Declaratively Producing Data Mash-ups

Sudarshan Murthy [§]

Applied Research Group, Wipro Technologies
53/1 Hosur Main Road, Bangalore 560068 India
sudarshan.murthy@wipro.com

David Maier

Department of CS, Portland State University
PO Box 751 Portland, Oregon 97207-0751 USA
maier@cs.pdx.edu

Abstract

Mash-ups extract *data fragments* from disparate sources, and combine and transform the extracted fragments for display. Currently, mash-up developers tend to employ *ad hoc* representations for mash-up data, view mash-ups as applications, and use *imperative scripts* to extract and transform data fragments. These approaches can make mash-up development hard, and the mash-ups' run-time performance poor. We address these concerns with our infrastructure to *declaratively* produce *data mash-ups*. In this paper, we introduce three parts of this infrastructure: *Sixml*, an XML language to uniformly represent a condensed mash-up; *Sixml DOM*, a means to manipulate and reconstitute mash-up parts *on demand*; and *Sixml Navigator*, an alternative path navigator to reconstitute and format a mash-up using *queries* in existing languages. We also present the highlights of an experimental evaluation.

1. Introduction

A *mash-up* [13] combines, transforms, and displays data from heterogeneous sources, likely using only *fragments* of each source's content. A mash-up can be an application that consumes data from different sources, or it can be data that includes content from other sources. This paper focuses on *data mash-ups*.

Consider an application that allows individual reviewers to comment on arbitrary regions of *any* document (not just HTML). In this setting, a *review report* of comments over all documents, along with the excerpt of each commented region, would be a data mash-up because the excerpts come from different documents. Figure 1 shows a review report that includes excerpts from PDF and Microsoft® (MS) Word fragments.

This paper describes parts of our mash-up infrastructure designed to address two problem areas in producing mash-ups such as that in Figure 1: At *design time*, represent, organize, and augment references to a mash-up's source fragments; and at *run time*, easily and efficiently extract the referenced fragments, combine the extracted fragments with augmentations, and transform the combination to different forms (such as a review report).

We have used our infrastructure to build both map and non-map mash-ups. In this paper, we use a non-map mash-up (the review report) for illustration. The tool *Mash-o-matic* [13] uses the same infrastructure to produce map-based mash-ups.

For simplicity, we limit the discussion in this paper to the XML model, but we have used the techniques discussed in the relational model as well.

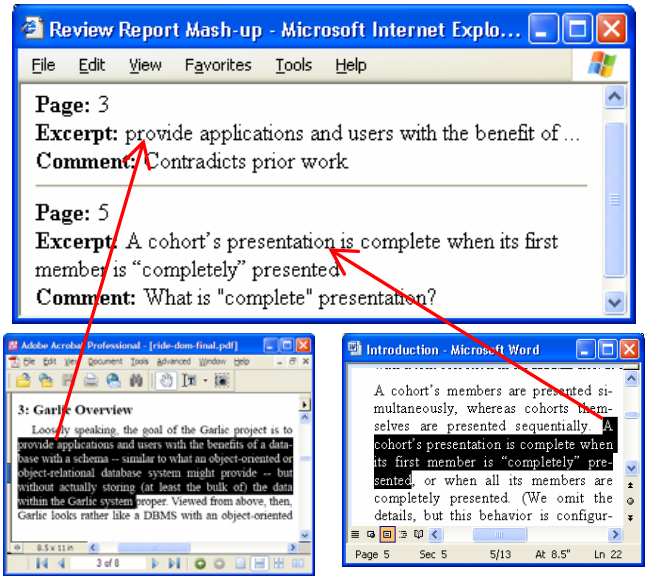

Figure 1: A data mash-up (top) and its information sources

In the rest of this section, we illustrate the problems we address in mash-up production. Section 2 gives an overview of our solution. Sections 3, 4, and 5, describe Sixml, Sixml DOM, and Sixml Navigator (three components of our mash-up infrastructure), respectively. Section 6 presents the results of an evaluation, Section 7 briefly reviews related work, and Section 8 concludes the paper.

Representation issues: A developer should be able to use the schema most appropriate for his application. He

International Conference on Management of Data
COMAD 2008, Mumbai, India, December 17--19, 2008

[§] Work done at Portland State University

should be able to associate any part of the mash-up's data with external sources, and optionally populate parts from external sources. For example, Figure 2 shows a plausible comment structure. In this structure, the developer should be able to indicate that the comment is about an external fragment, indicate that the *run time* value of the attribute `excerpt` is the text excerpt of the commented region, and allow the commenter to reference other external sources in the comment text.

```
<Comment excerpt="">
  Contradicts prior work
</Comment>
```

Figure 2: A barebones comment structure

Currently, there is no means of expressing such structures: XLink [22] provides a means to express links in XML documents, but it does not support *transclusion* [16] (that is, inclusion by obtaining data via a reference to the source). Active XML [1] supports transclusion from web services, but only into elements, not into parts such as attributes.

Run-time issues: A mash-up needs to extract, combine, and transform external data at run time, but the current suite of XML tools do not allow integrated access to referenced fragments. For example, a developer can use the Document Object Model (DOM) [4] or use an XPath expression [23] to access document parts (such as the attribute `excerpt`), but neither DOM nor XPath can automatically assign the excerpt of a referenced fragment to any part of an XML document. Thus, each developer needs to implement the procedures to extract data from heterogeneous sources.

Tasks such as listing comments in order of the page containing the commented regions (or computing the number of comments in each page) require information that exists in the *context* of a referenced fragment, but is not explicit in the fragment. Currently, there is no means of representing and obtaining context information from referenced fragments.

Development issues: If excerpts and context information (such as page number) are available in the XML document, a mash-up can be easily and efficiently constructed, manipulated, and transformed using existing data management techniques. For example, with appropriate changes in representation, comments structured as in Figure 2 can be *declaratively* transformed to the HTML review report of Figure 1 using XSLT templates (such as those shown in Figure 11).

However, developers tend to view mash-ups as applications, and use *imperative client-side scripts* (for example, JavaScript [6] code running inside a web browser) to extract, combine, and transform data. Also, developers frequently implement their own version of common database operators such as sort and aggregate. This approach can increase development effort and hurt a mash-up's runtime performance.

2. Solution Overview

Our infrastructure to produce data mash-ups provides *declarative* solutions to the problems illustrated thus far, and allows developers to fully exploit the XML tools at their disposal. We first describe a conceptual approach to producing mash-ups, and then introduce the parts of our infrastructure.

A conceptual approach: Figure 3 shows the three steps in the conceptual mash-up production process. Dotted arrows indicate data flow and solid arrows indicate control flow. The boxes indicate process steps, likely performed cooperatively by an application developer and an application user. (For simplicity, we use only the term *developer* in the following paragraph.)

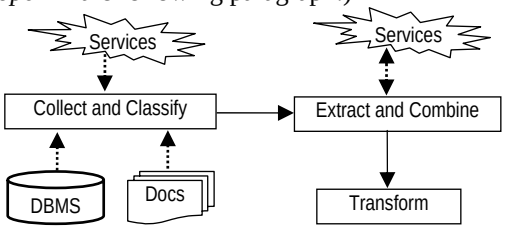

Figure 3: The mash-up production process

In the `Collect and Classify` step, the developer collects references to different data fragments, creates structures over the collected references, and elaborates the collected data. For example, the developer defines the comment structure; a reviewer creates comments, adds references to commented regions, and supplies comment text. A mash-up produced in this step is in a *condensed* form because it does not yet include data from each referenced source.

