Presentation Dynamism in XML

Functional Programming meets SMIL Animation

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

Simon Thompson
Computing Laboratory
University of Kent
Canterbury, Kent, UK
+44 1227 823820
S.J.Thompson@ukc.ac.uk

Peter King
Department of Computer Science
University of Manitoba
Winnipeg, MB, Canada
+1 204 474 9935
prking@cs.UManitoba.ca

ABSTRACT

The move towards a semantic web will produce an increasing number of presentations whose creation is based upon semantic queries. Intelligent presentation generation engines have already begun to appear, as have models and platforms for adaptive presentations. However, in many cases these models are constrained by the lack of expressiveness in current generation presentation and animation languages. Moreover, authors of dynamic, adaptive web content must often use considerable amounts of script or code, thus breaking the declarative description possible in the original presentation language. Furthermore, the scripting/coding approach does not lend itself to authoring by non-programmers. In this paper we describe a set of XML language extensions that bring tools from the functional programming world to web authors, extending the power of declarative modeling for the web. The extensions are described 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, DOM, function, modeling, parameter, event, SMIL, SVG, time, XML,

1. INTRODUCTION

Web authors are turning more and 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. 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 or Java. Since most content authors are not programmers, this is often awkward.

Modern presentation generation systems, such as [14], 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 [10] language extensions that will enhance these language standards. The specific extensions are inspired by constructions from functional programming languages, and include:

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

The paper outlines these proposed extensions, discusses how they may be integrated into existing languages and implementations and illustrates their effect in examples based on SMIL animation, XHTML and SVG graphics. In this introduction 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 might gain from the latter.

1.1 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 [25], styling and presentation [5, 11] as well as languages to describe complex multimedia elements such as 2D graphics [16], 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 latter, lower-level, description is usually provided by a system program of some sort, such as an interpreter.

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 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

WWW 2003, May 20-24, 2003, Budapest, Hungary.
ACM xxx.

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 [21] 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 author 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. Furthermore, SVG demonstrates some awkward interactions between animation and other useful features such as the template support provided by `<symbol>` and `<use>`. In particular, the model for `<use>` precludes animating child elements, precludes variant animation instances (e.g. varying duration or values per `<use>`), and makes interactive animation awkward (with the shadow element event-handling model).

1.2 Authoring in a programming language

An alternative approach to multimedia authoring, discussed for example in [3], consists in making use of a programming language. Of its nature, a general purpose programming language is guaranteed to provide 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 system for Functional Reactive Animation [8]. 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.

Constraint programming languages [13] also provide a suitable host for multimedia description. A document is described in terms of a collection of constraints on the spatial [20] and temporal [1] layout of the components, and a constraint solver is used to find a presentation that meets all the constraints. Constraints can be seen as a restricted form of logical formula.

More general logics, such as variants of interval temporal logic [2] can also be used to specify high-level presentations as a collection of logical requirements. 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 [17], which, *inter alia*, also discusses the tension between authoring and programming in the Tbag model.

1.3 When the DSL is not enough – adding functionality

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 speed of the movement to be determined by the speed of a mouse gesture. 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; or
- begin an animated figure when the figure is scrolled into view; or
- define a 'template' button with motion animation, and then use instances of the template that vary the button appearance and animation.

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 almost impossible 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, many authors will prefer to work with some authoring tool. 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 the previous section. 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 three notions, calculation, event-predicates and templates. In the case of SMIL these extended features will support the use cases mentioned earlier in this section amongst others; these features can also be seen to 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 the standard.

The remainder of the paper is organized as follows. Section 2 introduces three common use-case scenarios that we use to illustrate our various extensions. Sections 3,4 5 and 6 detail these extensions, section 7 presents experience and issues with a prototype implementation, and section 8 presents some further

examples that illustrate these extensions and their utility. Section 9 presents our ideas for future work in the direction of adding functional features to XML languages, and concludes.

2. USE-CASE SCENARIOS

We describe three primary use-case scenarios that motivate and help to explain our extensions. These were chosen to span a range of common Web content models, and to highlight shortcomings in current content description mechanisms. We refer to these as we discuss the extensions in detail.

2.1 Arrow/Missile scenario

Consider a game-like scenario in which a projectile is fired at a moving target. The course of an arrow is fixed when the arrow is fired, whereas a guided missile can track the course of its target in flight. We want a simple way to describe motion towards a target, for both arrow and guided missile behavior variants.

