xlinkit: links that make sense

Christian Nentwich, Wolfgang Emmerich and Anthony Finkelstein

Department of Computer Science
University College London
Gower Street, London WC1E 6BT
{c.nentwich,w.emmerich,a.finkelstein}@cs.ucl.ac.uk

ABSTRACT
We describe xlinkit, a rule-based hyperlink generation and consistency checking service. xlinkit takes sets of XML documents and delivers XLink linkbases that contain semantically relevant links between elements of those documents. We present a case study illustrating the application of xlinkit and give an account of our approach to rule definition and checking.

KEYWORDS: Link generation, document management, consistency checking, semi-structured data

INTRODUCTION
For an effective hypertext it is necessary for related information to be linked so as to to support navigation. Since manual link construction is difficult and error prone, it is desirable to automate this process. xlinkit provides automated hyperlink generation in the face of large data volume and helps to ensure the global consistency (freedom from contradiction) of distributed documents.

xlinkit exploits the semantic richness of distributed XML documents by enabling the specification of rules that relate document types. These rules are evaluated to provide meaningful hyperlinks, which are stored in a *linkbase*.

The key contribution of this work is the development of a lightweight hyperlink generation and consistency checking service. The building blocks of this service are a first-order logic rule language for relating document elements, an interpretation of that language in terms of hyperlinks and a distributed document management scheme. We present an efficient implementation of this service, openly available at `http://www.xlinkit.com`.

BACKGROUND
Our background is in software engineering. The main concepts in this paper have their origin in the problem of managing the consistency of distributed, heterogenous software and system models. We use open and standardised web and hypertext technologies, which permits the application of ideas in this area to a much broader class of problems. This section presents the rationale behind our choice of the tools and standards on which the work is based.

The eXtensible Markup Language (XML) [1] has gained acceptance in the business world as a heterogeneous data exchange and web publication format. As a consequence, a lot of business data is now encoded in a semi-structured format, which makes it easy to give meaning to document elements.

Along with XML comes a set of powerful languages for linking and element extraction. XPath [2] allows the formulation of queries on XML documents which yield a set of elements when executed. This technology is a crucial part of our approach as we use it to define the sets of elements that are to be linked.

XLink [4] provides linking functionality for XML documents. The expressive power of XLink goes beyond what has been available in HTML. In particular, links can have more than two endpoints and any element in an XML file can act as a link. Furthermore, XLinks do not have to be stored in the documents that are being linked. This allows us to assemble a set of links in a linkbase, that is out-of-line from the documents that are being linked. When combined with XPointer [3] or XPath, XLink is able to link not only at the document level but also between elements contained within the documents.

EXAMPLE
We have conducted a number of case studies in order to evaluate the scalability and applicability of our approach. A simple case study will now be introduced as a running example. The Department of Computer Science at University College London makes its curricula available to staff and students over an intranet. It maintains a curriculum for the degree programmes offered by the department as well as a syllabus for each course. XML was chosen as the format to hold the

information because it can easily be processed on a variety of platforms and can be transformed into a number formats, including HTML.

Figure 1 shows a sample syllabus file. It contains a code identifying the course, several elements presenting administrative information and a description of the content of the course (omitted from the figure).

```
<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>
```

Figure 1: Sample shortened syllabus file in XML

Figure 2 shows a fragment of the department's curriculum. The document contains one entry per degree programme offered by the department. Each degree programme is divided into years and for each year, the mandatory and optional courses are listed.

In total, the curriculum and syllabus data comprise 50 files. The curriculum and syllabi need to be displayed in HTML format for students to be able to access the information in a web browser. In addition, hyperlinks have to be established from the courses listed in the curriculum to the correct syllabus information. Due to the amount of data involved and the frequent changes to the curriculum, manual linking is not a feasible option.

The content of the files is overlapping. For example, the meaning of "course code" in the curriculum file is the same as the meaning in the syllabi. This semantic equivalence, if made explicit, can be used to establish links between entries in the curriculum and definitions in syllabus file. Our approach to defining these relationships is to write rules. For the given example, we can write a rule of the form *"for all courses in the curriculum, there must be some syllabus with the same code"*.