In the `Extract and Combine` step, the developer *reconstitutes* the condensed mash-up data by extracting the various external data and combines the extracted data with the data added in. In the `Transform` step, the developer *formats* the reconstituted data by transforming it to match display needs. For example, the developer transforms the reconstituted comment data to an HTML review report.

We aim to help a developer easily and efficiently produce each of the three conceptual forms of data mash-ups: condensed, reconstituted, and formatted.

The mash-up infrastructure: Figure 4 shows a reference model for our mash-up infrastructure. Arrows denote inter-module dependency. A gray module is an existing XML query processor. Modules filled with horizontal lines are parts of our infrastructure described elsewhere. The module filled with vertical lines uses our infrastructure. The modules with clear background are our new contributions and are described in this paper.

Sixml (pronounced 'siks-m&l) [10] is an XML language to represent a data mash-up. It provides a uniform means to associate parts of an XML document with external data fragments, including a way to declaratively specify that a part is populated with external data.

We call a reference to an external fragment a *mark* [3]. XML content associated with a mark is *superimposed*

information (SI), because the use of a mark has the effect of overlaying new information (content and structure) on existing information fragments.

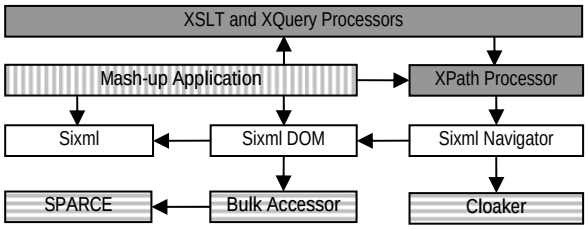

Figure 4: A reference model for the mash-up infrastructure

Sixml is SI represented as XML. A *Sixml document* is an XML document some of whose parts are associated with marks using instances of element types we define. Section 3 describes Sixml.

Sixml DOM is an extension of DOM to manipulate Sixml documents at run time. In Sixml DOM, if the value of a part (such as an attribute and text) of a document is declared to be the excerpt from a mark, it will be so populated, when that part is accessed. The external value is retrieved *automatically* and *on demand* (that is, when the value is first accessed). Section 4 describes Sixml DOM.

The *Sixml Navigator* is an alternative path navigator for use with traditional query processors to combine XML content with data retrieved from marks, and to query the combined *bi-level information* using existing languages unchanged. For example (as shown in Figure 11), with the Sixml Navigator, the review report mash-up can declaratively include the page number of each commented region. Section 5 describes the Sixml Navigator.

Using our infrastructure, a developer prepares a *condensed form* of a mash-up using Sixml; *reconstitutes* the mash-up using Sixml DOM; and reconstitutes and *formats* the mash-up with traditional query processors that use the Sixml Navigator. In our approach, little or no development effort is needed to express the use of marks and to extract data from marks. A mash-up executes more efficiently because external data is obtained on demand and because the amount of interpreted code is lower. (Our components are compiled to executables.)

We now briefly discuss the parts of our infrastructure not discussed in this paper. SPARCE [15] is our middleware to interact with arbitrary mark types. Thus far, we have used SPARCE to support marks of the following types: HTML, PDF, XML, MS Office applications, and several audio and video formats. Support for other types can be easily added. The *bulk accessor* [14] efficiently retrieves excerpts and other information from a large number of marks. The *cloaker* selectively hides parts of data from Sixml Navigator so that certain classes of queries execute more efficiently.

3. Preparing Condensed Mash-ups

A developer prepares a *condensed form* of a mash-up using *Sixml*, our language to represent a data mash-up. This process includes determining the overall structure for the mash-up, and determining which parts of the mash-up are associated with marks and which parts are reconstituted at run time using data obtained from marks.

Encoding how a mark is associated with a mash-up part is a key problem in representing a mash-up. The encoding should be amenable to validation using standard schema constructs, it should not constrain the types of content with which marks are associated, and its serialization should result in mark up that is uniform and comprehensible. One encoding solution is to develop conventions (for example, use comments with specific structure and contents) to encode the association, but conventions cannot be validated using standard schema constructs.

We choose to encode a mark associated with a mash-up part as an *element* because, as we will soon illustrate, an element satisfies all three of the aforementioned encoding needs.

Mark associations: A *mark association* is an element of a type Sixml defines [10] to associate marks with mash-up parts. The mark-association element types are defined using XML Schema [25], and belong to the namespace http://schema.sixml.org. In this paper, we use the prefix sixml with this namespace, but, unless needed, omit namespace information from the main text. Also, for simplicity, we give an instance the same name as its type, and describe the types using only instances.

Marks may be associated with *six kinds* of mash-up content: element, attribute, text, CDATA, comment, and processing instruction (PI). Mark association element types are defined for each of these kinds, but, we present only the types related to elements, attributes, and text.

```
<Comment excerpt="" xmlns:sixml="http://schema.sixml.org"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <sixml:TMark>Contradicts prior work
  <sixml:Descriptor xsi:type="sixml:XPointer">
   <pointer>http://www.w3.org/#element(/1/2)</pointer>
  </sixml:Descriptor>
 </sixml:TMark>
 <sixml:AMark sixml:target="excerpt" sixml:valueSource="true">
  <sixml:Descriptor xsi:type="sixml:SPARCE">
   <Agent>AcrobatAgents.PDFAgent</Agent>
   <Doc location="file://c:/ride-dom-final.pdf"/>
   <Subdoc page="3" startWord="395" endWord="439"/>
  </sixml:Descriptor>
 </sixml:AMark>
 <sixml:EMark>
  <sixml:Descriptor xsi:type="sixml:SPARCE">
   <Agent>OfficeAgents.MSWord</Agent>
   <Doc location="c:\abc.doc"/>
   <Subdoc startChar="45" endChar="53"/>
  </sixml:Descriptor>
 </sixml:EMark>
</Comment>
```

Figure 5: A condensed mash-up represented using Sixml

Figure 5 shows the data of Figure 2 represented as a Sixml segment. It has a mark association each for an element, an attribute, and text content. The element EMark associates an MS Word mark with the element Comment. The AMark associates a PDF mark with the attrib-

ute excerpt (with the help of the AMark attribute target). The TMark associates a mark with the text content it contains. (The variety of marks used in the document serves to illustrate different aspects of our infrastructure. In reality, the elements EMark and the AMark are likely to use the same mark.)

Through the attribute valueSource of a mark association, a mash-up developer declaratively specifies that an attribute's or a text content's value is reconstituted at run-time using marks. For example, by setting valueSource to true for the AMark in Figure 5, the developer declares that the *run-time reconstituted value* of the attribute excerpt is the text excerpt retrieved using the associated mark. However, the run-time value of the text content wrapped in TMark is *not* reconstituted from the associated mark because the attribute valueSource is missing for that mark association. (The default value of valueSource is false.) Section 4 further discusses this attribute.

