Checking Distributed Software Engineering Content

Christian Nentwich, Wolfgang Emmerich and Anthony Finkelstein
Department of Computer Science
University College London
Gower Street, London WC1E 6BT, UK
{c.nentwich,w.emmerich,a.finkelstein}@cs.ucl.ac.uk

ABSTRACT

We describe the foundations of xlinkit, a technology for checking the consistency and linking the elements of distributed, heterogenous XML documents, and its application to software engineering. We do so by providing a formal semantics for a rule language which relates document elements and by showing how we used the language to express the constraints of the UML Core package. We outline how we implemented xlinkit as a lightweight web service using open standard technology and present the results of an evaluation against several sizeable UML models provided by industrial partners.

Keywords

Consistency management, UML, document management, automatic link generation

1. INTRODUCTION

This paper presents a formal basis for checking the consistency and linking the elements of distributed, heterogeneous data in XML. It shows how this can be applied to software engineering content, most notably the UML. It outlines a service, xlinkit, that has been developed to support this approach and shows evaluation results which demonstrate its practicability. xlinkit leverages open standards such as XML, XLink and XPath in order to bridge heterogeneity problems and allow Internet scale distribution of development activities.

The key contributions of this paper are an intuitive first-order language for expressing consistency relationships of elements in XML documents, the formal specification of a transparent semantics that enables links to be generated between elements from formulae in the language, and a lightweight architecture for delivering this as a service via the Web.

We show in particular how we made use of our language to express the constraints of the UML Foundation/Core package and to write rules that relate UML documents to Z specifications. We present an evaluation of the performance of our implementation against several industrial size models. xlinkit may be used at `http://www.xlinkit.com`.

2. BACKGROUND

The eXtensible Markup Language (XML) [2] has gained acceptance in the business and software development world as an open standard and as a mechanism for bridging data heterogeneity problems.

XML has simplified the creation of domain-specific markup languages. Software engineering is an obvious application area where many languages have been developed, for example the next generation of the CASE Data Interchange Format (CDIF)[8] will be based on XML, ADML [20] defines an XML DTD for software architecture interchange based on ACME [12], Enterprise JavaBeans [15] specify their deployment descriptors in XML and Ant [1] provides a dependency checking and make system based on XML. Most importantly for the examples in this paper, the XML Metadata Interchange (XMI) [19] standard supports the storage of MOF-compliant models in XML format, in particular it includes a DTD for the UML as an example application.

XML is accompanied by a set of powerful technologies, which we will briefly review. XPath [3] supports the selection of sets of elements from XML documents by standardising a language for paths in trees. XLink [4] is the linking language for XML. An XLink consists of a set of locators which identify the resources connected by the link. XLink greatly improves the linking facilities available for hypertext authors over those available in HTML anchors: it can link more than two documents, links do not have to be inside the documents being linked (*out-of-line* links) and link traversal behaviour can be specified. When combined with a language like XPath, XLink can be used at a fine-grain level to relate elements rather than simply documents.

3. MOTIVATION

The Unified Modeling Language (UML) [18] is widely used in software development. One of the reasons for its popularity is that it combines a graphical notation with a semi-formal language. This language can be used together with the notation to provide multiple views of a system.

Like every other formal language, the UML has a syntax,

and models expressed in it have to obey certain static semantic constraints. If any of these constraints are violated, by definition, inconsistency is introduced into the model. Unfortunately, as a result of distribution of development teams and process organisation, it is almost impossible for all constraints to be satisfied at any one time. For example, one constraint prescribes that if a class type is instantiated - e.g. in a sequence diagram - then that class type must have been defined - e.g. in a class diagram. If a developer decides to model some interactions first in order to get a better idea of how the final system could work, he may deliberately decide to leave the model in an inconsistent state.

Most modeling tools that implement the UML offer some sort of consistency checking mechanism. More often than not, these tools do not enforce the complete set of constraints in the UML standard [18]. Furthermore, UML models can be split using XMI as an interchange format and maintained in a distributed fashion, further complicating the issue.

Software development is often undertaken by distributed teams who use a variety of tools and specification languages. Interchange formats like XMI can help to alleviate problems of integration between tools that build on the same language, but offer little support otherwise. As an example, in a mission-critical system it may be necessary to introduce Z schemas corresponding to the elements in the UML model. If the Z specification lacks a schema for a certain UML element, the combined model is inconsistent with respect to this regime.

In the remainder of this paper we assume that distributed developers will be able to store their content in XML. We believe this assumption is reasonable given the growing number of XML specifications and the current status of most widely used industrial tools. We will also assume that the documents are somehow accessible over the Internet, be it in a database repository or via a protocol like HTTP.

The following sections present our language for specifying consistency rules between heterogeneous specifications, a semantics for evaluating formulae in this language, and our deployment model using XML.

4. RULE DEFINITION

We now define our rule language, which is used to assert consistency relationships between document elements.

