Declarative Extensions of XML Languages

Simon Thompson
Computing Laboratory
University of Kent
Canterbury, UK CT2 7NF
s.j.thompson@kent.ac.uk

Peter R. King
Department of Computer Science
University of Manitoba
Winnipeg MB, R3T 2N2, Canada
prking@cs.Umanitoba.ca

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

ABSTRACT
We present a set of XML language extensions that bring notions from functional programming to web authors, extending the power of declarative modelling for the web. Our previous work discussed expressions and user-defined events. In this paper, we discuss how one may extend XML by adding definitions and parameterization; complex data and data types; and reactivity, events and continuous "behaviours". We consider these extensions in the light of World Wide Web Consortium standards, and illustrate their utility by a variety of use cases.

Categories and Subject Descriptors
I.7 [Document and Text Processing]: Document Preparation – Languages and Systems; H.5.4 [Hypertext/Hypermedia]:

General Terms
Design, Human Factors, Languages.

Keywords
XML, functional, declarative, type, data type, event, behaviour.

1. INTRODUCTION
Many Web authors make use of W3C language standards [10] as powerful yet simple to use authoring tools. These standards promote a *declarative* approach to defining complex document manipulations, permitting the author to describe *what* is to happen rather than *how* the effect is to be achieved. Similarly, functional programming embodies a declarative approach to programming. Our work examines how features from functional programming may extend the declarative authoring model of XML-based languages, in particular addressing situations when authors, needing additional capabilities not provided in the XML language [3], are forced to work outside the declarative dictum in an imperative scripting or programming language. In [6] we showed how *expressions* and user-defined *events* could be added to XML. In this paper we discuss a variety of further extensions, including a generalized type mechanism, expression evaluation, and a more powerful model of dynamic behaviours.

DocEng'07, August 28–31, 2007, Winnipeg, Manitoba, Canada.

2. PARAMETERIZED DEFINITIONS
We present a general mechanism to support template definition and parameterization in XML-based languages. Our proposed mechanisms contrast with those provided by such HTML template languages as Smarty [8], in that we use an entirely declarative means of expression.

2.1 Definitions and Instances
Consider the following SVG/SMIL [7,9] fragment:

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

This defines a red circle and a particular 5 second motion animation. If one requires an animation comprising 100 such circles of varying colours and durations, then in XML one would need to reproduce this code fragment 100 times, making 100 sets of changes to the attributes fill and dur. Our proposed extensions are related to the SVG `<symbol>` and `<use>` elements but with semantics that are different enough that we define new elements `<template>` and `<instance>`. We include a mechanism for *parameterization* allowing for more flexible template instantiation. The following example illustrates these notions:

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

Within the template element: each formal parameter is specified using a `<param>` element and its name attribute. A default value may be assigned to the parameter using the value attribute. Created instances supply values for actual parameters, as in:

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

Here, the y coordinate of the rectangle position is calculated as a dynamic expression using proposals to be discussed in section 4.

2.2 Naming

In order to refer to instances independently, a mechanism is required to associate a local identifier space with each instance. Local id-spaces also enable the use of each instance to be exposed as a true DOM copy, rather than as a shadow copy as used by SVG [7], and also enable the children to be selected by style sheets, to be targeted by external animations or XMLE event bindings [12], and to be referenced by scripts.

We have investigated two approaches to local identifier spaces. The first uses structured ID references such as homeBtn/bg where the instance introduces a new ID scope, and so bg is found as a descendent of homeBtn. A second solution requires no XML parser changes. In this case, the interpreter changes local (within the template) ID definitions and local ID references (ID-REFs to local IDs), inserting the value of the `<instance>` ID as a prefix. Thus, the following animation declarations show the two references to the background rect element for the home and search button instances in the menu example:

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

3. REACTIVITY

In this section we examine ways in which events and continuously evolving behaviours can be defined, to set out the design space for adding them to XML. This marks a major improvement over what can currently be achieved with XML languages. In SMIL [9] one is restricted to reactions based on a limited number of pre-defined events, although the general event description and handling mechanism of XML Events [12] extends this power somewhat. HTML Template Languages do not provide for any dynamic behaviour at runtime, they generate HTML and so cannot extend the DOM. Thus, referring to our example earlier, using, say Smarty, one could readily generate HTML to represent the 100 animated circles, but one could not define a (user defined) event to be raised when, say, two of the circles collide, nor any changes in the animated behaviour which occur upon such an event being raised. It is this wider class of dynamic behaviour that our extensions seek to address.

