Flexible Consistency Checking

Christian Nentwich, Wolfgang Emmerich, 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

Ernst Ellmer
Zühlke Engineering GmbH
Düsseldorfer Strasse 40a
D-65760 Eschborn-Frankfurt, Germany
ee@zuehlke.com

Abstract

The problem of managing the consistency of heterogeneous, distributed software engineering specifications is central to the development of large and complex systems. We show how this problem can be addressed using xlinkit, a lightweight framework for consistency checking that leverages standard Internet technologies. xlinkit provides flexibility, strong diagnostics, and support for distribution and document heterogeneity. We use xlinkit in a comprehensive case study that demonstrates how design, implementation and deployment information of an Enterprise JavaBeans system can be checked for consistency.

1 Introduction

The problem of managing the consistency of a set of heterogeneous, distributed software engineering specifications is central to the development of large and complex systems. Yet, in practice we find that tools cannot cope, and developers frequently adopt ad-hoc and inflexible approaches.

We show how this problem can be addressed using xlinkit, a generic, flexible and lightweight framework for checking the consistency of distributed, heterogeneous documents over the Internet. We demonstrate xlinkit's novel semantics for first order logic, which produces hyperlinks between inconsistent elements instead of boolean values, and show how it can provide powerful diagnostics across specification boundaries.

Based on a categorisation of the consistency constraints that arise during software development, we derive a set of requirements for consistency management services. We then demonstrate that xlinkit can meet these requirements in a fully worked case study.

The case study demonstrates how consistency constraints can be checked during the development of systems based on Enterprise JavaBeans. We show checks between design, implementation and deployment information.

An evaluation package for xlinkit, a web service and interactive demonstrations are available at `http://www.xlinkit.com`, and form a useful accompaniment to this paper.

The paper progresses from here as follows: we provide a brief introduction to XML technologies, the infrastructure on top of which xlinkit is built. This is followed by a detailed account of the types of consistency constraints we aim to support with this work and an introduction to xlinkit. Our case study then shows the practical application of xlinkit, followed by a brief section on the visualization of diagnostics. We conclude with a review of related work in the area of software engineering, and a conclusion that summarises what we have learnt and outlines future work.

2 XML Technologies

xlinkit, the consistency checking mechanism presented in this paper, relies on XML technologies. Is is therefore necessary to provide some technical background. This section can safely be skipped by readers familiar with the DOM, XLink and XPath.

The eXtensible Markup Language (XML) [8] is a recommendation of the World Wide Web Consortium (W3C) for specifying markup languages. Users can create their own markup elements and attributes. The relationships of the markup elements, such as which elements can be contained in others, can be controlled by providing a grammar, in the form of a Document Type Definition (DTD) or XML Schema [16]. XML files can then be checked, or *validated*, against the grammar to protect processing applications from syntax errors.

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)[15] will use XML, ADML [34] defines an XML DTD for software architecture interchange based on ACME [21], Enterprise JavaBeans [23] specify their deployment descriptors in XML and Ant [2] provides a dependency checking and make system based on XML.

In this paper, we will make reference to a number of XML languages. The most prominent amongst those is the XML Metadata Interchange (XMI) [32] language. XMI supports the storage of Meta-Object Facility (MOF) [29]-compliant models in XML format, in particular it includes a grammar for the UML as an example application. XMI DTDs are automatically generated and can be slightly verbose. Figure 1 shows an abbreviated fragment describing a class. XMI was primarily created as a standard for exchanging models between CASE tools provided by different vendors, however its encoding as an XML-based language means that it can be readily used by other applications. It is straightforward, for example, to translate XMI documents into a vector graphic markup language so as to make UML models accessible on the web [27].

```
<Foundation.Core.Class xmi.id = 'S.10007'>
   <Foundation.Core.ModelElement.name>
      BrowseReportWorkFlow
   </Foundation.Core.ModelElement.name>
   <Foundation.Core.ModelElement.visibility xmi.value = 'public'/>
   <Foundation.Core.GeneralizableElement.isRoot xmi.value = 'false'/>
   <Foundation.Core.GeneralizableElement.isLeaf xmi.value = 'true'/>
   <Foundation.Core.GeneralizableElement.isAbstract xmi.value = 'false'/>
   <Foundation.Core.Class.isActive xmi.value = 'false'/>
   ...
</Foundation.Core.Class>
```

Figure 1: Fragment of UML model in XMI

The Document Object Model (DOM) [3] is an application programming interface (API). It specifies a set of interfaces that can be used to manipulate XML content. XML content is represented in the DOM as an abstract syntax tree structure. The interfaces contain methods for manipulating nodes in the tree, traversal and event handling. The DOM provides a convenient mechanism for representing XML documents in memory and is implemented by most major XML parsers and XML databases.