It may be beneficial, for paedagogic reasons, to start with an example of the kind of relationship we are trying to express. Constraint 1 for associations in the UML Foundation/Core package (given in Appendix A), *"The AssociationEnds must have a unique name within the association"* can be expressed abstractly as $\forall a(\forall x \in a(\forall y \in a(name(x) = name(y) \rightarrow x = y)))$, where a will be assigned to the associations and x and y refer to the association ends inside the association. This rule can now be evaluated against a set of documents in order to establish their consistency status with respect to the rule.

The first step in defining the language is to decide on a mechanism for obtaining the elements to work with. We use the XPath language to build up sets of elements. In the following, we use a notation for XPath queries due to Wadler [24]: $\mathcal{S}[\![p]\!]_x$ selects all nodes matching pattern p with x as the context node - the context node becomes the relative root for the selection. For example, $\mathcal{S}[\![schemadef]\!]_{/z}$ selects all `schemadef` elements underneath the `z` element, which is also the root of the document.

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

Figure 1: Rule language abstract syntax

Having selected the elements we are going to relate, we can formally specify constraints between them. Figure 1 shows the abstract syntax for our language - a restricted form of first order logic that uses XPath to select sets.

As a simplified example, we can express constraint 2 for classifiers (see Appendix A) in the UML standard, *"No Attributes may have the same name within a Classifier"*. Let C be the set of classifiers. As a simplification, we assume that all classifiers only have attributes as subelements. We can write $\forall c \in C(\forall a \in c(\forall b \in c(\mathcal{S}[\![name]\!]_a = \mathcal{S}[\![name]\!]_b \rightarrow same(a, b))))$, i.e. if two attributes have the same name as a subelement it follows that they are the same. This rule represents a triangular constraint that has to hold between a classifier and each pair of attributes contained within the classifier.

In our implementation we provide an XML representation as a concrete syntax for our language. This syntax is presented in Section 6.

5. RULE CHECKING

A particularly important contribution of our work is the generation of hyperlinks from rules as a meaningful diagnostic for the consistency status of a set of documents. We will explain this link generation strategy by presenting the formal semantics of our language, interspersed with some examples. This semantics is completely transparent to the user writing the rules, who only cares about the quality of the generated links and the straightforward syntax.

A formula in our rule language expresses a desirable or undesirable combination of elements types contained within the document set. When such a formula is applied to actual documents, elements will be found that either conform to it or violate it. Our strategy for highlighting the consistency status of a set of documents is to link together elements that

satisfy rules with a *consistent link* and elements that violate rules with an *inconsistent link*.

A link consists of a set of *locators*. Each locator identifies the element it is pointing to. We now introduce some simplified formal notation for link representation: Let Σ be our alphabet and $S = \Sigma^\star$ be the set of strings over Σ. Since a path expression is just a string, and a locator essentially consists of a path expression, the set of strings is also the set of locators. We define the set of sets of locators as $Locators = \wp(S)$. The set of states a link can take is defined as $C = \{Consistent, Inconsistent\}$ and finally the set of consistency links is $L = C \times Locators$.

Before defining an evaluation strategy, we also need to introduce some auxiliary functions, shown in Figure 2: *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. To preserve space, we introduce the infix operator $\times$ which 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* deals with variable bindings: a binding $B = S \times S$ maps a variable name to a string uniquely identifying a node. *bind* can be used to introduce new variable bindings into a set of bindings.

$$
\begin{aligned}
first(x, y) &= x \\
second(x, y) &= y \\
\\
flip &: L \to L \\
flip((Consistent, y)) &= (Inconsistent, y) \\
flip((Inconsistent, y)) &= (Consistent, y) \\
\\
linkcartesian &: L \to L \to L \\
linkcartesian(x, y) &= (first(x), \\
&\quad second(x) \cup second(y)) \\
\\
\times &: L \to \wp(L) \to \wp(L) \\
x \times Y &= \{linkcartesian(x, y) \mid y \in Y\} \\
\\
bind &: B \to \wp(B) \to \wp(B) \\
bind(b, B) &= \{b\} \cup B
\end{aligned}
$$

Figure 2: Auxiliary functions

We will first define our evaluation function for the *rule* non-terminal in Figure 1 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 3. We do not define a truth assignment for the top level *rule* non-terminal since we are not really interested in the overall truth of the formula - we are interested in link generation.

Figure 4 shows the complete link generation semantics for our language. The function $\mathcal{R} : rule \to \wp(L)$ takes a rule and returns a set of consistency links. Since a rule consists of a universal quantifier, the function will build a set of nodes using a path expression, assign the node 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.

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 can demonstrate the behaviour of the quantifiers using an example. Suppose we have a formula of the form $\forall x \in X(\exists y \in Y(x = y))$. Suppose also that the sets X and Y consist of the elements shown in Table 1. We use the notation X_i to address the ith element in set X. The rule evaluation will bind $'a'$ to x and call the existential quantifier's evaluation function. Stepping through the destination set, the equality comparison returns false for the first entry, so the entry is ignored. On the second entry, it returns true. The existential quantifier generates a new link of the form $(Consistent, \{Y_2\})$. 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 thus prepends its current node X_1 to all links returned by the subformula. The set of consistency links is now $\{(Consistent, \{X_1, Y_2\})$.

