1 Multimedia Aspects

The author needs the ability to introduce typed media elements into the artifact, and also needs the ability to specify attributes (parameters) for these elements. It should be pointed out that we are not, however, concerned in this study with the *creation* of media elements, but rather with the integration of existing media elements into a presentation. One may regard such activity as the construction of increasingly complex structured media items. Media element attributes may be static or may be dynamic. Static attributes are fixed for the presentation, and in our example the balls each possess static attributes including their *shape*, *colour*, and *size*. Dynamic attributes are those that change in value as the artifact is played. In the example the *speed*, *direction* and *position* of each ball may be regarded as dynamic attributes. In a more complex example, one might have other dynamic attributes such as *orientation*, *luminosity*, *volume*, and so on.

With respect to temporal properties, real-time specification is needed for such items as the speeds of the balls in the game. Temporal synchronization is also required — for example, the five balls move in *parallel* — as is reactivity which is both internal — to accommodate, for example, the change of ball motion when it is hit — and external — to permit the user to interact with the game. Spatial placement of the balls in appropriate initial positions for the game, and specification of the regions in which the balls will move once the game is underway, are also needed.

The overall artifact is an interactive animation, and support must be provided for *continuity*, that is, the requirement that objects preserve their dynamic attribute values between various components of the complete animation. In addition, some non-multimedia features are needed, for example, integer arithmetic, or some equivalent mechanism, to count the number of balls that have been hit, and numeric computation to derive ball trajectories at various stages of the game.

2.3.2 Document Engineering Aspects

We wish to use this example to discuss a number of higher-level aspects that will arise during the creation of this artifact. Thus, we are interested in the level of support given by the tools to top-down modular design of the artifact. We are also interested in support for testing and prototyping individual modules as they are developed, and also for other aids such as module versioning. Thus, it would perhaps be appropriate to develop first a prototype with, say, a single ball, and then use language features to extend this prototype to the five balls required in the problem as described. Equally, it would make sense to develop the part of the game dealing with counting the balls as they are hit quite independently from the parts of the game that deal with the balls moving up and down.

2.3.3 Methodological Aspects

As indicated earlier, the three tools that we are considering were designed primarily to support different types of activity. We wish to distinguish between the *design* of an artifact and the rendering *code* that is produced, perhaps as the result of such a design.

A second question related to authoring methodology is that of granularity. That is, when developing the design or even the code for a particular module, what level of detail is required before the module can be tested or prototyped?

Third, since we are dealing with three different paradigms (programming language, graphical tool, and special purpose notation), we wish to identify and explore more generally the influence of the particular paradigm on the approach of an author to the creation of the artifact.

2.4 Details of the Study

In the three sections that follow we take each of the three paradigms in turn, first MCF, then FRP, and then SMIL 2.0. This order has been chosen as we wish to keep the three versions of the chosen artifact as similar as we reasonably can, and therefore it makes sense to begin with the paradigm most closely concerned with supporting the design of the artifact. In each case, we first provide some more detailed explanation of each paradigm and then show how our animation would be authored using the paradigm in question. We also provide some commentary for each paradigm, considered separately, based on the requirements analysis in the previous subsection. Following our separate discussion of each paradigm, we then discuss the paradigms from a comparative viewpoint.

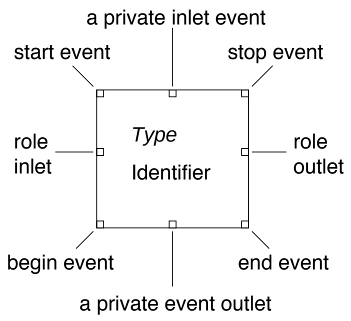


Figure 1: The visual representation of an MCA.

3 The Animation in MCF

3.1 An Overview of MCF

Media Construction Formalism [14, 15], MCF, is a visual tool for multimedia artifact design. MCF targets multimedia design issues, rather than issues directly associated with implementation or with rendering. MCF comprises both a formalism and an associated interactive design environment. MCF provides the designer with a graphical language in which to specify and manipulate complex temporal structures and multimedia behaviours in a generic and reusable fashion. MCF is inspired by work on design patterns [1, 17] as used in software engineering [8], and by the notion of actors and roles [11]. MCF allows a designer to express a multimedia scenario as a data-flow of actors from one role to another.

To accomplish such a design, the user is provided with a visual language in which abstractions, called Media Construction Abstractions (MCA), can be created and manipulated. Such a media construction abstraction corresponds to some component of the multimedia artifact under design; see Section 3.2 for a complete MCF design. Within an MCF diagram, MCAs are connected in a network. The interpretation of this network follows data-driven data-flow semantics. MCAs may be parameterised and an MCF diagram is strongly typed. Actual parameters, called *actors*, flow along the arcs within the diagram, and correspond to the objects in the media artifact. Within an MCA, an actor is represented by a formal parameter, termed a *player*. The activity that an actor undertakes within an MCA is termed its *role*. Thus, in formal terms, within an MCA, a role is defined in terms of its player. At run time, an actual actor becomes the player of the role. In addition to actors, a second class of items, *events*, flows between MCAs; events are used to represent reactivity.

Figure 1 illustrates the appearance of an MCA. Actors enter on the left hand side and exit on the right. Events enter on the top and leave on the bottom. Each MCA has two standard input events, `start` and `stop`, which may be invoked either explicitly by the flow of an event from some other MCA, or implicitly by, respectively, the arrival of a complete set of input actors or the termination of the role. Each MCA also has two standard outlet events, `begin` and `end`, which may be used to initiate and to terminate other MCAs.

Further events, known as `private`, may be specified within the MCA for specifying interaction with other MCAs.

