xlinkit: A Consistency Checking and Smart Link Generation Service

CHRISTIAN NENTWICH, LICIA CAPRA, WOLFGANG EMMERICH, and ANTHONY FINKELSTEIN
University College London

xlinkit is a lightweight application service that provides rule-based link generation and checks the consistency of distributed Web content. It leverages standard Internet technologies, notably XML, XPath, and XLink. xlinkit can be used as part of a consistency management scheme or in applications that require smart link generation, including portal construction and management of large document repositories. In this article we show how consistency constraints can be expressed and checked. We describe a novel semantics for first-order logic that produces links instead of truth values and give an account of our content management strategy. We present the architecture of our service and the results of two substantial case studies that use xlinkit for checking course syllabus information and for validating UML models supplied by industrial partners.

Categories and Subject Descriptors: D.2.6 [**Software Engineering**]: Programming Environments; C.2.4 [**Computer-Communication Networks**]: Distributed Systems; H.5.4 [**Information Interfaces and Presentation**]: Hypertext/Hypermedia; I.7.1 [**Document and Text Processing**]: Document and Text Editing; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages; D.3.4 [**Programming Languages**]: Processors

General Terms: Languages, Management, Verification

Additional Key Words and Phrases: Consistency management, constraint checking, automatic link generation, XML

1. OVERVIEW

This article describes xlinkit, a lightweight application service that provides rule-based link generation and checks the consistency of distributed Web content. The article is supplemented by the online demonstrations at `http://www.xlinkit.com`.

The operation of xlinkit is quite simple. It is given a set of distributed XML resources and a set of potentially distributed rules that relate the content of those resources. The rules express consistency constraints across the resource types. xlinkit returns a set of XLinks, in the form of a linkbase, that supports

Authors' address: Department of Computer Science, University College London, Gower St., London WC1E 6BT UK; email: {c.nentwich,l.capra,w.emmerich,a.finkelstein}@cs.ucl.ac.uk.

navigation between elements of the XML resources. The major contribution of this article is a new linking semantics for our first-order logic-based language, which returns hyperlinks between inconsistent elements instead of boolean values.

xlinkit leverages standard Internet technologies. It supports document distribution and can support multiple deployment models. It has a formal foundation and evaluation has shown that it scales, both in terms of the size of documents and in the number of rules.

With this thumbnail description in mind, it is easiest to motivate and to explain xlinkit by reference to a simple example. This example is given in Section 3. It is preceded by some essential background on the technologies that we build on.

2. BACKGROUND

The article assumes some familiarity with XML (Extensible Markup Language) [Bray et al. 2000] and XSLT (Extensible Stylesheet Language Transformations) [Clark 1999]. It also makes significant reference to technologies related to XML, specifically XLink [DeRose et al. 2001], the XML linking scheme and XPath [Clark and DeRose 1999], which supports addressing of the internal structures of an XML resource. We make some reference in the article to the XML DOM (Document Object Model) [Apparao et al. 1998], the API for XML resources, though this article does not require a detailed understanding of it. For details of XML and related technologies, good sources are the World Wide Web Consortium (W3C) and the Organization for the Advancement of Structured Information Standards (OASIS). We briefly give an overview of the main XML technologies referred to in this article.

The Document Object Model (DOM) facilitates the manipulation of XML data through an application program interface. It specifies a set of interfaces that can be used to manipulate XML content. XML content is represented in the DOM as an abstract tree structure, consisting of DOM *nodes*. The interfaces contain methods for manipulating nodes in the tree, such as listing the children of nodes and adding and deleting nodes, traversal, and event handling. The DOM provides a convenient mechanism for representing XML documents in memory and is implemented by most major XML parsers.

Since the initial specification of XML, several languages have emerged as "core" languages that provide additional hypertext infrastructure to applications that have to deal with XML. XPath is one of these core languages. It permits the selection of elements from XML documents by specifying a tree path in the documents. For example, the path `/Catalogue/Product` would select all `Product` elements contained in any `Catalogue` elements in an XML file. XPath also supports the restriction of selected elements by predicates and contains several functions, including functions for string manipulation. We use XPath for selecting sets of nodes from DOM trees.

XLink is a another core infrastructure language. It is the standard linking language for XML and provides additional linking functionality for Web resources. HTML links are highly constrained, notably: they are unidirectional

```
<Catalogue>
     <Product xlink:type="extended">
         <Name>Haro Shredder</Name>
         <Code>B001</Code>
         <Price currency="GBP">349.95</Price>
         <Combines xlink:type="locator"
                   xlink:href="oldcatalogue.xml#/Catalogue/Product[1]"
                   xlink:label="component 1"/>
         <Combines xlink:type="locator"
                   xlink:href="oldcatalogue.xml#/Catalogue/Product[2]"
                   xlink:label="component 2"/>
     </Product>
</Catalogue>
```

Fig. 1. Sample XLink.

and point-to-point; have a limited range of behaviors; link only at the level of files unless an explicit target is inserted in the destination resource; and, most significantly, are embedded within the resource, leading to maintenance difficulties.

XLink addresses these problems, allowing any XML element to act as a link, enabling the user to specify complex link structures and traversal behaviors and to add metadata to links. Figure 1 shows some XML markup that uses XLink to turn an element into a link. The `Product` element has an `xlink:type` attribute attached to it—XLink-aware processors will now recognize this element as a link. The element contains two elements of type `locator`, which will be recognized as link endpoints. This link now links together the first and second `Product` elements in `oldcatalogue.xml`. Most importantly, it links together two elements in a file without inserting any links directly into the file; that is, it is an external link with respect to those files.

Since such “extended links” can be managed separately from the resources they link, it is possible to compile “linkbases,” XML files that contain a collection of XLinks. Linkbases can then be selectively applied to establish hyperlinks between resources. The XLink language contains further constructs for specifying behavior during link traversal and traversal restriction, which we do not currently make use of.

3. EXAMPLE

We now introduce an example that is used throughout the article. Wilbur’s Bike Shop sells bicycles and makes information about their company available on the Internet and on a corporate intranet. Wilbur’s uses XML for Web publication and information exchange.

The information collected by Wilbur’s is spread across several Web resources:

—*a product catalog* containing product name, product code, price and description;

—*advertisements* containing product name, price, and description;

—*customer reports* listing the products purchased by particular customers;

—*service reports* giving problems with products reported by customers.

```
<Catalogue>
     <Product>
         <Name>Haro Shredder</Name>
         <Code>B001</Code>
         <Price currency="GBP">349.95</Price>
     </Product>
     <Product>
         <Name>Dyno NFX</Name>
         <Price currency="GBP">119.95</Price>
         <Code>B003</Code>
     </Product>
</Catalogue>
```

Fig. 2. Wilbur's product catalogue extract.

```
<Advert>
     <ProductName>Dyno NFX</ProductName>
     <Price currency="GBP">119.95</Price>
     <Description>BMX Bike. Dyno expert frame.
                  Coaster brake or freewheel.
     </Description>
</Advert>
```

Fig. 3. Wilbur's sample advertisement.

Wilbur's has only one product catalog, but many advertisements, customer reports, and service reports. The information is distributed across different Web servers.

It should be clear that much of this information, though produced independently, is closely related. For example: the product names in the advertisements and those in the catalog; the advertised prices and the product catalog prices; the products listed as sold to a customer and those in the product catalog; the goods reported as defective in the service reports and those in the customer reports; and so on.

Relationships among independently evolving and separately managed resources can give rise to inconsistencies. This is not necessarily a bad thing, but it is important to be aware of such inconsistencies and deal with them appropriately. In view of this, Wilbur's would like to check their resources to establish their position.

For the example which follows we will concentrate on the relationship between the product catalog and the advertisements. Figure 2 shows an extract from the product catalog and Figure 3 shows a sample advertisement. Samples of the other resources can be found in Appendix A.

This relationship requires a check:

—*Are all the product names in the advertisements mentioned in the catalog?*

Other checks might include:

—*Do the advertised prices and the product catalog prices correspond?*
—*Are the products listed as sold to a customer in the product catalog?*
—*Did we sell the goods reported as problematic to the customer reporting the problem?*

We define these checks as rules and assemble them in a rule set. We describe our rule language and the assembly of rule sets in the following sections. The document set is the collection of documents we want to check against the rules. In this example we have a set of adverts, a set of customers, and a set of service reports. Both the document set and the rule set are identified by URLs.

We define a transparent semantics for generating hyperlinks, depending on the status of the documents with respect to the rules. Thus:

—*Are all the product names in the advertisements mentioned in the catalog?*
Result: Links between the product advertised and the corresponding product entry in the catalog. If there is no corresponding product in the catalog, the advertisement will not be linked to anything, and a rule reference is included for diagnostic purposes.

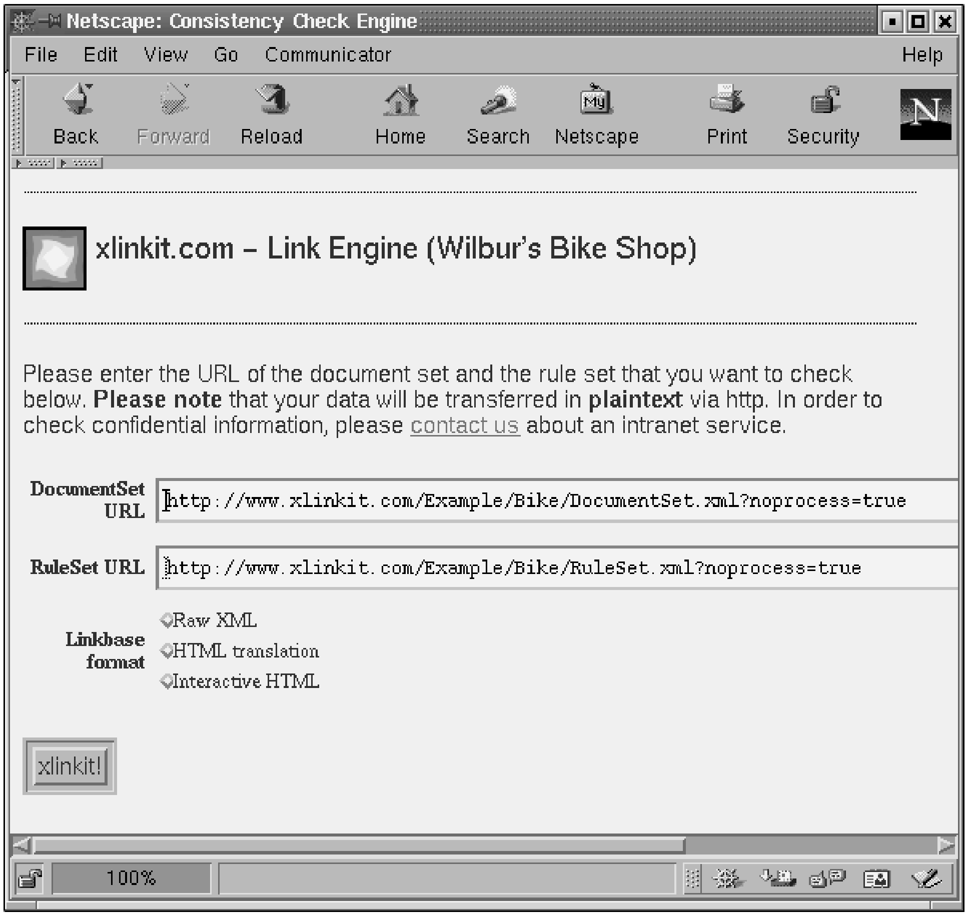
—*Do the advertised prices and the product catalog prices correspond?*
Result: When the advertised price does not match, link to the offending advert and include a rule reference for diagnostic purposes.

—*Are the products listed as sold to a customer in the product catalog?*
Result: Links between the product entry in the customer record and the corresponding product entry in the catalog. If there is no corresponding product in the catalog, link to the customer record and include a rule reference for diagnostic purposes.

—*Did we sell the goods reported as defective to the customer reporting the problem?*
Result: Links between the product with the problem and the product entry in the customer record. If there is no corresponding product entry in the customer record, link to the defective product and include a rule reference for diagnostic purposes.