XPath [9] is one of the foundational languages in the set of XML specifications. It permits the selection of elements from an XML document by specifying a tree path in the document. For example, the path `/java/class` would select all `class` elements contained in the `java` element, which is the root element. 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 [10] is an XML markup language that provides additional linking functionality for web resources. HTML links are highly constrained, notably: they are unidirectional and point-to-point; have a limited range of behaviours; 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 behaviours and to add meta-data to links. Figure 2 shows how to use XLink to turn an element into a link. The `Foundation.Core.AssociationClass` element has an `xlink:type` attribute attached to it – XLink aware processors will now recognise this element as a link. The element contains two elements of type `locator`, which will be recognised as link endpoints. This link now relates two classes contained in `model1_xmi.xml` and `model2_xmi.xml`, respectively. Most importantly, it links together two *elements* without the need of inserting any links directly into the files. It is this feature of linking elements that enables us to combine XLink and DOM trees to turn multiple DOM trees into a graph. This will be important in the software engineering applications of xlinkit demonstrated in this paper as the links will be used to highlight static semantic relationships among multiple, distributed DOM trees.

```
<Foundation.Core.AssociationClass xmi.id = '3' xlink:type="extended">
   <Foundation.Core.ModelElement.name>
      Transaction
   </Foundation.Core.ModelElement.name>
   <Foundation.Core.ModelElement.visibility xmi.value = 'public'/>
   <Foundation.Core.Class.isActive xmi.value = 'false'/>
   <Connects
       xlink:type="locator"
       xlink:href="model1_xmi.xml#//Foundation.Core.Class[@xmi.id='1']"
       xlink:label="class 1"/>
   <Connects
       xlink:type="locator"
       xlink:href="model2_xmi.xml#//Foundation.Core.Class[@xmi.id='2']"
       xlink:label="class 2"/>
   ...
</Foundation.Core.Class>
```

Figure 2: Sample XLink

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 behaviour during link traversal and traversal restriction, which cannot be covered here.

3 Consistency Management

3.1 Constraints

Throughout the development lifecycle, many types of consistency constraints appear, including *horizontal* constraints between artifacts at the same stage of the lifecycle and *vertical* constraints between stages, for example between the design and implementation. Developers are aware of such constraints, yet they are rarely explicitly expressed let alone automatically checked. Instead, ad-hoc techniques such as inspection and manual comparisons are used to maintain consistency. It is our goal to aid developers in the task of managing consistency, by providing them with a more systematic approach and tools for automation. In order to achieve this, we first categorise the types of constraints that occur in practice, and then derive a set of requirements for tools that provide support support for these types of constraints.

In our categorisation, we use the Unified Modeling Language (UML) [31] as a guiding example. The UML is widely used in software development. It combines a graphical notation with a semi-formal language, based on the OMG's Meta-Object Facility (MOF) [29]. The language and notation provide support for capturing a system from a variety of perspectives, including static perspectives such as class diagrams and dynamic ones such as sequence diagrams.

Specification languages used in software engineering typically come with a set of static constraints that valid specifications have to obey. These constrain the relationships between the elements of the underlying semantic models of the language. We call such constraints *standard constraints*, because they are generally standardised when a specification language is formally defined. For example, the UML has a syntax, and models expressed in it have to obey static semantic constraints. The UML specification expresses these constraints in the Object Constraint Language (OCL) [31]. If any of the constraints are violated, by definition inconsistency is introduced into the model.