MCAs fall into various classes. Many MCAs are of the class behavioural. A behavioural MCA corresponds to a basic media operation, such as playing an audio-clip, or moving an animated object from one position to another. Other MCA classes correspond to various forms of control structures, including iteration, conditionals, composition, interaction and so forth. The definition of MCF is a recursive one;

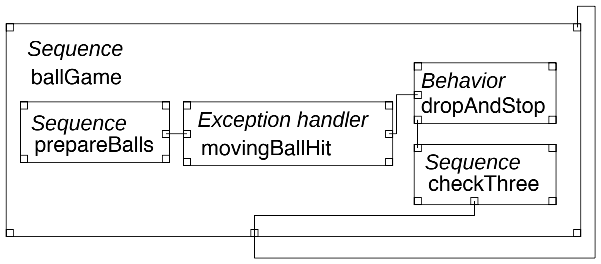


Figure 2: The MCF diagram of the entire game.

thus one may have iterations of conditionals, and so forth.

Temporal properties may be specified within a behavioural MCA, and thus by implication, may be associated with any other class of MCA. Both temporal and reactive relationships among sets of MCAs may also be specified.

The MCF editor is a typed, structured editor, which ensures that an MCF design is always syntactically correct. MCF diagrams can therefore be readily interpreted, and thus a design prototyped, within a variety of target languages. MCF is under development, and while the current version makes use of Java, one aspect of this paper is to explore the relationship between MCF and both FRP and SMIL 2.0.

3.2 The Game in MCF

We present four MCF diagrams that illustrate various stages in the design of an artifact corresponding to this simple game. Figure 2 contains a high-level view of the entire game, and comprises four MCA boxes, constituting three main steps to be considered. In the first MCA, `prepareBalls`, each of the five balls is created and assigned its parameters, namely, its initial position and speed. The balls flow as actors to the second MCA, `movingBallHit`, which controls the up and down motion and captures a successful hit (mouse click). When a ball is hit, it passes as an actor to the third stage, where it obeys the *Behavior* `dropAndStop`, which in turn triggers the MCA `checkThree`, used to terminate the game.

In addition to these actors (the balls), there are also two instances of events. The link between the outlet `begin` of `dropAndStop` and the `start` inlet of `checkThree` is one such event flow, which causes `checkThree` to be invoked whenever an actor enters `dropAndStop`, that is, whenever a ball is hit. A second event flow occurs between a private outlet event of `checkThree`, which we will describe in detail below, and which activates the `stop` event of the MCA `ballGame`, thus terminating the game.

The other diagrams contain more details of the three steps comprising the animation. For the purposes of this paper, it is not necessary to give full details of the roles within the various MCAs, since our purpose here is to illustrate the use of MCF in a design. Equally, details of the semantics of the various MCA classes (*Iteration*, *Behavior* and so on) are unnecessary for the purposes of this paper, but may be found in Nanard et al. [14, 15].

Figure 3 reveals the internal structure of the MCA `prepareBalls`. It will be observed that this MCA consists first of an iteration of length five, each step of which creates a new ball and hands it along to a second MCA, `placeBall`. The effect is that each of the five balls is first created and then placed in its appropriate starting position in the presentation window. Actors may contain state information, thus the index of each ball may be used to calculate its initial coordinates, and these coordinates may in turn be passed to the second stage of the animation.

Figure 4 reveals the internal structure of the MCA representing the central part of the game, the balls

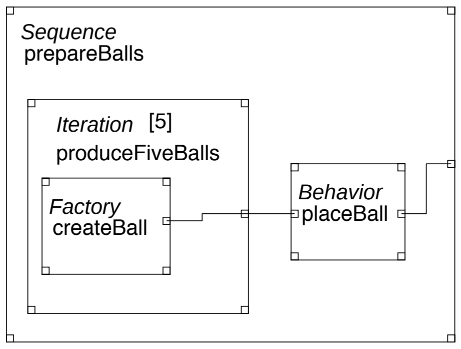


Figure 3: Creating the balls.

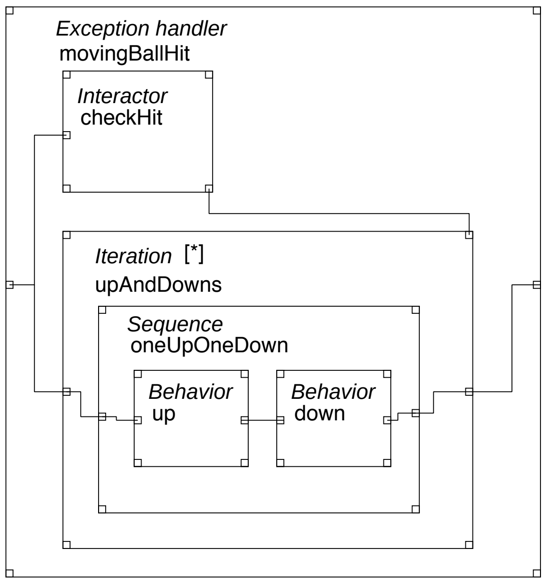


Figure 4: The balls moving up and down until they are hit.

moving up and down until they are hit. `MovingBallHit` comprises two MCAs operating in parallel. The *Iterator* `upAndDowns` consists of an indefinite iteration of a sequence of two behaviours, representing upward and downward motion respectively. The *Interactor* MCA `checkHit` will, by definition, activate its `end` event when the condition within its role, namely, the successful hit of the ball, is satisfied, and by virtue of the connection of this event to the `stop` event of `upAndDowns`, this *Iterator* is terminated at this point and the ball actor passed onto the third stage.

One aspect of the semantics of MCF is important at this point, and concerns the *cloning* of roles. In this MCF diagram, a succession of ball actors will enter the sole inlet role of the MCA `movingBallHit`. As successive actors enter the same role inlet of an MCA, the role is considered to be internally cloned. Such cloning, naturally, takes account of the different state information of the various actors. Therefore, since we have observed that the MCA `prepareBalls` passes along the five balls to the MCA `movingBallHit`, this MCA in its turn specifies the up and down motion consonant with the internal state of the ball, interrupted by the hit, of each of the five balls. The synchronisation in parallel of these five entities is, again, guaranteed by the semantics of MCF.