The checks are made by submitting the document set and the rule set URLs to the check engine which makes the checks and returns the URL of an XLink linkbase. Figure 4 shows the submission form that is passed to the check engine. Because the linkbase is itself XML, we can apply a stylesheet to render it in HTML for review or deliver in source XML. Figure 5 shows an HTML representation of a linkbase. Users can click on the consistency links; a servlet will then retrieve the two XML files that are being linked, convert them to HTML, and highlight the linked elements.

Most "off-the-shelf" browsers do not yet implement support for extended links of the sort that xlinkit produces—only limited support is available for simple links, that is, XLinks embedded in documents from browsers like Amaya [Consortium 2000] or the latest releases of Mozilla [Mozilla 2000]. One way to make the linkbase navigable is to first "fold" it into the resources. This entails applying an XLink processor to fetch the resources referenced in the linkbase, convert the extended links into simple links, and integrate them into the resources in the appropriate place. This task cannot be performed by a stylesheet, as stylesheet languages are typically designed to transform documents into one output document. Instead, we use our own XLink processor, XtooX, for this purpose—there are a number of similar processors available.



xlinkit.com – Link Engine (Wilbur's Bike Shop)

Please enter the URL of the document set and the rule set that you want to check below. **Please note** that your data will be transferred in **plaintext** via http. In order to check confidential information, please contact us about an intranet service.

DocumentSet URL	http://www.xlinkit.com/Example/Bike/DocumentSet.xml?noprocess=true
RuleSet URL	http://www.xlinkit.com/Example/Bike/RuleSet.xml?noprocess=true
Linkbase format	◇Raw XML ◇HTML translation ◇Interactive HTML

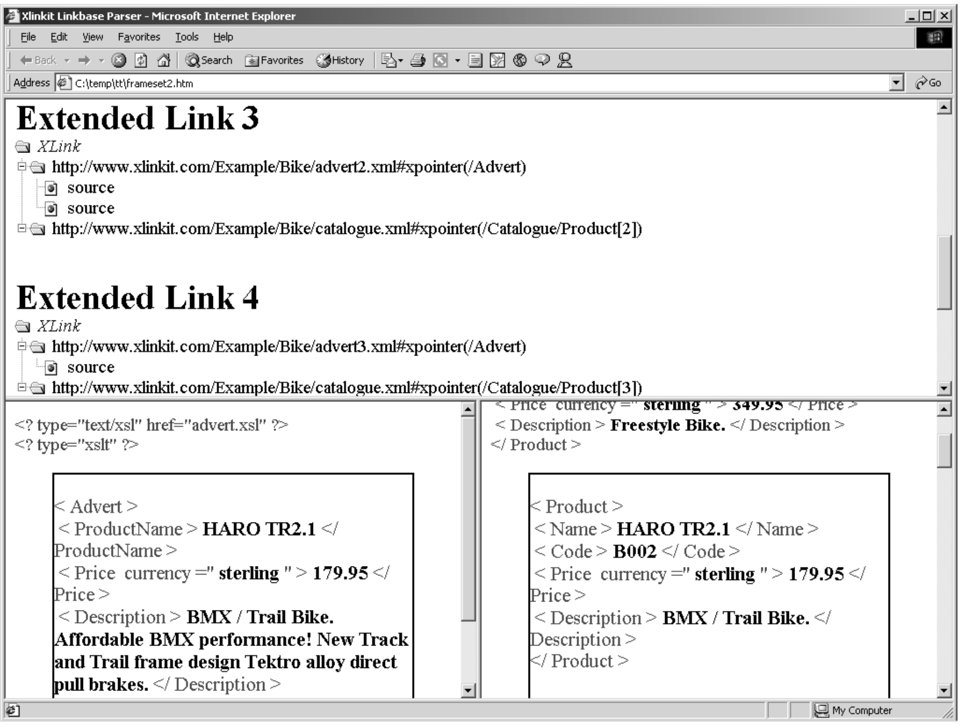
Fig. 4. Web submission form.

The resulting XML resources can then be handled in the familiar manner, that is, by applying stylesheets to render into a browsable hyperlinked HTML presentation. In the case of our example, we deliver a product catalog site that links to the advertisements.

4. RULE LANGUAGE

This section presents our set-based rule language, which serves to express consistency constraints between distributed documents. We outline a simple formal basis for the language and formalize our example rule.

Our rule language uses XPath to select sets of elements, which are then related via constraints. We use a notation for evaluating XPath expressions and for the formalization of the DOM due to [Wadler 1999]: the function $\mathcal{S}[\![p]\!]_x$ creates a set of nodes by evaluating the path expression p with x as the context node. (For example, $\mathcal{S}[\![\mathit{Price}/@\mathit{currency}]\!]_{/\mathit{Advert}}$ in Figure 3 would return a one-element set containing a text node with the string "GBP"). We later refine this


Xlinkit Linkbase Parser - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\temp\tt\frameset2.htm
Go
My Computer

Extended Link 3

XLink

http://www.xlinkit.com/Example/Bike/advert2.xml#xpointer(/Advert)
source
source
http://www.xlinkit.com/Example/Bike/catalogue.xml#xpointer(/Catalogue/Product[2])

Extended Link 4

XLink

http://www.xlinkit.com/Example/Bike/advert3.xml#xpointer(/Advert)
source
http://www.xlinkit.com/Example/Bike/catalogue.xml#xpointer(/Catalogue/Product[3])

```
<? type="text/xsl" href="advert.xsl" ?>
<? type="xslt" ?>

< Advert >
 < ProductName > HARO TR2.1 </
ProductName >
 < Price  currency =" sterling " > 179.95 </
Price >
 < Description > BMX / Trail Bike.
Affordable BMX performance! New Track
and Trail frame design Tektro alloy direct
pull brakes. </ Description >
```

```
 < Price  currency =" sterling " > 349.95 </ Price >
 < Description > Freestyle Bike. </ Description >
</ Product >

< Product >
 < Name > HARO TR2.1 </ Name >
 < Code > B002 </ Code >
 < Price  currency =" sterling " > 179.95 </
Price >
 < Description > BMX / Trail Bike. </
Description >
</ Product >
```

Fig. 5. Sample linkbase in HTML.

function to allow for variable context information, so that variables can be referred to in path expressions.

When specifying a rule, we want to express a relationship of one set of nodes with one or more other sets of nodes. For example, the set of all `Advert` elements in Wilbur's advertisements has to be consistent with the set of all `Products` in their product catalog.

We rephrase the question "*Are all the product names in the advertisement the same as in the catalog*?" more formally as an assertion: "*For all Advert elements, there exists a Product element in the Catalog element where the ProductName subelement of the former equals the Name subelement of the latter.*" If this condition holds, a consistent relationship exists between the Advert element and the Product element being considered. Otherwise, the Advert element is inconsistent with respect to our rule.

Figure 6 shows an abstract syntax for our rule language. The language is a restricted form of first-order logic, where no functions are allowed and all sets in the model are finite, since they are generated by an XPath processor. We do not make any restrictions on the path expressions that can be used, so the full range of string manipulation and namespace functions in XPath is available, as is the support for externally bound variables—which we use to bind variables from our quantifiers. We can express our sample rule directly in this language as

$$\forall a \in \text{“/Advert”}\ (\exists p \in \text{“/*/Product”}(\text{“\$a/ProductName”}=\text{“\$p/Name”}))$$

$$\begin{array}{rcl}
\textit{rule} & ::= & \forall\textbf{var} \in \textbf{xpath}(\textit{formula}) \\
\textit{formula} & ::= & \forall\textbf{var} \in \textbf{xpath}(\textit{formula}) \mid \\
 & & \exists\textbf{var} \in \textbf{xpath}(\textit{formula}) \mid \\
 & & \textit{formula}\ \textbf{and}\ \textit{formula} \mid \\
 & & \textit{formula}\ \textbf{or}\ \textit{formula} \mid \\
 & & \textit{formula}\ \textbf{implies}\ \textit{formula} \mid \\
 & & \textbf{not}\ \textit{formula} \mid \\
 & & \textbf{xpath} = \textbf{xpath} \mid \\
 & & \textbf{xpath} \neq \textbf{xpath} \mid \\
 & & \textbf{same var var}
\end{array}$$

Fig. 6. Rule language abstract syntax.

Table I. Sample Sets for Transitive Closure

Node 1	id=‘1’	child=‘2’
Node 2	id=‘2’	child=‘3’
Node 3	id=‘3’	

The language restrictions imply that we cannot express constraints that require any form of infinity. For example, the constraint *for all elements x, the children of x are prime numbers* would require quantification over the integers to express the latter half of the constraint, and thus cannot be expressed in this language. Nevertheless, its power is great enough to express a wide range of static semantic constraints, including those of the Unified Modeling Language [Object Management Group 2000a].

Some rules require the added power of a transitive closure operator. For example, if Wilbur's bikeshop were to offer composite products such as bikes made from several components, they might want to check that composite components are not parts of themselves. It is then not enough to check whether a part of a component equals the component itself, since the part may itself be made up from several parts—the transitive closure of the part-whole hierarchy has to be computed.

We have enriched the XPath language with a transitive closure operator. The basic form of the operator is `closure(base,transition)`, where `base` and `transition` are both XPath expressions. The operator first evaluates the `base` expression to build up a set of nodes. It then evaluates the `transition` expression with each node in the previously built set of nodes as the context node. The resulting node set is added to the base set. This process continues until the transition expression leads to an empty set or a cycle is found. The current base set minus the initial base set is then returned as the closure set.

As an example, consider the set of nodes shown with their subelements in Table I. If we invoke `closure(id(‘1’),id(./child))`, where $id(s)$ is a standard XPath function that retrieves a node based on an attribute of type ID, the operator constructs the base set containing node 1. It then executes `id(./child)` with node 1 as the context node, which leads to node 2. Node 2 is added to the base set and the transition expression is evaluated again, leading to node 3, which is added to the base set. Finally, for node 3, evaluating the transition expression leads to an empty set. Node 1 is now subtracted from the base set and the set containing node 2, and 3 is returned as the result.

5. LINK GENERATION

Our approach is always to take a tolerant view of inconsistency. Inconsistencies are not always accidental or undesirable, and we do not force their immediate resolution; instead we aim to provide diagnostic information that enables document owners to decide on further action, be it the resolution or toleration of inconsistency.

In this scenario, the priority of a consistency management system shifts from prevention of inconsistency—for example, through disallowing updates, to strong diagnostics that pinpoint precisely the elements that cause the inconsistency. We use hyperlinks called *consistency links* to connect consistent or inconsistent elements. If a number of elements form a consistent relationship, they will be connected via a *consistent link*. If they form an undesirable relationship, with respect to some rule, they are connected via an *inconsistent link*.

We have made the process of link generation transparent to the rule writer by defining a new semantics for our first-order logic. Instead of returning boolean values, we generate hyperlinks. The remainder of this section will explain this new semantics in detail and show how links are generated for our example.

A consistency link consists of a set of *locators*. Each locator points to a exactly one node in a DOM tree. Links, and hence consistency relationships, are not restricted to connecting two elements in this case, but can form relationships between n elements, where $n \geq 1$. Let N be the set of nodes contained in the DOM trees of the documents that are being checked. We define the set of sets of locators as $Locators = \wp(N)$. The set of states a link can take is defined as $C = \{Consistent, Inconsistent\}$, and finally the set of consistency links is $L = C \times Locators$.

In order to support variable bindings in quantifiers and XPath expressions, we need to define a variable environment. Let Σ be our alphabet and $S = \Sigma^\star$ be the set of strings over Σ. The set of all legal variable names, as defined by the XPath specification, is then V, where $V \subset S$. A variable environment, or a collection of "bindings," is a set of tuples mapping variable names to sets of DOM nodes. The set of variable environments E is thus defined as $E = \wp(V \times \wp(N))$.