X	Y
'a'	'c'
'b'	'a'
'c'	'f'

Table 1: Sample sets for rule evaluation

For node X_2, the existential quantifier will not find any assignment that makes its subformula true. 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 - $\{(Inconsistent, \{X_2\})\}$. 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:

$$
\begin{array}{rcl}
\mathcal{F} & : & formula \rightarrow boolean \\
\mathcal{F}[\![\forall \mathbf{var} \in \mathbf{xpath}(formula)]\!]_\beta & = & \mathcal{F}[\![formula]\!]_{bind((\mathbf{var},x_1),\beta)} \wedge \cdots \wedge \mathcal{F}[\![formula]\!]_{bind((\mathbf{var},x_n),\beta)} \\
 & & \mid x_i \in \mathcal{S}[\![\mathbf{xpath}]\!]_/ \\
\mathcal{F}[\![\exists \mathbf{var} \in \mathbf{xpath}(formula)]\!]_\beta & = & \mathcal{F}[\![formula]\!]_{bind((\mathbf{var},x_1),\beta)} \vee \cdots \vee \mathcal{F}[\![formula]\!]_{bind((\mathbf{var},x_n),\beta)} \\
 & & \mid x_i \in \mathcal{S}[\![\mathbf{xpath}]\!]_/ \\
\mathcal{F}[\![formula_1 \textbf{ and } formula_2]\!]_\beta & = & \mathcal{F}[\![formula_1]\!]_\beta \wedge \mathcal{F}[\![formula_2]\!]_\beta \\
\mathcal{F}[\![formula_1 \textbf{ or } formula_2]\!]_\beta & = & \mathcal{F}[\![formula_1]\!]_\beta \vee \mathcal{F}[\![formula_2]\!]_\beta \\
\mathcal{F}[\![formula_1 \textbf{ implies } formula_2]\!]_\beta & = & \mathcal{F}[\![formula_1]\!]_\beta \rightarrow \mathcal{F}[\![formula_2]\!]_\beta \\
\mathcal{F}[\![\textbf{not } formula]\!]_\beta & = & \neg\mathcal{F}[\![formula]\!]_\beta \\
\mathcal{F}[\![\mathbf{xpath_1} = \mathbf{xpath_2}]\!]_\beta & = & \mathcal{S}[\![\mathbf{xpath_1}]\!] = \mathcal{S}[\![\mathbf{xpath_2}]\!] \\
\mathcal{F}[\![\mathbf{xpath_1} \neq \mathbf{xpath_2}]\!]_\beta & = & \mathcal{S}[\![\mathbf{xpath_1}]\!] \neq \mathcal{S}[\![\mathbf{xpath_2}]\!] \\
\mathcal{F}[\![\textbf{same } \mathbf{var_1}\ \mathbf{var_2}]\!]_\beta & = & \mathcal{S}[\![\mathbf{var_1}]\!] = \mathcal{S}[\![\mathbf{var_2}]\!]
\end{array}
$$

Figure 3: Rule language - truth value semantics

$$
\begin{array}{l}
\{(Consistent, \{X_1, Y_2\}), \\
\ (Inconsistent, \{X_2\}), \\
\ (Consistent, \{X_3, Y_1\})\}
\end{array}
$$

Intuitively, these links make sense. X_1 and Y_2 form a desirable relationship with respect to this rule and thus have been linked using a consistent link. For X_2, we could not find a matching element and have thus created an inconsistent link. X_2 is inconsistent with the whole of the system rather than a particular element, so it stands alone.

Productions which contain only terminals, such as the definition of **equals** do not introduce new variables nor contain subformulae. Their linking semantics thus is to always return an empty set. We are left with the task of defining the behaviour of the logical connectives.

Because of space limitations we will concentrate on the **and** operator. Suppose we have a formula of the form $\forall x \in X(\exists y \in Y(x = y) \wedge \exists z \in Y(x \neq z))$. Assume that x is currently bound to X_1 and we evaluate the existential quantifiers. Assume furthermore that for this binding the first existential quantifier returns the set of links

$$\{(Consistent, \{Y_1\}), (Consistent, \{Y_2\})\}$$

and the second existential quantifier returns

$$\{(Consistent, \{Y_3\}), (Consistent, \{Y_4\})\}$$

Intuitively, we would like our links to express that the current assignment of x is consistent with respect to both subformulae of the **and** operator at the same time. We achieve this by computing a 'cartesian product' between the links produced by the subformulae: for each link in the first set of links and for each link in the second set of links, we generate a new link containing the union of locators of both links. The result returned by the **and** connective in the example is thus the set:

$$
\begin{array}{l}
\{(Consistent, \{Y_1, Y_3\}), (Consistent, \{Y_1, Y_4\}) \\
\ (Consistent, \{Y_2, Y_3\}), (Consistent, \{Y_2, Y_4\})\}
\end{array}
$$

This set will be passed back to the universal quantifier which will generate the final set of links (for the current assignment of x):

$$
\begin{array}{l}
\{(Consistent, \{X_1, Y_1, Y_3\}), (Consistent, \{X_1, Y_1, Y_4\}) \\
\ (Consistent, \{X_1, Y_2, Y_3\}), (Consistent, \{X_1, Y_2, Y_4\})\}
\end{array}
$$

The semantics for the remaining connectives were similarly derived to yield meaningful links. The whole semantics was implemented in Haskell and tested against simulation data structures to evaluate its usability.