3.1 Events and Behaviours

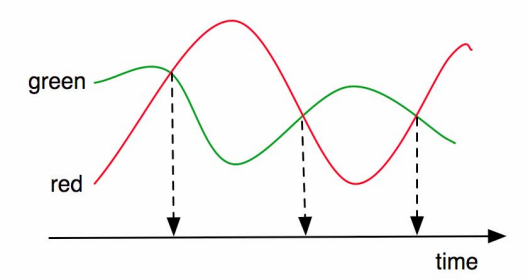
In [6] we outline a proposal for the support of user-defined events, how such events are raised and how they are handled. In the light of recent developments we see that these events can be subsumed under the general event description and handling mechanism of XML Events [12], which views events as atomic, and being initiated externally to the browser. It is possible to build a set of combining forms for events, the most obvious one taking a set of events into a single event which fires when and only when one of the set fires.

We first introduce the notion of behaviours, which were discussed in [6] and were inspired by the Fran model of reactivity [4]. A behaviour is a data value that evolves in time. The (calculated) expressions that we will describe in section 4 are used to define such dynamic behaviours. Motivating examples of external behaviours would include

- a numerical value arising from a sensor in the environment, measuring something like temperature or pressure;
- a tuple of values (R,G,B) representing the colour of an artifact.

Behaviours can also be internal, arising as an artifact of computation, such as a saw tooth function representing the fractional part of the current time.

Composite behaviours can be built from simpler ones. Moreover. behaviours can depend on events, and vice versa. A particular scenario involves two temperature sensors, red and green:



A composite behaviour would, for example, be given by an expression defining the maximum value of the two sensors at any point in time and "lifted" to become a behaviour.

Behaviours can be Boolean, such as the condition that the red value is greater than the green. Boolean behaviours give rise to events, which are triggered by the condition becoming true. For example, an event is triggered when the red graph crosses the green. The firing of this event can trigger other changes: for example, if the sensors represent conditions at two different sites, it is possible to switch between two webcam images of the sites. Similarly, a behaviour may respond to the occurrence of a particular event, such as a mouse button press. The mechanism underlying this uses an event handler in the definition of behaviour. Because of the constraints of space we omit the XML code for these behaviours and events in this scenario.

Thus new 'internal' events are created, as mentioned earlier. The presence of a behaviour in a computed value requires a different computation model on those parts of the document that are dependant on the behaviour. For instance, an image which depends on a behaviour will be an animated image. Further, in [1] we discuss the relationship between finite behaviours, as typically found in SMIL, and infinite behaviours, as found in functional languages.

3.2 Encapsulation

In order to associate events and handlers, we have identified two options which we now describe and illustrate:

1. Events and handlers are combined, as in Fran behaviours [4]

```
<value type="int"
       initial = "345"
       change  = "increment"
       trigger = "foo.click">
```

2. Values and their "handlers" are separated:

```
<value type="int"
       initial = "345"
       handler = "fred">

<handler name = "fred"
         change  = "increment"
         trigger = "foo.click">
```

Given we are adopting the XML Events approach, which separates events and handlers, we adopt the second form.

4. DATA TYPES AND EVALUATION

In [6] we propose an expression language supporting calculation of values of a limited repertoire of types, namely integers, floats, Booleans and strings. Calculations are restricted to expressions formed from a set of built-in operations, and can occur in a limited set of contexts. We now discuss extensions of this earlier proposal in two directions, better support for complex, composite, data types, and broader evaluation of expressions over these types.

Typed functional programming languages such as Haskell [5] provide a variety of structured data types. These types include finite "enumerated" types, records, arrays, collections, disjoint unions and their combination in variant records. These languages also support definitions of collections, homogeneous or heterogeneous, and types may in general be recursive.

While it is possible to express many such types in XML, using schema languages, or using (deeply) nested XML elements as a model of data structures languages, our proposal is to provide a richer set of types than such approaches can achieve, and to provide a more compact and readable XML-based concrete syntax for types and structured values. The design of the particular embedding of a readable syntax for structured data types and values within XML can take a number of directions. Attribute values containing structured XML fragments would violate the XML model. Such structures could be represented implicitly through use of id-refs as names for each layer of the structure; this would be legal XML, at the cost of readability and abstraction. We are still exploring alternatives. In any case, the introduction of such types necessitates both static and dynamic type checking; see Section 5 below for discussion.