Before defining an evaluation strategy, we also need to introduce some auxiliary functions, shown in Figure 7: *flip* flips the consistency status of a link to its opposite *linkcartesian* takes two links, x and y, and produces a new link with the status of x and a set of locators consisting of the union of the sets of locators from x and y. The infix operator $\times$ takes a link and a set of links and produces a new set by applying *linkcartesian* between the single link and every individual link in the set. Finally, *bind* is used to introduce a new variable into a variable environment. In practice, this function will perform a check to make sure that

$$\begin{aligned}
\mathit{first}(x, y) &= x \\
\mathit{second}(x, y) &= y \\
\\
\mathit{flip} &: L \to L \\
\mathit{flip}((\mathit{Consistent}, y)) &= (\mathit{Inconsistent}, y) \\
\mathit{flip}((\mathit{Inconsistent}, y)) &= (\mathit{Consistent}, y) \\
\\
\mathit{linkcartesian} &: L \to L \to L \\
\mathit{linkcartesian}(x, y) &= (\mathit{first}(x),\ \mathit{second}(x) \cup \mathit{second}(y)) \\
\\
\times &: L \to \wp(L) \to \wp(L) \\
x \times Y &= \{\mathit{linkcartesian}(x, y) \mid y \in Y\} \\
\\
\mathit{bind} &: (V \times \wp(N)) \to E \to E \\
\mathit{bind}(x, e) &= \{x\} \cup e
\end{aligned}$$

Fig. 7. Auxiliary functions.

no variable is bound twice. There is no function to retrieve variables from an environment, as this task is implicitly performed by the XPath processor.

Given this new set of functions, we now modify the function $\mathcal{S}$ for evaluating XPath expressions. We formally define the function as $\mathcal{S} : S \to E \to \wp(\wp(N))$. In words, the function takes a string, which must be an XPath expression or be rejected at runtime by a parsing mechanism, and a variable environment, and returns a set of DOM nodes. The function $\mathcal{S}[\![p]\!]_e$ will thus select a set of nodes using the path p given the variable environment e.

In practice, the function will evaluate the expression argument on all documents that are being checked and then compute the union of the set of result nodes, making it possible to address all documents independent of their location. Note that we have removed the context node from the function: instead, path expressions in xlinkit must either be relative to variables, e.g., "$p/Name" in our example, in which case the context node is resolved by the processor via the variable environment, or be absolute, e.g., "/Advert," in which case the context node is the root node.

We now define our evaluation function for the *rule* nonterminal in Figure 6 and then progressively define the semantics of the various *formula* productions. Our semantics will be supported by the standard first-order logic truth evaluation semantics shown for completeness in Figure 8. We do not define a truth assignment for the top-level *rule* nonterminal, since we are not really interested in the overall truth of the formula—we are interested in link generation. It should be noted though that when the top-level nonterminal evaluates its subformula, it will pass an initial binding context containing its quantifier variable and the selected nodes.

Figure 9 shows the complete link generation semantics for our language. The function $\mathcal{R} : \mathit{rule} \to \wp(L)$ takes a rule and returns a set of consistency links.

$$
\begin{aligned}
\mathcal{F} &: \mathit{formula} \rightarrow \mathit{boolean}\\
\mathcal{F}\llbracket \forall \mathbf{var} \in \mathbf{xpath}(\mathit{formula})\rrbracket_{\epsilon} &= \mathcal{F}\llbracket \mathit{formula}\rrbracket_{bind((\mathbf{var},\{x_1\}),\epsilon)} \wedge \ldots \wedge\\
&\ \mathcal{F}\llbracket \mathit{formula}\rrbracket_{bind((\mathbf{var},\{x_n\}),\epsilon)} \mid x_i \in \mathcal{S}\llbracket \mathbf{xpath}\rrbracket_{\epsilon}\\
\mathcal{F}\llbracket \exists \mathbf{var} \in \mathbf{xpath}(\mathit{formula})\rrbracket_{\epsilon} &= \mathcal{F}\llbracket \mathit{formula}\rrbracket_{bind((\mathbf{var},\{x_1\}),\epsilon)} \vee \ldots \vee\\
&\ \mathcal{F}\llbracket \mathit{formula}\rrbracket_{bind((\mathbf{var},\{x_n\}),\epsilon)} \mid x_i \in \mathcal{S}\llbracket \mathbf{xpath}\rrbracket_{\epsilon}\\
\mathcal{F}\llbracket \mathit{formula}_1 \textbf{ and } \mathit{formula}_2\rrbracket_{\epsilon} &= \mathcal{F}\llbracket \mathit{formula}_1\rrbracket_{\epsilon} \wedge \mathcal{F}\llbracket \mathit{formula}_2\rrbracket_{\epsilon}\\
\mathcal{F}\llbracket \mathit{formula}_1 \textbf{ or } \mathit{formula}_2\rrbracket_{\epsilon} &= \mathcal{F}\llbracket \mathit{formula}_1\rrbracket_{\epsilon} \vee \mathcal{F}\llbracket \mathit{formula}_2\rrbracket_{\epsilon}\\
\mathcal{F}\llbracket \mathit{formula}_1 \textbf{ implies } \mathit{formula}_2\rrbracket_{e} &= \mathcal{F}\llbracket \mathit{formula}_1\rrbracket_{\epsilon} \rightarrow \mathcal{F}\llbracket \mathit{formula}_2\rrbracket_{\epsilon}\\
\mathcal{F}\llbracket \textbf{not } \mathit{formula}\rrbracket_{\epsilon} &= \neg\mathcal{F}\llbracket \mathit{formula}\rrbracket_{\epsilon}\\
\mathcal{F}\llbracket \mathbf{xpath_1} = \mathbf{xpath_2}\rrbracket_{\epsilon} &= \mathcal{S}\llbracket \mathbf{xpath_1}\rrbracket_{\epsilon} = \mathcal{S}\llbracket \mathbf{xpath_2}\rrbracket_{\epsilon}\\
\mathcal{F}\llbracket \mathbf{xpath_1} \neq \mathbf{xpath_2}\rrbracket_{\epsilon} &= \mathcal{S}\llbracket \mathbf{xpath_1}\rrbracket_{\epsilon} \neq \mathcal{S}\llbracket \mathbf{xpath_2}\rrbracket_{\epsilon}\\
\mathcal{F}\llbracket \textbf{same } \mathbf{var_1}\ \mathbf{var_2}\rrbracket_{\epsilon} &= \mathcal{S}\llbracket \mathbf{var_1}\rrbracket_{\epsilon} = \mathcal{S}\llbracket \mathbf{var_2}\rrbracket_{\epsilon}
\end{aligned}
$$

Fig. 8. Rule language—truth value semantics.

$$
\begin{array}{lcl}
\mathit{status} & : & \mathit{bool} \rightarrow C\\
\mathit{status}\ \top & = & \mathit{Consistent}\\
\mathit{status}\ \bot & = & \mathit{Inonsistent}\\
 & & \\
\mathcal{R} & : & \mathit{rule} \rightarrow \wp(L)\\
\mathcal{R}\llbracket \forall \mathbf{var} \in \mathbf{xpath}(\mathit{formula})\rrbracket & = & \{(\mathit{status}(\mathcal{F}\llbracket \mathit{formula}\rrbracket_{bind((\mathbf{var},\{x\}),\{\})}), \{x\})\times\\
 & & \ \mathcal{L}\llbracket \mathit{formula}\rrbracket_{bind((\mathbf{var},\{x\}),\{\})} \mid x \in \mathcal{S}\llbracket \mathbf{xpath}\rrbracket_{\{\}}\}\\
 & & \\
\mathcal{L} & : & \mathit{formula} \rightarrow \wp(L)\\
\mathcal{L}\llbracket \forall \mathbf{var} \in \mathbf{xpath}(\mathit{formula})\rrbracket_{\epsilon} & = & \{(\mathit{Inconsistent}, \{x\}) \times \mathcal{L}\llbracket \mathit{formula}\rrbracket_{bind((\mathbf{var},\{x\}),\epsilon)}\\
 & & \ \mid x \in \mathcal{S}\llbracket \mathbf{xpath}\rrbracket_{\epsilon} \wedge \mathcal{F}\llbracket \mathit{formula}\rrbracket_{bind((\mathbf{var},\{x\}),\epsilon)} = \bot\}\\
\mathcal{L}\llbracket \exists \mathbf{var} \in \mathbf{xpath}(\mathit{formula})\rrbracket_{\epsilon} & = & \{(\mathit{Consistent}, \{x\}) \times \mathcal{L}\llbracket \mathit{formula}\rrbracket_{bind((\mathbf{var},\{x\}),\epsilon)}\\
 & & \ \mid x \in \mathcal{S}\llbracket \mathbf{xpath}\rrbracket_{\epsilon} \wedge \mathcal{F}\llbracket \mathit{formula}\rrbracket_{bind((\mathbf{var},\{x\}),\epsilon)} = \top\}\\
\mathcal{L}\llbracket \mathit{formula}_1 \textbf{ and } \mathit{formula}_2\rrbracket_{\epsilon} & = & \{x \times \mathcal{L}\llbracket \mathit{formula}_2\rrbracket_{\epsilon} \mid x \in \mathcal{L}\llbracket \mathit{formula}_1\rrbracket_{\epsilon}\}\\
\mathcal{L}\llbracket \mathit{formula}_1 \textbf{ or } \mathit{formula}_2\rrbracket_{\epsilon} & = & \mathcal{L}\llbracket \mathit{formula}_1\rrbracket_{\epsilon} \cup \mathcal{L}\llbracket \mathit{formula}_2\rrbracket_{\epsilon},\\
 & & \qquad \mathit{if}\ \mathcal{F}\llbracket \mathit{formula}_1\rrbracket_{\epsilon} = \mathcal{F}\llbracket \mathit{formula}_2\rrbracket_{\epsilon}\\
 & & \mathcal{L}\llbracket \mathit{formula}_1\rrbracket_{\epsilon},\ if\ \mathcal{F}\llbracket \mathit{formula}_1\rrbracket_{\epsilon} = \top\\
 & & \mathcal{L}\llbracket \mathit{formula}_2\rrbracket_{\epsilon},\ if\ \mathcal{F}\llbracket \mathit{formula}_2\rrbracket_{\epsilon} = \top\\
\mathcal{L}\llbracket \mathit{formula}_1 \textbf{ implies } \mathit{formula}_2\rrbracket_{\epsilon} & = & \mathcal{L}\llbracket \mathit{formula}_2\rrbracket_{\epsilon},\\
 & & \qquad \mathit{if}\ \mathcal{F}\llbracket \mathit{formula}_1\rrbracket_{\epsilon} = \top \wedge \mathcal{F}\llbracket \mathit{formula}_2\rrbracket_{\epsilon} = \top\\
 & & \{x \times \mathcal{L}\llbracket \mathit{formula}_2\rrbracket_{\epsilon} \mid x \in \mathcal{L}\llbracket \mathit{formula}_1\rrbracket_{\epsilon}\},\\
 & & \qquad \mathit{if}\ \mathcal{F}\llbracket \mathit{formula}_1\rrbracket_{\epsilon} = \top \wedge \mathcal{F}\llbracket \mathit{formula}_2\rrbracket_{\epsilon} = \bot\\
 & & \{\mathit{flip}(x) \mid x \in \mathcal{L}\llbracket \mathit{formula}_1\rrbracket_{\epsilon}\},\ otherwise\\
\mathcal{L}\llbracket \textbf{not } \mathit{formula}\rrbracket_{\epsilon} & = & \{\mathit{flip}(x) \mid x \in \mathcal{L}\llbracket \mathit{formula}\rrbracket_{\epsilon}\}\\
\mathcal{L}\llbracket \mathbf{xpath_1} = \mathbf{xpath_2}\rrbracket_{\epsilon} & = & \{\}\\
\mathcal{L}\llbracket \mathbf{xpath_1} \neq \mathbf{xpath_2}\rrbracket_{\epsilon} & = & \{\}\\
\mathcal{L}\llbracket \textbf{same } \mathbf{xpath_1}\ \mathbf{xpath_2}\rrbracket_{\epsilon} & = & \{\}
\end{array}
$$

Fig. 9. Rule language—link generation semantics.

Since a rule consists of a universal quantifier, the function will build a set of nodes using a path expression, assign the nodes in the set to the quantifier variable in turn, and ask the subformula to return a set of links. Depending on the truth value of the subformula for the current assignment, the function generates a consistent or inconsistent link by prepending its current variable assignment to all links returned by the subformula.

Table II. Sample Sets for Rule Evaluation

Advert	Product
ProductName='a'	Name='c'
ProductName='b'	Name='a'
ProductName='c'	Name='f'

The quantifiers in the *formula* productions behave similarly. Both the universal and existential quantifiers will first evaluate their XPath expression, which may now include references to variables bound to some node in a parent formula, and then bind each node in the resulting node set to their variable in turn, calling the subformula evaluation. As far as link generation is concerned, the existential quantifier generates consistent links if the subformula is true for the current assignment, prepending its own current node to the links returned by the subformula. The universal quantifier generates an inconsistent link every time a subformula is false, again prepending its current node to the links returned by the subformula.