Figure 5 associates one mark each with an element, attribute, and text, but any number of marks may be associated with any of the aforementioned six kinds of content. (A schema can control the number of mark associations.) For example, another EMark (AMark) added to Comment associates another mark with Comment (excerpt). We omit discussing multiple mark associations with text, but the type definitions online [10] cover that case.

Typed mark associations: In Figure 4, a mark association has the same name as its type, but we provide two means to give the association any valid XML name: Associate a schema (available online [10]) with a mark association, or add the attribute sixml:type to denote the type of the mark association. For example, the value "sixml:EMark" for the new attribute means the association is of type EMark. The type of a mark association that does not use either of these alternatives is suggested by the element's qualified name (as in Figure 5).

Mark descriptors: Each element named Descriptor in Figure 5 describes the external fragment a mark references. This element, called a *mark descriptor*, typically includes information such as the location of an external document and the region (or regions) of interest within the document. Generally, the constituents of a mark descriptor vary according to the linking protocol (such as SPARCE or XPointer [24]) used to identify the external fragment, and also according to the type of information linked (such as PDF and XML). SPARCE is mentioned in Section 2. An XPointer *pointer* identifies regions of an XML document.

A mark descriptor is of the *abstract* type Descriptor. This type is typically derived once for each linking protocol. We do not constrain the internal structure of a mark descriptor. Figure 5 shows instances of the example descriptor types we have defined for SPARCE and XPointer. The attribute xsi:type in each descriptor gives the qualified name of the instantiated type. (XML Schema, not Sixml, requires the use of this attribute.) The elements EMark and AMark use SPARCE descriptors. In these descriptors, the sub-element Agent names the software component (called a *context agent*) that SPARCE uses to access external fragments. The descriptor in EMark references the Characters 45–53 in an MS Word document. The descriptor in AMark corresponds to the span of words 395–439 on Page 3 of a PDF document.

The descriptor in TMark uses the XPointer *element()* scheme to address the second child (the element body) of the document element (html) in the XHTML document at http://www.w3.org/, the home page of W3C.

4. Reconstituting Mash-ups

We now present an overview of *Sixml DOM*, a means of creating, manipulating, and reconstituting (all at run time) a condensed mash-up. Figure 6 shows a simplified class diagram for Sixml DOM. DOM defines the shaded classes and the relationships among those classes.

A developer can use DOM to manipulate a condensed mash-up, but that approach has several disadvantages: DOM does not automatically reconstitute mash-up parts; it requires the developer to be aware of mark-association schemas; and, to create mark associations, the developer would need to know where and how the mark-association elements should be inserted into the DOM tree. Sixml DOM addresses problems such as these without compromising performance.

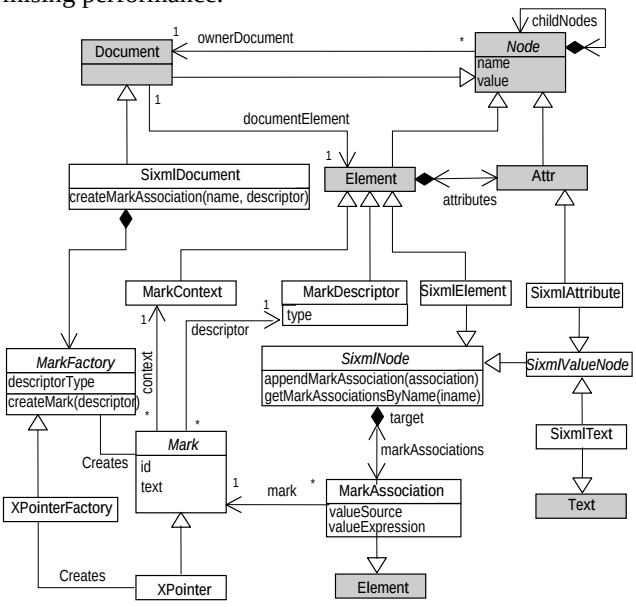

Figure 6: A simplified class diagram for Sixml DOM

Sixml nodes and documents: A node with which marks may be associated is called a *Sixml node*, and is represented by the class SixmlNode. A Sixml node that can contain a value is a *Sixml value node*, represented by SixmlValueNode. Per DOM, nodes of the following types are allowed to have a value: attribute, text, CData, comment, and PI.

Marks may be associated with elements and the aforementioned value node types, but, for simplicity, we

limit this discussion to elements, attributes, and text. The classes SixmlElement, SixmlAttribute, and SixmlText represent these node types, respectively. These classes respectively extend the DOM classes Element, Attr, and Text. In addition, SixmlElement extends SixmlNode; SixmlAttribute and SixmlText extend SixmlValueNode.

The class SixmlDocument extends the DOM class Document. It overrides the factory methods for the types of nodes with which marks may be associated. For example, it overrides the method createAttribute to create an instance of the class SixmlAttribute instead of the DOM class Attr.

Mark associations: A mark association pairs a *target* Sixml node with a mark and assigns a name to the pairing. A node may be associated with different marks using the same name, but a name may be used only once for a node-and-mark pairing. A node may be associated with any number of marks, but the node's schema determines the number of marks with which a node may be associated. The class MarkAssociation represents a mark association.

A mark association has no child nodes. It is attached to a target node, but it is not a child of the target. This relationship between a mark association and its target is similar to the relationship between an attribute and its owner element (as defined in DOM).

A mark association is created using the SixmlDocument factory method createMarkAssociation. The mark association thus created is added to the target node using the method appendMarkAssociation in SixmlNode. Methods to add a mark association at a specific location in the list of mark associations, to replace a mark association, and to remove a mark association are also defined.

The mark associations for a Sixml node can be retrieved using the relationship markAssociations. Mark associations with a specific name can be retrieved using the method getMarkAssociationsByName.

Marks and mark descriptors: The class Mark models a mark created from a mark descriptor (included in a mark association). A mark is created using a *mark factory* chosen based on the type of the mark's descriptor. (As seen in Figure 5, the attribute `xsi:type` gives the descriptor type). Typically, a mark factory class and a mark class are implemented for each linking protocol. Figure 6 shows the mark factory class and mark class for XPointer pointers.

The class MarkDescriptor models a mark descriptor (that is, the element `Descriptor` in Figure 5). Though retrieved from a mark association, at *run time*, a descriptor element does not have a parent. This constraint allows an implementation (if it chooses) to return the same descriptor element when a mark is used more than once in a document.

Mark context: A mash-up occasionally uses information besides the text excerpt of an external fragment. For example, generating a review report in which comments are listed in page order, needs placement information (not text excerpt), for each commented region.

In general, a mash-up may use the text excerpt, page number, font name, and any other information available in the *context* of a mark (that is, in the original setting of the referenced fragment).

What constitutes context information varies across mark types, and even across marks of the same type. For example, an audio mark has *duration*, but a PDF mark does not. An MS Word mark to text in a table has a *row number*; an MS Word mark to text not in a table does not.

We represent context information for a mark as a hierarchy: A *context kind* collects related information (including sub-kinds). Information at the leaf level of a context hierarchy is a *context element*. This hierarchical representation allows the context information for any mark to be modeled as XML. The root of context information is always the element `sixml:Context`, but a mark implementation determines the internal structure and content of this element. Figure 7 shows the partial context information retrieved from the PDF mark referenced by the element `EMark` in Figure 5. The elements `Content`, `Presentation`, and `Placement` denote context kinds. Their sub-elements represent context elements. The text content of a context element is that element's value.