Conversely, the presence of a more or less rigid structure in the files leads to the creation of implicit dependencies. For

```
<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>
```

Figure 2: Curriculum fragment

example, if a course code is mentioned in the curriculum, there should now be a syllabus defining that course, otherwise the curriculum is flawed. If a prerequisite course code is mentioned in a syllabus, then that code must not lead to the creation of a pre-requisite "cycle", which would make it impossible for a student to fulfill the entry requirements for that course.

The duplication and spreading of information across several distributed files, a necessary consequence of the loosely federated approach the department takes to the documentation of its programmes, has introduced the possibility of inconsistency. Alongside the ability to link related document elements, we provide a diagnostic that draws the user's attention to elements that introduce inconsistency.

RULES

We now explain how we define rules that relate elements in distributed documents.

In order to relate elements, we first have to define which elements we are going to work with. The XPath language allows us to retrieve elements from XML documents and group them into sets of DOM nodes. For example, the XPath query `/syllabus/identity/title/text()` returns the set of all syllabus titles.

We will use a formal notation due to Wadler [17] to simplify the presentation of XPath queries. The function $\mathcal{S}[\![p]\!]_x$ selects the pattern p from the document with context node x. The pattern is always relative to the context node. If a specific node has to be retrieved with an absolute index, the root node $/$ can be used as the context node.

Going back to the rule *"for all courses in the curriculum, there must be some syllabus with the same code"*, we first have to establish the sets we are going to work with. The set of all courses in the curriculum can be obtained using the query $\mathcal{S}[\![/Curricula/Curriculum/Year/Course]\!]_/$. The set of all course definitions in syllabus files is obtained by executing the query $\mathcal{S}[\![/syllabus/identity]\!]_/$ for each syllabus file.

The rule given in natural language above can be translated trivially into our language. For convenience, let C be the set of all courses in the curriculum and I be the set of all course identities from the syllabi. We can write

$$\forall c \in C(\exists i \in I(\mathcal{S}[\![Code]\!]_c = \mathcal{S}[\![code]\!]_i))$$

Figure 3 gives the abstract syntax for the entire language. The language in its current form is quite simple. It basically corresponds to the constructs allowed in general first-order logic, with the following restrictions: the sets we are working on are sets of DOM nodes and are always finite, the only predicates allowed are equality and inequality and no functions are allowed. Under these conditions, a truth table for a particular rule together with its sets of nodes can be established in finite time - thus our evaluation function will always terminate.

$$\begin{aligned}
rule &::= \forall \mathbf{var} \in \mathbf{xpath}(formula) \\
formula &::= \forall \mathbf{var} \in \mathbf{xpath}(formula) \mid \\
&\quad \exists \mathbf{var} \in \mathbf{xpath}(formula) \mid \\
&\quad formula\ \mathbf{and}\ formula \mid \\
&\quad formula\ \mathbf{or}\ formula \mid \\
&\quad formula\ \mathbf{implies}\ formula \mid \\
&\quad \mathbf{not} formula \mid \\
&\quad \mathbf{xpath} = \mathbf{xpath} \mid \\
&\quad \mathbf{xpath} \neq \mathbf{xpath}
\end{aligned}$$

Figure 3: Rule language syntax

In practice, we use XML as a concrete syntax for our language, cheXML. Encoding the rules in XML allows users to use the same tool environment for editing that they would be using for processing their documents. It will also be possible to write meta-rules that check properties of other rules. Figure 4 shows our example rule in its XML encoding.

Because the rules are in XML, they can also be rendered into an HTML representation using an XSLT stylesheet. Figure 5 shows some rules rendered in a browser – the prefix format of the formulae in XML has been transformed into the usual first order logic infix.