When putting any specification language, including the UML, to practical use it very frequently transpires that the expressive capabilities provided by the language are not sufficient for a particular purpose and require extension or adaptation. The specification language may not be semantically rich enough to capture all the information necessary, for example many design languages obscure architectural information or design patterns. Conversely, the language may be overly expressive and insufficiently concise for a particular practical purpose, for example many languages lead to specifications that are too complex to be shown to stakeholders. These problems can sometimes be circumvented by restricting the language or introducing naming conventions for specification artifacts. Such restrictions or conventions will invariably give rise to additional semantics not catered for by the language. It is thus important to properly specify the extension method, and to ensure that it is applied consistently. This can be accomplished by introducing *extension constraints* that supplement the standard constraints of a language.

In the case of the UML, it was recognised that the basic metamodel of the UML was not rich enough to express the architectural properties of distributed systems that build on middleware. The UML standard foresees such problems and includes a general extension framework centered around *stereotypes* and *tagged values*, which can be used to annotate model elements in a UML model with additional information. Certain extensions are peculiar to a domain, such as the domain of systems built on middleware, and have been standardised: the UML Profile for EJB [23] lists a set of stereotypes and tagged values for designing systems built on top of Enterprise JavaBeans [23] and the UML Profile for CORBA [30] provides similar features for CORBA [33]-based systems. These extension frameworks also supply a set of constraints that specify the properties that model elements annotated with the standardised stereotypes have to obey.

Two further classes of constraints often appear at different stages of the development lifecycle. For example, if we wish to check that the implementation of a system is consistent with the design, we have to introduce additional constraints that relate them. Or if we wish to refine the classes in our UML class diagrams into Z schemas, we need constraints between the Z specification and the UML model. In both examples, constraints arise naturally as the different specification or implementation languages denote the same elements of the semantic model. We refer to these constraints as *integration constraints.* Integration constraints also arise when one specification language is restricted to interoperate cleanly with another: when using the UML to design a Java-based system, we may want to check that the design does not include any multiple inheritance between classes.

Finally, developers frequently introduce additional constraints for their specifications. For

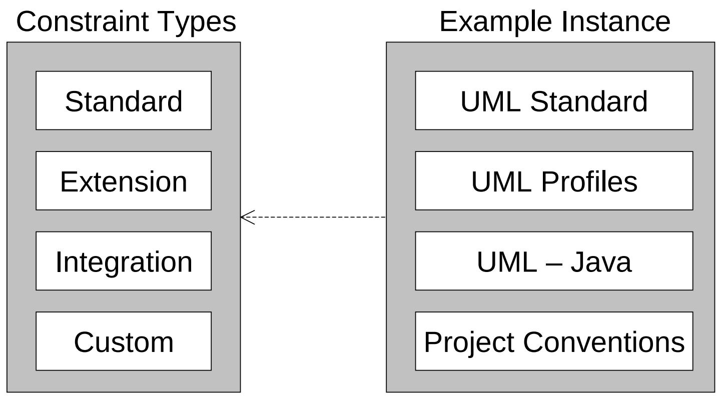


Figure 3: Constraints types with examples

example, software development organisations may introduce conventions that they want their software engineers to adhere to, the most prominent example being naming conventions. These *custom constraints* may be defined to ensure consistency between different products produced by the company, or be specific to a certain project. Such constraints are mostly informally specified and checked manually. Figure 3 gives an overview of the types of constraints that we have identified, and gives a UML-based project as an example instance.

3.2 Tool Support

Effective use of specification languages in the construction of large and complex systems requires CASE tools, which often include mechanisms for checking the constraints of the language they are built to support. A good CASE tool will let its user check on demand whether a specification is consistent with respect to the constraints or will provide interactive checking as changes are made. The majority of CASE tools currently on the market provide rather static support for consistency checking, and most support only standard constraints. Providing support for checking all four types of constraints, standard, extension, integration and custom constraints, is not trivial and gives rise to a number of problems.

The first problem is flexibility. Many CASE tools still handle only standard constraints, and in many cases they are hardcoded into the tool. It is impossible to support the three other types of constraints with such tools. These tools generally treat all constraints as equal and apply them all at once. It is not possible for the user to choose to ignore certain constraints or to delay the resolution of inconsistency.