The class MarkContext represents the context information retrieved from a mark. Sixml DOM obtains context information *on demand* from marks. Each mark implementation is responsible for retrieving context information from its marks, possibly using appropriate applications (such as Adobe® Acrobat® for PDF marks).

```
<sixml:Context>
  <Content>
    <Text>provide … system</Text>
  </Content>
  <Presentation>
    <FontName>Times New Roman</FontName>
    <FontSize>11</FontSize>
  </Presentation>
  <Placement>
    <Page>3</Page>
  </Placement>
</sixml:Context>
```

Figure 7: Partial context information for a PDF mark

Reconstituting a node's value: A mark association attached to a value node (such as an attribute and text content) may have the attribute valueSource. A target node's value is reconstituted at run time, if this attribute is `true` for at least one of the mark associations of the node. Specifically, the reconstituted value of a node is the *concatenation* of the values obtained from each of its mark associations for which valueSource is `true`.

The attribute valueExpression of a mark association decides which context value is contributed to a reconstituted node. If this attribute is missing, or is empty, the text excerpt (obtained from the property text of Mark) is the contributed value. Otherwise the value of this attribute is an XPath expression over the context information. For example, the path expression `"Placement/Page"` over the data in Figure 7 contributes the value `"3"`. The element `AMark` in Figure 5 assigns a mark's text excerpt to its target attribute because valueExpression is missing.

Reconstituting a mash-up: Conceptually, a condensed mash-up is reconstituted in three steps. First, the mash-up document is represented as a tree in DOM. This step represents mark associations as regular elements. Second, each mark association element is attached to it target node. Third, each node that derives its value from marks is reconstituted (lazily).

For example, Figure 8 shows the DOM tree (the result of Step 1) for the condensed mash-up in Figure 5. Attribute names are prefixed by the symbol @; content of a text node is placed in quotes. A solid edge denotes a parent-child relationship; a dotted edge indicates a non-child relationship. The mark association elements are shown as children of the document element Comment. Mark descriptors are omitted for brevity. The value of the attribute excerpt is empty because DOM is unaware of mark association semantics.

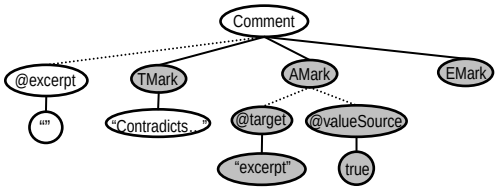

Figure 8: A simplified DOM tree for the data in Figure 5

Figure 9 shows the Sixml DOM tree generated (after Step 2) from the DOM tree in Figure 8. A dashed line connects a mark association node with its target node. The element EMark is now a mark association attached to Comment. AMark is a mark association of the attribute excerpt. The text node previously a child of TMark is now a child of Comment and TMark is a mark association attached to the text node. The reconstituted value of excerpt (after Step 3) is shown partially.

The white nodes in Figure 9 represent reconstituted data. The gray nodes represent mark associations and their content.

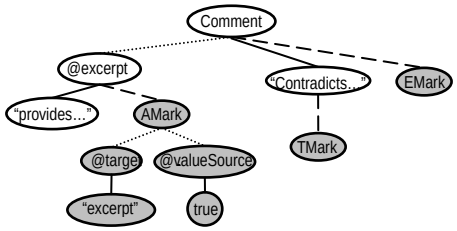

Figure 9: A simplified Sixml DOM tree for Figure 5

Serializing a reconstituted mash-up: Serializing a reconstituted mash-up document condenses the document. Sixml DOM uses a special serializer to serialize a document, because mark associations (for example, the gray nodes in Figure 9) are not visible to the DOM serializer.

The Sixml DOM serializer can place the elements EMark and AMark anywhere in the list of the children of the parent element, but it must preserve the order of the associations within a target node. For example, EMark and AMark in Figure 9 may be serialized in any order, but the order of the mark associations of the attribute excerpt must be preserved. The tree ordering of mark associations is necessary for a reconstituted node because the value of such a node is the concatenation of the string values obtained from its mark associations (and string concatenation is not commutative.)

To ensure proper serialization, the Sixml DOM serializer always writes mark associations for a target node in tree order. Also, when serializing an element, it first serializes all child nodes (as in DOM) and their mark association, followed by the mark associations of each attribute in tree order, followed by the mark associations of the element itself. For example, Figure 5 is a serialization of the Sixml DOM tree in Figure 9.

Using Sixml DOM: A developer can manipulate both XML and Sixml documents using Sixml DOM, because Sixml DOM supports the complete DOM functionality and it exposes the complete DOM interface. Also, he can open a condensed mash-up using Sixml DOM, but access the reconstituted parts (for example, the white nodes in Figure 9) using only the DOM interface. The developer needs to use the Sixml DOM interface only to explicitly retrieve mark descriptors and context information.

The procedure WriteComment (to print comment details) in Figure 10 illustrates the ease with which a mash-up can be manipulated and reconstituted using Sixml DOM. The parameter c is of type SixmlElement so that mark context can be explicitly accessed. Line 2 uses the Sixml DOM interface to retrieve context information from the first mark associated with the input comment. Line 3 retrieves the context element named Page, and Line 4 prints the page number. Line 5 uses the DOM interface to retrieve the reconstituted excerpt of the commented region, and Line 6 prints the added comment text. The execution of this procedure can be traced starting with the node Comment in Figure 9, and using the context information in Figure 7.

```
1. procedure WriteComment(SixmlElement c)
2.   XmlElement ctxt = c.markAssociations[0].context
3.   XmlNode page = ctxt.getElementsByTagName("Page")[0]
4.   Writeln("Page: ", page.firstChild.nodeValue)
5.   Writeln("Excerpt: ", c.getAttribute("excerpt"))
6.   Writeln("Comment: ", c.firstChild.nodeValue)
```

Figure 10: Printing reconstituted comment details

5. Formatting Mash-ups

In this section, we present a means of reconstituting and formatting a condensed mash-up using declarative queries in existing languages.

5.1 The Need for Declarative Querying

The procedure in Figure 10 illustrates that Sixml DOM provides a developer an easy way to reconstitute a mash-up, but its imperative approach can be both tedious and inefficient for tasks such as printing comments in order of the page containing the comments.

An alternative approach is to use *queries*. The benefit of this approach is that XML query languages tend to be *declarative*, and XML query processors can process large amounts of data efficiently.

For example, Figure 11 shows a pair of XSLT templates to format condensed comment data as an HTML review report sorted by page number. Key parts of the template are bolded. The template that matches elements named `Comment` is similar to the procedure in Figure 10.

The template that matches the root node *declaratively* invokes the template for each `Comment`, in order of the page number containing the commented regions. (Section 5.2 discusses retrieving page number.) An equivalent imperative procedure (with or without our infrastructure) would need more development effort, and likely executes slower, especially if implemented as a client-side script.