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.

2.2 Begin-when-viewed scenario

In a long scrolling document, we have figures that are animated to illustrate concepts in the accompanying text. We want each animation to begin only when the particular figure is scrolled into view (either directly with scrolling UI, or indirectly via hyperlink scrolling, etc.). Especially for a longer animation, this allows the document presentation to be 'in sync' with the user as she reads.

2.3 Menu scenario

A common UI component in documents is a navigation menu, composed of buttons that hyperlink to other pages, or begin content within the page when clicked. The menu buttons may be composed of complex graphics, and include roll-over or other interactive behaviors, so that much of the definition of each button is common or shared, and varies only in the details (such as the position, text label, or color). Menus are used across a wide range of documents, in multimedia as well as simple hypermedia.

3. EXPRESSIONS

The expression language we propose forms the basis for our dynamic attribute values and our event predicates. In our 'arrow/missile' scenario, we calculate the projectile motion based upon the position of the target, where in the 'begin-when-viewed' scenario we define an event predicate as a Boolean expression using object dimensions and the scroll position. The first case uses simple expressions, while the second is a more complex Boolean combination of simpler expressions, and illustrates the value of a fully featured expression language.

In defining the expression language we have chosen names and definitions similar to those used in [7]. We have, however, imposed a number of constraints for authoring simplicity and runtime safety. A complete definition of the current form of our expression language is to be found at the following URL: http://www.cs.ukc.ac.uk/people/staff/sjt/PDXML/Expr.htm; here we concentrate on some of its more significant features and omit most details of syntax.

3.1 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 it is *strongly typed*: all types can be computed and verified prior to presentation. Furthermore, there are no coercions (automatic type conversions) between types in the model and in particular, therefore, there is no conversion between the numeric and Boolean types in our model. We believe that most authors will find such a type safe model more natural and less error prone. We contend that 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)" …/>
```

3.2 Types and operators

3.2.1 Data types

Our choice of data types is motivated to a very large degree by the application domain. Numeric types are needed as they are widely used when computing the evolution of animated values. Booleans 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').

The numeric type consists of floating-point numbers and the Boolean type contains the two values `true` and `false`. More specifically, the Number type contains numbers, the special values NaN (not a number) and positive and negative Infinity. Integer and floating-point literals are given in the usual IEEE format: integer literals are (optionally signed) strings of digits, and floating point literals consist of a decimal number with fractional part and an optional integer exponent following the symbol `E` or `e`. Boolean literals are, as usual, defined by the keywords `true` and `false`. String literals are enclosed between single quotes (since double quotes are used to delimit XML attribute values).

3.2.2 Operators

We have included a typical set of 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 "`?:`". The first argument of `?:` is a Boolean; if this evaluates to *true*, then the result of the second argument is returned; otherwise, the result of the third argument is returned. This operator plays the role of the conditional statement to be found in traditional programming languages.

Binary operators have a level of precedence as well as defined associativity (left, right or none). Our definitions are, as far as possible, compliant with IEEE standards; full details appear at the URL given in the introduction to this section.

In accordance with our rule that that expressions are well typed, if an operator is applied to an argument of the incorrect type, then the *undefined* result is returned from the evaluation. We also make the general assumption that if an *evaluated* argument to an

operator has the value *undefined*, then the result of the operation is also *undefined*. In practice, when an expression evaluates to *undefined*, the effect is the same as if an author specified an illegal attribute value. The resulting behavior is defined by the integrating language – in SMIL Animation, for example, an *undefined* result for a `from` or `to` attribute will cause the animation to have no effect.

3.2.3 Language Functions

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

- simple numeric functions whose role is largely to supplement the set of arithmetic operators, including *abs, max, min, floor,* etc.
- functions which return Boolean values, such as *isFinite* etc. that serve to provide the means to add a higher degree of security to the code being produced,
- mathematical functions, such as *cos, tan, sqrt, exp* etc. which are frequently needed in the computation of animation paths,
- environment functions: these functions return information about the current environment, such as the current time.

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 the particular domain. We expect that implementation techniques will extend to these domain specific elements in a straightforward way.

It should be noted that our model provides no facility for the author to define functions for herself. 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.

3.2.4 Domain-specific values

Each domain will have 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.

One set of properties we see as common to many applications exposes the mouse position in a simple manner. The *mouseX* and *mouseY* properties are exposed on all elements that can raise a mousemove event. The actual values follow the definition given in [6] 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.

4. 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 prefix ('`calc`') and enclosing parentheses, similar to CSS functional notations.

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

```
<animate attributeName="width" dur="5s"
    to="calc(body.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 page:

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

4.1 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?