Our overall goal is to produce a set of links that will make it easy to spot problems. We are therefore keen to obtain the minimal set of links that completely expresses the consistency status of the documents that are 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 the formula $\forall x \in X(\forall y \in X(x = y \rightarrow same(x, y)))$. Suppose we define the set X as $X = \{'a', 'a', 'b'\}$ (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. We show the values of the nodes rather than their identifiers for paedagogic purposes). 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 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 which introduce looping into the scheme only

$$
\begin{array}{lcl}
status & : & bool \rightarrow C \\
status\ \top & = & Consistent \\
status\ \bot & = & Inonsistent \\
\\
\mathcal{R} & : & rule \rightarrow \wp(L) \\
\mathcal{R}[\![\forall \mathbf{var} \in \mathbf{xpath}(formula)]\!] & = & \{(status(\mathcal{F}[\![formula]\!]_{bind((\mathbf{var},x),\{\})}), \{x\}) \times \mathcal{L}[\![formula]\!]_{bind((\mathbf{var},x),\{\})} \\
 & & \mid x \in \mathcal{S}[\![\mathbf{xpath}]\!]_{/}\} \\
\\
\mathcal{L} & : & formula \rightarrow \wp(L) \\
\mathcal{L}[\![\forall \mathbf{var} \in \mathbf{xpath}(formula)]\!]_\beta & = & \{(Inconsistent, \{x\}) \times \mathcal{L}[\![formula]\!]_{bind((\mathbf{var},x),\beta)} \\
 & & \mid x \in \mathcal{S}[\![\mathbf{xpath}]\!]_{/} \wedge \mathcal{F}[\![formula]\!]_{bind((\mathbf{var},x),\beta)} = \bot\} \\
\mathcal{L}[\![\exists \mathbf{var} \in \mathbf{xpath}(formula)]\!]_\beta & = & \{(Consistent, \{x\}) \times \mathcal{L}[\![formula]\!]_{bind((\mathbf{var},x),\beta)} \\
 & & \mid x \in \mathcal{S}[\![\mathbf{xpath}]\!]_{/} \wedge \mathcal{F}[\![formula]\!]_{bind((\mathbf{var},x),\beta)} = \top\} \\
\mathcal{L}[\![formula_1\ \mathbf{and}\ formula_2]\!]_\beta & = & \{x \times \mathcal{L}[\![formula_2]\!]_\beta \mid x \in \mathcal{L}[\![formula_1]\!]_\beta\} \\
\mathcal{L}[\![formula_1\ \mathbf{or}\ formula_2]\!]_\beta & = & \mathcal{L}[\![formula_1]\!]_\beta \cup \mathcal{L}[\![formula_2]\!]_\beta,\ if\ \mathcal{F}[\![formula_1]\!]_\beta = \mathcal{F}[\![formula_2]\!]_\beta \\
 & & \mathcal{L}[\![formula_1]\!]_\beta,\ if\ \mathcal{F}[\![formula_1]\!]_\beta = \top \\
 & & \mathcal{L}[\![formula_2]\!]_\beta,\ if\ \mathcal{F}[\![formula_2]\!]_\beta = \top \\
\mathcal{L}[\![formula_1\ \mathbf{implies}\ formula_2]\!]_\beta & = & \mathcal{L}[\![formula_2]\!]_\beta,\ if\ \mathcal{F}[\![formula_1]\!]_\beta = \top \wedge \mathcal{F}[\![formula_2]\!]_\beta = \top \\
 & & \{x \times \mathcal{L}[\![formula_2]\!]_\beta \mid x \in \mathcal{L}[\![formula_1]\!]_\beta\}, \\
 & & \quad if\ \mathcal{F}[\![formula_1]\!]_\beta = \top \wedge \mathcal{F}[\![formula_2]\!]_\beta = \bot \\
 & & \{flip(x) \mid x \in \mathcal{L}[\![formula_1]\!]_\beta\},\ otherwise \\
\mathcal{L}[\![\mathbf{not}\ formula]\!]_\beta & = & \{flip(x) \mid x \in \mathcal{L}[\![formula]\!]_\beta\} \\
\mathcal{L}[\![\mathbf{xpath_1} = \mathbf{xpath_2}]\!]_\beta & = & \{\} \\
\mathcal{L}[\![\mathbf{xpath_1} \neq \mathbf{xpath_2}]\!]_\beta & = & \{\} \\
\mathcal{L}[\![\mathbf{same\ xpath_1\ xpath_2}]\!]_\beta & = & \{\}
\end{array}
$$

Figure 4: Rule language - link generation semantics

execute their loops n times for a node set of size n. Secondly, the run-time 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 behaviour sounds problematic, it is not a problem in practice. Most rules from the UML Core, which represent a complex scenario by our standards, require at most 3 levels of nesting. In addition, empirical results show that the evaluation is fast enough for the theoretical complexity to be ineffectual.

