Behavioral Reactivity and Real Time Programming in XML

Functional Programming meets SMIL Animation

Peter King
Department of Computer Science
University of Manitoba
Winnipeg, MB, Canada
prking@cs.UManitoba.ca

Patrick Schmitz
Ludicrum Enterprises
San Francisco, CA, USA
cogit@ludicrum.org

Simon Thompson
Computing Laboratory
University of Kent
Canterbury, Kent, UK
S.J.Thompson@kent.ac.uk

ABSTRACT

XML and its associated languages are emerging as powerful authoring tools for multimedia and hypermedia web content. Furthermore, intelligent presentation generation engines have begun to appear, as have models and platforms for adaptive presentations. However, XML-based models are limited by their lack of expressiveness in presentation and animation. As a result, authors of dynamic, adaptive web content must often use considerable amounts of script or code. The use of such script or code has two serious drawbacks. First, such code undermines the declarative description possible in the original presentation language, and second, the scripting/coding approach does not readily lend itself to authoring by non-programmers. In this paper we describe a set of XML language extensions, inspired by features from the functional programming world, which are designed to widen the class of reactive systems which could be described in languages such as SMIL. The described features extend the power of declarative modeling for the web by allowing the introduction of web media items which may dynamically react to continuously varying inputs, both in a continuous way and by triggering discrete, user-defined, events. The two extensions described herein are discussed in the context of SMIL Animation and SVG, but could be applied to many XML-based languages.

Categories and Subject Descriptors

H.5.1 [**Information Interfaces and Presentation**]: Multimedia Information Systems – *Animations*, I.3.6 [**Computer Graphics**]: Methodology and Techniques – *languages, standards*.

General Terms

Design, Standardization, Languages, Theory, Verification.

Keywords

Animation, declarative, continuous, events, functional programming, expressions, modeling, behaviors, SMIL, SVG, time, XML, DOM.

1. INTRODUCTION

Web authors are turning more to W3C language standards as powerful yet simple to use authoring tools. These languages are declarative, providing a domain-level description of both content and presentation. However, when authors need additional capabilities not provided in the language, they are forced to work in an imperative scripting or programming language, such as ECMAScript [6] or Java. Since most content authors are not programmers, this use of script is often awkward. Moreover, modern presentation generation systems, such as [18], rely on the structure and semantics of declarative languages, and often cannot easily integrate imperative content extensions. Similarly, the use of script or code is problematic in data-driven content models based upon XML and associated tools.

In this paper we will motivate and describe a set of XML [9] language extensions that will enhance these language standards. The extensions capture certain run-time dynamic behavior in the XML declarative dictum, and make provision for presentation dynamism in response to continuously-varying (or 'fluid') user input. Thus the behavior of the artifact being specified may change in response to some other behavior. Such dynamic behavior change occurs in two related forms. The first form may be termed *event-based*, and consists of a discrete presentation state change occurring when a particular property becomes true. For example, one might switch audio sources when the signal strength of one has become less than the strength of the other, or one might notify an investor when a stock value has reached a certain level. The second class of application, termed *continuous dependence*, permits fluid output to change in a continuous fashion in response to a dynamic input. To give two simple examples, the position of an image may track the mouse position, or a continuous meter or slider may follow the volume of an audio signal. In general terms, in this type of application, the output behavior is a continuous function of some other dynamic behavior

While it would be possible for an XML user to achieve the effects just described by making use of, say, ECMAScript, our intent is to integrate these added capabilities into the existing XML languages. In this way authors may make use of these features while remaining within the familiar XML programming style. In order to provide this added functionality, we propose two extensions, both inspired by constructions to be found in declarative, functional programming languages, namely:

- attribute values defined as dynamically evaluated expressions,
- custom (or 'author defined') events based on predicate expressions.

The paper outlines these proposed extensions and discusses how they may be integrated into existing XML based languages and implementations. We have chosen to illustrate the effect of these extensions by examples based on SMIL animation, XHTML and SVG graphics, but the extensions could be applied to other XML-based languages.

Conference '00, Month 1-2, 2000, City, State.

The remainder of the paper is organized as follows. Section 2 contrasts the paradigms of declarative authoring and functional programming. Section 3 discusses in more detail the sort of reactive dynamism we have in mind, and contrasts our notions with more traditional event driven reactivity. We support the discussion with a number of use-case scenarios. Sections 4 5 and 6 detail the extensions to XML that we are proposing in order to support this new notion of behavior-driven reactivity. Section 7 discusses our implementation experience and describes approaches towards both a prototype and a production-level implementation. Section 8 presents a number of case studies that illustrate our extensions and their utility to authors in extending the declarative authoring paradigm. Section 9 presents our ideas for future work in the direction of adding functional features to XML languages, and concludes.

2. DECLARATIVE AUTHORING AND PROGRAMMING

In this section we will review both the W3C XML-based and the functional language-based approaches to authoring, and we will then outline in general terms what we feel the former can gain from the latter.

2.1 Authoring in W3C language standards

Many W3C language standards promote a *declarative* approach to defining complex document manipulations. These languages include standards for XML document transformation [23], styling and presentation [3, 10] as well as languages to describe complex multimedia elements such as 2D graphics [15], and timing, synchronization and animation [18]. A declarative language permits the author to create a high-level description that explains *what* is to happen rather than *how* the effect is to be achieved. This low-level realisation is the responsibility of the platform in which the artifact is rendered.

In the XML paradigm, the high-level description is a structure or *element* containing sub-elements and values of various parameters (or *attributes*). Consider a simple example written in SVG and SMIL Animation:

```
<circle cx="20" cy="20" r="100" fill="red">
  <animateMotion dur="5s" from="0,0" to="50,50"/>
</circle>
```

This fragment defines a red circle and a motion animation, moving the circle down and right over the course of 5 seconds. The circle is described by a `circle` element with values for various key attributes: its position, radius and colour; a component element describes how the element is `animated`.