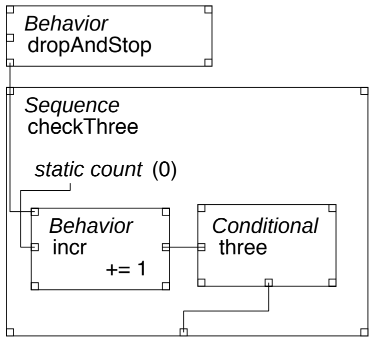


Figure 5: The termination of the game.

The final diagram, Figure 5, controls the termination of the game. The actor representing a hit ball enters the *Behavior* `dropAndStop`, which activates the *Sequence* `checkThree`. The *Conditional* MCA `three` is assumed to activate the depicted private event when the count has reached three, and this event stops the entire game by virtue of its connection to the stop event of the outermost MCA.

3.3 MCF Discussion

We now discuss the MCF design approach in the light of the criteria discussed in Section 2.3 and consider the extent to which MCF provides the various aspects described therein.

MCF provides two methods for the incorporation of media items into an animation. The first approach is to permit the author to reference and incorporate an existing media element from some other network or local source. Pre-existing images, videos, sound-clips and so forth can be introduced into a MCF design by naming them with a URI-like syntax. The second approach is to permit the author to construct composite items, frequently animated items, using the actor/player/role model embodied in MCF. Actors and players are typed, and their types may be composite, and may have parameters that serve as the attributes discussed in Section 2.3.1. No formal distinction is made between static and dynamic attributes, rather all attributes are dynamic by default as they may be manipulated by the composition laws within an MCA box.

MCF provides a comprehensive set of temporal relationships, including reactive relationships. The Allen temporal relations [2], including parallel and sequential composition, real-time delays, internal and external reactivity are all provided for in MCF. Nanard et al. [14, 15] fully discuss these matters; we will not repeat the details here. On the other hand, the MCF model does not contain any features whatsoever directly supporting spatial relationships. MCF contains no specific feature enabling the author to specify, say, a region, in the sense of a geometrical region in which an item will be placed. Rather, such a specification, as well as the specification of any subsequent animation applied to such a region, must be programmed explicitly within an MCA. It follows that there is little direct correlation between spatial and temporal relations, and no formal notion of combined spatio-temporal relationships [13, 21].

Turning to interactive animation, the actor/player model can be seen to provide continuity, since actors preserve their state from one MCA to the next. Thus, the author/designer simply includes those quantities for which such continuity is an issue as parameters of the actor/player, and defines the type of the actor accordingly. Equally, since the behavioural description in an MCA may incorporate any features from the host programming language, the inclusion of arithmetic and other computations is straightforward.

MCF was designed, in large part, with the document engineering issues from Section 2.3.2 in mind. Once again, Nanard et al. [14, 15] discuss many of these matters in some detail. MCF supports modularity of design both through the MCA itself, which possesses many of the aspects of a typed object, and through the features that MCF has for encapsulation. Any set of adjacent MCAs, where adjacent means linked by actor paths, may be encapsulated and treated as a single composite MCA. In addition, such encapsulation permits the author to view and to work on and to test a partial design at an arbitrary level of granularity. A single MCA or, equally, an encapsulated MCA set may be regarded as a plug-in component, which supports the notion of re-use.

On the other hand, at least in its present form, MCF does not provide explicit support for version control.

4 The Animation in Fran

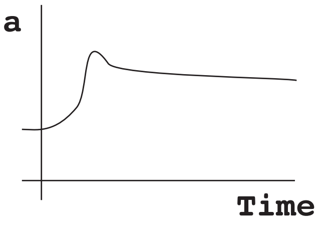
4.1 An Overview of Fran

Fran is a system for **F**unctional **R**eactive **An**imation [5, 6, 16], which is available as a library for the functional programming language Haskell [10]. Fran's creator had previously implemented a similar system in C++ [7], but moved to work within a modern functional programming language because of the support it provided for modelling by means of functions as well as the high degree of productivity that functional languages afford. Tutorial introductions to Haskell and Fran can be found in [19] and [20], respectively. Some brief explanations of Haskell notation are included in the text.

Fran models the world by means of **behaviours**, which evolve in continuous time, and **events**, which occur at discrete time points. Behaviours are functions over `Time`:

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

where `type Time = Double`. The behaviour type, `Behavior`, is a parameterised type with `a` being a type variable; thus `Behavior Int` is the type of functions from `Time` to `Int`, `Time ->Int`.



An animated image will be of type `Behavior Image` (or `ImageB`), and the x-position of the image may be given by a `Behavior Double` (or `RealB`). One way of viewing a value of type `Behavior a` is as the *history* of a variable of type `a` over the evolution of a system. In a similar way a behaviour can trace the history of a whole system by tracking the values of its state variables over time.

Events are seen as sequences of event occurrences

```
type Event a = [(Time,a)]
```

The Haskell type `(Time,a)` consists of pairs of values, from `Time` and `a` respectively. The type `[(Time,a)]` consists of lists of such pairs. Each pair in the list consists of a time at which the occurrence happens, and the value returned. For example, a keyboard event would be modelled by an event of type `Event Char`. An example keyboard might be

```
[(2.1,'b'),(3.2,'e'),(3.5,'e'), ...]
```

where the first occurrence of the event happens at time `2.1` and carries the value `'b'`, indicating which key has been pressed when. An event that does not return a value, such as a left mouse button click, is represented by a value of type `Event ()`[1].

Haskell is a lazy functional language, in which values are evaluated only when they are needed. Moreover, data structures are evaluated incrementally, again by need, and so in particular lists may be infinite. Therefore events are simply modelled by sequences of event occurrences which might or might not be infinite, which, in turn, are modelled by lists in Fran.

The Fran library provides a collection of functions over the core `Behavior` and `Event` types, including functions by which the discrete and continuous aspects of a Fran animation may interact with each other. The library functions include

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

which takes two real behaviours `x` and `y` and an animated image and imposes the motion described by `x` and `y` on the moving image. `moveXY` thus composes motions.

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

overlays two animated images one over the other, giving a parallel composition of animations.