The templates in Figure 11 use XSLT *as is* (that is, no feature specific to bi-level querying is used), and they access context information as if it is contained in the input document (though it is not, as evidenced in Figure 5).

```
<xsl:template match="/">
 <xsl:apply-templates select="//Comment">
   <xsl:sort
    select="sixml:EMark/sixml:Context/Placement/Page"/>
 </xsl:apply-templates>
</xsl:template>

<xsl:template match="Comment">
 <P>
   <xsl:value-of select="concat('Page: ',
    sixml:EMark/sixml:Context/Placement/Page)"/>
 </P>
 <P><xsl:value-of select="concat('Excerpt: ', @excerpt)"/></P>
 <P><xsl:value-of select="concat('Comment: ', text())"/></P>
</xsl:template>
```

Figure 11: Formatting a condensed mash-up using XSLT

5.2 Expressing Bi-level Queries

We call queries such as those in Figure 11 *bi-level queries* because they work on *bi-level information*, which is a combination of reconstituted mash-up parts and information from the context of referenced external data. We now show how bi-level queries over a mash-up are expressed in existing languages without using language extensions.

Bi-level queries in XPath: Any approach to facilitating bi-level queries must provide access to marks and mark contexts. Also, the approach should make query expression easy and aid efficient query execution.

One approach is to use the XPath data model (XDM) [26] *as is*. For example, the Sixml data in Figure 5 would be represented as a tree similar to that in Figure 8. In this approach, the simple expression `./sixml:EMark` navigates from an element to mark association, but navigation from an attribute to mark associations requires the expression `./sixml:AMark[@target=$name]`, where `$name` is a variable bound to the name of the target attribute. However, creating variable bindings requires the use of XSLT or XQuery, making query expression hard.

Another approach is to extend XDM by attaching mark associations to target nodes (as in Sixml DOM), and introduce a new axis *marks* to navigate from a node to its marks, and a function *context* to access mark context. This approach extends both XDM and the XPath language.

Our approach is to extend only XDM as follows:

1. Allow child elements for *any* Sixml node.
2. Like Sixml DOM, attach a mark association to its target Sixml node, but unlike Sixml DOM, make a mark association a child of its target node.
3. Unlike Sixml DOM, represent a mark descriptor and mark context as children of a mark association.
4. Like Sixml DOM, reconstitute a Sixml node's value based on the attribute `valueSource` in the attached mark associations.

Extensions 1 and 2 allow the mark associations for a Sixml node to be selected simply by following the child axis. For example, the mark associations for an attribute can be selected using the simple XPath expression `./*`.

Extension 3 allows the use of the child axis to select mark descriptors and contexts. For example, the expression `sixml:EMark/sixml:Context/Placement/Page` selects the context element containing the page number for a commented region.

Extension 4 allows easy access to reconstituted data. For example, the value of the attribute returned by the expression `@excerpt` executed in the context of the element `Comment` would be the text excerpt of the corresponding commented region.

Bi-level queries in XSLT and XQuery: Both XSLT and XQuery [27] only provide ways to manipulate parts of an XML document already selected using XPath expressions. (For example, see Figure 11.) Thus, a mash-up can be formatted in these languages without using extensions as long as path expressions are executed over a tree represented in our extended XDM.

5.3 Executing Bi-level Queries

We now give an overview of our architecture (shown in Figure 12) for a bi-level query processor, based on an alternative path navigator called *Sixml Navigator*.

In our architecture, an XPath processor is composed of two classes: XPathNavigator defines an abstract *path navigator* to traverse a document tree. XPathEvaluator uses a path navigator to evaluate an expression.

For example, to evaluate the expression `/Comment`, the XPath evaluator (*evaluator* for short) first moves the path navigator (*navigator* for short) to the root node of the document tree. It then moves the navigator to the first child of the root node, then to the next sibling of the first child, and so on until the navigator reports an element `Comment`, or until navigation fails.

In this approach, the navigator freely determines what nodes are exposed to the evaluator: It can report nonexistent nodes, and it can omit existing nodes. In Figure 12, the class SixmlNavigator is a navigator that exercises

this freedom. In the rest of this section, we describe how SixmlNavigator supports bi-level querying.

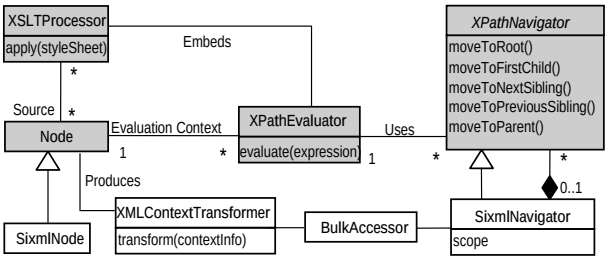

Figure 12: Architecture of a bi-level query processor

Navigating mark associations, descriptors, and context: We support access to mark associations, descriptors, and contexts using the XDM extensions outlined in Section 5.2. Specifically, when the evaluator seeks child elements of a Sixml node, we include the mark associations attached to that node. Also, when the evaluator seeks the child elements of a mark association, we include the mark descriptor and context information.

We use the *bulk accessor* [14] mentioned in Section 2 to retrieve context information from a mark. The bulk accessor improves the *scalability* of the bi-level query processor by efficiently retrieving context information from a large number of marks, or from marks into a large number of documents. The bulk accessor uses the class XMLContextTransformer to transform context information into the XML model. For brevity, we omit the details of the bulk accessor and the XML context transformer.

Cloaking data: Presenting a mark association as a child of its target can reduce the performance of *SI-only queries* (that is, queries that examine and return only reconstituted data such as the attribute `excerpt`, and the added data such as comment text). For example, the expression `text()`, intended to return only the comment text, also returns mark associations because they are represented as child nodes of the comment text. Eliminating mark associations requires the use of XSLT or XQuery because XPath cannot remove a child node from a result node. However, using XSLT or XQuery can slow down execution because both languages always construct new result nodes. Also, when evaluating this expression, the navigator *unnecessarily* visits the `EMark` element.

We use the notion of *query scope*, modeled by the property scope in class SixmlNavigator, to improve the performance of SI-only queries (and other classes of queries we omit for brevity). When query scope is *SI*, we omit mark associations from navigation, thus examining and returning only SI nodes. When the scope is *Associations*, we navigate mark associations, but leave out the contained mark descriptors. When the scope is *Descriptors*, we include mark descriptors in the navigation.

Cloaking makes it easy to run *ad hoc* queries and to perform data-exploration activities. The module *Cloaker* in Figure 4 cloaks data based on query scope.

Bi-level queries in XSLT and XQuery: As stated in Section 5.2, both XSLT and XQuery provide ways to manipulate data already selected using XPath expressions. Thus, each XSLT and XQuery processor has an embedded XPath evaluator. In our approach, XSLT and XQuery bi-level queries are executed simply by embedding an XPath evaluator that uses an instance of SixmlNavigator.

6. Evaluation

We have evaluated Sixml by using it to define the schema of applications such as the *Superimposed System Information Browser* (SSIB) [11] to manage information related to networked computers; the *Superimposed Scholarly Review System* (SISRS) [12], a tool to assist in reviewing scholarly publications such as conference papers; and *Mash-o-matic* [13], a utility to build map-based mash-ups. The running example in this paper is based on SISRS.

We have evaluated Sixml DOM and Sixml Navigator by implementing them, using the implementations to produce mash-ups in applications such as SSIB, SISRS, and Mash-o-matic, and by running experiments.

We present here only the implementation and experimental evaluation of Sixml DOM and Sixml Navigator.