The SMIL 2.0 Animation module provides a small *domain-specific language* (DSL) for describing the animation of properties in a document. The language contains certain primitive constructs (elements) for functions such as changing a property over time or moving a target object along a path, and provides a model for composing multiple animations on a given property. Details of the animation (such as the duration and the property values to interpolate) are specified as attribute values (`dur`, `from` and `to` in the example). The DSL approach has the advantage over the use of script or code that document semantics are machine independent while at the same time machine understandable. Thus, documents can be interchanged among authoring tools and rendered consistently across a range of presentation implementations.

The XHTML+SMIL integration [22] has provided additional experience with animation and adaptation, especially with the flow layout model provided in HTML/CSS. Because the size and position of elements can often be determined only at presentation time (and can vary depending on user preferences), it is often impossible to specify associated values for animation at authoring time. A more flexible definition of animation values is needed; with current technology this dynamic animation can only be achieved using script- or code-based extensions.

SVG integrated and extended SMIL Animation to support dynamism for vector graphics. While this was an important early integration, the model was intentionally simple, and as a result, many animations can be described only with the use of script.

2.2 Authoring in a programming language

An alternative approach to multimedia authoring consists in making use of a programming language [1,2]. Of its nature, a general purpose programming language provides the author with sufficient power to implement virtually anything, but the costs of complexity, lack of portability, and the overhead associated with the implementation of such a language are such that it is by no means clear that use of a programming language is a preferable approach to authoring. Moreover, multimedia authors and designers are generally not programmers. Nevertheless, functional programming languages like Haskell [19] have been shown to provide a suitable substrate for embedding DSLs of various kinds [12], including the Fran [7] system, and the more recent Yampa system [4] for Functional Reactive Programming and Animation. These models provide a powerful and flexible programming environment, but require a level of sophistication well beyond the authoring model of SMIL. Moreover, it is well nigh impossible to integrate programmed models into current authoring tools.

Other approaches to multimedia description can be found in [14] and a comprehensive overview of timing issues in multimedia and computer graphics is given in [16], which, *inter alia* discusses the tension between authoring and programming in the Tbag model.

2.3 Functional Programming

Before continuing the discussion of authoring we give a brief overview of functional programming, since this inspires the extensions we propose below. Functional programming embodies a declarative approach to programming, and so extends the declarative authoring model of the XML-based languages.

A functional program consists of a number of definitions. A program is executed by *evaluation*: an *expression*, built up using the definitions from the program together with built in functions and values, is evaluated to give its result.

As an example, given the definitions (using the syntax of the Haskell [19] language)

```
width = 42; height = 27
perimeter x y = x*2 + y*2
```

evaluation of the expression

```
perimeter width height
```

gives the result `138`. Evaluation proceeds by expanding out definitions and applying primitive operations; the evaluation can be written step by step thus:

```
perimeter width height
  = perimeter 42 27
  = 42*2 + 27*2
  = 84 + 54
  = 138
```

Thus, in a pure functional language such as Haskell evaluation is without side-effects: at each stage equals are replaced by equals. An expression simply stands for its value. For example, the expression `perimeter width height` describes the perimeter of a rectangle with height 27 units and width 42. Thus, expressions are *declarative*. (It is important to realise that in most imperative languages, evaluation of expressions can and usually does change the state of the underlying system; in other words evaluation has *side effects*: this is not the case here).

Such functional languages can form an effective basis for multimedia authoring, as discussed in [2]. In particular, Fran [7] and Yampa [4] are two Haskell-based programming systems in which reactive multimedia artifacts may be created using the concepts of Functional Reactive Programming (FRP) [21].

2.4 When the DSL is not enough

The declarative DSL approach to multimedia authoring provides a number of benefits. A user is presented with a clear model of what can and cannot be achieved. SMIL animation, for example, allows movement along spline paths, but does not allow the motion to be determined by, say, the current mouse position. The motion is expressed in the language of the author, using terms such as *duration* and *extent*, rather than at the level of the implementation engine, which draws images in particular places at particular times.

Here's the problem, though. The language is, by its very nature, limited, and authors will want to express things that the DSL itself cannot. In the case of SMIL animation one might want to

- animate motion from the current position of the mouse to the layout position of some element
- begin an animated figure when it is scrolled into view
- switch between a pair of audio sources based on relative signal strength

An ECMAScript or Java program could be used to generate or modify SMIL content, but the code is non-trivial to write. Moreover, once one works outside the DSL all its nice properties are lost: document structure and presentation is defined by low-level imperative instructions from which it is difficult to reconstruct a declarative description of the intended behavior. Scripting is for programmers, whereas the DSL can be used by a much wider group of authors whose only requirement is knowledge of the domain itself. Furthermore, authoring tools can read and write ("round-trip") a DSL, and can exchange DSL documents between tools, but there is no way tools can reasonably interpret or present an animation description defined in script; authors must become programmers to be effective.

There are two approaches to tackling the DSL/scripting mismatch. The first is to *embed* the DSL in a higher-level language, as discussed in Section 2.2. The second approach is to *extend* the DSL is various ways, consistent with the declarative approach. This approach preserves the ability of domain authors to work in the language whilst extending its expressiveness. In this paper, we adopt the second approach. Specifically, we add two notions, *calculation*, and *event-predicates*. In the case of SMIL these extended features will support the use cases mentioned earlier in this section, as well as many others. Furthermore, these features can provide a general model for the extension of other XML-based languages.

It is worth noting that SMIL Animation was designed specifically to support extension, and that the SVG integration itself includes extensions to support SVG-specific functionality (e.g., the `<animateTransform>` element). Our approach is aligned with the spirit of these standards.

3. REACTIVE SYSTEMS -- EVENTS AND BEHAVIORS

The traditional view of a reactive system models reaction to occurrences of discrete events. For example, each time the `left button press' event occurs, a certain action may take place. This view pertains to the XML language family as well; in SMIL, for example, begin and active end times can be specified to be relative to events that are raised in the document playback environment, including user events such as mouse clicks. The reactivity can be two-way, so that the system can also initiate event occurrences. SMIL makes provision for events raised by media players like a *mediaComplete event*, and events raised by the presentation engine itself such as a *repeat* event. Thus events can also be used inside a system to achieve temporal coordination between its various components.