6. XML DEPLOYMENT

Our language depends on the ability to select sets of elements for inclusion into checks and on an output format for the resulting links. In practice, we employ XPath to achieve the former and XLink for the latter. This makes it possible to provide a consistency checking environment which is based entirely on open and extensible standards.

We provide an encoding of our entire rule language in XML. Using XML for writing rules allows users to use the same tool environment for editing that they would be using for processing their documents. In addition, since elements of the rules can be selected using XPath, it will be possible to write meta-rules that check other rules should the need arise. Figure 5 shows an example rule which states that we have to create a Z schema somewhere for each class in our UML model. The syntax of the XMI paths in the figure has been abbreviated for clarity.

The rule will be parsed and checked by our implementation.

```
<globalset id="$classes"
    xpath="//Foundation.Core.Class[@xmi.id]"/>
<globalset id="$stateschemas"
    xpath="/z/schemadef[@purpose='state']"/>

<consistencyrule id="r1">
  <description>
    Every class in the UML model must have a
    state schema in a Z specification
  </description>

  <forall var="c" in="$classes">
    <exists var="s" in="$stateschemas">
      <equal op1="$c/ModelElement.name/text()"
             op2="normalize-space($s/text()[1])"/>
    </exists>
  </forall>
</consistencyrule>
```

Figure 5: Example rule in XML

The resulting set of links will be stored in an XLink *linkbase*, an example of which can be seen in Figure 6 - the XMI paths have been abbreviated again. This file now contains the complete consistency status of the participating documents with respect to the set of rules that were checked.

As they stand, the linkbases are not directly usable. We have developed several options for making the results of the consistency more accessible. The out-of-line links in the linkbases can be *folded* back into the files they are pointing to. For example, the link shown in Figure 6 would cause one link to be inserted in the UML model, pointing to the Z

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

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

    <xlinkit:State>consistent</xlinkit:State>
    <xlinkit:Locator
       xlink:href="meeting2.xml#//Class[1]"/>
    <xlinkit:Locator
       xlink:href="meetingz.xml#/z/schemadef[2]"/>

  </xlinkit:ConsistencyLink>

</xlinkit:LinkBase>
```

Figure 6: Sample result linkbase

schema, and one link to be created in the Z schema, pointing to the UML model. We could now provide a link-aware stylesheet or program to display the XMI file and the Z file and thus provide a consistency link browser.

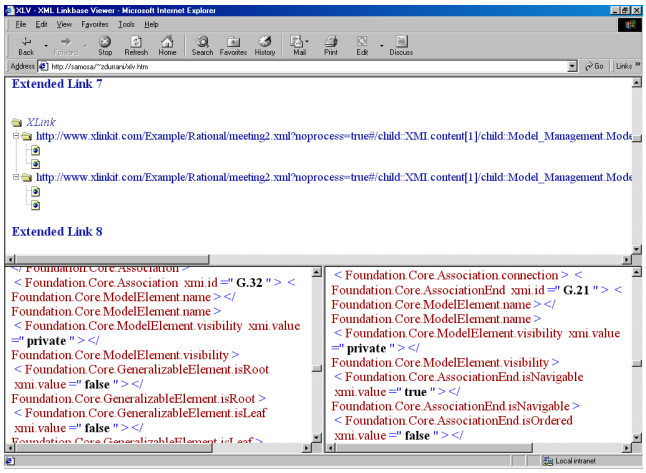

Figure 7: Interactive linkbase servlet

Alternatively, we can take the XML representation of the linkbase and make the linkbase itself more user-friendly by transforming it into HTML and making it interactive. Figure 7 shows a screenshot of a servlet we have developed for this purpose. The user can select a pair of locators, which will bring up the two documents they are pointing to in the bottom frames. The documents are rendered in HTML and the linked elements are centered.

We have developed a tool which takes a linkbase and produces a graph showing which documents and which elements are being linked. It does so by arranging the elements linked by the linkbase in a graph and outputting the graph in SVG [9], a vector graphics markup language that can be displayed using a browser plugin. Figure 8 shows a sample graph produced by the servlet.

7. DOCUMENT MANAGEMENT

We provide a general framework for managing the submission of documents and rules to our check engine. It is infeasible to check every document against every rule and it is certainly not necessary to check every document every time. The aim of this framework is to provide a structured mechanism to select which rules are supposed to be applied to which documents.

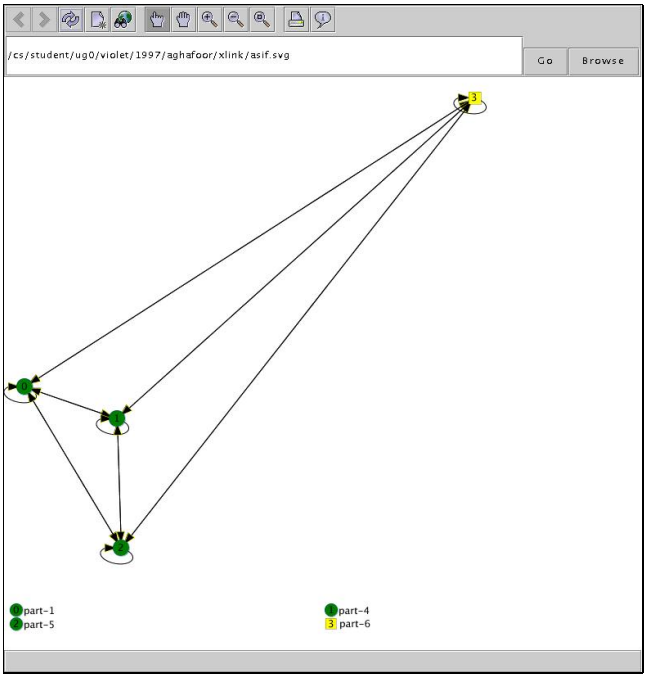

Figure 8: Linkbase graph