4.1.1 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.

4.1.2 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: Does the author want an animation to update midstream, or would she prefer that it use the value it 'saw' when the animation began? To illustrate this dichotomy, consider the two variations on 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.x)" …/>
```

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.

Now to back up a bit, in practice we generally want to model references to changing values using dependency-relation graphs, so that we can efficiently re-compute dependent expressions when a given value changes (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 need only mark all dependent expressions as *out-of-date*; the animation engine will then re-calculate the expression at the next sample[1].

If we reconsider calculation frequency assuming the dependency graph model is also in place, we can collapse the cases 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 can take an additional parameter that indicates the desired calculation frequency. Allowed values are `always` and `atStart`, with `always` assumed as the default[2]. Thus our arrow use-case is specified:

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

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

5. 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 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 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 [9] 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 [6]). This is inspired by the Fran event model [8]. For example, an author could define an 'enterView' event that indicates when an image appears in the current user agent

---

[1] 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 should in turn mark 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 is further room for 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.

[2] 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.

window[3] (e.g., to note when a user scrolls a figure into view). In an XHTML integration, this could be described using the syntax:

```
<event target="img1" type="enterView"
    predicate="img1.top <= body.clientHeight" />
```

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 3. The event will fire once as soon as the condition is true (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.

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).

6. TEMPLATES & PARAMETERIZATION

6.1 Introduction

The need to create of a number of similar objects from a common template arises naturally in multimedia design, just as it does in other areas of system development. Consider our 'menu' scenario of a web page with a sidebar containing a menu of navigational buttons. The buttons have much in common, but differ in *color*, *position* and *label*. The buttons will have common animation behaviors, such as *highlighting* when the cursor passes over the button. A frequent mode of design is first to produce a correct animation of a single button, and then to copy and modify this animation for the other buttons. Thus, having developed a blue button labeled `Home`, and placed at position (10, 25), an author can use this as a template for the creation of 4 other buttons, with varying colors, individual labels and positioned down the page as a menu. Although this effect can be achieved by explicit copy and modify, such an approach is hard to maintain, and inefficient to express, bloating the description file. A better approach is to define a template with parameters for color, label and the button number (from which the position is calculated). The template can then be instantiated with different property values at each instance.

SVG provides a mechanism for creating elements which may be used as such templates, including `symbol` elements, graphic elements, and so on. Moreover, SVG provides a means for instantiating such elements, the `<use>` element, described in section 5.6 of [16]. An element instantiated in this fashion references some other named element, and indicates that the graphical contents of that element is to be included and/or drawn in place of the `<use>` reference. Thus, using the SVG model, one may copy the definition of a button symbol as:

```
<use xlink:href= "#button"  …/>
```

and thereby may instantiate as many copies are needed.

However, the `<use>` element in SVG has some severe constraints. In particular, the `<use>` element does not enable one to change attribute values when re-instantiating a definition. Thus, one cannot easily change the color attribute from blue to green, or the label attribute from `Home` to `Search`. Because of the shadow DOM model in SVG 1.0, there is neither an Information Set [24] representation of the instance nodes, nor DOM element nodes. As such, there is no way to target animations to nodes within an instance tree – e.g., one cannot target an animation to a color property on an element within the template instance. The SVG model also precludes registering event handlers on nodes within the template instance – this makes it difficult to define interaction and timing on instance nodes. Further, the identifiers appearing within the various instantiated copies of the element are identical across the various instantiations, severely limiting the utility of ID values and references.

By way of contrast, within the more general domain of programming languages and design tools, the notion of the instantiation of such a template and of creating variants when instantiating is commonplace. Support is generally provided by linguistic mechanisms such as object creation and parameterization, and identifier scopes.

6.2 Proposed solution

Our goal is to provide facilities that remain within the declarative idiom of XML. Our proposed extensions are inspired by the SVG `<symbol>` and `<use>` elements illustrated above, but the semantics are different enough that we define new elements `<template>` and `<instance>`.

We then propose a simple mechanism for the specification of formal parameters within `<template>` elements, and the provision of actual parameter values within `<instance>` elements. Parameterization allows for more flexible instantiation, increasing the useful domain for templates and making the language more efficient as more cases can be represented as templates, thereby reducing file size.

Within the template element:

- each (formal) parameter is specified using a `<param>` element and its `name` attribute;
- a default value may optionally be assigned to the parameter using the `value` attribute; omitting this is equivalent to specifying the special value "" (the empty string), and allows *no value* to be specified. When used for an attribute value within the template, this effectively specifies that the language default be used;
- a formal parameter may be referenced anywhere within the template content that defines it; the reference is designated by prefixing the parameter name with the '$' character.

The following example illustrates these notions:

```
<template id="button">
   <param name="color"  value="blue" />
```

[3] A more complex predicate could also consider the height of the image, and fire when the image is fully in view.

```
   <param name="label" />
   <param name="num"  value ="0" />
   <rect id="bg" width="100" height="40"
         style="fill:$color"
         x="10" y="calc(25+$num*(40+5)">
      <text>$label</text>
      <animateColor id="rollover"
          begin="bg.mouseover" end="bg.mouseover"
          attributeName="fill" to="yellow" />
   </rect>
</template>

<instance id="homeBtn" xlink:href="#button">
   <param name="label" value="Home"/>
</instance>
<instance id="goBackBtn" xlink:href="#button">
   <param name="label" value="Go Back"/>
   <param name="num"  value="1" />
</instance>
<instance id="searchBtn" xlink:href="#button">
   <param name="color"  value="green" />
   <param name="label" value="Search"/>
   <param name="num"  value="2" />
</instance>
```

The template describes a generic button with default color, rollover behavior etc. Each instance defines a button in the menu, specifying the label, etc. The position is calculated as a dynamic expression based upon the button number, and shows how we can combine our extension features.

In order to make it possible to refer to each instantiated copy independently, a mechanism is required to associate a local identifier space with each such instance. Support for local id-spaces has two further advantages.

- In the first place, it will enable the use of each instance to be exposed as a true DOM copy, rather than as a shadow copy (as used by SVG); any tools that query the DOM need not be modified to work with instances.
- Further, the presence of local ID name-spaces enables the children to be selected by style sheets, to be targeted by external animations or XMLE event bindings [9], and to be referenced by script.

We have investigated two possible approaches to the provision of such separate identifier spaces. The first is a general solution using structured ID references (e.g. `homeBtn/bg` where the instance introduces a new ID scope, and so `bg` is found as a descendent of `homeBtn`), analogous to that used by a compiler when instantiating objects in a scope-based object-oriented language. In the longer term, the DOM and XML Info Set models will need to address the issues associated with compound documents and fragment transclusion, and may well incorporate such a model for local ID-spaces.

However we do not propose such an approach at this stage, since it would require major changes to existing XML parsers and to the DOM model; in our view changes of such a scale would be inappropriate at this time. For present purposes, therefore, we propose a second solution, which requires no XML parser changes. Our proposal for our template extension translates local IDs and references. The language interpreter will change all local (within the template) ID definitions and local ID references (i.e. ID-REFs to local IDs), inserting the value of the `<instance>` ID as a prefix. Thus we can express the two distinct references to the background `rect` element for the *home* and *search* button instances in the menu example, as in the following animation declarations:

```
<animate targetElement="homeBtn.bg" …/>
<animate targetElement="searchBtn.bg" …/>
```

It should be observed that since dot '.' is a legal ID char, the code created by this scheme will conform to current XML syntax and will function in the desired manner, mimicking the structured ID solution. This simple approach has the advantage of providing a mechanism for separate name spaces within the correct XML framework, and will give us a useful means for experimenting with this facility while making use of existing XML parsers[4].

7. IMPLEMENTATION EXPERIENCE

7.1 Calculated expressions and predicates

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 is a subset of ECMAScript syntax). 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 propertychange (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 hope to explore this route.

7.2 Templates and parameterization

We are currently developing `<template>` and `<instance>` element prototypes, again using IE behaviors. The `<template>` implementation is trivial, and just ensures that the element is removed form the layout and display graphs (setting the CSS `display` property to "none"). The `<instance>` implementation clones the `<template>` content (except for the `<param>`

[4] There is minimal risk of ID clash, if the document includes an element with an ID that matches one of our synthesized ID values. However authors and tools can easily avoid this.

elements), and then replaces parameter references and maps local ID definitions and references. Any "`calc()`" expressions are handled (orthogonally) by the expression behavior described above.

7.3 Observations

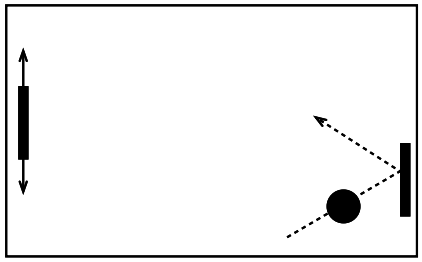
The experience we gained building out prototype leads us to believe that high quality, efficient implementations of our proposed functionality are entirely feasible. The model for dependency graph maintenance is quite similar to that for timing references in SMIL Timing; we expect that much design – if not code – can be borrowed from this. The most complex aspect of the implementation will be to optimize animated value references.

8. CASE STUDIES

In this section we briefly outline a number of other use-cases that we have considered, and indicate for each where our extensions would be of use.

8.1 Pong

This 1970s arcade version of table tennis provides further examples of the mechanisms outlined in the paper. The example was chosen to stress test the extensions we propose, by dint of its open-ended and computational nature.



The aim of the game is for each player to keep the ball in play by controlling the vertical position of his or her paddle. The two paddles are controlled by external user events or behaviors. The SMIL definition of the paddle would be a template with parameters corresponding to the horizontal position and to the events that cause the paddle to move upwards or downwards. There would be two instances of this template, one for each paddle.

The ball continues to move in a straight line – a straightforward animation – until it hits either a paddle or a wall. This impact will be effected by processing an event generated by a predicate over the position of the ball, and the processing of the event will require a calculation of the modification to one of the horizontal or vertical components of the ball's motion.

More advanced effects can cause the ball to flatten a little on contact with the paddles, or one can apply a more complex transformation to the ball on impact with a moving paddle. The players' score can be shown as an animated string, and other attributes – such as the color of the ball – can also depend on the players' performance.

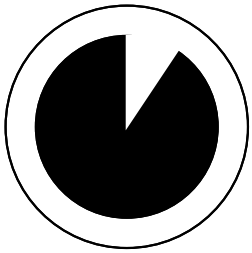
Among the aspects of Pong that we could not easily model were:

- multi-user input – this would require DOM support for the associated input devices, or equivalent.
- the definition of variables for the score values, etc. We considered and rejected several kludge solutions, using extension (a.k.a. *expando*) attributes or numeric properties (e.g., `width` and `height`) on hidden elements.

8.2 Other case studies

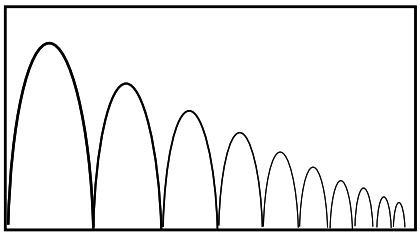
A variety of smaller examples illustrate our approach.

Clocking to show the progress of an operation, registering progress by animating an angle.



This would be implemented by animating the angle through which a geometrical figure is rotated, which can be can be done either by explicit placement of the coordinates of a figure or by animating an SVG-style rotation transformation. The animation progress (i.e., the animated angle value) would be linked with a `calc()` expression to some other animated property (such as a `percentLoaded` property for media).

Decaying bounce. To describe the behavior of a ball bouncing, a degree of damping needs to be applied to the motion on each bounce. This can be achieved by defining the height of the bounce motion as an expression based upon an `iRepeat` value exposed by SMIL timing.



Spatialized or layered audio: We tie the stereo balance to x position of the audio source, or surround sound to an x-y-z position. Alternatively, the mouse position can be used dynamically to determine the balance (mix) between a number of audio channels (sources).

Oscilloscope. An oscilloscope gives a visual display of a sliding window 'snapshot' of an audio signal. A similar display can show a time segment of an evolving floating point value of any sort. A treatment of this example in Fran is given in [4].