We propose in this paper that systems should be able to react to continuous behaviors, that is, behaviors which evolve continuously in time. Such behaviors can come in many forms, including audio and video feeds, as well as information from sensors or behaviors exported by other reactive systems. We distinguish two types of reactivity which we wish our XML extensions to support, (a) dynamism in response to *dynamic events*, and (b) *continuous real-time dependence* on a continuously varying behavior. We now discuss and illustrate both of these situations in turn.

3.1 Dynamic Events

A dynamically evolving behavior may generate a sequence of events. These events are dynamic but discrete. Some simple examples include:

- *Thresholds and turning points:* A number of useful case-studies are derived by considering turning points and threshold values in, say, a continuously varying real-valued signal.
  - We may detect when a real-valued behavior passes a certain threshold: when a temperature exceeds 25 Celsius, for instance
  - We may detect the turning points in a real-valued behavior: representing a stock market average; when it starts to go down, we can present an alert to the user.
  - We may detect the turning point in a behavior derived from an external behavior. For instance, from an audio feed (an external behavior) we could derive the volume of the audio signal, which need not be an explicit part of the external behavior. We might then detect crescendi or other sorts of musical features from this derived volume information.

These case studies are further discussed in section 8, where the code for examples such as these in our extended XML is given.

- *Begin-when-viewed scenario:* Consider a long scrolling document with figures that are animated to illustrate concepts in the accompanying text. Each animation is to begin only when the particular figure is scrolled into view (either directly with scrolling UI, or indirectly via hyperlink scrolling, etc.).
- *Multiple dependence* The event in question may depend on more than one input behavior. Thus we might wish to detect when one of a pair of behaviors exceeds the other: at the point when the overall brightness of video feed A exceeds that of feed B we might wish to switch between feeds.

The final example here illustrates an important point. The event of one volume exceeding another is not a property of either of the inputs in isolation; it can only be raised in the presentation – which has access to the two inputs – and not in the systems which produce the inputs.

3.2 Continuous real-time dependence

In this more powerful case a dynamic property *p*, say, of some item *A*, say, is a function of (depends upon) the value of a dynamic property (or values of a set of dynamic properties) of some other item(s). Thus the property *p* must be continuously re-evaluated, and *A* redisplayed in accordance with the continuously changing *p*-values. Some simple examples include

- The background colour of a webpage may be a function of, say, *temperature* as measured by some sensor; the hotter the temperature the redder the background;
- The position of an image on the screen may follow a mouse position;
- Extending the stock exchange example given earlier, in addition to the threshold events, we may also display a set of *sliders* each tracing the value of a particular stock
- Many instances arise in video games, and although games are not the primary content on the web, they are a very common application of animation tools, and serve as a measure of the expressiveness of an animation model. Consider a game-like scenario in which a projectile is fired at a moving target. Unlike the course of an arrow, which is fixed when the arrow is fired, a guided missile must track the changing course of its target in flight.
- Several other examples of real-time dependence, complete with the corresponding extended XML code, appear in section 8 of this paper.

3.3 Continuously-varying values: behaviors

These examples illustrate the usefulness of what we have termed *behavioral reactivity*. In order to extend XML to incorporate these features we need two separate classes of language extension: an *event mechanism* and the ability to *evaluate expressions*. We need an extended event mechanism in order to handle the condition within the artifact, so that the artifact may properly react to the condition which has arisen. We need the ability to evaluate expressions of various types, both in order to determine when a condition such as those we have described arises, and so that the dynamic properties of behaviors can be appropriately evaluated.

The discussion in Section 2 gave an overview of declarative authoring and programming. Expressions were shown to denote values. It is not difficult to generalize this model to values which vary continuously with time. If the value `mouseY` gives the vertical coordinate of the mouse, then the expression

```
mouseY + offset
```

will itself vary with time. At each point in time, the value will be calculated by adding the (instantaneous) values of `mouseY` and `offset`, if indeed the latter varies in time also. Formally + is *overloaded* to work over time varying values as well as over simple numbers. We assume that such properties (DOM-only, or attribute values) are exposed on some element, and may be changed, either by targeted server updates, by some plug-in extension – such as a web-services client – which has access to and manipulates the property. These continuously varying values we term *behaviors* and behaviors and the expressions denoting them form the basis of our proposals. We discuss these two extensions, expressions and events, in the next sections of the paper.

4. EXPRESSIONS

An expression language forms the basis for our dynamic attribute values and our event predicates. In the above example of the background colour, we calculate, say, the *(R,G,B)* values based upon the numerical value of *temperature*, whereas in the above *turning point* scenario we define an event predicate as a Boolean expression which uses the values of the volume from the external audio feeds. The first case makes use of a simple arithmetic expression, while the second instance demonstrates a more complex Boolean combination of simpler expressions, and illustrates the value of a fully featured expression language.

The syntax for the expression language follows the syntax adopted for timing expressions in SMIL 2, combining property references of the form `{element}.{property}` with arithmetic operators and constants. SMIL 2 defines this syntax to allow timing values to be defined relative to other timing values and events, and so it is a natural extension for us to use this syntax to define animation values in terms of other animated values and object model properties.

We considered an XPath syntax for property references, but rejected it for several reasons:

- As defined, XPath allows references to XML attributes, but not to object model properties (e.g. CSS-OM values).
- Use of XPath would be inconsistent with SMIL 2
- Authors transitioning from script or code-based animation to our declarative syntax will be more familiar with the proposed syntax; choosing XPath could hinder learning and adoption.

A complete definition of the current form of our expression language is to be found at the following URL: http://www.cs.kent.ac.uk/people/staff/sjt/PDXML/Expr.htm; here we concentrate on some of its more significant features and omit most details of syntax.