```
<DocumentsSet name="UMLandZ">
  <Description>
    A couple of UML models and Z schemas
  </Description>

  <Document href="catalogue.xml"/>

  <Set href="Zschemas.xml"/>
</DocumentsSet>
```

Figure 9: Sample document set

Figure 9 shows a *document set*. It includes a `Document` elements which directly adds a document into the set and a `Set` element which includes a further document set. The latter can be used to form a hierarchy of document sets, perhaps representing a hierarchy of subsystems. At run-time, the hierarchy is flattened and all documents are loaded.

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. By default, data are retrieved from XML files using the `FileFetcher` class, however user-defined classes can override this behaviour. Using this mechanism, it is possible to read in 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 used as a repository for software engineering information.

```
<RuleSet name="ZIFRule">

  <Description>XMI vs. ZIF rules</Description>

  <RuleFile href="zifrules.xml"
            xpath="//consistencyrule[1]"/>

</RuleSet>
```

Figure 10: Sample rule set

Rules are treated similarly, the are stored in rule sets. Figure 10 shows a sample rule set. There is a `RuleFile` element for including a rule file directly and an `xpath` attribute for specifying precisely which rules to extract from that file. Although it is not shown in the figure, a `Set` element can again be used to include further rule sets.

8. ARCHITECTURE

We have implemented our technology as an openly accessible web service, Figure 11 shows its basic architecture. Users assemble their files in a document set and their rules in a rule set. They then type the URL of their sets into a browser form and submit it to the web server, which invokes a servlet.

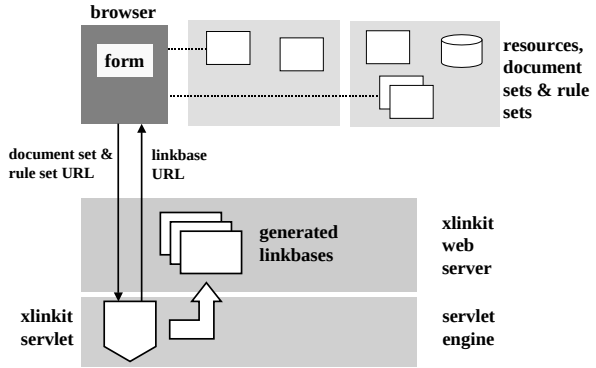

Figure 11: Web service architecture

The check engine executes the consistency rules and computes a set of links. These links are then stored locally in a linkbase with a unique identifier and a result page is generated which contains instructions for downloading or viewing the linkbase.

9. EVALUATION

We have evaluated our implementation in several case studies. This section presents the results we obtained by checking the UML Core constraints against a set of sample models: a small design model of a meeting scheduler, a medium size model taken from 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 meta-model derives from ModelElement. Our small model contains 93 elements, the medium size model has 610 elements. The number of elements contained in the industrial models ranges from 64 to 2834 elements. In terms of file size, the models range from around 100 kilobytes to 6 megabytes.

The rules given in Appendix A are rules from the UML Foundation/Core package. We have expressed all but 8 rules. Of those rules, some are enforced by the XMI DTD an do not have to be checked, some require transitive closure, which we have implemented in the past [17] but not ported to our new implementation yet, and some cannot be checked because the required information is not exported by XMI.

All results listed below were obtained by executing our check engine on a single-processor Intel machine running at 750 Mhz and using the IBM JDK 1.2 for Linux. We will first discuss the results obtained by checking the UML Core constraints in each file individually. Figure 12 shows the total time in seconds taken by each rule for all files. The 34 rules included in the check are listed in order in Appendix A. The longest time taken to check all rules against any individual file was 2.38 minutes, for the largest file in the set. The most amount of RAM consumed was 60 megabytes, again for the largest file in the set.

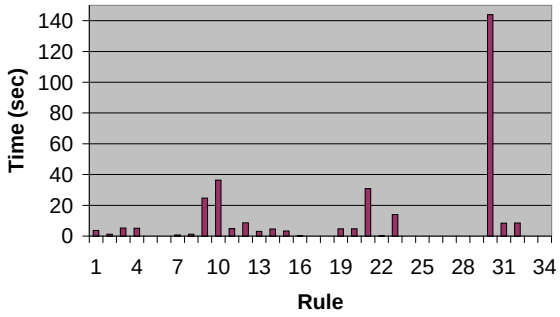

Figure 12: Rule totals for UML Core rules

We can observe several interesting properties from the figure. There is a large variance in the time taken by the different rules. This is due to two factors: Some 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. Secondly, the complexity of the XPath expressions in the rules varies greatly. Some expressions use straightforward tree paths whereas others require sophisticated functions like id lookup. This is a feature of the rather complex design 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!

We also observe that there are several rules which seem to take very little time or no time at all. What this shows is

that rules which do not apply do not consume any time - because the XPath selection fails very quickly if no match is found.

In total, over all files, 8101 inconsistent links were generated. Consistent link generation was turned off since we were only interested in finding inconsistencies. Although the number 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. The bulk of the inconsistencies was however caused by rules which were not implemented in Rational Rose. For example, the rule *"No opposite AssociationEnds may have the same name within a Classifier"*, implementing constraint 3 for `Classifier`, triggers 320 inconsistencies.