To take an example from the UML, the constraint that no inheritance cycles must be present in a class diagram expresses a concept fundamental to object-oriented technology and must hold in all models. By contrast, the constraint that requires all attributes in classes to have valid types is often consciously violated. The UML is used for a variety of purposes, ranging from high-level domain analysis models to detailed designs. It does not make sense to apply the same set of constraints to all models – some flexibility in choosing constraints is required, and is not present in current tools. This lack of flexibility represents a barrier if all four types of constraints are to be supported.

In addition, there is still a large number of CASE tools that *enforce* their constraints, further undermining the flexibility needed in software development. This behaviour imposes a process on the developer, who has to add objects to the specification in a certain order so as to maintain consistency at all times. It would be preferable to provide a checking service that notifies developers of inconsistency, but gives them the power to decide not to address it. This tolerant approach to inconsistency becomes more important as development artifacts at different stages of the lifecycle are checked against each other, as maintaining total consistency at all times between stages is both unnecessary and time consuming.

Secondly, checking integration constraints is a rather more difficult problem than checking any of the other types of constraints. Since specifications expressed in different languages are to be checked against each other, the problem of language heterogeneity must be addressed. It is possible to provide support for integration constraints in each individual CASE tool, by adding support for addressing objects in all other languages, but this approach is clearly unworkable. It thus makes sense to provide a checking service that does not reside in any one tool, but *between* tools. Such a checking service can bridge the heterogeneity gap between the tools by either reducing all specification languages to a common vocabulary such as first order logic – a difficult process that may cause prohibitive overhead – or by providing a mechanism for expressing relationships between artifacts in different languages directly.

Thirdly, proper support for integration and custom constraints requires the problem of distribution to be addressed. Developers working on a large and complex system cannot be expected to store their specifications, implementation and deployment artifacts in one central repository. In order to fully support integration constraints between heterogeneous artifacts, we thus have to assume that these artifacts may be distributed, and perform a distributed consistency check. This kind of mechanism is clearly beyond the state of the art of current tools, most of which do not even provide mechanisms for splitting up and distributing individual specifications, let alone heterogeneous artifacts.

Finally, many constraint languages have been designed to be very expressive, at the cost of making it hard to provide diagnostic feedback. For example, most tools that provide support for OCL can only supply information on whether a constraint has been violated or not. OCL constraints are attached to model elements, so a constraint can be either *true* or *false* for that particular element. This is inadequate as it imposes the burden of finding the cause of inconsistency on the developer – it does not state why, for example due to the presence of other information, the inconsistency has occurred. This approach may be feasible, though time consuming, when checking a single UML model since the developer can look through the model to find the cause of the inconsistency. If multiple, heterogeneous, and distributed specifications are checked against each other, the overhead of browsing through all of them in search of what caused an inconsistency will be prohibitive. It is necessary in this setting to provide diagnostics that pinpoint exactly the combination of elements in all specifications that cause an inconsistency.

We will show that xlinkit can solve these problems, by building a consistency checking system on top of a framework that makes distribution and heterogeneity its underlying assumptions. We will give examples of the diagnostic power of xlinkit, which produces links between inconsistent elements instead of boolean results. We will also demonstrate how consistency checking support throughout the development life-cycle can be achieved, once such a heterogeneous framework is deployed in practice. We will give an example of the kinds of standard constraints xlinkit can check by expressing some of the standard constraints of the UML and demonstrate xlinkit's flexibility by specifying extension, integration and custom constraints for EJB-based systems.

4 Overview of xlinkit

xlinkit is a framework for checking the consistency of distributed, heterogeneous documents. It comprises a language, based on first order logic, for expressing constraints between such documents, a document management mechanism and an engine that checks the documents against the constraints. A full description of xlinkit, including a formal specification of its semantics, its scalability and an evaluation can be found in [25].

xlinkit has been implemented as a lightweight mechanism on top of XML and makes use of hyperlinks as a diagnostic to pinpoint inconsistent elements by linking them. Because it was built on XML, xlinkit is very flexible and can be deployed in a variety of architectures, for example as a standalone program, a web-based checking service or as a distributed high-performance checker.