4.1 The typing mechanism

The expression language provides three types: numeric, string and Boolean types. The expression language is *typed*: if an operator is applied to an operand of an incorrect type, then the value *undefined* is returned. Moreover the language is *strongly typed*: all types can be computed and verified prior to presentation. Furthermore, while there are several explicit type conversion functions in the expression

language, there are no *automatic* type conversions (coercions) in the model, and in particular, therefore, there is no automatic conversion between the numeric and Boolean types. This is consistent with SMIL Animation, which requires animation values to be legal values for the animated property, and with DOM and CSS-OM, which define strongly typed interfaces. We believe that most authors will find such a type safe model more natural and less error prone: in the following fragment in which '-' has been mistyped as '<', the author would prefer to have the expression yield the value *undefined* (causing the animation to have no effect), rather than to have a Boolean quietly coerced to 0, causing the animation to behave in a subtly incorrect manner:

```
<animate from="calc(a+b)" to="calc(a<b)" …/>
```

4.2 Types and operators

4.2.1 Data types

Our choice of data types is motivated to a very large degree by the application domain, and is developed from concepts found in functional programming languages adapted to the declarative model found in XML. Floating point numbers are needed as they are widely used when computing the evolution of animated values. Booleans (`true` and `false`) are needed for use within events and predicates. Strings are used to convey information, and in a dynamic context it will be necessary to compute strings, (or at least to choose from among alternatives). For example, a different string might be generated according to the position of an object on a web page ('top' or 'bottom'). String literals are enclosed between single quotes (since double quotes are used to delimit XML attribute values).

4.2.2 Operators

While SMIL timing expressions allow only `+` and `–` operators, our set of operators is extended to be consistent with the tradition of functional expression languages, and includes the typical unary and binary, arithmetic, relational and Boolean operators. The Boolean operators for conjunction and disjunction are lazy: if their first argument evaluates to `false` (respectively `true`) then this value is returned without evaluating the second argument. In addition, we have provided a C-style ternary conditional operator, denoted "`?:`". If `b` is `true` then `b?e:f` yields `e` as the result; otherwise `f` is the result.

4.2.3 Language Functions

We provide a repertoire of numeric functions to supplement the basic arithmetic operators. The functions are of four types:

- simple numeric functions: *abs, max, min, floor,* etc.
- functions that return Boolean values: <, <=, etc.
- mathematical and geometrical functions: *cos, tan,* etc.
- environment functions: current time, etc.

The choice of functions given here represents a core of general functionality likely to be required across all application areas. The choice is not intended to be definitive or closed; language designers who integrate this module may extend the language to include functions relevant to their particular domain. We expect that implementation techniques will extend to these domain specific elements in a straightforward way.

Our model provides no facility for author-defined functions. This important constraint greatly simplifies the authoring model, and also provides a measure of 'safety' for the implementation (ensuring, for example, that expressions used with animation can be quickly evaluated at each animation sample). Our goal was to provide flexible expressions, not a programming language.[1]

4.2.4 Domain-specific values

In addition to the broadly applicable interfaces for attribute or property reference in DOM, CSS-OM et al., each language integration defines a specific or extended domain. The SVG DOM, for example, defines a set of properties and the rules for animating these. Each domain using our expression extensions will include a set of properties that expose OM (Object Model) values in a manner convenient for use in expressions. For example in SMIL Timing integrations, properties such as the current simple time or a Boolean *isActive* would likely be provided. Another example is the ad hoc extensions to the HTML DOM in wide usage by authors. Although not (yet) standardized, a shared subset of extensions is supported by major commercial browsers, and provides crucial application-dependent values such as `clientWidth` and `scrollTop`.

One set of properties we see as common to many applications exposes the mouse position in a simple manner. We describe *mouseX* and *mouseY* properties, exposed on all elements that can raise a mousemove event. The actual values follow the definition given in [5] for mousemove events, returning the position of the mouse relative to the container (which in turn is language specific). Expressions can reference these mouseX/Y properties on the root layout element (e.g. "body.mouseY") to get "global" mouse positions, or on a particular target element to get "local" mouse positions.

In our formal syntax, all domain specific properties must be defined in the DOM for the integrating language. We expect the inclusion of expression functionality to motivate richer DOM interfaces of this sort.

5. CALCULATION

We apply the expression functionality to animation attributes, allowing the values that describe the animation function to be expressed as calculated expressions. This approach provides more expressive power to authors, greatly increasing the range of animation use-cases that can be expressed, and also allows dynamic documents to be adaptive, in that animation function values can be defined in terms of other document properties that are computed or may change in response to user actions.

Expressions may be applied to any of the attributes used to describe the animation function values. This includes `from`, `to`, `by`, and `values`, as well as `path` for `<animateMotion>`. The expressions are called out for the parser with a reserved prefix (`'calc'`) and enclosing parentheses, similar to CSS functional notations, and to SMIL timing expressions like '`wallclock()`'. A `calc()` expression may also specify calculation frequency; we return to this point in section 5.1.2 below. As in SMIL Timing, the reserved prefix can be escaped for unusual authoring cases.

For example, to 'zoom' a box from the current size up to 80% of the main container width, we specify:

---

[1] The decision to impose this constraint was based in part upon private discussions we had with the MSIE development team regarding issues with their "dynamic property" functionality.

```
<animate attributeName="width" dur="5s"
    to="calc(main.width*0.8)" .../>
```

For target attributes that take simple scalar values, the result of the calculated expression must be a legal value for the specified attribute. Vector-valued attributes (e.g. position or transforms) are supported using the vector syntax of the `attributeType` domain, but allowing calc-values for each constituent of the vector value, as in the following example. To 'fly' an object from the right edge of a button to the position of a content container, we specify:

```
<animateMotion dur="5s"
    from="calc(btn.x+btn.width),calc(btn.y)"
      to="calc(content.x),calc(content.y)" …/>
```

An interactive example (using XHTML+SMIL) tracks 'tooltip' text with the mouse, and sets the tip string to indicate whether the mouse is on the upper or lower half of the main container:

```
<p>
  <t:set attributeName="left"
         to="calc(main.mouseX+20)"/>
  <t:set attributeName="top"
         to="calc(main.mouseY-5)"/>
  <t:set attributeName="innerHtml"
         to="calc((main.mouseY>(main.height/2))?
                  'Lower':'Upper')"/>
</p>
```

The continuous evaluation of the expression yields mouse tracking motion for the tooltip, as well as dynamic change to reflect which half of the page the mouse is in.