Based on these figures we can conclude that our approach scales well enough for industrial use. The maximum checking time for the largest model seems reasonable when one considers that it will not be necessary to always check all rules at the same time. In addition, most documents were much smaller than this worst case and modularisation can bring down checking time considerably.

10. RELATED WORK

There is a significant body of work on programming language environments, for example the Cornell Synthesizer Generator [21] and Gandalf [13]. These systems typically check the consistency of abstract syntax trees by evaluating the static semantic rules of the underlying programming languages. What distinguishes our approach is that documents can be distributed arbitrarily and checked without depending on a central repository.

Later work on software development environments such as ESF [22], IPSEN [16], and GOODSTEP [7] integrates tools for different languages and provide consistency checks that span across different documents. Most of the work on these environment has also assumed the existence of a centralised repository, which limits scalability and often requires commitment to a single vendor. We build on open standards such as XML and web transport protocols to obtain access to distributed resources in order to avoid these problems.

Recent work [23] on graph grammars [26] demonstrates their use in expressing UML class and sequence diagrams and specifying consistency relationships. In this approach specifications have to be translated into graph grammars. In addition, graph grammar systems typically make use of proprietary, centralised data structures, whereas we build on open standards and support Internet-scale distribution.

The viewpoints framework [11] allows the specification of multiple views of the same system. Multiple viewpoints may describe the same design fragment, leading to the possibility of inconsistency [10]. Work in this area also identifies the notion of a *consistency rule* [5] between distributed specifications. Our work complements the body of theoretical work on viewpoints with a concrete implementation on top of which such a framework could be built.

Schematron [14] enables the specification of assertions about the structure of documents and uses XSLT to evaluate the assertions. Schematron is a widely used, lightweight approach to semantic document validation. It does however not posses the expressive power of our language since, by using pure XPath expressions, it essentially builds on a boolean logic. It also does not provide support for checking inter-document relationships and does not generate links.

There is a substantial amount of work in the area of hypertext on automatic link generation [25]. Most approaches in this area focus on textual data, which by its nature exposes a minimal amount of semantics. Several information retrieval techniques such as similarity measures have been applied to link generation. Our approach makes use of the semantics exposed by the structure of XML and the rules in order to provide linkbases that are guaranteed to be meaningful.

Finally, our language and its implementation are the successors of previous prototype schemes [6, 17] and have been considerably improved with respect to expressiveness, by the introduction of first order logic with arbitrarily nested quantifiers and the removal of link generation annotation, and performance.

11. FUTURE WORK

Our current approach is static, meaning that it checks all documents against all rules in the rule set when a check is invoked. We have prototyped an incremental algorithm which performs a tree-diff operation between documents and computes a set of rules which need to be rechecked. We plan to integrate this algorithm into our web-based architecture and evaluate how it improves performance.

Our memory management strategy at the moment is to load all documents from a document set into memory as DOM trees in order to make them available for checking. This approach will not scale for large volumes of data and we will investigate options such as ordering our rules in order to minimise the amount of trees required in memory. Alternatively, it may be possible to exploit the caching mechanisms of XML databases, which can provide DOM trees directly without parsing, to circumvent the problem.

We are planning to investigate a refinement of our linking semantics. Consider the formula $a \wedge b \wedge c$. If this formula was a consistency rule and one of the three propositions was false, an inconsistent link of the form $(Inconsistent, \{a, b, c\})$ would be generated. This link shows that the three elements form an undesirable combination, but does not show which element is causing the problem. Consider the case where a and b are true and c is false. It would be preferable to have some diagnostic which identifies c as the cause of the problem. In order to successfully define such an evaluation function, we will have to find a generalisation of this simple example to arbitrary formulae in our language.