We discuss this semantics using our example rule $\forall a \in$ "/Advert" ($\exists p \in$ "/*/Product" ("$a/ProductName"="$p/Name")). Suppose that our documents contain three `Advert` elements and three `Product` elements, each shown with their subelement names and values in Table II. We use the notation X_i, where $X_i \in N$, to address the ith element in set X; for example, $Advert_2$ will address the element with value b as its product name.

In the first step, the rule evaluation will bind $Advert_1$ to a and call the existential quantifier's evaluation. Stepping through the *Product* set, the existential quantifier asks the equality predicate for a boolean result, comparing the values 'a' and 'c.' The result is false, so the existential quantifier ignores it. On the second entry in the *Product* set, the equality comparison returns true. In accordance with the semantics, the existential quantifier generates a new link of the form $(Consistent, \{Product_2\})$. For the third entry, the subformula returns false, so the link generated previously represents the whole set of links returned. The universal quantifier is now notified that the subformula has come out true for the current assignment. It prepends the current element $Advert_1$ to all links returned by the subformula, yielding the set of links $\{(Consistent, \{Advert_1, Product_2\})\}$. Here we have our diagnostic that tells us that the first advert is indeed consistent, and linking it to the information it is consistent with.

In the case of $Advert_2$, the existential quantifier will not find a product with the same name 'b.' As a consequence, its truth value will be *false* and it will return an empty set of links. The universal quantifier will obtain this truth value, and hence generate a new set of links—prepending its current assignment to the empty set of links returned by the existential quantifier—$\{(Inconsistent, \{Advert_2\})\}$. Evaluation of the third node will proceed similarly to that of the first node. The result is the union of all sets of links obtained by the universal quantifier:

$$\{(Consistent, \{Advert_1, Product_2\}),$$
$$(Inconsistent, \{Advert_2\}),$$
$$(Consistent, \{Advert_3, Product_1\})\}$$

Intuitively, these links make sense. $Advert_1$ and $Product_2$ form a desirable relationship with respect to this rule, and thus have been linked using a consistent link. For $Advert_2$, we could not find a matching element, and thus have created an inconsistent link. $Advert_2$ is inconsistent with the whole of the system rather than a particular element, so it stands alone.

Productions that contain only terminals, such as the definition of **equals** do not introduce new variables nor contain subformulas. Hence their linking semantics is to always return an empty set, since they would not be able to contribute any link locators. We can now also explain why these predicates are included in the xlinkit language, while also present in XPath. We could easily rewrite our example rule as:

$$\forall a \in \text{“/Advert”}\ (\exists p \in \text{“/*/Product[Name=\$a/ProductName]”})$$

The semantics of xlinkit's **equals** is equivalent to XPath's equality operator; that is, it compares two sets of nodes for equality. In the case of this rule, we would get an equivalent result by doing all the comparison work in XPath. It is however possible to specify rules that cannot be rewritten in this way. Take, for example, a rule, abstractly specified, of the form $\forall x(x = \text{‘5’} \vee \exists y(\sigma))$, where σ is any subformula. In this case it is not possible to rewrite the equality comparison into an XPath predicate, since it is joined in disjunction with an existential quantifier. If we want any links to be generated, the formula has to make use of xlinkit's **equals** predicate, since only xlinkit's constructs have link generation semantics.

Discussing the behavior of all the logical connectives is beyond the scope of this article. Suffice it to say at this point that the semantics given here has been tested with over 100 rules, involving all of the logical constructs, and produced good results in terms of properly highlighting the causes of inconsistency.

Our overall goal is to produce a set of links that will make it easy to spot problems. So we are eager to obtain the minimal set of links that completely expresses the consistency status of the documents that have been checked. Unfortunately it is possible for a set of links to contain redundant information. Consider the set

$$\{(consistent, \{X_1, Y_1\}), (consistent, \{Y_1, X_1\})\}$$

Since our links are bidirectional it is obvious that one of the links is redundant. Both links express the same meaning: The two elements contained within them form a desirable relationship. As an example of how this kind of redundancy arises in practice, consider a formula of the form $\forall x \in X(\forall y \in X(x = y \rightarrow same(x, y)))$. Suppose we define the set X as $X = \{\text{‘}a\text{’}, \text{‘}a\text{’}, \text{‘}b\text{’}\}$. (*Note*: X seems to be a multiset according to this notation. This is not the case in practice, since a set of nodes will contain nodes with unique identifiers. For pedagogic purposes, we show the values of the nodes rather than their identifiers.) If we evaluate the rule over X, we get the set

$$\{(inconsistent, \{X_1, X_2\}), (inconsistent, \{X_2, X_1\})\}$$

We deal with this problem by running a check over the resulting set of links, which checks if a link is a permutation of another link. If so, the link

```
<consistencyrule id="r1">
   <description>
       Each advert must refer to a product
       defined in the catalogue
   </description>

   <forall var="a" in="/Adverts">
       <exists var="p" in="/Catalogue/Products">
           <equal op1="$a/ProductName/text()"
                  op2="$p/Name/text()"/>
       </exists>
   </forall>
</consistencyrule>
```

Fig. 10. Consistency rule in XML.

is removed. The complexity of this process is $O(n^2)$, but is fast enough in practice.

We conclude the section with some observation about the complexity of our link generation semantics. First of all, we note that the evaluation function will always terminate, since the quantifiers that introduce looping into the scheme only execute their loops n times for a node set of size n. Second, the runtime complexity of the system is mainly influenced by the maximum nesting of quantifiers, i.e., it is $O(n^k)$ where k is the maximum level of quantifier nesting. Though this exponential behavior sounds problematic, it is not in practice. Most of the rules of the Unified Modeling Language, for example, which represents a complex scenario by our standards, require at most three levels of nesting. In addition, empirical results show that the evaluation is fast enough for the theoretical complexity to be ineffectual.

6. XML IMPLEMENTATION

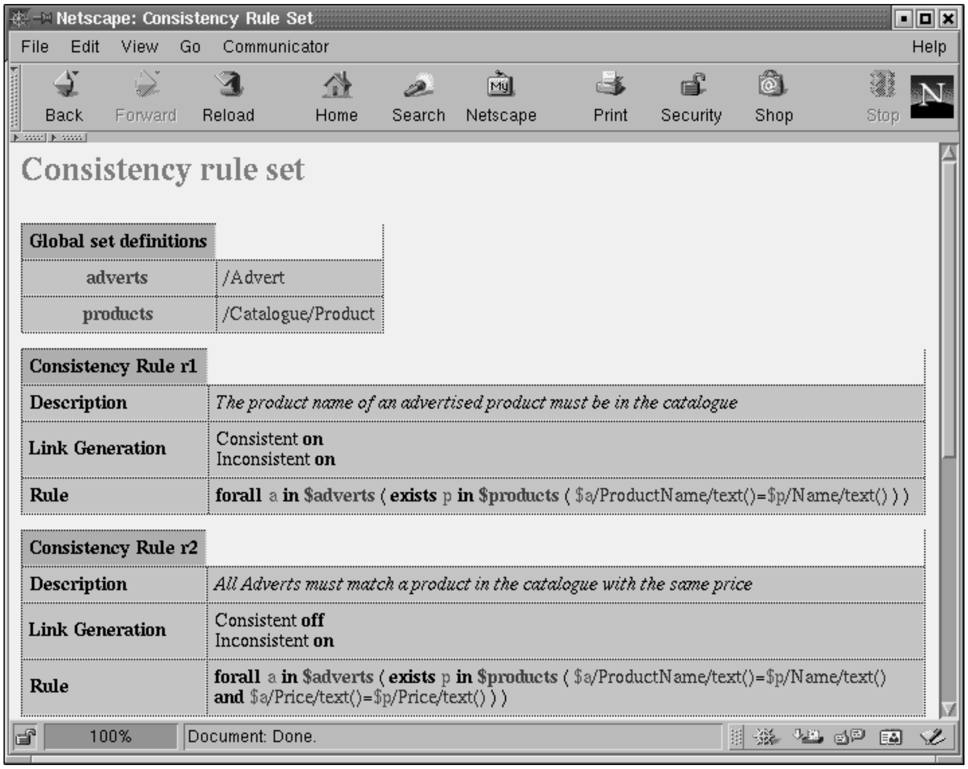
To define a concrete syntax for our language, we use an XML encoding. Encoding the language in XML has the advantage of blending more uniformly into the environment where it is going to be used. It also allows us to treat the rule files as targets that can be checked by other rules.

Presenting the encoding of the whole language is beyond the scope of this article, and the interested reader is referred to Appendix D for the complete DTD. Instead, we present two example rules expressing constraints for Wilbur's bike shop.

Our first example will be the now familiar rule *"For all Advert elements, there exists a Product element in the Catalog element where the ProductName subelement of the former equals the Name subelement of the latter."* Figure 10 shows a rule file that specifies this rule in XML format.

A rule consists of two main parts: the first entry in a rule is a `description` element, which is a natural language description of the rule that can be used for diagnosis. The following `forall` elements contains the formula that specifies the constraint.

We have written a stylesheet that transforms the rules from XML to HTML to make them more accessible for reading and browsing. The first-order logic



Consistency rule set

Global set definitions	
adverts	/Advert
products	/Catalogue/Product

Consistency Rule r1	
Description	*The product name of an advertised product must be in the catalogue*
Link Generation	Consistent **on** Inconsistent **on**
Rule	**forall** a **in $adverts** (**exists** p **in $products** ($a/ProductName/text()=$p/Name/text()))

Consistency Rule r2	
Description	*All Adverts must match a product in the catalogue with the same price*
Link Generation	Consistent **off** Inconsistent **on**
Rule	**forall** a **in $adverts** (**exists** p **in $products** ($a/ProductName/text()=$p/Name/text() **and** $a/Price/text()=$p/Price/text()))

Fig. 11. Rules in HTML.

```
<xlinkit:LinkBase
    xmlns:xlinkit="http://www.xlinkit.com"
    docSet="file://DocumentSet.xml"
    ruleSet="file://RuleSet.xml">
    <xlinkit:ConsistencyLink
        ruleid="rule.xml#/id('r1')">
        <xlinkit:State>consistent</xlinkit:State>
        <xlinkit:Locator
            xlink:href="advert1.xml#/Advert[1]"/>
        <xlinkit:Locator
            xlink:href="catalogue.xml#/Catalogue/Product[1]"/>
    </xlinkit:ConsistencyLink>
</xlinkit:LinkBase>
```

Fig. 12. Sample linkbase in XML.

formulas are translated from their XML prefix form back into infix. Figure 11 shows the translated rules.

The consistency links that are generated as a result of a check are also presented in XML, in the form of XLink *linkbases*. Figure 12 shows a sample linkbase containing only one XLink. The link indicates that it is connecting two *consistent* elements. It contains two locators that reference the elements using a URL and an XPath expression. Note that the remaining attributes

```
<DocumentsSet name="BikeDoc">
        <Description>Wilbur’s complete collection</Description>

        <DocFile href="catalogue.xml"/>

        <Set href="Adverts.xml"/>
        <Set href="Customers.xml"/>
        <Set href="Services.xml"/>
</DocumentsSet>
```

Fig. 13. Sample document set.

required in an XLink have been omitted here, as they are defined by default in the linkbase DTD.

The linkbases can be postprocessed in several ways. Figure 5 shows our servlet for interactive linkbase browsing. The user can click on a pair of locators and the servlet juxtaposes the documents and linked elements in two frames at the bottom. Using our linkbase processor, XtooX, we can also *fold* the linkbases back into the files; that is, we can take the externally defined links and insert them back into the files that they are pointing to. The link in Figure 12 would cause the insertion of a link in `advert1.xml`, linking to the first `Product` element in `catalog.xml`—and conversely, a link would be inserted in `catalog.xml`, linking to the `Advert` element in `advert.xml`. This mechanism can be used to produce a web of inconsistency information between files. We also show in the evaluation how the mechanism can be used for the automatic construction of linked standard Web content in HTML.