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

The `Behavior` `untilB b1 e b2` acts like `b1` until the event `e` occurs, when it switches to the behaviour `b2`. In this way an event affects behaviours: for instance, `untilB` can be used to make a mouse click change the motion of an animated ball.

```
predicate :: Behavior Bool -> User -> Event ()
```

This function generates an event from a Boolean behaviour `b` (and the environment, which is encapsulated in the `User` argument): the event occurs at the times when `b` becomes `True`. `predicate` may be used to generate an event when a falling ball reaches a certain point on the screen, for example.

Other, more specialised, functions are introduced in the course of the case study.

4.2 The Game in Fran

The video game is modelled in Fran by making each of the balls an animated image (an `ImageB`). These images can react to external events, such as a button press, or internal events, such as the point at which a variable takes on a particular value. The balls are initially programmed separately, and overlaid in parallel using the `over` function; in a second version of the game the balls are coordinated and the game is ended once three balls are hit.

The Ball Game: Hit All Balls

A single ball is animated in the program given in Figure 6. Using the `moveXY` library function discussed in Section 4.1, the top level definition of the moving ball is thus given by

```
moveXY x y (withColor colour circle)
```

[1] `()` is the one-element type, whose element is also written `()`

```
circ :: User -> (RealB,ColorB,RealB,RealB) -> ImageB

circ user (horiz,colour,phase,speed)
  = moveXY x y (withColor colour circle)
    where

    x = horiz

    y = untilB (sin(pi*time*speed - phase)) changePos (-1)

    changePos
      = whenSnap lButton apart isClose
        where
        apart       =  distance2 mousePos (point2XY x y)
        isClose x p = (p <=0.2)
```

Figure 6: A single reactive ball.

```
balls :: User -> ImageB
balls user
  = circ user ((-0.8),blue,0.4,1.3)
    `over`
    circ user ((-0.4),purple,0.2,0.5)
    `over`
    circ user (0,green,1.3,1)
    `over`
    circ user (0.4,yellow,0.9,1.1)
    `over`
    circ user (0.8,red,0.6,0.8)
```

Figure 7: Five balls moving in parallel.

where x and y are real behaviours. Note that function application in Haskell is denoted by juxtaposition: the function name is followed by its actual parameters. In this case x is the constant horiz, since the balls move in a vertical line. A ball that bounces has its vertical position given by sin(pi*time*speed - phase); this motion is oscillatory.

In the reactive game, the behaviour continues in this way until the ball is hit: the vertical position y then changes to (-1). Hitting the ball is given by the event changePos, and y is given by

```
y = untilB (sin(pi*time*speed - phase)) changePos (-1)
```

The top-level definition of changePos states

```
changePos = whenSnap lButton apart close
```

In this definition, the function whenSnap is applied to three arguments:

- an event, lButton: the press of the left mouse button;
- a behaviour, apart: the distance between the mouse position and the centre of the circle;
- a test of a value, isClose: is a real value bounded by 0.2?

The effect of whenSnap is to snapshot the behaviour at the time when the event occurs; if the value produced meets the test, then an occurrence of the result event is produced. In this case, the event changePos will fire when the button is pressed *only if* the mouse and circle centre are sufficiently close when the button is pressed.

As is evident from Figure 6, we have in fact defined a function that gives a moving image (an ImageB) that depends on some input parameters. The first parameter encapsulates the environment of the image, or the 'User' in Fran terminology. The second contains the parameters for colour, x-position, speed and phase.

This parametricity allows a five ball game to be defined by taking five instances of balls, and overlaying the five over each other in parallel, as in Figure 7.

The Game: Hit Three Balls

The balls in the first version are independent of each other, and the game consists, in effect, of five quite separate games played one on top of the other. To achieve coordination, it is necessary for each ball to make its changePos event visible. This is done by modifying the circ function to return both an animated image and an event:

```
circEvent :: User -> (RealB,ColorB,RealB,RealB) -> (ImageB, Event ())
```

Given the five events, they can be merged to give the event

```
stopEv = ev1 .|. ev2 .|. ev3 .|. ev4 .|. ev5
```

which fires each time any of the balls is hit; the number of occurrences is then counted by an integer-valued behaviour, stopped:

```
stopped = accumCB (+) 0 stopEv
```

and the game is ended when the value of this behaviour reaches three

```
stopGame = predicate (stopped >* 2) u
```

Full Details

The full details of these programs can be found at

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

Also accessible from the web site are variants of the programs closer in detail to the MCF formalism.

4.3 Discussion of Fran and Functional Reactive Programming

An FRP system is modelled in an abstract and implementation-neutral way: rather than describing *how* certain values are made to evolve over time, the FRP model provides a description of *what* the evolving value is. The description of a complete system will comprise various behaviours and events:

- Event occurrences can produce changes in behaviours (for example, `untilB` in Fran).
- Event occurrences can be generated from behaviours (for instance, `predicate` in Fran).
- Events and behaviours evolve concurrently: their descriptions refer to the same time domain.
- The FRP model will be embedded in a programming language, and thus the descriptions of behaviours and events can use the facilities of the host language, including arithmetic, data structures, definition and parameterisation mechanisms and so forth.

An FRP description of a system is thus a mathematical model, in principle unconstrained by implementation concerns. However, its realisation in a particular system like Fran will impose limitations:

- A system like Fran contains a finite set of domain-specific primitives, which may not cover all the operations of the domain.
- Concern for implementation efficiency may well restrict the primitives further. For example, rather than allowing arbitrary functions from `Time` to values – an *extensional* approach – the system may use an internal representation of functions. Such an *intensional* view of functions allows optimisations: for instance, it is unnecessary to sample a constant function more than once.

Turning to the categories of Section 2.3, we first consider the *multimedia* features provided by FRP and Fran. Because of its abstraction, FRP can embrace all types of multimedia: behaviours can be described directly, and ‘external’ media such as videos can be described by means of the event model. Media attributes can be described by behaviours; spatial placement is given by `moveXY` and (real-time) temporal specification is easily embraced by the model. Continuity is up to the user: behaviours can change continuously or not as is required by the particular scenario. Fran is hosted in the general purpose language Haskell, and so arithmetic and (parameterised) definitions are available.