Given this language, it is possible to write rules for simple relationships like the ones in our case study in a relatively

```
<consistencyrule id="r1">
  <description>
      Each course in the curriculum
      must have a syllabus
  </description>

  <forall var="c" in="/*/Curriculum/Year/Course">

      <exists var="s" in="/syllabus/identity">
          <equal op1="$c/Code/text()"
                 op2="$s/code/text()"/>
      </exists>

  </forall>
</consistencyrule>
```

Figure 4: Example rule in XML

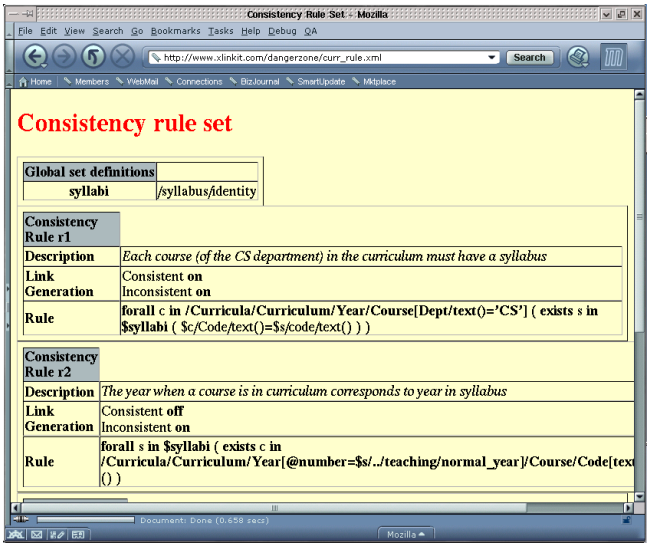

Figure 5: Example rule in HTML

intuitive manner. At the same time, the expressiveness of the language allows for relationships such as the ones required in complex software engineering notations such as the UML [15].

LINK GENERATION

The crucial contribution of our approach is the ability to generate meaningful links from the rules that relate the documents. This section presents our semantics for evaluating the rules.

Looking back to the example rule, we can define an intuitive goal for link generation: For each course in the curriculum, if we find a course in some syllabus that matches, we want to generate a link that ties the two together. If, for some entry in the curriculum, we cannot find a matching syllabus, then there is nothing to link to. In that case we would probably like a link pointing to the offending entry and some sort of reference to the rule which has been violated.

We will distinguish between two types of links: *consistent* and *inconsistent* links. We establish a consistent link be-

tween elements if the relationship set out in the rule holds and an inconsistent link if the combination of elements being linked contradicts a rule. To simplify the discussion, we introduce a formal notation for links. A link has a consistency status as indicated above and one or more *locators*, which represent the endpoints of the link. Formally, we write $consistent[e_1, e_2, .., e_n]$ or $inconsistent[e_1, e_2, .., e_n]$, where the locators e_i refer to some element being linked. In the actual implementation, these links will become extended XLinks and the locators will be XLink locators.

The syntax in Figure 3 shows that a rule consists of a universal quantifier which contains a formula. A formula will consist of an equality comparison or a quantifier or operator that consists of further formulae. Our goal is to obtain a set of links by evaluating a rule. We achieve this by defining a recursive strategy whereby a formula can be evaluated to yield a set of links. The locators contained in a link will be derived from the current assignment of quantifier variables in the formula when the link is being composed. A formula that recursively consists of subformulae will take the links created by the subformulae and append its own locators or links.

Curriculum set (C)	**Syllabus set (I)**
Course (Code="1B10")	identity (code="3C03")
Course (Code="3C03")	identity (code="1B10")
Course (Code="9C99")	

Table 1: Source and destination node sets

We will explain this link generation strategy using our simple example. Suppose we have selected the courses to be processed from the curriculum and the course information from the syllabi using XPath. Table 1 shows the two sets. For illustration purposes, we will use a shortcut for addressing the elements in the sets: C_n denotes the n-th element in set C. In xlinkit, this addressing is implemented using XPath.