6.1 Implementation

Sixml DOM and Sixml Navigator are implemented using the .NET Framework [17]. The shaded classes in Figures 5 and 11 are included in the .NET Framework. We have implemented the other classes in C#.

We have implemented Sixml DOM using two *alternative* strategies: extending DOM and revising DOM. The *extension strategy* implements uses inheritance. In this approach, both DOM and Sixml DOM are simultaneously available. Nodes can be created using the document classes Document and SixmlDocument, but marks can be associated only with a node created from SixmlDocument. Sixml nodes can be accessed using DOM interfaces, but mark associations are accessible only with Sixml DOM.

The *revision strategy* adds Sixml capability to DOM from the ground up. For example, the methods of the class SixmlNode are added directly to the DOM class Node. In this approach, all XML documents are Sixml documents.

The extension approach does not require access to the source code of the base implementation, but run-time efficiency can vary based on the availability of source code. The revision approach requires access to the source of the base DOM implementation, but the run-time performance can be better than the extension counterpart.

We have just one implementation of Sixml Navigator, but *three* implementations of Sixml DOM: an extension-strategy implementation based on Microsoft's distribution of .NET; and an extension-strategy implementation and a revision-strategy implementation based on Mono's distribution (Version 1.2.5.1) [8] of .NET.

We refer to the three Sixml DOM implementations as *Microsoft Extension* (MSX), *Mono Extension* (MNX), and

Mono Revision (MNR). We refer to the base DOM implementation for MSX as *Microsoft Base* (MS), and refer to the base of MNX and MNR as *Mono Base* (MN). We have the source code for MN, but not for MS. We used the same source code for MSX and MNX and adapted much of that source code in MNR.

We had initially implemented only the MSX version of Sixml DOM, but its performance overhead (compared to its base, MS) seemed excessive. We then implemented MNX and MNR to test if the overhead can be reduced.

6.2 Experiments

All implementations were compiled using MS Visual Studio 2005. All experiments were run using the MS distribution of the .NET Common Language Runtime (Version 2.0) running on an Intel® Core Duo 1.66 GHz processor with 1 GB of main memory. The operating system was MS Windows XP (Service Pack 2).

Table 1 summarizes the Sixml documents used in the experiments. The first set of documents comes from the SISRS application, the second set comes from the SSIB application. The number against each document indicates the *size scale factor*. For example, the document SISRS-2 has twice the number of mark associations as SISRS-1. SSIB-8 has eight times the number of mark associations as SSIB-1. The third column lists the number of *external documents* each Sixml document references. The column 'Total' shows the number of *external fragments* referenced. SISRS documents reference PDF fragments; SSIB documents reference MS Excel fragments.

Table 1: Sixml documents used in the experiments

			#Mark associations			
Sixml doc.	**Size (MB)**	**#External docs.**	**EMark**	**AMark**	**TMark**	**Total**
SISRS-1	0.2	53	1,908	53	0	1,961
SISRS-2	0.4	106	3,816	106	0	3,922
SISRS-4	0.8	213	7,668	213	0	7,881
SISRS-8	1.6	426	15,336	426	0	15,762
SSIB-1	3.2	18	0	25,922	12,961	38,883
SSIB-2	6.5	18	0	51,850	25,925	77,775
SSIB-4	13.0	18	0	103,710	51,855	155,565
SSIB-8	26.1	18	0	207,426	103,713	311,139

6.2.1 Sixml DOM

In general, MSX has the fastest response, and MNX the slowest response. MSX is faster because its base, MS, is faster than MN, the base of MNX and MNR [7]. MNR is faster than MNX because it does not have the inheritance overheads of MNX, and Sixml DOM capability is added at optimal locations within the base implementation.

Scalability: In this experiment, we test how the runtime performance scales up with the number of mark associations. Here, we open each document and traverse all mark associations in the document (using the relationship markAssociations in Figure 6). We then compute a *speed scale factor* for each document in a set as the ratio of the time to traverse mark associations in the document to the time to traverse mark associations in the first document in the set (SISRS-1 and SSIB-1).

Figure 13 plots the speed scale factor for both SISRS and SSIB datasets for each Sixml DOM implementation. For example, in MSX, traversing the mark associations in SISRS-2 takes 2.3 times the time needed to traverse the mark associations in SISRS-1. In MNR, the speed scale factor for SISRS-2 is only 2. (SISRS-2 has twice the number of mark associations as SISRS-1.)

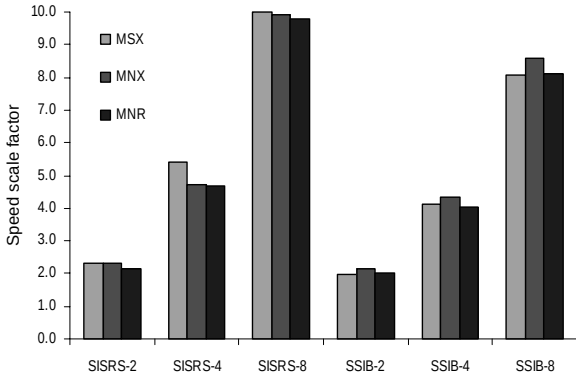

Figure 13: Speed scale factor to traverse mark associations

The speed scale factor of MNR is always less than or equal to that of MSX, for any document. We also computed the speed scale factors to traverse SI. The trends were similar to that for mark associations.

Savings from using Sixml DOM: Using Sixml DOM to manipulate a Sixml mash-up at run time has several benefits (such as automatic reconstitution), but we wanted to test if using Sixml DOM also saves time over DOM.

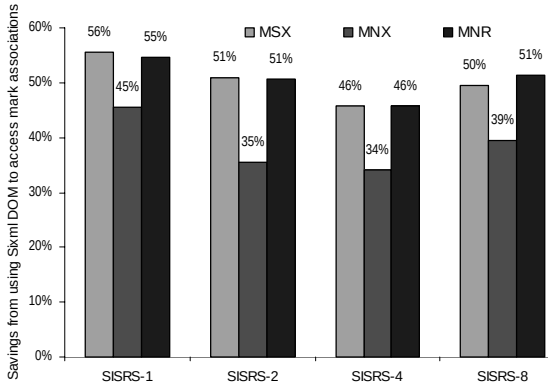

Figure 14: Percentage time saved due to Sixml DOM over DOM when accessing mark associations

In this experiment, we compute the percentage savings (or overhead) in time to traversing mark associations and SI using Sixml DOM over the time to traverse the same data using DOM. For brevity, we report results for only the SISRS dataset. The trends for SSIB were similar.

Figure 14 shows the savings due to Sixml DOM when accessing only mark associations in the SISRS dataset: Sixml DOM always saves time. MNX saves the least, and

the savings from MNR are comparable to that from MSX. The savings from MNR drops four percentage points from the first document to the last, but the drop is six percentage points for MSX. That is, MNR scales better.

Figure 15 shows the savings due to Sixml DOM when accessing only SI in the SISRS dataset. Negative savings denote overhead. In all cases, the savings decline as the amount of SI increases. MSX and MNX have overheads for SISRS-4 and SISRS-8, but MNR saves in all cases. Also, as with mark associations, MNR scales better.