7. CONTENT MANAGEMENT

The selection of documents and rules to be checked against each other has to be managed. It is not feasible to always check every document against every rule, and it is certainly not necessary to check every document every time. In our bike shop example, marketing people may be interested in the status of adverts, whereas a customer relations department may be interested in the status of customer reports. Some support for partitioning documents and rules is needed to support flexible consistency management.

We use document sets, which contain a selection of documents taken from resources, and rule sets which contain several rules. A document set together with a rule set can then be submitted for checking.

Figure 13 shows a sample document set. Document sets form a hierarchy in that they consist of documents and possibly further document sets. In the figure, the `DocFile` directive is used to add a file directly into the set, while the `Set` directive includes further sets. At check time, the hierarchy is flattened and resolved into a single set. To find out whether a document needs to be checked against a rule, we check if the XPath expressions in the rule’s set definition can be applied.

Our method of retrieval of document information is not limited to XML content stored in files. Instead, we abstract from the underlying data store by providing *fetcher* classes. It is the responsibility of a fetcher to liaise with some data store in order to provide a DOM tree representation of its content.

```
<DocumentsSet name="BikeDoc">
    <Description>Wilbur’s complete collection</Description>

    <DocFile href="catalogue.xml"/>

    <Set href="Adverts.xml"/>
    <Set href="Customers.xml"/>

    <DocFile fetcher="JDBCFetcher"
        href="jdbc:mysql://www.xlinkit.com/testdb?user=wilbur#
              select * from report"/>
</DocumentsSet>
```

Fig. 14. Document set with SQL resource.

```
+-------------------------+-------------+--------------------------+
| productname             | productcode | description              |
+-------------------------+-------------+--------------------------+
| HARO SHREDDER           | B001        | Found a problem in ...   |
| HARO TR2.1              | B002        | Found a problem while... |
+-------------------------+-------------+--------------------------+
<rows>
     <row>
          <productname>HARO SHREDDER</productname>
          <productcode>B001</productcode>
          <description>Found a problem in ...</description>
     </row>
     <row>
          <productname>HARO TR2.1</productname>
          <productcode>B002</productcode>
          <description>Found a problem while...</description>
     </row>
</rows>
```

Fig. 15. Relational table XML representation.

By default, data are retrieved from XML files using the FileFetcher class; however, user-defined classes can override this behavior. Using this mechanism, it is possible to read in the content that follows a legacy format and translate it into a DOM tree, to read data from network sockets or to construct a DOM tree from a relational or object-oriented database.

As a proof of concept, we provide a JDBC fetcher, which executes a query on a database and translates the resulting table into a DOM tree. Figure 14 shows a version of Wilbur’s bikeshop document set where the service reports have been put into a relational database. The fetcher attribute in the DocFile directory overrides the default FileFetcher to select the JDBCFetcher class.

The JDBC fetcher class executes the SQL query on the relational database and transforms the resulting table into a DOM tree. Figure 15 shows a sample table of service reports fetched from Wilbur’s database by executing the JDBC query from the document set. Shown below the table is the XML representation, containing one row element for every row stored in the table and using the column name data from the data dictionary for the element names inside the rows.

```
<RuleSet name="BikeRules">
        <Description>Rules related to the Bike environment</Description>

        <RuleFile href="bike_rule.xml"
          xpath="/ConsistencyRuleSet/ConsistencyRule"/>
</RuleSet>
```

Fig. 16. Sample rule set.

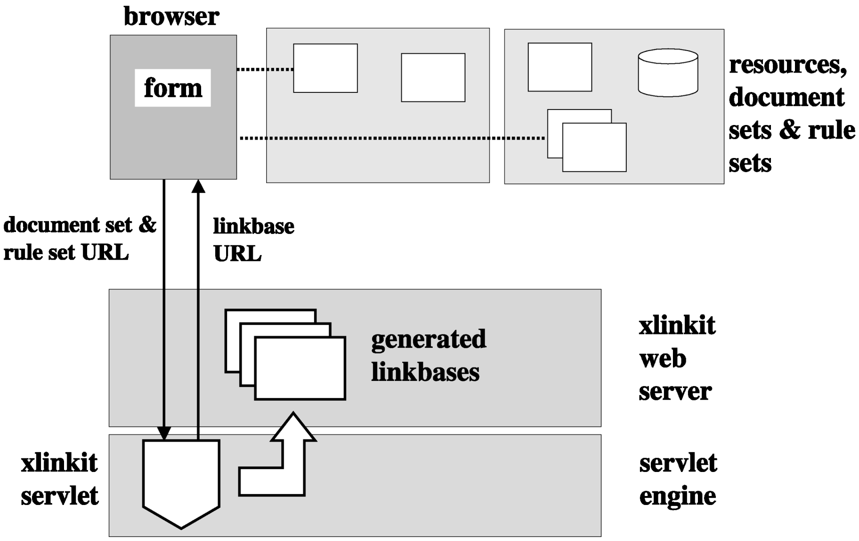


Fig. 17. Architecture overview.

Rule sets are managed in a similar fashion. A rule set contains references to rules and further rule sets. Figure 16 shows a sample rule set. A `RuleFile` element is used to specify a rule file to load and an `xpath` attribute specifies which rules from that file to actually include. The path `/ConsistencyRuleSet/ConsistencyRule` will match all `ConsistencyRule` elements included in the rule file. If that is not desired, a more constrained path such as `/ConsistencyRuleSet/ConsistencyRule[@id='r1']` could be used, which only loads the rule whose `id` attribute is equal to $r1$.

8. ARCHITECTURE

We have implemented a publicly accessible, free-to-use Internet service. Our architecture is very simple and its basic structure is shown in Figure 17.

We have implemented the check engine as a Java servlet, which is hosted on an Apache Web server running the Apache JServ servlet engine. Users are presented with the form shown in Figure 4 to enter the URL of the document set and rule set to be checked.

When the form is submitted, a new servlet instance is created to deal with the request. The servlet itself uses the Xerces XML parser from the Apache XML project to parse the documents and rule files. After checking the rules, the

```
<syllabus>
     <identity>
         <title>Concurrency</title>
         <code>3C03</code>
         <summary>The principles of concurrency
                  control and specification</summary>
     </identity>
     <teaching>
         <normal_year>3</normal_year>
         <term>1</term>
         <taught_by>
             <name>Wolfgang Emmerich</name>
             <pct_proportion>100</pct_proportion>
         </taught_by>
     </teaching>
     <subject>
         <prerequisites descr="">
             <pre_code>1B11</pre_code>
         </prerequisites>
     </subject>
</syllabus>
```

Fig. 18. Sample shortened syllabus file in XML.

servlet writes an XML file containing the generated links to the Web server's local storage. The servlet then generates a result page that contains the URL of the link base and returns it back to the browser client. The input form also gives the user a choice whether to return the raw XML file containing the links or to add a processing directive for it to be translated into HTML using a stylesheet. Please refer back to Figure 5 for an example of the latter.

9. EVALUATION

This section presents two case studies that we used to evaluate the expressiveness of our rule language and the scalability of our implementation. Our major goal was to find out if xlinkit can be applied to a real-world example. In addition, we also wanted a "stress-test" scenario for performance, scalability, and expressiveness. Our first study checks the consistency of course syllabus information and the second study performs a validation of multiple software engineering documents.

The Department of Computer Science at University College London recently introduced a new curriculum and associated course syllabi. In order to provide high-quality information in the wide variety of different representations required, it decided to adopt XML as a common format. The system has to hold a curriculum and provide links to the syllabi for students, depending on which degree program they are pursuing. Figure 18 shows a sample abbreviated syllabus file for a course. Each course is held in a separate XML file. The curricula for degree programs are kept in a single file. For each degree program, the mandatory and optional courses are listed, grouped by the year in which they can be selected. Figure 19 shows a fragment from the curricula file.

The process of syllabus development is highly decentralized, with different people providing additions and corrections to course syllabi. Curriculum files

```
<Curricula>
    <Curriculum>
        <Programme>
            <Title>CS</Title>
            <Award>BSc</Award>
        </Programme>
        <Year number="1">
            <Constraint>6 compulsory half-units,
                        2 optional half-units, no more than 1
                        optional half-unit can be non-programme.
            </Constraint>
            <Course value="Standard">
                <Name>Computer Architecture I</Name>
                <Code>1B10</Code>
                <Theme>Architecture</Theme>
                <Type requirement="C" level="F"/>
                <Dept>CS</Dept>
            </Course>
            ...
        </Year>
        ...
    </Curriculum>
</Curricula>
```

Fig. 19. Curriculum fragment.

contain information related to the individual syllabus files. For example, course codes mentioned in the curriculum files have to be part of a syllabus definition. Altogether, ten rules were identified as necessary to preserve the consistency of the system. The complete list of rules can be found in Appendix B.

It is desirable for navigation purposes to provide hyperlinks from the curriculum to individual courses. However, manually adding links from the curriculum file to all 48 syllabus files would be error-prone, as files get deleted and courses renamed. It is preferable to use the semantically equivalent information in the files (e.g., the course codes) to generate the hyperlinks automatically. We used xlinkit to achieve both goals.

Figure 20 shows the time used for checking each rule against all 52 documents. The syllabus files were all around 5 kilobytes in size and the curriculum is 110 kilobytes in size. Checking was performed on a 700 Mhz Intel machine with 128 Mb of RAM, running Mandrake Linux 8.0 with kernel 2.4.9 and the IBM JDK 1.3. The total checking time was 11.1 seconds, with the most complex rule taking 8 seconds to check. In total, 410 consistent and 11 inconsistent links were generated.

The exceptional checking time on rule 5 was caused by a transitive closure operation. Our current implementation of this operator, outlined in Section 4, is still a proof of concept. It uses a rather naive algorithm and has not been optimized for efficiency.

Our second goal in the case study was to provide a fully linked HTML version of the department's curriculum to be browsed by staff and students. One of the rules for the curriculum is that every course listed in the curriculum must have a syllabus definition. If the rule is satisfied, a consistent link is generated

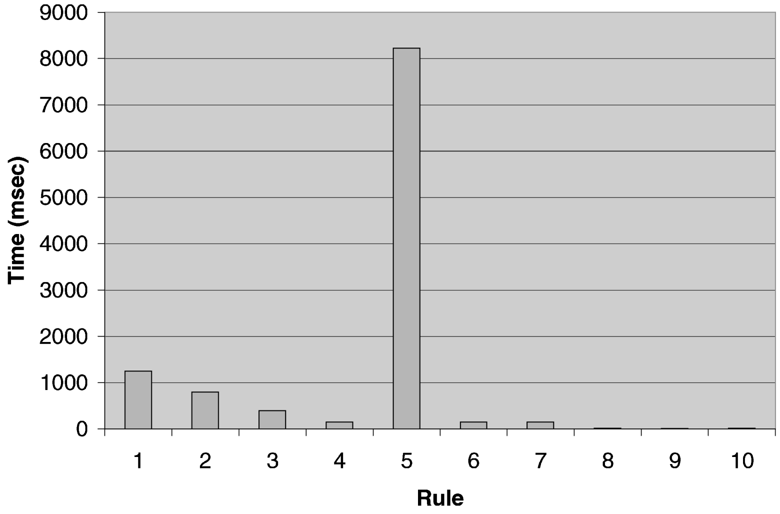


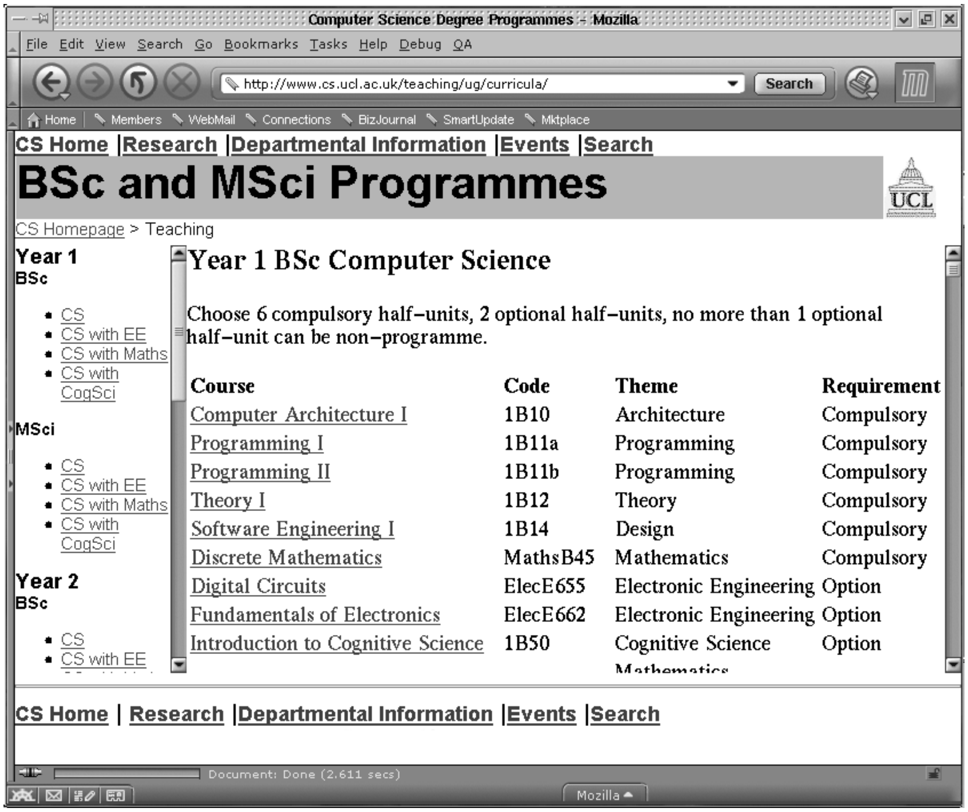
Fig. 20. Syllabus study timings.