5.1 Computation model

In its simplest form, the computation of expressions is performed using a stack calculator with a few built-in functions and value references. However, the resolution of the references to Object Model values introduces two key questions:

- Which value for a property should be used?
- When should the referenced value be sampled? That is, when and how often should we re-calculate the expression?

5.1.1 Resolving OM value references

There are three possibilities for the type of values to use in value references:

1. the **author-specified** value,
2. the **computed** value (e.g. CSS OM computed-style property values),
3. the **animated** value (e.g. `SVGAnimatedNumber animVal` values).

We conducted a number of experiments and considered a broad range of use-case scenarios. We concluded that specified values are rarely useful in practice and could be ambiguous for things like CSS properties in which the value could be specified in many different ways. We note that the use of computed values may be appropriate in applications outside animation, e.g. for property values in CSS or XSL stylesheets. In our application domain however, where the values are used in the specification of animation functions, we concluded that the use of animated values would make the most sense to authors. Thus, when a referenced property is the target of animation(s), the animated value is used in the expression; when the property is not animated, the calculated value is used.

5.1.2 Expression calculation frequency

We describe the sampling rate for referenced values as the *calculation frequency* of the expression, and have identified four distinct models of when evaluation takes place:

1. once at parse time, for values that are effectively constants (e.g. user-agent window size),
2. after layout is complete, for values that depend upon styling and layout (e.g. position of an inline element),
3. each time an animation begins,
4. each time an animation is sampled.

For applications to other domains such as CSS and XSL property specification, only cases 1 and 2 apply. However, even in these domains there is the issue of handling changes to the referenced values (e.g. if script changes a value, or if user interaction forces a re-layout). Such changes should cause the engine to re-compute the expression that uses the values. But in the context of animation, the question then arises of whether the author wants an animation to update midstream, or prefers that it use the value it 'saw' when the animation began. To illustrate this dichotomy, consider these two variations of the 'arrow/missile' scenario:

Launching an arrow at a moving target. When the arrow is launched, it is aimed at the current position of the target. But once launched, it cannot change its course; further motion of the target has no effect upon the arrow.

Launching a guided missile at a moving target. A guided missile is aimed just as the arrow would be, but it also tracks the target as it flies, and adjusts its motion accordingly.

Both use-cases could be expressed using syntax like:

```
<animateMotion to="calc(target.x),
                   calc(target.y)" …/>
```

In the first case we need to specify that once calculated, the `to` value should remain fixed, while in the second case the `to` value should be re-calculated on each sample.

In order to efficiently re-compute dependent expressions when a given value changes, we model references to other values using dependency-relation graphs (this can be compared to cache maintenance). For a sampled animation, there is no point in re-calculating more often than the animation is sampled, and so a change to a referenced value just marks all dependent expressions as *out-of-date*; the animation engine then re-calculates the expression at the next sample[2].

When we reconsider calculation frequency assuming the dependency graph model is also in place, we can collapse the cases

[2] Dependency graphs can chain, as expressions reference values that are animated in turn by animations defined with expressions. As a dependent value marked "out-of-date", it in turn marks any expression "out-of-date" that references the animation target value. Naturally, cycles in the graph must be detected and broken (just as for SMIL Timing references). There are further optimizations that take into account the semantics of animation composition. Also, the traversal order of the animation tree may generate forward references to animated values, and so update of the "cached" expression values is slightly more complex than described. Nevertheless, these principles of optimized computation (cache maintenance) still apply.

for frequency models 1, 2 and 4 into one case; for this, we re-compute an expression every time we sample the animation graph, but if (and only if) a referenced value has changed. Cases 1 and 2 will change infrequently, but are covered by this simple rule. Case 3 is then distinct in that it *ignores* changes to referenced values once an animation has begun.

To provide authoring control over this behavior, `calc()` expressions may explicitly specify the desired calculation frequency as a second parameter. Allowed values are `always` and `atStart`; `always` is the default[3] and has been used in all the illustrations thus far. The arrow use-case is specified as follows:

```
<animateMotion to="calc(target.x, atStart),
                   calc(target.x, atStart)"…/>
```

The guided missile case could either specify `always` or just use the default semantics.

6. EVENTS AND PREDICATES

In many animation use-cases, we need to know when a certain condition is true, and to take action in response. Object models typically provide a set of events to indicate a range of interaction conditions (e.g., mouse events) as well as document conditions (e.g., media download and mutation events). These can be used declaratively to bind actions to the events - e.g., in SMIL, to begin or end an animation when an event occurs. However, there is no means for the author to declare and name new events specific to the document content. Authors are forced to resort to code, and the implementation of conditional events is non-trivial even for programmers.

High-level languages for simulation, games and concurrent programming support the definition of conditions and associated events, albeit programmatically – outside the domain of XML authors. Early drafts of the event syntax of XML [8] included a step in this direction, supporting declaration of a new event based upon existing events, with timing constraints when integrated with SMIL. This functionality was removed in later drafts.