Going through the example rule

$$\forall c \in C(\exists i \in I(\mathcal{S}[\![Code]\!]_c = \mathcal{S}[\![code]\!]_i))$$

the forall evaluation will bind c to the first entry in the set, C_1 (the course with code "1B10"). This binding is passed on to the existential quantifier evaluation, which iterates through set I and performs an equality comparison for each entry. For the course at I_2, this comparison is true, so we use I_2 as a locator and store a link of the form $consistent[I_2]$. Since the existential quantifier has found an assignment that makes its subformula true, it will itself return true. The universal quantifier obtains this information and produces a cartesian product of the element it is currently considering, C_1 and the links returned by the existential quantifier. The result is the set of links

$$\{consistent[C_1, I_2]\}$$

The same strategy has the existential quantifier returning $consistent[I_1]$ when C_2 is bound to c by the universal quantifier. The set of links is now

$$\{consistent[C_1, I_2], consistent[C_2, I_1]\}$$

Finally, when c is bound to C_3, the existential quantifier will not find an entry in set I where the equality comparison holds because the course with code $9C99$ has no syllabus definition. It will therefore return false and produce an empty set of links. The universal quantifier uses this information to store the link $inconsistent[C_3]$.

We have by this means obtained a set of consistency links that relate corresponding elements and highlight combinations of elements that make our rule fail. It should be pointed out that the generated links are not always two-dimensional: rules with n nested quantifiers can generate links of cardinality n. Note also that we have managed to generate our links transparently, by defining a suitable semantics for the evaluation of first-order logic formulae, a key contribution of our work.

```
<xlinkit:LinkBase
       docSet="DocumentSet.xml"
       ruleSet="RuleSet.xml"
       xmlns:xlinkit="http://www.xlinkit.com">

    <xlinkit:ConsistencyLink
          ruleid="curr_rule.xml#id('r1')">

        <xlinkit:State>consistent</xlinkit:State>
        <xlinkit:Locator
            xlink:href="curricula.xml#
                        /Curriculum[1]
                        /Year[1]/Course[1]"/>
        <xlinkit:Locator
            xlink:href="computer_architecture.xml#
                        /identity[1]"/>

    </xlinkit:ConsistencyLink>
    ...
</xlinkit:LinkBase>
```

Figure 6: XML Linkbase

Our links are presented as linkbases in an XML encoding using the XLink standard. Figure 6 shows a link in a sample – slightly simplified – linkbase that was generated using our link generation engine. The linkbase contains all diagnostic information required to locate consistent or inconsistent elements.

DOCUMENT MANAGEMENT

Before we can evaluate our rules and generate links, we have to inform the system which documents are to be checked against which rules. It is thus necessary to provide a way of organising the documents.

Our link generation engine always takes a set of documents and a set of rules as input. Documents can be arranged in a hierarchy of *document sets*. Figure 7 shows what a document

set for our curriculum would look like. The `Document` element is used to instruct the system to include an XML document into the set and the `Set` element includes a further document set, which can itself include further sets and files. This hierarchy of sets and files is flattened at runtime and all documents are parsed and loaded as a DOM tree.

```
<DocumentSet name="SyllabusDoc">
    <Description>
        Course syllabi and curriculum
    </Description>

    <Document href="curriculum.xml"/>

    <Set href="Syllabi.xml"/>
</DocumentSet>
```

Figure 7: Curriculum document set

Our document retrieval mechanism is not tied to any specific data storage mechanism. We abstract from low-level storage details by defining *fetchers*. It is a fetcher's responsibility to liaise with a data source and return a DOM tree. By default, the *FileFetcher* fulfills this task by parsing an XML file. We have also implemented a simple yet powerful database retrieval mechanism using JDBC. Figure 8 shows how the curriculum would be retrieved if it were stored in a relational database. By abstracting from the underlying mechanisms we enable the generation of links between documents that are stored in heterogeneous and distributed data sources.

```
<DocumentSet name="SyllabusDoc">
    <Description>
        Course syllabi and curriculum
    </Description>

    <DocFile fetcher="JDBCFetcher"
             href="jdbc:mysql://www.xlinkit.com
                   /db#select * from curriculum"/>

    <Set href="Syllabi.xml"/>
</DocumentSet>
```

Figure 8: Database retrieval

The rules are treated similarly and arranged in a *rule set*. Figure 9 shows the set for the curriculum. The `RuleFile` element is used to add rules from a file – further precision can be achieved by using the `xpath` attribute to pick out specific rules. In this example, all rules are selected. Although the figure does not show it, a `Set` element can be used in a rule set to include further rule sets, similar to the document set mechanism.