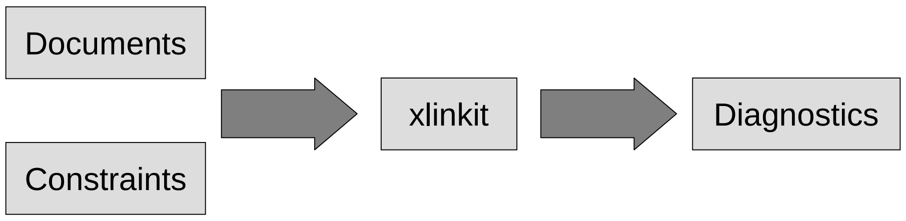


Figure 4: xlinkit basic operation

Figure 4 gives an abstract overview of the operation of xlinkit. Documents and constraints are submitted to xlinkit and checked. xlinkit then returns some diagnostic information that shows which elements in which documents have violated or satisfied a particular constraint.

We will explain the operation of xlinkit, starting with the document and constraint submission process, to the actual checking and then the diagnostics, using a small paedagogic example. We will assume that two developers are working on the same system, one specifying a UML model and the other working on a Java implementation, and that they are working separately and store their data on different machines. We wish to check the simple constraint that each class in the UML model has to be implemented as a Java class.

```
<DocumentSet name="UMLandJava">
    <Description>
        A UML model and some Java files
    </Description>

    <Document href="http://host1/UMLmodel.xml"/>
    <Document href="http://host2/Main.java" fetcher="JavaFetcher"/>
    <Set href="http://host2/ClassSet.xml"/>
</DocumentSet>
```

Figure 5: Sample document set

The developers somehow have to make their specifications available to xlinkit for checking. xlinkit provides *document sets* for including documents in a check and *rule sets* for selecting constraints. Figure 5 shows the document set for our example – it contains `Document` elements that instruct xlinkit to load the documents directly from the given URL and a `Set` element that includes a further document set, in this case a set containing further Java files. Constraints are similarly assembled into *rule sets*, and rule sets can contain further rule sets. Since the sets can be arbitrarily distributed, it is possible to assemble useful rule sets from third parties and to add or remove rule sets in a check to vary its "strictness".

The `fetcher` attribute of the `Document` element in the figure is used by xlinkit for dealing with input that is not in XML format. A *fetcher* is a plug-in that is capable of loading particular file type and turning into a DOM tree. The fetcher attribute is set to *FileFetcher* by default, a fetcher class that loads XML files. xlinkit builds on this fetcher mechanism to abstract from the underlying data source and make a wide variety of formats available for checking: we have written fetcher classes that load relational tables, making it possible to check consistency between distributed databases, and a class that parses Java source files and makes them available as a DOM tree (further details will be presented in the case study). It is thus possible to use xlinkit to check consistency between arbitrary data sources, as long as their content can be transformed into a DOM tree.

In order to explain the operation of xlinkit and to understand the diagnostics it produces as output, we will have to discuss its underlying formalism. xlinkit uses a language based on first order logic that has been adapted for use with XML and has been restricted to make it decidable in polynomial time. Figure 6 gives the abstract syntax of the language. Note that all sets in the language are sets generated by evaluating XPath expressions over the documents that are being checked, and that there are no functions allowed.

We can write the example constraint in the xlinkit language abstractly as

$$\forall c \in \text{“}\mathit{umlclasses}\text{”} (\exists j \in \text{“}\mathit{javaclasses}\text{”} (\text{“}\$c/\mathit{name}\text{”} = \text{“}\$j/\mathit{name}\text{”}))$$

where the expressions in quotes are placeholders for XPath expressions.