We define an XML syntax that leverages our expression support to model author-declared events. Events are generated from Boolean expressions; when this expression (or *predicate)* evaluates to *true*, an event is raised on a target element (following the model of [5]). This is inspired by the Fran event model [7]. For example, an author could define an 'enterView' event that indicates when an image appears in the current user agent window (e.g., to note when a user scrolls a figure into view). In an XHTML integration, the following code would declare the event and would specify the condition on which the event is raised or fired (for simplicity here, we leave namespace issues to the language integrator):

```
<event target="img1" type="enterView"
    predicate="(img1.top + img1.height) <=
         (main.scrollTop + main.scrollHeight)" />
```

The `target` attribute indicates (as an ID-REF) the element on which to raise the event; `type` declares the event type for binding references; `predicate` is an expression as defined in section 4.

[3] In other applications domains where timing does not play a central role (e.g. for property definition in CSS or XSL stylesheets), the distinction is meaningless and so this syntax option need not be supported - the default behavior of "always" will correctly apply.

The event will fire once as soon as the condition is true (or as soon as the document loads when the condition is initially true). It will not fire again unless the predicate is *reset*, either because:

- the predicate changes to *false*.
- the event element itself resets, e.g., in integration with SMIL Timing when the element restarts.

To react to the event, an animation can be defined to begin or end in response to the `enterView` event, using SMIL 2.0 syntax.

Note that the calculation frequency for event predicate expressions is fixed — by definition — to be `always`. We considered an additional attribute to preclude an event being raised more than once. In integration with SMIL, this may be unnecessary as a similar semantic is provided by SMIL Timing. If the integrating language allows the `<event>` element to support SMIL timing, events are only raised when the element is active (between the begin and end times); the author can then leverage the SMIL `restart` attribute to ensure that the event is raised at most once.

Some common use-case scenarios for event predicates include collision events, limit-conditions (when a property goes above or below a certain threshold) and state modeling (relating the values of a set of properties).

7. IMPLEMENTATION EXPERIENCE

Three different approaches to the implementation of our XML extensions, each with differing goals, are being pursued, and in this section we outline these methods; a full discussion of implementation details is beyond the scope of this paper.

7.1 A prototype for expressions

We developed a prototype implementation for our expressions, leveraging the MS Internet Explorer 6 support for XHTML+SMIL. The extensions were developed using the IE *behavior* mechanism. Our behavior located `calc()` expressions in the attribute values for animation elements, and then evaluated the expressions using the JScript engine (since our syntax can be easily mapped to a subset of ECMAScript). The animation attributes were then set via DOM interfaces to the resulting expression values, replacing the "`calc()`" strings. This works in part because the "`calc()`" strings are illegal values for the animation attributes, which causes the animations to have no effect (until the behavior provides legal expression result values). This first version was not unlike the support in IE for *dynamic CSS properties* (an inspiration for our extension), but applied to animation attribute values.

The next step was to refine the behavior to parse the expressions in the behavior, implementing a stack calculator and modeling the dependency graphs using property mutation events (provided in the DOM). Unfortunately, the IE implementation does not raise property change (i.e. mutation) events for animated CSS properties. We added a brute force work-around to get notifications, but the propagation of changes through dependent expressions sometimes lags by one sample. A native implementation would resolve this.

Since we cannot inject our behavior code into the animation sampling traversal in IE, we cannot always optimize expression calculation (cache maintenance) to only recalculate once per sample. Also, without access to the animation composition engine, we are not able to optimize the dependency graph (e.g., ignoring dependent expression changes for an animation element A when a higher priority, non-additive animation B cancels or overrides the

effect of A). A more robust and better-optimized version of this code could be developed in an open-source implementation, such as the Batik implementation of SVG. We are currently exploring this approach.

7.2 Implementation by transformation

Our second implementation approach, still to a certain extent a prototype, is to transform a document written in our extended XML into an equivalent program in a language which provides all of the timing semantics which we need and for which a reasonably robust processor (interpreter or compiler) exists. The particular language we have chosen in this respect is Yampa [4], which, as mentioned earlier, is a Haskell-based domain-specific embedded language that uses the concepts of Functional Reactive Programming (FRP). Since Yampa embodies a fully-fledged programming language, Haskell, expression evaluation is, of course, given. Additionally, Yampa supports dynamic behaviors of the type discussed in section 3.3, and provides for the dynamic creation of discrete events, as discussed in section 6. Therefore, although XML and Haskell/Yampa differ considerably in their syntactic form, at the semantic level all of the timing and reactivity that is found in, say, SMIL and all of the additional event and behavior mechanisms discussed in this paper, can be readily expressed in Yampa. Thus, this approach to implementation requires two activities:

1. specification of the semantics of our extended XML, notably the timing and the reactive aspects, in their Yampa equivalent; this paper exercise has been completed;
2. provision of a transformation from extended XML to Yampa,

Since a document written in our extended form of XML is valid XML, the transformation to Yampa syntax may be specified in XSLT and an existing XSLT processor used to effect the transformation. It should be emphasized that this transformation need address only syntax; the transformation considers the source document in extended XML and the target document in Yampa as character strings with no inherent semantics. It is only when the target string is processed by the Yampa interpreter that the semantics of the constructions within that target string become significant.

The advantage of this approach is that it is a relatively simple one to implement, and will be of great use as a prototype and as our extended XML is further developed. Its major drawback is one of efficiency, and in particular the lack of optimization of animated attribute value references.

7.3 Towards a production-level implementation

The experience we gained building our prototypes leads us to believe that high quality, efficient implementations of our proposed functionality are entirely feasible. The approach we intend to take towards a production-level implementation will leverage open source browser projects, such as Mozilla and Batik, to which would be added the appropriate timing semantics. This represents future work.

It is interesting to compare the second and third implementation approaches. In the transformation approach, the timing and other semantics are free in the sense that they are provided by the semantics of the target language and by its processor; we need to deal with (just) the syntax of extended XML. In the third approach the syntax is free, and the problem is one of temporal semantics, which will naturally involve considerably more effort. However, the model for dependency graph maintenance is quite similar to that for timing references in SMIL Timing; we expect that much design – if not code – can be borrowed from this. The most complex aspect of the implementation will, once again, be to optimize animated value references.