PERFORMANCE

The curriculum was stored in a single XML file and 49 syllabus definitions were stored in one file each. In addition, two "study-plans", files in which students can define the courses they wish to attend, were included. In total, 9 rules were checked.

The link generation was performed on a single processor 750

```
<RulesSet name="CurrRules">
    <Description>
        Rules for the CS curriculum
    </Description>

    <RuleFile href="curr_rule.xml"
        xpath="//consistencyrule"/>
</RulesSet>
```

Figure 9: Curriculum rule set

Mhz Intel machine with 128 megabytes of memory. The total time for all rules was 5.5 seconds and 40 megabytes of RAM were consumed. Figure 10 shows the time taken for each rule in milliseconds. Rules 1 and 5 are somewhat slower because they produce more than 90% of the links. Rule 6 is formulated using three nested quantifiers and is also noticeably slower.

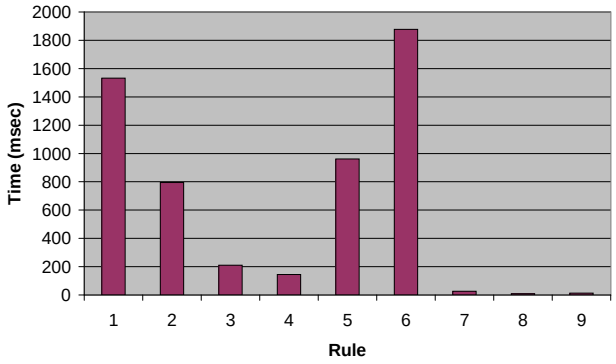

Figure 10: Case study timings

The rules yielded 410 consistent links and 11 inconsistent links in total.

DELIVERY

Linkbases generated by our tool contain complete diagnostic link information. They are however of limited direct use to a human user. This section presents various strategies of making the linking information useable.

One strategy with any out-of-line set of links is to reinsert the links into the documents they are linking – *folding* the links. Using this approach together with our tool, it is possible to produce complete web pages without manually authoring any links. Our curriculum case study used a commercial XLink processor called X2X [7] by Empolis to fold the generated links into the curriculum and syllabus definitions. It was then possible to apply an XSLT stylesheet to obtain a complete and fully-linked set of web pages for access by staff and students. Figure 11 shows the final version of the website after link insertion and transformation.

If a linkbase is going to be used for diagnostic purposes, such as finding and rectifying inconsistencies in a set of distributed documents, a more dynamic representation is desirable. We provide a servlet that uses XSLT to transform the

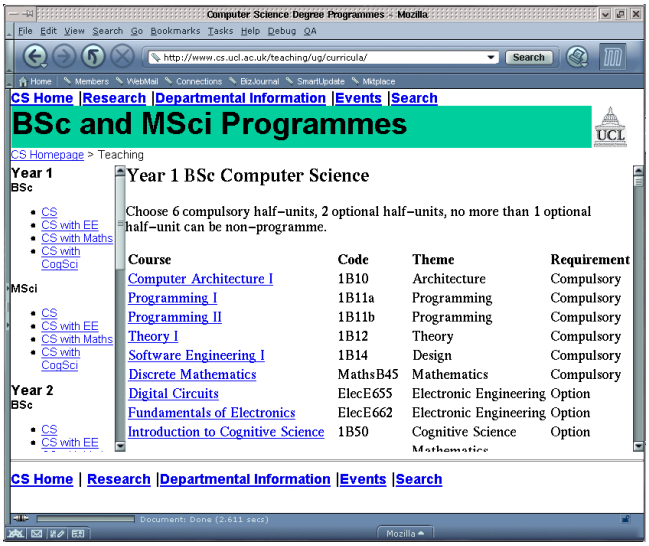

Figure 11: Finished curriculum screenshot

linkbase into HTML and lets the user traverse the links. Figure 12 shows a screenshot of the servlet in action. When the user clicks on one of the consistency links, the participating documents are loaded by another servlet and juxtaposed in the two bottom frames. The related elements inside the XML files are selected using an XPath processor, framed and centered to simplify comparison.