The *document engineering* features of the FRP/Fran model have a number of aspects.

- The host language, Haskell in the case of Fran, supports top-down development. Indeed, one way of viewing the FRP approach is as giving an executable specification of a multimedia artifact.
- On the other hand, FRP lends itself to high-level logical specification: one can write relationships between behaviours that need to be refined into executable specifications in Fran or indeed a more concrete system.

- As an embedded language, it is trivial to use the definition facilities of the host language to generalise a definition (of a bouncing ball, say) into one that can be instantiated in a variety of ways.

Finally, turning to the methodological aspects of FRP and Fran, Fran is explicitly designed to hide from the user the details by which the animation is achieved. Users are not required to refresh buffers and so forth, but rather to describe what effect should be achieved. In developing a Fran system one can produce a series of prototypes that describe key parameters of the system. These evolving values can be shown as changing numbers or strings on a simple display; once their interaction is validated the system can use them to animate more complex graphics.

A drawback of Fran from our authoring perspective is its generality and (paradoxically) low-level API. For Fran – and indeed maybe FRP – to be usable by a non-specialist it needs to supply a higher-level API, in which, for instance, the `User` argument is hidden, and sequential composition of finite behaviours is supported. Such an API could also encapsulate general design patterns as higher-order functions, and thus prevent the naive user from writing non-terminating recursions and other 'black holes'. This represents work in progress for the authors.

5 The Animation in SMIL 2.0

5.1 A Brief Introduction to SMIL 2.0

SMIL (Synchronized Multimedia Integration Language) [23] is a free-standing modelling language that allows an author to specify the spatial and temporal layout of a multimedia artifact, as well as the internal and external reactivity of the artifact. The World Wide Web Consortium has added an animation module to SMIL 2.0, which became a Recommendation in September 2001. Because SMIL 2.0 is an XML application, the author writes an XML source document which then can be rendered by a SMIL player. The author places the media items spatially by assigning values to spatial attributes of the XML elements representing the items. The author places the media items temporally by combining the media item within such constructs as `<par>` (presents all elements within the `<par>` in parallel) or `<seq>` (present the elements within the `<seq>` one at a time in the order they appear within the `<seq>`). Simple animations of media items, such as stretching an item vertically or horizontally or moving an item, can be specified using such animation elements as `<animate>` or `<animateMotion>`.

5.2 The Game in SMIL 2.0

Our SMIL 2.0 version of the game comprises three parts. Each ball is a simple GIF image represented by an `<img>` element:

```
<!-- prepareBalls -->

<img id="ball1" src="ball.gif" region="ballregion1" left="0" top=="250"/>
<img id="ball2" src="ball.gif" region="ballregion2" left="0" top=="250"/>
<img id="ball3" src="ball.gif" region="ballregion3" left="0" top=="250"/>
<img id="ball4" src="ball.gif" region="ballregion4" left="0" top=="250"/>
<img id="ball5" src="ball.gif" region="ballregion5" left="0" top=="250"/>
```

The popping up and down of a ball may be achieved by a repeated sequence of two animations that move the ball up and then down. The repeated popping sequence ends when the player hits the ball. The ball then is returned to its starting place by means of a third animation.

```
<!-- Ball movements -->

<seq id="ball1Motion">
  <par id="movingBallHit1" end="ball1.activateEvent" >
    <seq id="upAndDowns1" repeat="indefinite" >
      <animateMotion id="up1" targetElement="ball1" dur="2s"
                   from="0,250" to="0,0" />
      <animateMotion id="down1" targetElement="ball1" dur="2s"
                   from="0,0" to="0,250" />
    </seq>
  </par>
  <animateMotion id="dropAndStop1" targetElement="ball1"
                 to="0,0" dur="0s" />
</seq>
```

This specification moves only `ball1`. A copy must included for each of the other five balls, with appropriate changes to the various attributes referring to the particular ball.

To count the three hits, we use a sequence of length three; the content of each step in the sequence is empty and each of the three steps terminates when one of the balls is hit. The end of the third such hit causes the end of this counting sequence, and thereby the end of the parent `<seq id="stopAll">` container, which is also the signal to end the game.

```
<seq id="stopAll">
  <seq repeatCount="3">
   <seq end="upAndDowns1.onend; upAndDowns2.onend;
             upAndDowns3.onend; upAndDowns4.onend;
             upAndDowns5.onend">
   </seq>
  </seq>
 </seq>
```

These three pieces of the game must happen in parallel, and must therefore be children of a container `par` element.

```
<par endsync="stopAll">
  <!-- prepareBalls -->
  ...
  <!-- Ball movements -->
  ...
  <!-- stopAll -->
  ...
</par>
```

Full Details

A working version of this program may be found at

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

This site also contains two other versions of the game, one in which the balls move in two dimensions, and another in which the balls roll off once the game is over. These three versions were all implemented in Internet Explorer's HTML+TIME 2.0 (a dialect of SMIL 2.0) and require Internet Explorer version 5.5 or higher to be played.

5.3 SMIL 2.0 Discussion

We now consider SMIL 2.0 in the light of the criteria discussed in Section 2.3.

SMIL 2.0, as its name indicates, is designed specifically for the integration of multimedia items within composite media artifacts. Thus multimedia items such as images, video, audio, and so on can be included by referring to appropriate elements using an URI-like notation. Such integration is, naturally enough, subject to the implementation understanding media items of the type in question.

SMIL 2.0 contains provision for a rich set of temporal attributes. As has been demonstrated above, synchronisation of simple or composite media items is achieved using `<par>` and `<seq>` elements. In addition, SMIL 2.0 provides a number of temporal *attributes* (in the XML sense) such as `begin`, `end`, `dur`, and so on, which allow for real-time relative and absolute placement of items. Indeed, it can be easily seen that one can specify temporal relationships, including all of the Allen relations [2] together with real-time delays. In addition, SMIL 2.0 provides for repetition, via the (`repeat` and `repeatDur` attributes), and as we have seen in our example, one may use such attributes to provide a limited form of integer arithmetic that allows counting event occurrences.