8. CASE STUDIES

In order to further illustrate the various aspects of our approach, we detail several of the use-cases introduced above. We omit unrelated syntax details to concentrate on our extensions. We have chosen examples which could currently only be implemented by means of scripting in order to reinforce the point that our work makes the work of authors substantially easier allowing them to express complex ideas within a completely declarative framework.

8.1 Thresholds in a real-time stock-watcher

Two common features for a stock watch application are a clear indication of the daily price change and an alert to the user when the current price passes a specified threshold. In our example, a web page has been generated for a user portfolio (e.g. by transforming XML describing the stocks into an XHTML+SMIL presentation). In addition, a small agent connects to a web service to receive real-time quotes for the portfolio stocks, and stores the values in an XML data-island within the page. In the first example of a threshold event, the page defines events for Sell-Alerts when the price for a stock falls below the configured threshold:

```
<event target="stock1" type="belowSellPrice"
   predicate="stock1.price <= stock1.sellPrice" />
<div id="Alert1" begin="stock1.belowSellPrice">
   …{author-defined content for alert}… </div>
```

A related threshold function allows the display of the price to emphasize the movement from the previous close with color:

```
<t:set attributeName="color"
   to="calc((stock1.price < stock1.prevClose)?
             'red':'green')"/>
```

The significant point here is that an agent can update real-time information and the presentation can respond. An authoring tool can provide simple tools to bind a palette of threshold behaviors to actions, animations, etc.

8.2 Threshold/turning points for an audio application

In this example, a music player application manages a playlist of music as well as an audio feed from a VOIP intercom object. The intercom audio is quiet except when someone is *buzzing* the user. We want to switch from the music to the intercom when someone is trying to contact us. Leveraging SMIL timing syntax for interrupt semantics, we can say:

```
<event target="voipObj" type="buzz"
        predicate="voipObj.currLevel > .1" />
…
<t:excl>
  <t:priorityClass peers="pause">
    <t:audio id="music" begin="0"
             src="{playlist}" />
    <t:audio begin="voipObj.buzz" src="…" />
  </t:priorityClass>
</t:excl>
```

A complete implementation would define additional events to switch back to the music, etc.

8.3 Continuous real-time dependence

A common feature in audio UI is a level meter to indicate the intensity of the audio being played. Leveraging a current level property on the audio player from our previous example, we could define a simple meter that tracks the music continuously and uses color to indicate peaks:

```
<div id="VU">
  …
  <t:set attributeName="width"
         to="calc(music.currLevel*100)" />
  <t:set attributeName="backgroundColor"
         to="calc((music.currLevel < 0.9 )?
                  'green':'red')"/>
<div/>
```

This continuous dependence can be extended, making the (RGB values of) the color depend continuously on `music.currLevel`, or indeed on the relationship of the levels of a number of different audio sources.

8.4 Symphony with microphone placement

This example allows music students to explore the different instruments that make up a symphony orchestra, as a piece of music is played. The example assumes an image of the orchestra, an image of a microphone that the user can drag around (it could also be animated) and individual music tracks for each section or even for each instrument. The audio elements are assigned a spatial position corresponding to the image and as the microphone moves, the volume of each audio track is adjusted as a function of the distance from the microphone to the audio position.

```
<img id="orch" src="orchestra.jpeg" …/>
<img id="mic"  src="mic.jpeg" …>
   <t:set attributeName="left" dur="indefinite"
          to="calc(orch.mouseX)" …/>
   <t:set attributeName="top" dur="indefinite"
          to="calc(orch.mouseY)" …/>
</img>
<t:par>
   <!-- Violins are at 100, 250 -->
   <t:audio src="violins.mp3"
      volume="calc(min(15,
              (100-sqrt(pow((mic.x-100),2)
                       +pow((mic.y-250),2)))))…/>
   <!-- Woodwinds are at 200, 75 -->
   <t:audio src="woodwinds.mp3"
      volume="calc(min(15,
              (100-sqrt(pow((mic.x-200),2)
                       +pow((mic.y- 75),2)))))…/>
   <!—And so on for additional instruments… -->[4]
</par>
```

In the example, audio elements play at a minimum volume of 15 (on a 0 to 100 scale), and up to 100 as the microphone position approaches the instrument audio position. The distance is computed using standard library methods for power and square-root functions, and a minimum function is applied to preclude the complete muting of instrument audio. In the same spirit, it would be possible to highlight the score for the closest instrument in a filmstrip view of the full score, scrolling in synchronization with the music.

[4] Some XML-based languages, including SVG, allow user-defined *templates*, which are a form of function definition. We have also examined the possibility of adding a more general form of functions to these languages, as reported in [17].

9. FUTURE WORK AND CONCLUSIONS

Our future work will proceed in three related directions: one concerned with widening our experience with the authoring implications of our extensions, a second concerned with further integration with W3C language standards, and the third with more basic XML extensions to accommodate the scoped ID model.

9.1 Authoring and implementation issues

All of the experience we have to date with use of these proposed language extensions has been with hand authoring of a limited set of use cases. While we do contend that the use cases in the paper are varied and complex enough to justify the utility of these extensions, we nonetheless need to explore further to demonstrate that the extended DSL lends itself to reasonable authoring. We have outlined in section 7 our plans for proceeding from our current prototype implementation(s) towards a full-scale production quality implementation; we are naturally anxious to complete our exploration of such further case studies and the further XML extensions that they may motivate before proceeding towards this level of implementation.

9.2 Language integration and extensions

9.2.1 Integration with SMIL Timing

The currently proposed extensions are deliberately all orthogonal to the timing model. This simplification is appropriate as a first approach, but at the same time we intend to consider whether we can apply `calc()` values to timing attributes so that timing can be computed. What timing attributes would be appropriate? What issues arise with this additional dynamism in the SMIL timing model? What further interesting examples would timing computations allow?

9.2.2 Extensions relating to expressions

Our current extensions allow expressions in the animation attributes `from`, `to`, `by`, `values` and `animateMotion::path`. We wish to investigate the question of how limited is this in practice, and in particular, what other items would it be useful or feasible to animate?