from the course entry in the curriculum to the syllabus defining the course. We used XtooX, as in the example section, to fold all consistent links from this rule back into the XML file containing the curriculum. We then only had to provide a simple XSL stylesheet that transforms the XML file and simple links into an HTML representation. Figure 21 shows the “production version” of the curriculum Website, as generated by xlinkit.

Our second case study uses our rule language to express some of the static semantic constraints of the Unified Modeling Language (UML) [Object Management Group 2000a] and checks them against several models stored in XMI [Object Management Group 2000b], an XML-based metamodel interchange format that supports the UML. The typical scenario for this study is a distributed development team working on the same model and producing their own additions and copies of documents. If frequent merging of the documents is not feasible, due to geographical separation, for exapmple, checks can be used to ensure consistency.

We have expressed all but three constraints of the UML `Foundation.Core` package, the package dealing with static information such as classes; they are listed in Appendix C. Of the three that were not expressed, two are enforced by the XMI DTD and do not have to be checked, and one requires information that is not supplied by XMI. Figure 22 shows one constraint for Associations as expressed in the xlinkit XML format.

We have applied these constraints to several UML models: a small design model of a meeting scheduler [Feather et al. 1997], a medium-size model shipped as an example with Rational Rose and 19 industrial-size models provided by an investment bank. We use the number of ModelElement objects contained in each model as a measure of scale, since almost everything in the UML metamodel derives from ModelElement. Our small model contains 93 elements; the medium size model has 610 elements. The number of elements contained in


Computer Science Degree Programmes - Mozilla
File Edit View Search Go Bookmarks Tasks Help Debug QA
http://www.cs.ucl.ac.uk/teaching/ug/curricula/
Search
Home
Members
WebMail
Connections
BizJournal
SmartUpdate
Mktplace
CS Home |Research |Departmental Information |Events |Search
UCL
CS Home | Research |Departmental Information |Events |Search
Document: Done (2.611 secs)
Mozilla

BSc and MSci Programmes

CS Homepage > Teaching

Year 1
BSc

- CS
- CS with EE
- CS with Maths
- CS with CogSci

MSci

- CS
- CS with EE
- CS with Maths
- CS with CogSci

Year 2
BSc

- CS
- CS with EE

Year 1 BSc Computer Science

Choose 6 compulsory half–units, 2 optional half–units, no more than 1 optional half–unit can be non–programme.

Course	Code	Theme	Requirement
Computer Architecture I	1B10	Architecture	Compulsory
Programming I	1B11a	Programming	Compulsory
Programming II	1B11b	Programming	Compulsory
Theory I	1B12	Theory	Compulsory
Software Engineering I	1B14	Design	Compulsory
Discrete Mathematics	MathsB45	Mathematics	Compulsory
Digital Circuits	ElecE655	Electronic Engineering	Option
Fundamentals of Electronics	ElecE662	Electronic Engineering	Option
Introduction to Cognitive Science	1B50	Cognitive Science Mathematics	Option

Fig. 21. Automatically generated links in the curriculum.

the industrial models ranges from 64 to 2834 elements. In terms of file size, the models range from around 100 kilobytes to 6 megabytes.

All results listed below were obtained on a 600 Mhz Intel machine with 384 Mb of RAM, running Redhat Linux 6.1 with kernel 2.2.19 and the IBM JDK 1.3. We will discuss the results obtained by checking the UML Core constraints against the industrial models. It took a total of 4 minutes to check all rules against all files, counting only the time taken to check individual rules and ignoring parsing overhead. While parsing takes more than 2 minutes in total over all files, the variation in performance between XML parsers means that including it would introduce unnecessary noise into the evaluation. Figure 23 shows the time taken for each rule over all files.

We can observe several interesting properties from the figure. Most rules take roughly the same amount of time to check, but three rules stand out as taking much longer. This is due to two factors: These rules apply to more files than others; for example, almost every model has classes whereas few have association classes, i.e., the association class rules do not apply in many cases. Second, the complexity of the XPath expressions in the rules varies greatly. Some expressions use straightforward tree paths, whereas others require expensive functions like id lookup. This is a feature of the rather complex design

```
<consistencyrule id="a1">
  <description>
      The AssociationEnds must have a unique name within the Association
  </description>
  <linkgeneration>
      <consistent status="off"/>
      <eliminatesymmetry status="on"/>
  </linkgeneration>

  <forall var="a" in="$associations">
      <forall var="x" in="$a/Foundation.Core.Association.connection/
                           Foundation.Core.AssociationEnd">
          <forall var="y" in="$a/Foundation.Core.Association.connection/
                              Foundation.Core.AssociationEnd">
              <implies>
                <equal op1="$x/Foundation.Core.ModelElement.name/text()"
                       op2="$y/Foundation.Core.ModelElement.name/text()"/>
                <same op1="$x" op2="$y"/>
              </implies>
          </forall>
      </forall>
  </forall>
</consistencyrule>
```

Fig. 22. Sample rule from the UML Foundation/Core package.

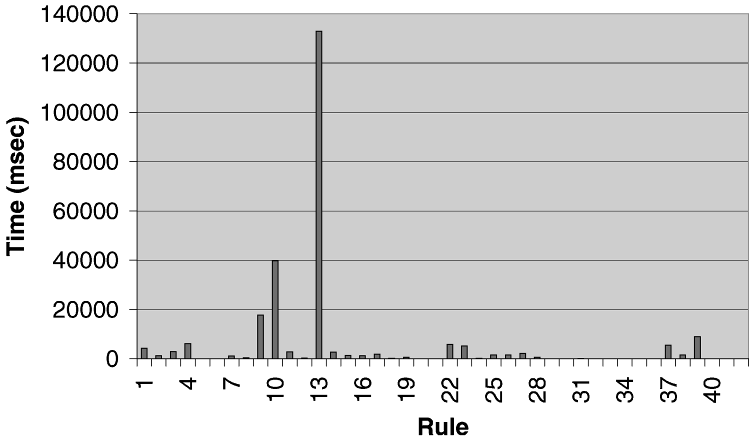


Fig. 23. Rule totals for UML core rules.

of XMI. XPath selection is the single most expensive process in rule checking, and hence the complexity of the path has the greatest impact—far greater than the complexity of the formula in terms of nested quantifiers! The rule that takes longest to check makes use of features in XPath, the `name` function and the union operator, that do not seem to be well implemented in the XPath processor we use and hence take longest to check.

In total, over all files, more than 8,000 inconsistent links were generated. Consistent link generation was turned off, since we were only interested in

```
<xlinkit:ConsistencyLink ruleid="assoc.xml#//consistencyrule[@id=‘r1’]">
 <xlinkit:State>inconsistent</xlinkit:State>
 <xlinkit:Locator xlink:href="meeting2.xml#/XMI/XMI.content[1]/
  Model_Management.Model[1]/Foundation.Core.Namespace.ownedElement[1]/
  Foundation.Core.Association[1]"/>
 <xlinkit:Locator xlink:href="meeting2.xml#/XMI/XMI.content[1]/
  Model_Management.Model[1]/Foundation.Core.Namespace.ownedElement[1]/
  Foundation.Core.Association[1]/Foundation.Core.Association.connection[1]/
  Foundation.Core.AssociationEnd[1]"/>
 <xlinkit:Locator xlink:href="meeting2.xml#/XMI/XMI.content[1]/
  Model_Management.Model[1]/Foundation.Core.Namespace.ownedElement[1]/
  Foundation.Core.Association[1]/Foundation.Core.Association.connection[1]/
  Foundation.Core.AssociationEnd[2]"/>
</xlinkit:ConsistencyLink>
```

Fig. 24. Sample consistency link generated from UML model.

finding inconsistencies. Figure 24 shows one of the consistency links generated by the rule shown previously in Figure 22. It shows clearly how the association has been linked to the two association-ends with equal names, thus exhaustively specifying the ternary relationship that caused the inconsistency.

Although the number of inconsistencies seems large, given that the models were exported from a CASE tool, it can be explained. Some of the models included in the check were analysis or high-level design models, so they were incomplete with respect to definition of fundamental data types, had operation parameter types missing, and similar problems. If this system were to be used in practice, developers could identify a suitable subset of rules and assemble them into a "high-level model" ruleset that would be more permissive.

While we believe the timing results in our two case studies were satisfactory for standalone consistency checks, they may not be if frequent checks are necessary. A downside of our current implementation is that it checks all documents against all rules every time a check is invoked. In an interactive environment, an incremental scheme that performs a smaller check, depending on the changes made to documents, would be preferable.

Another problem we have encountered is that of memory usage. Our check engine needs to retain the DOM trees for all documents in memory in order to be able to execute XPath queries on them. In the case studies, this was not a problem, however we have checked some software engineering documents that required a considerable amount of memory. We are investigating a number of strategies to address this problem: A distributed supervisor-worker architecture for very large datasets, where individual workers handle a number of documents and send back results of XPath queries, the use of an XML database with caching features to avoid retaining the entire DOM tree in memory, and a scheduling system that loads documents on demand when they need to be queried.

10. APPLICATIONS

xlinkit is a highly generic technology. It can be applied wherever we want to establish links between Web resources, broadly construed, where those links reflect relationships between resource types. In particular, rather than directly authoring and maintaining links, xlinkit can provide semantically aware link generation.

Our principal interest derives from our software engineering background. Thus we have worked on applications largely in this area, most notably managing the consistency of complex development models produced by distributed teams.

A large range of other applications primarily focusing on link generation and content management have been worked on by us or our partners. For example, information about important customers can be found in many places in sales files, service agreements, problem reports, logistics, and supply records. xlinkit can be used to build a Web-based customer relationship management system that allows us to navigate between all the pieces of information which reflect the interests of a single customer.

eCRM (Electronic Customer Relationship Management) of this form is an example of a broad class of lightweight intranet portals. Many organizations have information in many different databases scattered across different sites. xlinkit can be used to build portals that can deliver coordinated access to this information and diagnose consistency problems.

The idea of delivering Web content on multiple channels such as Web-TV, phones, PDAs is now common. Unfortunately, content has to be adapted for each channel to make a high-value service. Content adaptation risks inconsistency with its attendant problems. xlinkit can be used to support navigation between information presented in different channels and identify problems. Web sites that aggregate content can use xlinkit to add value by providing content-relevant navigation without directly authoring links.

We are investigating applications of xlinkit in the financial domain, in particular for checking the consistency of financial trading information. We have used xlinkit both to validate derivative trading data encoded in FpML [Gurdel 2001] and also to match trading data between counterparties.

Other applications, which have not been fully evaluated but appear promising, are consistency of information in service-level agreements, security policy, and network management policy.

11. RELATED WORK

This account of related work is not intended to be a survey of consistency management, for which we refer to Nuseibeh et al. [2000]. Below, we highlight some key comparison points and work that has had a particular influence on xlinkit.