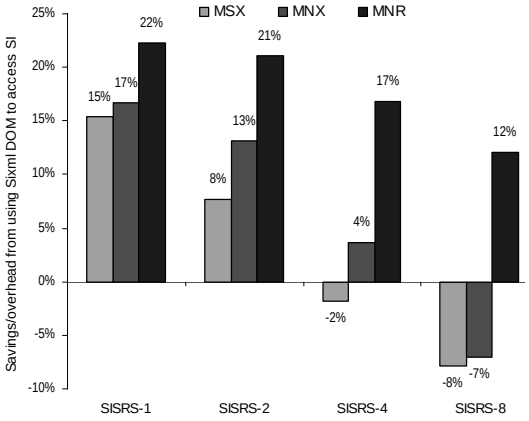

Figure 15: Percentage time saved (lost) due to Sixml DOM over DOM when accessing SI

Overhead to traverse XML data: We also tested the performance of the Sixml DOM implementations when traversing XML data containing no mark associations. We report results for three XML documents: *SIGMOD Record* 1999, the XML index of issues of ACM SIGMOD Record [2] for the year 1999; *XMark*, a document from the XMark benchmark [19]; and *MBench*, a document from the Michigan benchmark [18]. The salient features of these documents are, respectively: Size 484 KB and tree depth 4; 113.7 MB, depth 8; and 14.7 MB, depth 16.

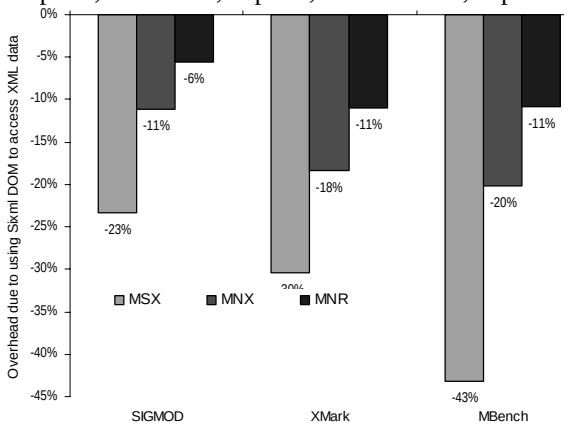

Figure 16: Percentage time lost due to Sixml DOM over DOM when traversing pure XML data

Figure 16 shows the percentage overhead to traverse the three XML documents. MNR has the least overhead and MSX has the most overhead. In general, the performance of Sixml DOM when traversing an XML document is similar to that of accessing SI in a Sixml document. (In fact, we use the same code to access SI and XML data in Sixml DOM.) For example, the trends in Figures 15 are similar to those in Figure 15. (Figure 16 is oriented such that it can be easily compared with Figure 15.)

Summary: MSX has the best absolute performance when traversing mark associations, SI, and pure XML data. MNR performs better with growing number of marks and has the least overhead. MNX underperforms MNR because it has inheritance overheads.

It is better to use Sixml DOM to access Sixml documents, but DOM is better for some pure XML documents.

Both the extension and revision strategies of implementing Sixml DOM have merits. Sixml DOM can be fast (as in MSX) and have low overheads (as in MNX and MNR) if the base DOM implementation is fast and the source code for the base is available. That is, the speed of MNX and MNR could be improved by improving MN. The overheads in MSX could be reduced with compile-time access to the source code for MS.

6.2.2 Sixml Navigator

The Sixml Navigator makes it easier to query Sixml data, but we wanted to test if the Sixml Navigator also saves time compared to using the traditional navigator (that is, the class `XPathNavigator` in Figure 12). In this experiment, we compute the percentage savings in time to retrieve mark associations and SI using the Sixml Navigator over the time to retrieve the same data using the traditional navigator. For this experiment, the two navigators were used with the XPath and XSLT query processors included in the .NET Framework.

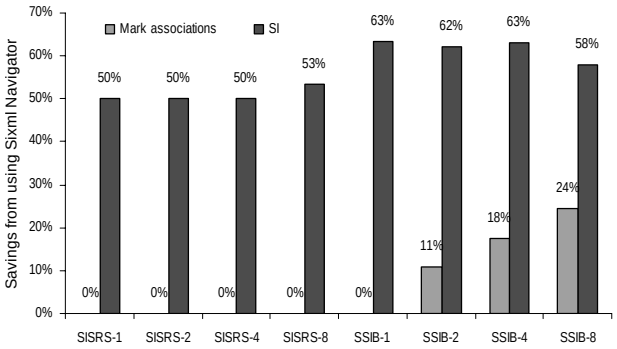

Figure 17: Percentage time saved by using Sixml Navigator over the traditional navigator

The first set of bars in Figure 17 shows the percentage time saved when traversing mark associations: The savings from the Sixml Navigator increases as the number of mark associations increases. The second set of bars shows the percentage time saved when retrieving SI: In all cases, the Sixml Navigator provides considerable savings over the traditional navigator.

The better performance of the Sixml navigator is due to cloaking. For example, with query scope *SI*, the simple XPath expression * suffices to retrieve SI using the Sixml Navigator. However, with the traditional navigator, the

same task needs an XSLT style sheet with eight templates, involving 23 XPath expressions.

We have also conducted experiments specifically to illustrate the benefits of cloaking (but omit presenting them, for brevity). For example, retrieving all comment text using the expression `//text()` saves 39% time with query scope *SI* compared to using the scope *Associations*. Section 5.3 discussed query scope.

In summary, the Sixml Navigator lets a developer exploit existing XML query languages and processors to mash disparate data fragments, even if the fragments' sources are not represented as XML.

7. Related Work

Damia [20] is a tool to produce data mash-ups from XML sources and from sources that can be transformed to XML. Each source is transformed to XML and represented using a variation of XDM, and parts of the transformed XML are processed using special operators. A mash-up may use only parts of a source, but the *complete* source is transformed to XML.

In our approach, only the reconstituted fragments are represented as XML, and the reconstitution is *on demand*. A mash-up can be reconstituted and formatted using existing query languages and query processors unchanged.

Yahoo! Pipes [29] is a visual editor to assemble data mash-ups using *complete* information sources, not fragments. It supports operations such as sort and filter over web feeds, but it does not support the expression and manipulation of a mash-up using standard XML tools. (Yahoo! Pipes might internally represent a network of pipes as XML, but that representation is not exposed.)

In general, both Yahoo! Pipes and Damia are designed to assist non-technical people assemble mash-ups. Our infrastructure allows a developer to produce mash-ups, and might form the basis for a tool such as Yahoo! Pipes and Damia.

Active XML (AXML) [1] provides a means to describe parts of an XML document intensionally using *service-call* elements which encode calls to web services. No special DOM is defined to manipulate an AXML document, but a special query processor lazily executes service calls, and *replaces* a service-call element with the results of the call.

AXML data references programs (which are web services), but Sixml data references data. In AXML, external data (that is, the result of service calls) is not necessarily related to the data specified extensionally, and it is not possible to distinguish external data from extensional data. In Sixml, the division between SI and the external data is always apparent. AXML uses a schema language extension to express the type of the result of a function call, but the schema of a Sixml document can be expressed using only the standard XML Schema constructs.

An AXML service-call element can supply the content of a regular XML element, but it cannot supply values of parts such as attributes. In our approach, external values can be assigned to attributes, text, and other document parts. In this paper, we have not described a means to supply the content of an element, but we do have the designs for a mark association type called `EContent` to achieve this goal.