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

Figure 6: Rule language abstract syntax

In order to present this constraint to xlinkit, we use an XML rule language as a concrete syntax. Figure 7 shows the constraint written as an xlinkit *consistency rule* ready to be checked. It should be noted that the consistency rule makes no references to where the elements indicated in the constraint should be retrieved from - the language is *location transparent* and data sources could be arbitrarily distributed. At run-time, xlinkit will try to apply the XPath expressions in the constraint to all documents in the document set and thus build up a set of nodes to be checked.

```
<globalset id="$classes"  xpath="//Foundation.Core.Class[@xmi.id]"/>
<globalset id="$javaclasses" xpath="/java/class"/>

<consistencyrule id="r1">
  <description>
    Every class in the UML model must be
    implemented as a Java class
  </description>

  <forall var="c" in="$classes">
     <exists var="j" in="$javaclasses">
        <equal op1="$c/Foundation.Core.ModelElement.name/text()"
               op2="$j/@name"/>
     </exists>
  </forall>
</consistencyrule>
```

Figure 7: Sample constraint in XML

An important contribution of xlinkit is the definition of a new semantics for this restricted form of first-order logic. We go beyond the boolean evaluation of such formulae that returns *true* or *false* and specify a new semantics in terms of hyperlinks that link consistent or inconsistent elements. This semantics takes the DOM nodes currently assigned to variables of quantifiers in the formulae and turns them into link endpoints. It was especially designed to determine which parts of a formula to “blame” depending on whether the formula returns true or false for a particular element. It discards irrelevant information and links together only those node that have directly contributed to the boolean result of the formula. As a simple example, in the formula $a \wedge b$, if a is true and b is false, the formula fails due to b and only due to b, and hence b would be included in the link. xlinkit is thus able to provide much better diagnostic information to the user, who can now see which combination of elements causes an inconsistency rather than just get information if a particular element has violated a rule. xlinkit’s semantics has been formally specified [25] and evaluated in a number of case studies and has produced good results in each case.

```
<xlinkit:LinkBase date="Wed Oct 03 16:10:52 EDT 2001"
    docSet="file://DocumentSet.xml" ruleSet="file://RuleSet.xml"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:xlinkit="http://www.xlinkit.com">

    <xlinkit:ConsistencyLink ruleid="zrule.xml#/id('r1')">
        <xlinkit:State>consistent</xlinkit:State>
        <xlinkit:Locator
            xlink:href="http://host1/UMLModel.xml#//Foundation.Core.Class[@xmi.id='S.1']"/>
        <xlinkit:Locator
            xlink:href="http://host2/Main.java#/java/class"/>
    </xlinkit:ConsistencyLink>
    <xlinkit:ConsistencyLink ruleid="zrule.xml#/id('r1')">
        <xlinkit:State>inconsistent</xlinkit:State>
        <xlinkit:Locator
            xlink:href="http://host1/UMLModel.xml#//Foundation.Core.Class[@xmi.id='S.2']"/>
    </xlinkit:ConsistencyLink>
</xlinkit:LinkBase>
```

Figure 8: Sample result links in XLink linkbase

Figure 8 shows two hyperlinks in an XLink linkbase that may have been generated from our sample rule – the XPaths have been abbreviated for clarity. In this case, as by default, xlinkit has generated links between consistent elements, “consistent links”, and between inconsistent elements, “inconsistent links”. This default behaviour can be overridden to only provide inconsistent or consistent links. It can be seen that the consistent UML class has been linked to the Java class it conforms to and that the inconsistent UML class has not been linked to anything – because there is no matching Java class. In both cases, the user can refer to the `ruleid` attribute of the consistency link to see which rule generated the link. The linking semantics of xlinkit can deal not only with simple constraints like this but has been defined for all logical operators and arbitrary first order logic formulae. We will see several examples in the remainder of the paper of links that have been generated from more complex formulae.

We will show how xlinkit can be applied in a software engineering setting in the following section.

5 Supporting the EJB Development Lifecycle

5.1 Introduction

We now present a comprehensive case study that demonstrates how the different types of constraints we have identified arise in practice and how they can be addressed. In this case study we use xlinkit to check the standard, extension, integration and custom constraints that arise in different stages, and between stages, of the development life-cycle of a system based on Enterprise JavaBeans. We precede the study with a short overview of Enterprise JavaBeans.

Enterprise JavaBeans [23] are a popular technology for component-based software development. Components are implemented as *beans* that reside in a *container*. The fundamental idea underlying EJB is that the container should provide all the infrastructure services for the bean, such as transaction management, persistence, replication and communication with the underlying middleware, so that the developer can concentrate on implementing business logic.