Consistency management has been recognized as an important issue by the programming language and software engineering communities. Early work in this area can be found in publications on programming environments such as the Cornell Synthesizer Generator [Reps and Teitelbaum 1984], Gandalf [Habermann and Notkin 1986], or Centaur [Borras et al. 1988]. These environments typically provide syntax-directed editors. When the user has finished entering a construct, incremental consistency checks related to the static programming language semantics being used are carried out. These semantic checks are typically carried out on a centralized data structure such as an abstract syntax tree. Later work on software development environments (SDEs) such as IPSEN [Nagl 1996], Arcadia [Taylor et al. 1988], ESF [Schäfer and Weber 1989], ATMOSPHERE [Boarder et al. 1989] and GOODSTEP [Emmerich 1996] raised the complexity by integrating tools for different languages. The latter in particular allowed the specification of *semantic rules* [Emmerich 1996]. Checks for semantic integrity between documents could be triggered by user actions. Our approach represents a generalization, in that it builds on the open model of XML rather than specific programming formalisms. In addition, we allow for the distribution of the documents and provide diagnostics in the form of links.

A viewpoint [Finkelstein et al. 1992] allows developers to express a design fragment in some specification language, together with additional attributes describing the viewpoint. Multiple viewpoints can describe the same design fragment, leading to overlap and hence the possibility of inconsistency. The issues involved in inconsistency handling of multiperspective specifications are outlined in Finkelstein et al. [1994]. Research in the viewpoints area also introduces the idea of *consistency rules* [Easterbrook et al. 1994] between distributed specifications. The work on viewpoints has spun off our continuing interest in consistency management, and in particular our tolerant view in which consistency is not always enforced. For a detailed discussion, see Finkelstein [2000]. Although a lot of theoretical work on viewpoints and the associated consistency checking scheme has been done, no generic implementation was ever provided. Our work realizes these ideas by providing a concrete implementation on top of which a viewpoint framework can be built.

Traditional database integrity notions have been extended to cope with semistructured data [Buneman et al. 2000] and XML content in particular [Fan and Simeon 2000]. The fundamental goal of this work, and hence the approach, is different. Integrity constraints are present in databases to *prevent* inconsistency, from occurring. In many application domains, most notably software engineering, inconsistency cannot be prevented and is not necessarily undesirable—for a discussion of this approach in a database context, see Balzer [1991]. Hence the focus is not on the language as such, but on producing good diagnostics. The hyperlinks that we offer as diagnostics establish a clear relationship between inconsistent elements. We note also that traditional integrity constraints and the restricted path constraints in the first paper are not sufficient to express some of the constraints required in software engineering notations such as the UML.

The problem of verifying constraints on Websites is discussed in Fernandez et al. [1999] and applied in Fernandez et al. [2000]. It is important to distinguish between the goals of these approaches and our own goals: Our constraints check if a set of data is consistent, whereas the approaches in the paper check if *any instance* of a schema graph will satisfy the constraint. If the schema graph does not satisfy the property, modifications are suggested that will lead to valid instance graphs. Since we wish to tolerate inconsistency to introduce flexibility and because it is sometimes not possible to change the schemas of documents, for example when standardised schemas are used, we cannot adopt this approach, but instead focus on detecting inconsistencies in instance documents.

Standard query languages can and have been used to specify integrity constraints [Henrich and Däberitz 1996]. In the context of XML, such an approach would be feasible by using an XML query language such as XQuery [Chamberlin et al. 2001]. We regard such an approach as lower-level than xlinkit, since the user would have to “manually implement” the linking semantics for each query, rather than achieving the desired goal of specifying a declarative constraint. The user would, for example, have to write two queries for each constraint in xlinkit, one that selects the combination of consistent elements and one that selects the combination of inconsistent elements.

Even if only inconsistent elements are to be selected, xlinkit’s semantics for logical constructs has been defined very carefully to discard locators in links that do not add any information, the goal being to maximize the diagnostic value by discarding noise. For example, in the formula $a \rightarrow b$, where a and b are subformulas, if a is *true* and b is also *true* then we include the links returned by b into our results. Since a change in the truth value of a would not change the overall result, we discard the links returned by a as irrelevant. This semantics has been extensively tested and produced good results in all our case studies. If XQuery were to be used directly, the user would have to handcode the combination of elements to be included into links for all combinations of truth values of a and b, leading to huge queries, and that is without removing permutations of links. We also note that we would have to wait for a framework based on XQuery that includes proper document management, so as to achieve the distribution transparency that our service provides.

The hypertext community has worked on the problem of automatic link generation. For a survey of this topic we refer to Wilkinson and Smeaton [1999]. Much work in the area has focused on textual documents and many approaches based on information retrieval techniques such as similarity measures can be found. We exploit the structure afforded by XML, and its widespread use for storing data rather than textual information in our approach to provide a much richer and more fine-grained expression of linking semantics.

There is a growing body of work concerned with applications of hypertext in software engineering. The CHIMERA project [Anderson et al. 1994] demonstrates multiple document views and the capability of separating linking information from the underlying documents. It does not support consistency checking. CHIME [Devanbu et al. 1999] provides a framework for folding links into legacy software documents using information from software analysis tools. The work provides a strong case for the sort of browsing that our approach provides.

Our work has some analogies with Schematron [Jelliffe 2000], an XML structure validator that employs XSL and XPath to traverse documents and check constraints. Schematron was built as an alternative to traditional, grammar-based systems for document validation. The focus of our work is clearly different, as we are interested in relationships between distributed documents. Checking constraints between multiple documents can be achieved in Schematron using the XPath `document` function, which would, however, hardcode the names of documents. Our strict distinction between rule sets and document sets, and the transparency of our rules with respect to underlying storage, allows the same set of rules to be applied to multiple sets of documents. Also, Schematron does not generate hyperlinks, one of the important components of our approach.

Finally, xlinkit builds on two previous consistency checking schemes [Ellmer et al. 1999] which have substantially influenced the ideas on which it is based. In both cases these were standalone applications and used a rule language based on a much more restricted form of first-order logic. The language, architecture, and content management framework are novel and the genericity, scaleability, and performance of the xlinkit approach distinguish it from the earlier prototypes.

12. FUTURE WORK

A "static" application service such as ours does more work than is really necessary because it has to recheck all documents against all rules upon request. When documents are changed, we would like to recheck only those rules that are affected by the changes. Such an *incremental checking* scheme is certainly a barrier we have to overcome if our approach is to scale to very large datasets that need regular checking. We have already devised and prototyped an algorithm for determining the set of rules to be checked after changes and are planning to implement the scheme for testing and benchmarking.

Conflict resolution is a logical back-end of a consistency check and has not been discussed in this article. It is assumed that the user will refer to our linkbases as a diagnostic tool and then take action in accordance with some real-world process. While we believe that conflict resolution can never be fully automated, it should still be possible to set certain default actions for handling trivial inconsistencies. Integration with a workflow management system may prove valuable in this respect, and we will investigate this option. Achieving this goal without compromising the lightweight characteristics of xlinkit will, however, be a challenge.

The evaluation section has already mentioned the problem of maintaining a DOM tree for all documents in memory during a check. We are currently investigating both architectural styles and implementation mechanisms to address this problem. On the architectural side, a distributed architecture can be used to spread the memory load over several machines, whereas on the implementation side, a scheduling mechanism for document loading together with an XML database with XPath support may provide some benefits.

Our rule language has a rather limited range of predicates, basically consisting of equality operations. Even for such a simple operation as equality, a wide range of requirements can be found, depending on the application domain—for example inclusion of a particular business date in a cash flow. We are currently adding mechanisms to our evaluation engine that allow the dynamic definition of new predicates in Javascript to address this problem.

13. CONCLUSION

This article has described xlinkit, a lightweight application service that provides rule-based link generation and checks the consistency of distributed Web resources. xlinkit leverages standard Internet technologies. It supports document distribution and can support multiple deployment models. It has a formal basis, and evaluation has shown that it scales, both in terms of the size of documents and in the number of rules. We have identified some important applications and pointed to future directions for our work.

xlinkit is the product of long-standing research looking at consistency management. It is available as an open source package. Several research groups and industrial partners have already started to use xlinkit, and we are keen to see it applied further. The evaluation package and examples can be found at `http://www.xlinkit.com`.[1]

APPENDIX

A. WILBUR'S BIKE SHOP SAMPLE FILES

A.1 Product Catalog Sample

```
<Catalogue>
    <Product>
        <Name>HARO SHREDDER</Name>
        <Code>B001</Code>
        <Price currency="sterling">349.95</Price>
        <Description>Freestyle Bike.</Description>
    </Product>
    <Product>
        <Name>HARO TR2.1</Name>
        <Code>B002</Code>
        <Price currency="sterling">179.95</Price>
        <Description>BMX / Trail Bike.</Description>
    </Product>
</Catalogue>
```

A.2 Sample Advert File

```
<Advert>
    <ProductName>HARO SHREDDER</ProductName>
    <Price currency="sterling">349.95</Price>
    <Description>Freestyle Bike. Super versatile frame
        for dirt, street, vert or flat. New full cromoly
        frame. Fusion MegaTube axle extenders.
    </Description>
</Advert>
```

A.3 Sample Service Report File

```
<ServiceReport>
    <CustomerIdentity reg_number="3645"/>
```

[1] xlinkit is protected by PCT 9914232.5.

```
    <Report>
        <ProductName>HARO SHREDDER</ProductName>
        <ProductCode>B001</ProductCode>
        <ProblemDescr>Found a problem in ...</ProblemDescr>
    </Report>
</ServiceReport>
```

A.4 Sample Customer Report File

```
<CustomerReport>
    <CustomerIdentity>
        <FirstName>Licia</FirstName>
        <FamilyName>Capra</FamilyName>
        <Reg_Number>3645</Reg_Number>
    </CustomerIdentity>
    <Purchase>
        <ProductName>HARO SHREDDER</ProductName>
        <ProductCode>B001</ProductCode>
    </Purchase>
    <Purchase>
        <ProductName>Shimano LX Mountain Bike
                     Crank Set</ProductName>
        <ProductCode>A102</ProductCode>
    </Purchase>
</CustomerReport>
```

B. CURRICULUM CASE STUDY RULES

Table III. Curriculum Study Rules

1	Each course (of the CS department) must have a syllabus
2	The year of the course in the curriculum corresponds to the year in the syllabus
3	There must not be two courses with the same code
4	Each course listed as a pre-requisite in a syllabus must have a syllabus definition
5	A course cannot be a pre-requisite of itself
6	Each course in a studyplan is identified in the curricula
7	A student cannot take the same course twice
8	1st year BSc/CS and MSci: 6 compulsory half-units
9	1st year BSc/CS and MSci: 2 optional half-units
10	1st year BSc/CS and MSci: no more than 1 optional half-units can be Non-programme

C. UML FOUNDATION.CORE RULES

C.1 Association

[1] The AssociationEnds must have a unique name within the Association

[2] At most one AssociationEnd may be an aggregation or composition

[3] If an Association has three or more AssociationEnds, then no AssociationEnd may be an aggregation or composition

[4] The connected Classifiers of the AssociationEnds should be included in the Namespace of the Association

C.2 AssociationClass

[1] The names of the AssociationEnds and the StructuralFeatures do not overlap

[2] An AssociationClass cannot be defined between itself and something else

C.3 AssociationEnd

[1] The Classifier of an AssociationEnd cannot be an Interface or a DataType if the association is navigable from that end

[2] An Instance may not belong by composition to more than one composite Instance

C.4 BehavioralFeature

[1] All parameters should have a unique name

[2] The type of the Parameters should be included in the Namespace of the Classifier

C.5 Class

[1] If a Class is concrete, all the Operations of the Class should have a realizing method in the full descriptor

C.6 Classifier

[2] No Attributes may have the same name within a Classifier

[3] No opposite AssociationEnds may have the same name within a Classifier

[4] The name of an Attribute may not be the same as the name of an opposite AssociationEnd or a ModelElement contained in the Classifier

[5] The name of an opposite AssociationEnd may not be the same as the name of an Attribute or ModelElement contained in the Classifier

[2] For each Operation in a specification realized by a Classifier, the Classifier must have a matching Operation