Like Sixml, XLink [22] allows embedding of links in arbitrary XML documents. A linked resource may be *remote* (that is, external to the document that specifies the link) or *local*. An element is a *link element* if it has the attribute `xlink:type`, or if the element has the attribute `xlink:href`. A sub-element of a link element, called a *locator*, addresses a resource using a URI or an XPointer.

An XLink locator is comparable to a mark descriptor. The attribute `xlink:type` is similar to our attribute `sixml:type`, but we also allow a mark association's type to be conveyed via a schema. The XLink specification is silent about to which part of an embedding XML document a link corresponds, but we can reasonably assume that a link corresponds to an element. In contrast to XLink, we make explicit the target of a link and support links to both elements and non-elements. Also, we do not restrict locators to URIs and XPointers. Finally, XLink does not support deriving of XML content from linked resources.

DOM extensions have been defined for MathML [5] and Scalable Vector Graphics [21]. Both extensions define specialized classes for elements and attributes. A factory method chooses a class to instantiate a node based only on the node's qualified name. For example, in MathML DOM, an element named `math` (with the namespace URI `http://www.w3.org/1998/Math/MathML`) becomes an instance of the class `MathMLMathElement`, and `math` is the top-level element in each MathML document or segment. In our approach, a mark association element can be detected both by its name and by its type, and a mark association may be slipped into any element.

The XML query processors in the .NET Framework allow the use of alternative path navigators. We have come across code samples [9] illustrating alternative path navigators, but are yet to encounter the use of alternative path navigators as a part of a query-processing strategy.

8. Summary and Conclusions

We have identified three forms of a data mash-up: *condensed*, *reconstituted*, and *formatted*. We have described three parts of our infrastructure to declaratively produce XML data mash-ups and shown how each part helps work with the different forms of a mash-up: *Sixml* helps create a condensed mash-up by providing a means to embed in arbitrary XML documents links to disparate data fragments. *Sixml DOM* reconstitutes a mash-up by lazily reconstituting the component parts. It also provides a means to manipulate a mash-up. The *Sixml Navigator* provides a means to reformulate and format a mash-up using declarative queries in existing languages.

We have described two strategies to implement Sixml DOM and outlined three implementations of Sixml DOM. We have also presented the results from an experimental evaluation of both Sixml DOM and the Sixml Navigator. Experiments show that our implementations can efficiently reconstitute and format even mash-ups that reference thousands of external fragments.

The schema for the Sixml element types, the interface definitions for Sixml DOM, and the source code for the implementations are available from http://www.sixml.org.

References

1. Abiteboul, S., Benjelloun, O., Cautis, B., Manolescu, I., Milo, T., Preda, N. 2004. Lazy Query Evaluation for Active XML. In *Proceedings of SIGMOD 2004*, Paris, France.
2. ACM SIGMOD Online. ACM SIGMOD. www.sigmod.org/sigmod/record/xml/index.html.
3. Delcambre, L., Maier, D., Bowers, S., Weaver, M., Deng, L., Gorman, P., Ash, J., Lavelle, M., Lyman, J. 2001. Bundles in Captivity: An Application of Superimposed Information. In *Proceedings of ICDE 2001*, 2001, Heidelberg, Germany.
4. Document Object Model. W3C. http://www.w3.org/DOM/.
5. Document Object Model for MathML. 2003. W3C. http://www.w3.org/TR/MathML2/appendixd.html.
6. JavaScript. Mozilla Foundation. http://developer.mozilla.org/en/docs/JavaScript.
7. Jeswin, P. Xml Performance: XmlMark Revisited: Java, Mono and .Net. http://www.process64.com/articles/xmlmark1/.
8. Mono. Mono Project. http://www.mono-project.com/.
9. MSDN Library Archive. Microsoft Corporation. http://msdn.microsoft.com/archive.
10. Murthy, S. 2007. Sixml.org. http://www.sixml.org.
11. Murthy, S., Delcambre, L., Maier, D. 2006. Explicitly Representing Superimposed Information in a Conceptual Model. In *Proceedings of 25th International Conference on Conceptual Modeling (ER 2006)*, Nov. 6-9, Tucson, Arizona.
12. Murthy, S., Maier, D. 2004. SISRS: The Superimposed Scholarly Review System. http://sparce.cs.pdx.edu/pubs/SISRS-WP.pdf.
13. Murthy, S., Maier, D., Delcambre, L. 2006. Mash-o-matic. In *Proceedings of Sixth ACM Symposium on Document Engineering (DocEng 2006)*, Oct. 10-13, Amsterdam, Netherlands.
14. Murthy, S., Maier, D., Delcambre, L. 2008. Speeding up On-the-Fly Integration of DB and Exo-DB Data. In *Proceedings of Workshop on Information Integration Methods, Architectures, and Systems*, Apr. 11-12, Cancun, Mexico.
15. Murthy, S., Maier, D., Delcambre, L., Bowers, S. 2004. Putting Integrated Information in Context: Superimposing Conceptual Models with SPARCE. In *Proceedings of First Asia-Pacific Conference of Conceptual Modeling*, Jan. 22, 2004, Dunedin, New Zealand.
16. Nelson, T. H. 1999. Xanalogical Structure, Needed Now More than Ever: Parallel Documents, Deep Links to Content, Deep Versioning, and Deep Re-Use. *ACM Computing Surveys* 31 (4).
17. .NET Framework Developer Center. Microsoft Corporation. http://msdn.microsoft.com/netframework/.
18. Runapongsa, K., Patel, J.M., Jagadish, H.V., Chen, Y., Al-Khalifa, S. 2006. The Michigan Benchmark: Towards XML Query Performance Diagnostics. *Information Systems* 31 (2).
19. Schmidt, A., Waas, F., Kersten, M., Carey, M.J., Manolescu, I., Busse, R. 2002. XMark: A Benchmark for XML Data Management. In *Proceedings of Proceedings of the 28th international conference on Very Large Data Bases*, Hong Kong, China.
20. Simmen, D. E., Altinel, M., Markl, V., Padmanabhan, S., Singh, A. 2008. Damia: Data Mashups for Intranet Applications. In *Proceedings of SIGMOD 2008*, Vancouver, Canada.
21. SVG Document Object Model. 2003. W3C. http://www.w3.org/TR/SVG/svgdom.html.
22. XML Linking Language (XLink) Version 1.0. 2001. W3C. http://www.w3.org/TR/xlink/.
23. XML Path Language (XPath) Version 1.0. 1999. W3C. http://www.w3.org/TR/xpath.
24. XML Pointer Language (XPointer) Framework. 2003. W3C. http://www.w3.org/TR/xptr-framework/.
25. XML Schema Part 0: Primer Second Edition. 2004. W3C. http://www.w3.org/TR/xmlschema-0/.
26. XQuery 1.0 and XPath 2.0 Data Model (XDM). 2007. W3C. http://www.w3.org/TR/xpath-datamodel/.
27. XQuery 1.0: An XML Query Language. 2005. W3C. http://www.w3.org/TR/xquery/.
28. XSL Transformations (XSLT). 1999. W3C. http://www.w3.org/TR/xslt.
29. Yahoo! Pipes. Yahoo! Inc. http://pipes.yahoo.com.