SMIL 2.0 includes ample provision for spatial specification, in that one can place an item in a previously defined region. A region in SMIL 2.0, however, is more general than a simple spatial region. Indeed, in SMIL 2.0, every media item is played in a region, including audio items for example. If no region is explicitly defined for an element, then the player assigns one by default. In essence a region is an "area" for which one can control values pertaining to media items. A single media item can be playing in a region at a time, and the region may be animated, thereby changing the attributes of the media item. Such an animation might change the size of a displayed image, or increase the volume of a sound being played in the region, and so on. Thus, a region in SMIL 2.0 is similar to a "channel" in [9].

SMIL 2.0 contains the notion of "event" and in the example we have made use of this feature to track the number of balls that have been hit by counting the number of times the up and down motion of a ball has been stopped. In addition, SMIL 2.0 provides a set of predefined events, such as the "mouse" events exemplified in the example by our use of `activateEvent`, and such events may be used, for example, to start and stop arbitrary composite media segments. However, the ability of SMIL 2.0 to specify internal reactivity among media items within an artifact is limited. We shall discuss this matter more fully in Section 6, but for now note, by way of example, that it is difficult in SMIL 2.0 to create an event that would correspond to two of the balls colliding. The focus of W3C's Synchronized Multimedia Activity has not included the document engineering criteria of Section 2.3.2. As an XML-based application, SMIL 2.0 inherits the strengths and weaknesses of XML, which was designed to allow easy, platform-independent exchange of information, as well as the processing and display of that information. SMIL 2.0 itself does not in itself support versioning or modularity of design or testing, and provides no separation between the design and coding of a multimedia artifact.

6 Commentary

In this section, we draw some conclusions, based in large part on the experience we gained in developing the video game in these three paradigms.

6.1 Design or Implementation?

At the beginning of this paper we referred to a number of diverse aspects of authoring. The three systems target different facets of the authoring process.

MCF targets design issues, and accordingly provides features that can be understood and manipulated by designers at different levels of granularity and detail. The approach in MCF comprises two separate operations: the design of individual components of an animation, and the assembly of these components into an artifact. These two operations can be carried out independently of each other. In particular, a complete MCF diagram may be produced and its consistency verified before any 'code' implementing particular components is written. The term 'assembly' refers to a range of possibilities, including (possibly multiple) instantiation, temporal synchronisation, the use of conditional and iterative control structures, encapsulation and so forth.

SMIL 2.0, as its name indicates, targets media item integration, rather than artifact design. SMIL lacks facilities for design, and indeed, when we wrote the SMIL 2.0 code appearing in Section 5, we used the MCA diagrams developed previously as the basis for the implementation. One could well imagine a SMIL-based interpreter for MCF.

FRP provides some aspects of both design and authoring. The Haskell host language provides many of the higher level features referred to under 'assembly' above, and, at the same time, FRP code is executable, and therefore provides its own prototyping system. One could well imagine an FRP-based player for SMIL 2.0 and MCF, or a SMIL 2.0 realisation of aspects of FRP.

6.2 Programming Language or Markup Language?

Both Fran and SMIL 2.0 provide declarative descriptions of multimedia artifacts. Fran is a domain-specific language (DSL) for reactive animation, embedded in the general-purpose programming language Haskell. SMIL is also a domain-specific language, but one which stands alone.[2]

An embedded language supplies the user with a general-purpose infrastructure for describing multimedia artifacts. In our case study we use these facilities in a variety of ways.

- **Arithmetic** over integers is used in the calculation of the number of balls that have been hit. Arithmetic computations over reals define the trajectories of the balls.
- **Definitions** of components. A component of the artifact can be described once and used multiple times, perhaps in different positions. This is a special case of the next bullet point.
- **Parameterised definitions** of components. As can be seen in the case study, the definition of the ball can be parameterised on its x position, speed and so forth. Each time that the definition is used, these parameters can be given different values.
- A natural style of description for reactive behaviours defines the behaviour of the components in terms of each other: such **recursive** descriptions are valid in Haskell.

[2]We address the issue of *scripting* within SMIL 2.0 at the end of this section.

- **Design patterns** can be coded in a programming language such as Haskell, in which functions can take functions as arguments (or return functions as results). For example, the *horizontal placement* pattern used in laying out the balls could be passed the function constructing the particular objects as a parameter; in our case study this constructing function would be the function `circ`.

- The **type system** of Haskell ensures that any type errors in Fran programs are discovered at compile time. This contrasts with the XML worldview (on which SMIL 2.0 is based) in which all data items are described as strings; type errors in describing an item will, in general, be discovered only at run time.

In summary, Fran provides the user with a Turing-complete framework in which to write high-level descriptions of reactive behaviours. The modular features of Haskell facilitate modular development and, more importantly, abstraction in the language supports a user to write re-usable definitions. In the case study this support was evident in the single, parameterised, definition of the ball which could be reused immediately. Such reuse in turn supports maintenance: if one is to modify the behaviour of the balls there is a single point of maintenance, namely the definition of `circ`.

In comparison to Fran, SMIL 2.0 does not support arithmetic, and has no definition facilities. SMIL 2.0 cannot, therefore, support abstraction and reuse in the same way as Fran. SMIL 2.0 has, on the other hand, a variety of advantages over Fran.

- SMIL 2.0 belongs to the XML language family, and thus is supported by a wide range of tools and language bindings.

- SMIL 2.0 is integrated into a number of current browsers, including Internet Explorer (versions 5.5 onward).

- Evaluation of SMIL 2.0 does not (directly) involve the interpretation of a Turing-complete programming language.

As is evident from the discussion here, Fran provides a coherent programming model for reactive behaviours. SMIL 2.0 is, *prima facie*, a declarative language of more limited scope. It can be extended, however, by *scripts* written in JavaScript, say, as well as by integration with the Domain Object Model (DOM). The disadvantage of this approach is that the high-level declarative description is lost, and the extension of expressive power is gained at the cost of a much more operational view of the system.