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 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.

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? There are interesting questions with respect to our current typing model that we intend to pursue. All types currently are simple types; the question naturally arises of how useful would it be to introduce structured values such as tuples and sequences. We wish to investigate whether such structured types could be accommodated without going too far away from the declarative XML approach, and too far towards a 'full' programming language.

9.2.3 Extensions relating to TEMPLATES

We intend to investigate the role of templates and parameterization in the general XML context. We wish to consider the utility and implementation issues of template hierarchies, analogous to classes, and inheritance in the object-oriented world.

9.3 Scoped IDs in XML

As we have indicated above, our current approach to scoping rules and name spaces is ad hoc. We wish to investigate a more solid model for id-spaces similar to what is available in scope-structured programming languages. While a proper scope model would certainly increase complexity of interpreter and language complexity, but it would be less ad hoc.

We contend that this will be a requirement for DOM in the not too distant future, as more support is developed for document fragment references [22] and transclusion in practice. The model appearing in [23] will not work for many cases, and an alternative will be required (as discussed in [15].

9.4 Conclusions

The extensions presented in this paper are based upon programming language constructs that have proved their utility in multimedia authoring. We have demonstrated how they can be added to W3C languages while remaining entirely within the style and character of XML and still be processed by existing XML parsers. We have experimented with our extended version of SMIL Animation with a number of hand 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 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, and of course the general utility of parameterized templates in XML documents.

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 implementation has provided valuable insights, and raises additional interesting questions and issues to explore.

10. ACKNOWLEDGMENTS

The work of Dr. King is supported by a research grant from the Natural Sciences and Engineering Research Council of Canada. The authors wish to thank Lynda Hardman of CWI Amsterdam for providing encouragement and support during the initial stages of the work.

11. REFERENCES

[1] G.J. Badros and A. Borning. Cassowary: A Constraint Solving Toolkit, 1999

[2] H.Bowman, H. Cameron, P. King and S. Thompson, Mexitl: Multimedia in Executable Interval Temporal Logic, Formal Methods in System Design, to appear, 2003.

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

[4] H. A. Cameron, P.R. King and S.J. Thompson, Modeling Reactive Multimedia: Events and Behaviours, Multimedia Tools and Applications, to appear 2003.

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

[6] Document Object Model (DOM) Level 2 Events Specificatio", W3C Recommendation 13 November, 2000 Available at http://www.w3.org/TR/DOM-Level-2-Events/.

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

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

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

[10] Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation 6 October 2000. Available at http://www.w3.org/TR/REC-xml

[11] Extensible Stylesheet Language (XSL) Version 1.0. W3C Recommendation 15 October 2001. Available at http://www.w3.org/TR/xsl/

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

[13] K. Marriott and P. Stuckey. Programming With Constraints: An Introduction. 1998. The MIT Press

[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] L. Rutledge and P. Schmitz, Improving Media Fragment Integration in Emerging Web Formats. Proceedings of the International Conference on Multimedia Modeling 2001 (MMM01), November 5-7, 2001, Available at http://www.cwi.nl/~media/publications/mmm01b.pdf

[16] Scalable Vector Graphics (SVG) 1.0 Specification, W3C Proposed Recommendation, 19 July 2001. Available at http://www.w3.org/TR/SVG/.

[17] 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/

[18] Synchronized Multimedia Integration Language (SMIL 2.0),W3C Recommendation 07 August 2001. Available at http://www.w3.org/TR/smil20/.

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

[20] Lionel Villard, Cécile Roisi and, Nabil Layaïda, "An XML-based multimedia document processing model for content adaptation", Proceeding of Eighth International Conference on Digital Documents and Electronic Publishing, 14 September, 2000

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

[22] XML Fragment Interchange, W3C Candidate Recommendation 12 February 2001. Available at http://www.w3.org/TR/xml-fragment

[23] XML Inclusions (XInclude) Version 1.0, W3C Candidate Recommendation 17 September 2002. Available at http://www.w3.org/TR/xinclude/.

[24] XML Information Set", W3C Recommendation 24 October 2001. Available at http://www.w3.org/TR/xml-infoset/.

[25] XSL Transformations (XSLT) Version 1.0, W3C Recommendation 16 November 1999. Available at http://www.w3.org/TR/xslt