We have also been experimenting with a simple type model, analogous to templates in SVG, in which all types are simple types. We intend to explore the question of how useful would it be to introduce structured values such as tuples and sequences. We wish to investigate whether structured types could be added without going too far away from the declarative XML approach, and too far towards a 'full' programming language.

9.3 Related work and conclusions

The extensions presented in this paper are based upon programming language constructs that have proved their utility in multimedia authoring. We have presented a range of diverse use cases, and we have demonstrated how these extensions can be added to W3C standard languages while remaining entirely within the style and character of XML and still being capable of being processed by existing XML parsers. We have experimented with our extended version of SMIL Animation with many examples, and in each case the extensions have made the coding of the example easier to

achieve and simpler to understand than using, say, script or some other notation external to the DSL.

We believe that our work represents the first approach towards integrating functional language concepts within XML - that is, towards extending XML in this fashion. Other work [20,11] on integrating XML and FP consists of embedding XML structures and types within existing functional programming languages: XML almost becomes a domain specific language, almost the dual of our approach.

Our extensions maintain to a large degree the desirable separation of data from presentation details, advocated by the XML approach. Our extensions all retain an entirely declarative idiom. In point of fact, we are adopting a view of data that mirrors that found in such well-trodden linguistic ground as objects and abstract data types, where a data type comprises both its set of values and a set of operations applicable to that type. Thus, we consider an event, say `hits-wall` and an expression, say `compute-trajectory`, as properties of the type `flying-object`, just as its `shape`, `color`, `position`, etc. are. Furthermore, our main target is animation, in which temporal attributes such as those we have incorporated, are properly to be included among this set of properties.

In this regard, our work differs from that described in [13], which makes use a constraint-based approach to implement a system with somewhat similar goals. However, in [13], the authors are concerned with issues of static adaptation whereas our approach permits dynamic re-evaluation, as values are input or changed during the presentation. Our approach is deterministic; the constraints of [13] are non-deterministic.

We concentrated on integration with SMIL Animation, using XHTML+SMIL and SVG; nevertheless, as we explored the model, we came to see the utility of these tools for a broad range of applications, including expressions for CSS/XSL style properties, custom event declaration to complement the binding facilities in XMLEvents.

The extensions provide considerable power for authoring, but we have resisted all temptation to provide a full-scale programming language, recognizing that skilled multimedia authors are not necessarily (and should not have to become) programmers. Our experience with the prototype implementations has provided valuable insights, and raises additional interesting questions and issues to explore.

Acknowledgements

Dr. King is supported by a research grant from the NSERC of Canada. We wish to thank Lynda Hardman and Lloyd Rutledge for encouragement, feedback and support of this work.

10. REFERENCES

[1] M.C. Buchanan and P.T. Zellweger, Automatic temporal mechanisms, Proc. Multimedia'93, ACM Press, 1993.

[2] H. A. Cameron, P.R. King and S.J. Thompson, Modeling Reactive Multimedia: Events and Behaviors, Multimedia Tools and Applications, Vol 19, issue 1, January 2003

[3] Cascading Style Sheets, level 2, W3C Recommendation 12 May 1998. Available at http://www.w3.org/TR/REC-CSS2.

[4] Antony Courtney, Henrik Nilsson and John Peterson, The Yampa Arcade, ACM SIGPLAN Haskell Workshop 2003, p 7-18

[5] Document Object Model (DOM) Level 2 Events Specification, W3C. Available at http://www.w3.org/TR/DOM-Level-2-Events/.

[6] ECMAScript, third edition, 1999, http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM)

[7] C. Elliott. and P. Hudak. Functional Reactive Animation, ICFP97, ACM Press.

[8] An Events Syntax for XML, W3C Working Draft 12 August 2002. Available at http://www.w3.org/TR/xml-events/.

[9] Extensible Markup Language (XML) 1.0 (Second Edition), W3C. Available at http://www.w3.org/TR/REC-xml

[10] Extensible Stylesheet Language (XSL) Version 1.0. W3C. Available at http://www.w3.org/TR/xsl/

[11] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. ACM Transactions on Internet Technology, 3(2):117-148, 2003.

[12] Paul Hudak, "Building Domain-Specific Embedded Languages", *ACM Computing Surveys* **28A**(4),1996.

[13] K. Marriott, B. Meyer, and L. Tardif. Fast and efficient client-side adaptivity for SVG. WWW 2002 May 2002.

[14] Jacco van Ossenbruggen, *et al.*, "Towards Second and Third Generation Web-Based Multimedia", WWW10, May 1-5, 2001. Available at http://www10.org/cdrom/papers/423/.

[15] Scalable Vector Graphics (SVG) 1.0 Specification, W3C. Available at http://www.w3.org/TR/SVG/.

[16] Patrick Schmitz, Multimedia Meets Computer Graphics in SMIL2.0: A Time Model for the Web, WWW2002, May 7-11, 2002. Available at http://www2002.org/CDROM/refereed/382/

[17] P. Schmitz, S. Thompson and P King, Presentation Dynamism in XML Poster Session, WWW2003, Budapest.

[18] Synchronized Multimedia Integration Language (SMIL 2.0),W3C. Available at http://www.w3.org/TR/smil20/.

[19] S. Thompson. Haskell, The Craft of Functional Programming, Second Edition, Addison-Wesley, 1999

[20] M. Wallace, C. Runciman, Haskell and XML: Generic Combinators or Type-Based Translation? International Conference on Functional Programming, 1999, ACM Press.

[21] Zhanyong Wan, Walid Taha and Paul Hudak, Event-Driven FRP, PADL'02, 2002

[22] XHTML+SMIL Profile, W3C Note 31 January 2002, Available at http://www.w3.org/TR/XHTMLplusSMIL/

[23] XSL Transformations (XSLT) Version 1.0, W3C. Available at http://www.w3.org/TR/xslt