C.7 Component

[1] A Component may only contain other Components

C.8 Constraint

[1] A Constraint cannot be applied to itself

C.9 DataType

[1] A DataType can only contain Operations, which all must be queries

[2] A DataType cannot contain any other model elements

C.10 GeneralizableElement

[1] A root cannot have any Generalizations
[2] No GeneralizableElement can have a parent Generalization to an element which is a leaf
[4] The parent must be included in the namespace of the GeneralizableElement

C.11 Interface

[1] An Interface can only contain Operations
[2] An Interface cannot contain any ModelElements
[3] All Features defined in an Interface are public

C.12 Method

[1] If the realized Operation is a query, then so is the method
[2] The signature of the Method should be the same as the signature of the realized Operation
[3] The visibility of the Method should be the same as for the realized Operation

C.13 Namespace

[1] If a contained element, which is not an Association or Generalization has a name, then the name must be unique in the Namespace
[2] All Associations must have a unique combination of name and associated Classifiers in the Namespace

C.14 StructuralFeature

[3] The connected type should be included in the owner's Namespace

C.15 Type

[1] A Type may not have any methods
[2] The parent of a type must be a type

D. RULE LANGUAGE XML DTD

```
<!ELEMENT consistencyruleset (globalset*,consistencyrule+)>
<!ELEMENT globalset EMPTY>
<!ATTLIST globalset
    id ID #REQUIRED
    xpath CDATA #REQUIRED>
<!ELEMENT consistencyrule (description?,linkgeneration?,forall)>
<!ATTLIST consistencyrule
    id ID #REQUIRED>
<!ELEMENT description (#PCDATA)>
<!ELEMENT linkgeneration (consistent?,inconsistent?,
                          eliminatesymmetry?)>
<!ELEMENT consistent EMPTY>
<!ATTLIST consistent
    status (on | off) "on">
<!ELEMENT inconsistent EMPTY>
```

```
<!ATTLIST inconsistent
    status (on | off) "on">
<!ELEMENT eliminatesymmetry EMPTY>
<!ATTLIST eliminatesymmetry
    status (on | off) "off">
<!ELEMENT forall (exists|forall|and|or|implies|not|equal|
                  notequal|same|subset|intersect)?>
<!ATTLIST forall
    var CDATA #REQUIRED
    in CDATA #REQUIRED
    mode (exhaustive | instance) "exhaustive">
<!ELEMENT exists (exists|forall|and|or|implies|not|equal|
                  notequal|same|subset|intersect)?>
<!ATTLIST exists
    var CDATA #REQUIRED
    in CDATA #REQUIRED
    mode (exhaustive | instance) "exhaustive">
<!ELEMENT and (exists|forall|and|or|implies|not|equal|
               notequal|same|subset|intersect)*>
<!ELEMENT or (exists|forall|and|or|implies|not|equal|
              notequal|same|subset|intersect)*>
<!ELEMENT implies (exists|forall|and|or|implies|not|equal|
                   notequal|same|subset|intersect)*>
<!ELEMENT not (exists|forall|and|or|implies|not|equal|
               notequal|same|subset|intersect)>
<!ELEMENT equal EMPTY>
<!ATTLIST equal
    op1 CDATA #REQUIRED
    op2 CDATA #REQUIRED>
<!ELEMENT notequal EMPTY>
<!ATTLIST notequal
    op1 CDATA #REQUIRED
    op2 CDATA #REQUIRED>
<!ELEMENT same EMPTY>
<!ATTLIST same
    op1 CDATA #REQUIRED
    op2 CDATA #REQUIRED>
<!ELEMENT subset EMPTY>
<!ATTLIST subset
    op1 CDATA #REQUIRED
    op2 CDATA #REQUIRED
    size CDATA "0">
<!ELEMENT intersect EMPTY>
<!ATTLIST intersect
    op1 CDATA #REQUIRED
    op2 CDATA #REQUIRED
    size CDATA "0">
```

ACKNOWLEDGMENTS

We would like to thank Zeeshawn Durrani, who wrote the linkbase stylesheet, and our colleagues from earlier incarnations of this project, Danila Smolko, Ernst Ellmer, Andrea Zisman, and Torbjorn Revheim, for their contributions. We would also like to thank the Apache Software Foundation and its volunteers for its continued and free provision of high-quality tools such as Xerces and Xalan, which have greatly simplified our work. The XLink working group also deserves thanks, in particular we are grateful to Eve Maler for technical feedback. We thank the anonymous reviewers, who have produced detailed reviews and helped to improve this paper a lot. Finally, we gratefully acknowledge the financial support from Zuhlke Engineering for Licia Capra and Christian Nentwich.

REFERENCES

ANDERSON, K. M., TAYLOR, R. N., AND WHITEHEAD, E. J. 1994. Chimera: Hypertext for heterogeneous software environments. In *Proceedings of the European Conference on Hypermedia* (Edinburgh, UK, Sept. 1994).

APPARAO, V., BYRNE, S., CHAMPION, M., ISAACS, S., JACOBS, I., HORS, A. L., NICOL, G., ROBIE, J., SUTOR, R., WILSON, C., AND WOOD, L. 1998. Document Object Model (DOM) level 1 specification. W3C Recommendation http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001 (Oct.), World Wide Web Consortium.

BALZER, R. 1991. Tolerating inconsistency. In *Proceedings of the 13th International Conference on Software Engineering* (Austin, TX, May 1991), 158–165. IEEE Computer Society Press.

BOARDER, J., OBBINK, H., SCHMIDT, M., AND VÖLKER, A. 1989. Advanced techniques and methods of system production in a heterogeneous, extensible, and rigorous environment. In *Proceedings of the 1st International Conference on System Development Environments and Factories* (Berlin, 1989), N. Madhavji, W. Schäfer, and H. Weber, Eds. 199–206. Pitman Publishing.

BORRAS, P., CLÉMENT, D., DESPEYROUX, T., INCERPI, J., KAHN, G., LANG, B., AND PASCUAL, V. 1988. CENTAUR: The system. *ACM SIGSOFT Soft. Eng. Notes 13*, 5, 14–24.

BRAY, T., PAOLI, J., SPERBERG-MCQUEEN, C. M., AND MALER, E. 2000. Extensible markup language. Recommendation http://www.w3.org/TR/2000/REC-xml-20001006 (Oct.), World Wide Web Consortium.

BUNEMAN, P., FAN, W., AND WEINSTEIN, S. 2000. Path constraints in semistructured databases. *J. Comput. Syst. Sci. 61*, 2, 146–193.

CHAMBERLIN, D., FLORESCU, D., ROBIE, J., SIMEON, J., AND STEFANESCU, M. 2001. XQuery: A query language for XML. Working draft (Feb.), World Wide Web Consortium (W3C). http://www.w3.org/TR/xquery/.

CLARK, J. 1999. XSL transformations (XSLT). Tech. Rep. http://www.w3.org/TR/xslt (Nov.), World Wide Web Consortium.

CLARK, J. AND DEROSE, S. 1999. XML path language (XPath) version 1.0. Recommendation http://www.w3.org/TR/1999/REC-xpath-19991116 (Nov.), World Wide Web Consortium.

CONSORTIUM, W. W. W. 2000. Amaya. http://www.w3.org/Amaya/.

DEROSE, S., MALER, E., AND ORCHARD, D. 2001. XML linking language (XLink) version 1.0. W3C recommendation http://www.w3.org/TR/xlink/ (June), World Wide Web Consortium.

DEVANBU, P., CHEN, Y.-F., GANSNER, E., MULLER, H., AND MARGIN, J. 1999. CHIME—Customizable hyperlink insertion and maintenance engine for software engineering environments. In *Proceedings of the 21st International Conference on Software Engineering* (Los Angeles, CA, May 1999), 473–482. ACM Press.

EASTERBROOK, S., FINKELSTEIN, A., KRAMER, J., AND NUSEIBEH, B. 1994. Coordinating distributed ViewPoints: The anatomy of a consistency check. *Int. J. Concurrent Eng. Res. Appl. 2*, 3, 209–222.

ELLMER, E., EMMERICH, W., FINKELSTEIN, A., SMOLKO, D., AND ZISMAN, A. 1999. Consistency management of distributed documents using XML and related technologies. Res. Note 99–94, Dept. of Computer Science, University College London.

EMMERICH, W. 1996. GTSL—An object-oriented language for specification of syntax directed tools. In *Proceedings of the 8th International Workshop on Software Specification and Design* (1996), 26–35. IEEE Computer Society Press.

FAN, W. AND SIMEON, J. 2000. Integrity constraints for XML. In *Proceedings of the Symposium on Principles of Database Systems* (2000), 23–34.

FEATHER, M., FICKAS, S., FINKELSTEIN, A., AND VAN LANSWEERDE, A. 1997. Requirements and specification exemplars. *Automated Soft. Eng. 4*, 4.

FERNANDEZ, M., FLORESCU, D., LEVY, A., AND SUCIU, D. 1999. Verifying integrity constraints on Web sites. In *Proceedings of the 16th International Joint Conference on Articial Intelligence* (1999), 614–619.

FERNANDEZ, M., FLORESCU, D., LEVY, A., AND SUCIU, D. 2000. Declarative specification of Web sites with strudel. *VLDB J. 9*, 1, 38–55.

FINKELSTEIN, A. 2000. A foolish consistency: Technical challenges in consistency management. In *Proceedings of the 11th International Conference on Database and Expert Systems Applications* (*DEXA*). (London, Sept. 2000), 1–5. Springer Verlag.

FINKELSTEIN, A., GABBAY, D., HUNTER, H., KRAMER, J., AND NUSEIBEH, B. 1994. Inconsistency handling in multiperspective specifications. *IEEE Trans. Soft. Eng. 20*, 8, 569–578.

FINKELSTEIN, A., KRAMER, J., NUSEIBEH, B., FINKELSTEIN, L., AND GOEDICKE, M. 1992. Viewpoints: A framework for integrating multiple perspectives in system development. *Int. J. Soft. Eng. Knowl. Eng. 2*, 1, 21–58.

GURDEL, G. 2001. FpML version 1.0. http://www.fpml.org.

HABERMANN, A. N. AND NOTKIN, D. 1986. Gandalf: Software development environments. *IEEE Trans. Soft. Eng. 12*, 12, 1117–1127.

HENRICH, A. AND DÄBERITZ, D. 1996. Using a query language to state consistency constraints for repositories. In *Database and Expert Systems Applications* (1996), 59–68.

JELLIFFE, R. 2000. The Schematron assertion language 1.5. Tech. Rep. (Oct.), GeoTempo Inc.

Mozilla. 2000. Mozilla. http://www.mozilla.org.

NAGL, M., Ed. 1996. *Building Tightly Integrated Software Development Environments: The IPSEN Approach*. LNCS vol. 1170, Springer Verlag.

NUSEIBEH, B., EASTERBROOK, S., AND RUSSO, A. 2000. Leveraging inconsistency in software development. *IEEE Computer 33*, 4 (April), 24–29.

Object Management Group. 2000a. *Unified Modeling Language Specification*. Object Management Group.

Object Management Group 2000b. *XML Metadata Interchange (XMI) Specification 1.1*. Object Management Group, Framingham, MA.

REPS, T. W. AND TEITELBAUM, T. 1984. The synthesizer generator. *ACM SIGSOFT Soft. Eng. Notes 9*, 3, 42–48.

SCHÄFER, W. AND WEBER, H. 1989. European software factory plan—The ESF-profile. In *Modern Software Engineering—Foundations and Current Perspectives*, P. A. Ng and R. T. Yeh, Eds., Ch. 22, Van Nostrand Reinhold, 613–637.

TAYLOR, R. N., SELBY, R. W., YOUNG, M., BELZ, F. C., CLARCE, L. A., WILEDEN, J. C., OSTERWEIL, L., AND WOLF, A. L. 1988. Foundations of the Arcadia Environment Architecture. *ACM SIGSOFT Soft. Eng. Notes 13*, 5, 1–13.

WADLER, P. 1999. A formal semantics of patterns in XSLT. Markup Technologies.

WILKINSON, R. AND SMEATON, A. 1999. Automatic link generation. *ACM Comput. Surv. 31*, 4es (Dec.). Article 27.

Received January 2001; revised October 2001; accepted October 2001