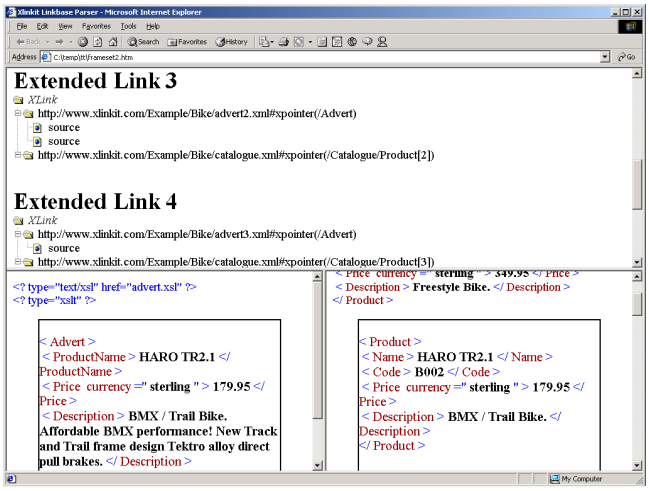

Figure 12: Dynamic linkbase servlet

APPLICATIONS

xlinkit has a wide variety of potential applications in enterprise data integration and content management. In particular it can be used to generate lightweight portals across heterogeneous information resources where these portals reflect 'business rules'. Content aggregation services which bring together XML feeds can use xlinkit to provide value-added linking wholly automatically.

Business areas in which there are obvious uses of xlinkit include those in which regulatory compliance requires linking of documents and information resources for audit purposes, such as pharmaceuticals and finance. In many financial trade settings there is a need to integrate information in heterogeneous back office systems and support reconciliation of trade information (conveniently in XML form) across diverse systems and between parties. Other areas characterised by large volumes of distributed and potentially inconsistent data where the xlinkit technology can be usefully applied include engineering product data management, customer relationship management and network policy management.

RELATED WORK

Our approach has its basis in work on management of consistency in software development environments. This spans the early work on programming environments like the Cornell Synthesizer Generator [16], which are based on abstract syntax trees, as well as later graph-based software development environments like IPSEN [12], and GOODSTEP [6].

We have worked extensively on the management of multiple views in software development [10]. This work has spawned the research into inconsistency handling [9] and has shaped our tolerant view in which consistency is diagnosed but not necessarily enforced [8, 14].

There has been much work in the hypertext community on automatic link generation. On the whole, this work has assumed relatively poorly structured text, making it hard to exploit semantic knowledge about the content of documents. There is a large body of research into similarity metrics and information-retrieval mechanisms for link generation. This work is surveyed in [18]. Our work exploits the structure provided by XML to produce links that 'make sense', that is are guaranteed to be meaningful.

Schematron [11] is an XML validator which uses XPath and XSLT to check documents against user-specified rules. Though the assertion language of Schematron bears some similarity to our rule language, it does not posess the expressiveness. Most significantly, the Schematron does not provide link generation.

Our rule language has gone through several revisions [5, 13] during which we have simplified it and improved its expressiveness. The language set out in this paper provides the expressiveness of full first-order logic and a transparent semantics for link generation.

FUTURE WORK

This section discusses the major unresolved issues in our current work and our strategy for tackling them.

Our prototype evaluates all rules against a document set, regardless of whether the documents have been changed or been checked before. It would be desirable to instead perform *incremental checking*, that is to only check those rules which are affected by a change in a participating documents.

We have devised an algorithm and will continue to investigate.

We have yet to deal with the problem of having to hold all participating documents in memory in order to execute a check. This strategy clearly does not scale if a large number of documents is involved. We will attempt to find a memory management scheme that can optimise the number of documents that have to be present in memory. Some support for this may be available in DOM-aware database management systems.