As a general conclusion we would argue that the approach in which a DSL is embedded in a declarative infrastructure such as Fran provides a more uniform and powerful programming model than the *ad hoc* augmentation of a SMIL 2.0 document by JavaScript and DOM.

6.3 Temporal or Reactive Model?

While it is possible in all three systems to specify both temporal and reactive relationships, the underlying model in SMIL 2.0 and MCF is quite different to that in FRP. Both SMIL 2.0 and MCF have at their core a temporal model of synchronisation, whilst the model in FRP is based on reactivity.

6.3.1 SMIL 2.0 and MCF: The Temporal Model

The temporal models found in SMIL 2.0 and MCF are similar, as noted in [14, 15]. Both SMIL 2.0 and MCF provide operators for parallel and sequential synchronisation: in SMIL 2.0 these are the `<par>` and

`<seq>` elements, whilst in MCF these two relationships have a visual form. Both SMIL 2.0 and MCF provide, in addition, real-time delays. MCF contains an explicit `delay()` operator. SMIL 2.0 provides attributes including `dur`, `begin` and `end`. These attributes may appear both within elementary media components — for example to effect the clipping of an object — and also within composite components, built using `<seq>` and `<par>`, for example. It can be readily shown that both SMIL 2.0 and MCF have temporal models encompassing those in, for example, [12] and [22].

To specify reactive relationships between media items, both SMIL 2.0 and MCF provide an event-driven mechanism. SMIL 2.0 provides both a set of predefined events, such as the mouse events occurring in the example, as well as events corresponding to the start and end of arbitrary media elements. Such events may be used to control the start and end of other media items. Thus, in the case study, the event `stopAll` arises when the third ball is hit, and is then used to terminate the entire animation. However, SMIL 2.0 does not provide a mechanism for specifying arbitrary, dynamically-occurring events. Thus, the definition of an event corresponding to two balls colliding, or a sequence of events corresponding to the times when the speed of a ball exceeds a certain maximum, are both problematical. In MCF, events traverse the MCA network in a similar manner to actors, and the occurrence of an event may be arbitrarily specified within an MCA. Thus the 'maximum speed' example can readily be handled, but the specification of an event depending on more than one behaviour remains difficult.

SMIL 2.0 provides an additional reactive mechanism, namely the `excl` element. Only one child of this element may play at any given time, and if any element begins playing while another child is already playing, the element that was playing is either paused or stopped. This feature, whilst useful in a number of situations, manifestly complicates the SMIL 2.0 timing model.

6.3.2 FRP: The Reactive Model

In FRP, all synchronisation is event-based. Thus, Fran contains, and indeed needs, no primitive definitions corresponding to the real-time operators of SMIL 2.0 such as `dur`, `begin` and `end`. Nor does FRP contain primitive operators embodying the familiar Allen relations [2]. However, the Fran operator `over` overlays two animations and thus realises the `<par>` element of SMIL 2.0, that is, Allen's operator *and*. Secondly, it is manifestly possible to *define* all of the SMIL 2.0 and Allen operators in Fran in a generic fashion using the function definition capabilities of Haskell.

The reactive model in FRP is a powerful one, and the use of the function `predicate` permits the specification of arbitrary events based on properties of behaviours. This permits the definition of those dynamically-occurring events that were identified in the previous section as being problematic for SMIL 2.0 and MCF. In [3], reaction to *external* behaviours is shown to be expressible in FRP. By way of illustration, consider an online multimedia news and weather bulletin in which we wish to display those locations in Canada with the current minimum and maximum temperatures, computed from external streams of time and temperature pairs from multiple locations. Such control cannot in general be achieved by defining (discrete) events to signal such properties, for two reasons. First, it would require the source(s) of the behaviours, the locations in Canada in our example, to provide events corresponding to all of the properties that might in the future be required. Second, the property in question may be associated with more than one such external behaviour — if we wish to display location B when its temperature has dipped below that of location A, for example.

We note that SMIL 2.0 itself provides no direct support for this form of reaction, and would require the inclusion of appropriate code in some other language. For example, most of the required functionality may be achieved within web pages by going outside of SMIL 2.0 and relying on the inclusion of appropriate applets or CGI scripts. Within MCF, this form of reactivity is also problematical and is difficult to achieve since the event model within MCF is discrete. Thus, one is driven to code the communication between the client which performs the computation and the servers which supply, in this example, the temperature values as a loop of discrete event-driven requests.

6.3.3 Temporal Transformations

SMIL 2.0 provides explicit abstractions for temporal transformation in the attributes `speed`, `accelerate`, `decelerate` and `autoReverse`. The meaning of these operators is intuitively fairly obvious, and they are discussed both in [18] and in [23]. Fran provides similar time transforms, which were used in the 'behaviour player' discussed in [3].

MCF does not provide such facilities as primitive operations of the language, but the effect of such operators could appear within the behaviour specified within an MCA.

6.4 Spatial Relationships

As we discussed in Section 5.3, SMIL 2.0 contains the notion of a region or channel containing arbitrary elements to which arbitrary transitions may be applied. In particular, regions are used for the spatial placement of media items in a display. Neither FRP nor MCF has explicit support for spatial placement, and both require spatial regions to be coded implicitly by means of parameters or within definitions. Thus the 'regions' in the FRP version of the game in Section 4.2 are defined by the five actual parameter values, -0.8, -0.4, 0, 0.4, and 0.8, of the centres of the five balls.

6.5 Underlying Formalism

Each of the systems presented is based on a model of reactive systems. These models provide different perspectives on the way that system descriptions are to be written and processed, and the models also determine the way in which one can verify properties of the systems described by them.

The FRP approach models the world by means of behaviours which evolve through time. Reactivity — both internal and external — enters the picture through an event model. System descriptions are given by sets of recursive equations, which are interpreted to give a reactive behaviour when an FRP expression is evaluated. Thus, an FRP program has a well-defined value. Contradictions or inconsistencies in the description will become behaviours that are undefined, and which result in a computational 'black hole'. This result is entirely consistent with the denotational semantics of Haskell, provided by its reduction to the λ-calculus.