Once inconsistencies have been detected, the problem of resolution arises. We are considering process integration and automated techniques for dealing with, though not eradicating, inconsistencies. We have worked in this area before and thus will be able to build on our experience.

12. CONCLUSION

We have presented the formal foundations of xlinkit, a generic technology for consistency checking and link generation, its deployment as an open, light-weight web service using XML, and its application to software engineering.

Our evaluation has demonstrated that we were able to express the constraints of the UML Foundation/Core package as consistency rules and check those rules against sizeable UML models in reasonable time.

We encourage experimentation and use of our work, which is available as an open web service and with free licenses for research and evaluation.[1]

13. ACKNOWLEDGEMENTS

We would like to thank our students who produced the linkbase processor, the interactive servlet and the SVG visualisation tool.

14. REFERENCES

[1] Apache Software Foundation. Ant. http://jakarta.apache.org/ant, 1999.

[2] 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, Oct. 2000.

[3] 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, Nov. 1999.

[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] S. Easterbrook, A. Finkelstein, J. Kramer, and B. Nuseibeh. Coordinating Distributed ViewPoints: The Anatomy of a Consistency Check. *Int. Journal of Concurrent Engineering: Research & Applications*, 2(3):209–222, 1994.

[6] 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.

[7] 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.

[8] J. Ernst. *CDIF – XML-based Transfer Format.* Electronic Industries Association, Engineering Dept.

[9] J. F. et al. Scalable Vector Graphics (SVG) 1.0. Candidate Recommendation http://www.w3.org/TR/2000/CR-SVG-20001102, World Wide Web Consortium, Nov. 2000.

[10] 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.

[11] 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.

[12] D. Garlan and Z. Wang. ACME-Based Software Architecture Interchange. In P. Ciancarini and A. Wolf, editors, *Coordination Languages and Models, Third International Conference, COORDINATION '99*, volume 1594 of *Lecture Notes in Computer Science*, pages 340–354. Springer, Amsterdam, The Netherlands, 1999.

[13] A. N. Habermann and D. Notkin. Gandalf: Software Development Environments. *IEEE Transactions on Software Engineering*, 12(12):1117–1127, 1986.

[14] R. Jelliffe. The Schematron Assertion Language 1.5. Technical report, GeoTempo Inc., October 2000.

[15] V. Matena and M. Hapner. Enterprise JavaBeans Specification v1.1. Technical report, Sun Microsystems, DEC 1999.

[16] M. Nagl. Building Tightly Integrated Software Development Environments: The IPSEN Approach. *Lecture Notes in Computer Science*, 1170, 1996.

[17] 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.

[18] Object Management Group. *Unified Modeling Language Specification*, March 2000.

[19] Object Management Group, 492 Old Connecticut Path, Framingham, MA 01701, USA. *XML Metadata Interchange (XMI) Specification 1.1*, Nov. 2000.

[20] Open Group. Architecture Description Markup Language (ADML) Version 1. Technical Report I901, Reading, UK, 2000.

[21] 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.

[22] W. Schäfer and H. Weber. European Software Factory Plan – The ESF-Profile. In P. A. Ng and R. T. Yeh, editors, *Modern Software Engineering – Foundations and current perspectives*, chapter 22, pages 613–637. Van Nostrand Reinhold, NY, USA, 1989.

[1] xlinkit is protected by PCT 9914232.5

[23] A. Tsiolakis. Consistency Analysis of UML Class and Sequence Diagrams bas ed on Attributed Typed Graphs and their Transformation. Technical Report 2000/3, Technical University of Berlin, March 2000. ISSN 1436-9915.

[24] P. Wadler. A formal semantics of patterns in XSLT. Markup Technologies, December 1999.

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

[26] A. Zündorf. *PROgrammierte GRaphErsetzungsSysteme – Spezifikation, Implementierung und Anwendung einer integrierten Entwicklungsumgebung*. PhD thesis, University of Aachen, 1996.

APPENDIX

A. UML CORE RULES EVALUATED

A.1 Assocation

[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

A.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

A.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

A.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

A.5 Class

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

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

A.7 Component

[1] A Component may only contain other Components

A.8 Constraint

[1] A Constraint cannot be applied to itself

A.9 DataType

[1] A DataType can only contain Operations, which all must be queries
[2] A DataType cannot contain any other model elements

A.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

A.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

A.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

A.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

A.14 StructuralFeature

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

A.15 Type

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