It will be necessary to propose strategies for dealing with consistency information. We hold a tolerant approach to inconsistency, however we do not suggest that it should be ignored. Consequently, we will look into possible approaches to resolution of inconsistencies using our linkbases.

CONCLUSION

In this paper we have shown how to relate the content of distributed XML files by writing rules that exploit the semantics exposed by the structure of the documents. Using the XPath language and a simple form of first order logic it is possible to write rules that are readable, yet powerfully expressive. We have shown how to evaluate these rules appropriately to obtain a set of links that provide diagnostic information and can be used for the automatic construction of web content.

xlinkit is protected by PCT 9914232.5. Licenses for xlinkit can be obtained free of charge for research and evaluation purposes. A free Internet service is available at `http://www.xlinkit.com`.

ACKNOWLEDGEMENTS

We would like to thank our student Zeeshawn Durrani for his work on the linkbase visualisation servlet and Wendy Hall and the Intelligence, Agents, Multimedia Research Group at Southampton University for their guidance.

REFERENCES

1. T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language. Recommendation http://www.w3.org/TR/2000/REC-xml-20001006, World Wide Web Consortium, October 2000.

2. J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. Recommendation http://www.w3.org/TR/1999/REC-xpath-19991116, World Wide Web Consortium, November 1999.

3. R. Daniel, S. DeRose, and E. Maler. XML Pointer Language (XPointer) Version 1.0. Candidate Recommendation http://www.w3.org/TR/2000/CR-xptr-20000607, World Wide Web Consortium, June 2000.

4. S. DeRose, E. Maler, D. Orchard, and B. Trafford. XML Linking Language (XLink) Version 1.0. Candidate Recommendation http://www.w3.org/TR/2000/CR-xlink-20000703, World Wide Web Consortium, July 2000.

5. E. Ellmer, W. Emmerich, A. Finkelstein, D. Smolko, and A. Zisman. Consistency Management of Distributed Documents using XML and Related Technologies. Research Note 99-94, University College London, Dept. of Computer Science, 1999. Submitted for Publication.

6. W. Emmerich. GTSL — An Object-Oriented Language for Specification of Syntax Directed Tools. In *Proc. of the 8th Int. Workshop on Software Specification and Design*, pages 26–35. IEEE Computer Society Press, 1996.

7. Empolis. X2X. http://www.empolis.co.uk, 2000.

8. A. Finkelstein. A Foolish Consistency: Technical Challenges in Consistency Management. In *Proceedings of the 11th International Conference on Database and Expert Systems Applications (DEXA)*, pages 1–5, London, UK, September 2000. Springer.

9. A. Finkelstein, D. Gabbay, H. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency Handling in Multi-Perspective Specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, 1994.

10. A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: a framework for integrating multiple perspectives in system development. *Int. Journal of Software Engineering and Knowledge Engineering*, 2(1):21–58, 1992.

11. R. Jelliffe. The Schematron Assertion Language 1.5. Technical report, GeoTempo Inc., October 2000.

12. M. Nagl. Building Tighly Integrated Software Development Environments: The IPSEN Approach. *Lecture Notes in Computer Science*, 1170, 1996.

13. C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: a Consistency Checking and Smart Link Generation Service. Research Note RN/00/66, University College London, Dept. of Computer Science, 2000. Submitted for Publication.

14. B. Nuseibeh, S. Easterbrook, and A. Russo. Leveraging Inconsistency in Software Development. *IEEE Computer*, 33(4):24–29, April 2000.

15. Object Management Group. *Unified Modeling Language Specification*, March 2000.

16. T. W. Reps and T. Teitelbaum. The Synthesizer Generator. *ACM SIGSOFT Software Engineering Notes*, 9(3):42–48, 1984. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, PA, USA.

17. P. Wadler. A formal semantics of patterns in XSLT. Markup Technologies, December 1999.

18. R. Wilkinson and A.F. Smeaton. Automatic Link Generation. *ACM Computing Surveys*, 31(4es), December 1999. Article No. 27.