SMIL is based on a timeline model, which is complicated in SMIL 2.0 by the presence of reactivity. SMIL elements are described by means of a set of constraints which can, for example, link the start of one item to the end of another. It is therefore quite possible to write sets of constraints that are not consistent, and thus which are not realisable. In early versions of SMIL it is possible to detect such inconsistencies statically. With the addition of reactivity, it is possible that the consistency of a system will depend upon an external event whose occurrence cannot be predicted statically: consistency thus becomes a dynamic property.

MCF is based on a temporal logic model which may in theory be used to verify temporal correctness. This may, as in the case of SMIL, place proof obligations on the behaviour of the user. In addition, the actor and MCA notions are based heavily on the actor role model discussed by Kristensen and Østerbye [11] and work on design patterns such as is described by Riehle [17].

7 Conclusions and Future Work

The discussions and comparisons appearing in Section 6 contain a number of indications of further work within the MCF and the FRP frameworks. With regard to FRP, our experience shows that there are some infelicities in that model. The fact that FRP behaviours are infinite, the complexity of the event model, and the lack of a standard library of temporal as opposed to reactive operators are all instances of this, and suggest items for further consideration. In addition, there is currently no higher-level user interface to FRP beyond the existing Haskell systems, and a special purpose interactive FRP editor would be a very useful addition.

An ongoing project, somewhat separate from what has been described in this paper, is the creation of a formal verifier for Haskell and therefore for FRP. Such a verifier would be an important component of more viable FRP user interface. A further question that we intend to investigate for both MCF and FRP is one that we alluded to earlier: to what extent can the notions of MCF and FRP encompass the broader SMIL 2.0 view of region or channel?

With regard to further work on MCF specifically, the current version is an experimental Java-based system. We wish to move toward a more solid MCF implementation, and intend to combine this work with our ongoing effort on FRP. Specifically, we are engaged in producing a formal definition of MCF semantics in FRP as a first and important step toward providing a transliterator from MCF to FRP. Given that FRP is a general purpose language and more complex than MCF, we are not intending to translate FRP to MCF, but at the same time work on expressing some FRP constructs in MCF will give us valuable insight on what subset of FRP is most appropriate in this work. We envisage that this in turn will render the task of producing a formal verifier for FRP simpler.

Acknowledgements

The authors are grateful to David Hadaller for his contributions to the HTML+Time version of the game; his work was supported by an NSERC Summer Undergraduate Student Research Award.

References

[1] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, 1977.

[2] James F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[3] Helen Cameron, Peter King, and Simon Thompson. Modelling Reactive Multimedia: Events and Behaviours. *Multimedia Tools and Applications*, 2002. To appear.

[4] Anthony Courtney. Frappe: Functional Reactive Programming in Java. In I.V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages '01*. LNCS 1955, Springer-Verlag, 2001. Available at `http://www.haskell.org/frappe/`.

[5] Conal Elliott. Composing Reactive Animations. *Dr Dobb's Journal*, July 1998.

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

[7] Conal Elliott, Greg Schechter, Ricky Yeung, and Salim Abi-Ezzi. TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications. In *Proceedings of SIGGRAPH '94*. ACM Press, 1994.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[9] Lynda Hardman, Guido van Rossum, and Dick C. A. Bulterman. Structured Multimedia Authoring. In *ACM Multimedia '93*, pages 283–289, 1993.

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

[11] Bent Bruun Kristensen and Kasper Østerbye. Roles: Conceptual Abstraction Theory and Practical Language Issues. *Theory and Practice of Object Systems*, 2(3):143–160, 1996.

[12] Thomas D. C. Little and Arif Ghafoor. Interval-Based Conceptual Models for Time-Dependent Multimedia Data. *IEEE Transactions on Knowledge and Data Engineering (Special issue on Multimedia Information Systems)*, 5(4):551–563, 1993.

[13] Mohammad Nabil, Anne H. H. Ngu, and John Shepherd. Modelling Moving Objects in Multimedia Databases. In *Database Systems for Advanced Applications*, pages 67–76, 1997.

[14] Jocelyne Nanard, Marc Nanard, and Peter King. Media Construction Formalism: Specifying Abstractions for Multimedia Scenario Design. *The New Review of Hypermedia and Multimedia, Special Issue on Time in Multimedia*, 6:47–87, 2000.

[15] Jocelyne Nanard, Marc Nanard, and Peter King. Media Construction Patterns: Abstraction and Reuse in Multimedia Application Specifications. In *Proceedings of Multimedia Computing and Networking 2001*. SPIE, 2001.

[16] John Peterson, Conal Elliott, and Gary Shu Ling. Fran Users' Manual. Available from `http://www.haskell.org/fran`, 1997–2000.

[17] Dirk Riehle. Composite Design Patterns. In *Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '97)*, pages 218–228. ACM Press, 1997.

[18] Patrick Schmitz. Multimedia Meets Computer Graphics in SMIL2.0: A Time Model for the Web. In *WWW 2002*, 2002.

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

[20] Simon Thompson. A Functional Reactive Animation of a Lift Using Fran. *Journal of Functional Programming*, 10, 2000.

[21] Michael Vazirgiannis, Yannis Theodoridis, and Timos Sellis. Spatio-temporal Composition and Indexing for Large Multimedia Applications. *Multimedia Systems*, 6(4):284–298, 1998.

[22] Thomas Wahl and Kurt Rothermel. Representing Time in Multimedia Systems. In *International Conference on Multimedia Computing and Systems*, pages 538–543, 1994.

[23] World Wide Web Consortium (W3C). *Synchronized Multimedia Integration Language (SMIL 2.0) Specification*, August 2001. (Available at http://www.w3.org/AudioVideo).

[24] Yale Haskell group and the Center for Computational Vision and Control. Frob: Functional Robotics. `http://haskell.cs.yale.edu/frob/`, 1998–1999.