The introduction of values and types allows data representing complex objects to be constructed and passed around between elements in a document. In [6] we imposed two essential restrictions. First, computed (or 'calculated') expressions could only occur in a limited, fixed, collection of contexts. Second, the operators and functions that could be used in expressions come from a limited repertoire that explicitly excludes user-defined functions. In the extensions proposed here we remove both of these restrictions, and permit computations on the values of attributes..

5. RELATED AND FUTURE WORK

There is a variety of proposals for user-defined functions in existing XML-based languages, but these are all more limited than our approach. Standard libraries of functions are defined XPath [13], as well as in ECMAScript [2]. XForms [11] permits certain values to be changed, such as forms input (as the user is typing), and slider controls, but this notion of computation is far more limited that our proposals.

XSLT 2.0 [14] allows the definition of functions (as opposed to templates) known as 'stylesheet functions'. A working draft of SMIL 3 [16] (State Modules) includes an expression attribute allowing a pre-defined set of functions from XPath, evaluated at runtime (the evaluation semantics are somewhat unclear). Although these models are more limited than our proposals, we have tried to be reasonably consistent with them.

We also note that what we propose is similar to what is found in some HTML template languages; Smarty [8], for example, uses expressions of the form {$x} where we use calc(x), but again, our proposal has the advantage of remaining entirely within a declarative mode of expression.

Some aspects of our proposed future work in this area have been identified in the foregoing descriptions. In addition, we propose to examine several questions relating to type definitions. Should type definitions be extracted from a document, or should types be defined explicitly? Equally, should types for parameters, functions and so forth be given in an explicit way, or should some form of type inference be used (to the extent that this is possible). Finally, should types be seen as particular fragments of XML schemata, or should they have a separate identity?

In addition, the prototype translator for the authors' current extended XML, implemented at the University of Manitoba, is being extended to handle the additions described here.

We are very grateful to the Royal Society for supporting an incoming short visit to the UK by Dr. King in March 2007.

6. REFERENCES

[1] Helen Cameron, Peter King, and Simon Thompson. Modelling Reactive Multimedia: Events and Behaviours *Multimedia Tools and Applications, 19(1), January 2003.*

[2] ECMAScript Language Specification, 3rd edition (December 1999),http://www.ecma-international.org/publications/ standards/ Ecma-262.htm, *last accessed (l.a.) 17/05/07.*

[3] Extensible Markup Language (XML), http://www.w3.org/XML/, *l.a. 17/05/07.*

[4] Fran version 1.16, http://conal.net/fran/, *l.a. 17/05/07.*

[5] Haskell 98 Language and Libraries, The Revised Report, *2002,* http://www.haskell.org/onlinereport/, *l.a. 11/05/07.*

[6] Peter King, Patrick Schmitz, and Simon Thompson. Behavioural Reactivity and Real Time Programming in XML: Functional Programming meets SMIL animation. In *Jean-Yves Vion-Dury, editor, ACM DocEng 2004,. ACM ,Press 2004.*

[7] Scalable Vector Graphics (SVG) Full 1.2 Specification, http://www.w3.org/TR/SVG12/, *la. 17/05/07.*

[8] Smarty Template Engine, http://smarty.php.net/ *l.a. 14/06/07.*

[9] Synchronized Multimedia Integration Language (SMIL 2.1), http://www.w3.org/TR/SMIL2/, *l.a. 17/05/07.*

[10] World Wide Web Consortium, http://www.w3.org/, *l.a. 17/05/07.*

[11] XForms 1.1, http://www.w3.org/TR/xforms11/ *l.a. 14/06/07.*

[12] XML Events 2, An Events Syntax for XML, http://www.w3.org/TR/xml-events, *l.a. 17/05/07.*

[13] XML Path Language (XPath) 2.0, http://www.w3.org/TR/xpath20/, *l.a. 17/05/07.*

[14] XSL Transformations (XSLT) Version 2.0, http://www.w3.org/TR/xslt20/, *l.a. 17/05/07*

[15] W3C The Forms Working Group, http://www.w3.org/MarkUp/Forms/, *l.a. 14/06/07.*

[16] SMIL 3.0 W3C Working Draft 20 December 2006, http://www.w3.org/TR/SMIL3/, *l.a. accessed 14/06/07*