Presentation Dynamism in XML

Functional Programming meets SMIL Animation

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

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

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

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 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 [JvO], 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 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 (*author defined*) events based on *predicates*,
- parameterized *templates* for document content.

The paper outlines these proposed extensions and discusses how they may be integrated into existing languages and implementations. The full paper elaborates the motivation for the declarative approach, and gives fuller descriptions of the extensions and their implementation. It also presents a set of use cases for our extensions and illustrates their effect in examples based on SMIL animation, XHTML and SVG graphics. The full paper is at **http://www.cs.ukc.ac.uk/~sjt/PDXML/PDXML.pdf**

2. EXPRESSIONS

The proposed expression language forms the basis for dynamic attribute values and for event predicates. Rather than reinventing the wheel in defining the expression language, we have chosen syntax and definitions similar to those used in scripting languages which will be familiar to many Web authors. At the same time, we have imposed a number of constraints for authoring simplicity and runtime safety.

The expression language is typed; there are three types, numeric, string and Boolean and 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.

Sets of unary and binary, arithmetic, relational and Boolean operators are included. There is a fixed repertoire of numeric functions to supplement the basic arithmetic operators. Further, a C-style ternary conditional operator, denoted "`?:`", has been provided. Each domain will have a set of properties and functions that expose 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. In many applications a simplified expression of mouse position may be provided.

Neither the language nor these domain specific functions and properties are intended to be definitive or closed; language designers may modify them as appropriate. However, authors are not permitted to define functions for themselves; this restriction simplifies the authoring model, and provides a measure of safety and efficiency for the implementation.

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

Expressions may be used as any of the attributes used to describe the animation function values including `from`, `to`, `by`, and `values`, as well as `path` for `<animateMotion>`. The expressions are called out to the parser with a prefix ('`calc`') and enclosing parentheses, similar to CSS functional notations.

Example To 'zoom' a box from the current size up to 80% of the page width:

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

One expression may depend on others; we keep track of these dependencies in a dependency graph. Expressions are computed using a stack discipline. However, values may, directly or indirectly, change with time. In our model, values are recalculated when (and only when) component sub expressions change. Re-calculation may not always be desired; whilst the flight of an arrow is not affected by its target moving, a guided missile will change course. Our model allows the choice between such possibilities.

3. EVENTS AND PREDICATES

In many animation use-cases, one needs to take action in response to a certain condition. 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). We define an XML syntax that leverages our expression support to model author-declared events. Events are generated from Boolean expressions (*predicates*); when a predicate evaluates to *true* an event is raised on a target element (following the CSS model).

Example An 'enterView' event in an XHTML integration that indicates when an image appears in the current user agent window (e.g., when a user scrolls a figure into view):

```
<event target="img1" type="enterView"
    predicate="img1.clientTop <=
          (body.scrollTop + body.clientHeight)" />
```

The `target` attribute indicates the element on which to raise the event (as an ID-REF); `type` declares the event type for binding references; `predicate` is an expression as defined in section 3.

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).

4. TEMPLATES & PARAMETERIZATION

SVG provides a mechanism for creating elements which may be used as templates, including `symbol` elements and graphic elements. Moreover, SVG provides a means for instantiating such elements, the `<use>` element, which indicates that the graphical contents of some other named element is to be included and/or drawn in place of the `<use>` reference. However, the `<use>` element in SVG has some severe constraints.

- The `<use>` element does not enable one to change attribute values when re-instantiating a definition.
- Because of the shadow DOM model in SVG 1.0, there is neither an Information Set representation of the instance nodes, nor DOM element nodes. Thus, one cannot target animations to nodes within an instance tree.
- The SVG model precludes registering event handlers on nodes within the template instance, which makes it difficult to define interaction and timing on instance nodes.
- The identifiers appearing across the various instantiated copies of the element are identical, which limits the utility of ID values and references.

By way of contrast, within the domain of programming languages and design tools the notion of instantiating and creating variants of such a template is commonplace. Support is generally provided by mechanisms such as object creation and parameterization, and identifier scopes. We propose a simple mechanism for the specification of formal parameters within `<template>` elements, and the provision of actual parameter values within `<instance>` elements. 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;
- 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.

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 instance. Support for local id-spaces also means that the use of each instance may be exposed as a true DOM copy, rather than as a shadow copy (as used by SVG), and enables the children to be selected by style sheets, referenced by script and so on.

We have investigated two possible approaches to the provision of such separate identifier spaces. The first is a general solution using structured ID references 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 a model supporting such local ID-spaces. This approach, however, would require major changes to existing XML parsers and to the DOM model; we therefore propose a second solution, which provides a mechanism for separate name spaces within the existing XML framework. Our proposal 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 and a '.' as a prefix. 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.

Implementation experience. We have developed a prototype implementation for the features described in the paper, building on the support for XHTML+SMIL in Microsoft Internet Explorer 6. Details of this are given in the full paper.

5. FUTURE WORK AND CONCLUSIONS

Our experience with the prototype implementation has provided valuable insights, and further issues in three particular directions. We intend to widen our experience with the authoring implications of our extensions. We intend to consider further integration with W3C language standards. We wish to develop more basic XML extensions to accommodate the scoped ID model.

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 by writing a number of examples by hand, 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 other notation external to SMIL.

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

6. REFERENCES

[JvO] Jacco van Ossenbruggen, *et al.*, "Towards Second and Third Generation Web-Based Multimedia", WWW10, 2001.