Implementing an EJB requires the developer to supply the following artifacts: A *remote interface* specifying the methods the bean supplies to client objects, a *home interface* specifying the methods the bean supplies to its container, and a *bean implementation*, containing the business logic and event methods that are called when the life cycle of the bean changes. A distinction is made here between *entity beans*, which represent persistent data, and *session beans*, which implement control flow.

When the bean is ready to be deployed into the container, the developer must further provide a *deployment descriptor*, expressed in XML, that describes the type of the bean and its quality-of-service contract with the container – for example whether persistence of the bean's data will be managed by the bean or by the container.

In the case study, the Unified Modeling Language (UML) [31] was used as the design language of choice. While the UML is semantically rich enough to express the implementation details of an EJB-based system, the expressiveness of the design model can be improved by clearly identifying the features supplied by the EJB architecture, that is by making the architecture explicit. It is possible by these means to distinguish the high-level components supporting the EJB architecture from implementation details.

The UML Profile for EJB [12] – the "EJB profile" or just "profile" from here on – identifies a set of stereotypes that can be used to label model elements that represent EJB artifacts, and the relationships between them. Figure 9 shows a fragment of a UML model annotated in this way. Since the stereotypes expose new semantics for the model, they cannot be combined arbitrarily, but have to obey the constraints set out in the profile.

Given the UML model, the implementation and the deployment information, we can now specify a framework for managing the consistency of the system as artifacts are added, changed and removed during the lifecycle. Figure 10 shows the checks that we want to perform: we want to check the UML model internally, first by checking that the standard constraints set out in the UML specification are obeyed, and then that the extension con-

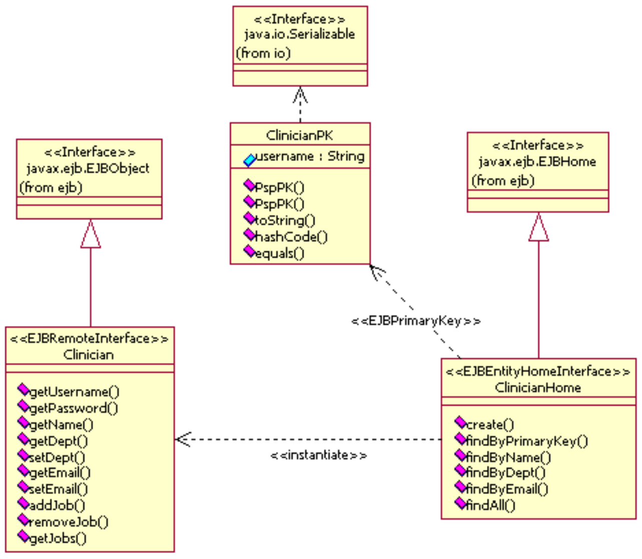


Figure 9: EJB-Profile compliant UML design of a single bean

straints of the EJB profile hold; we wish to check the integration constraints between the UML model and the deployment descriptor and implementation, between the deployment descriptor and the implementation, and additional custom constraints inside the implementation. We will discuss each of these types of checks using an example taken from the study.

Our first check will determine if the UML model is a valid model. The UML specification sets out the constraints that have to hold for each metaclass in the model. We have expressed a subset of those constraints – those of the **Foundation.Core** package, which deals with static information such as class diagrams – for use in this case study. There are 34 constraints in total, they are listed in Appendix A. These constraints are clearly *standard constraints*. Figure 11 gives an example of such a constraint expressed in xlinkit's notation.

We then check the constraints of the EJB profile against our model. We have chosen to express a subset of those constraints for this case study. This subset specifies the relationships of EJB elements that make up the "external view" of a system: remote and home interfaces, EJB methods, and primary key classes. We have most constraints in this subset, apart from the constraints on EJB methods and RMI interface inheritance. These constraints are no more complex than the ones we have expressed and would not add significant value to this study. The complete list of constraints we have expressed, 16 in total, is given in Appendix B. These constraints are *extension constraints* since they supplement or customize the UML for use in a particular application domain. Figure 12 gives an example of an EJB profile constraint.

We also check the UML design against the Java implementation and against the deployment descriptor, and the Java implementation directly against the deployment descriptor.

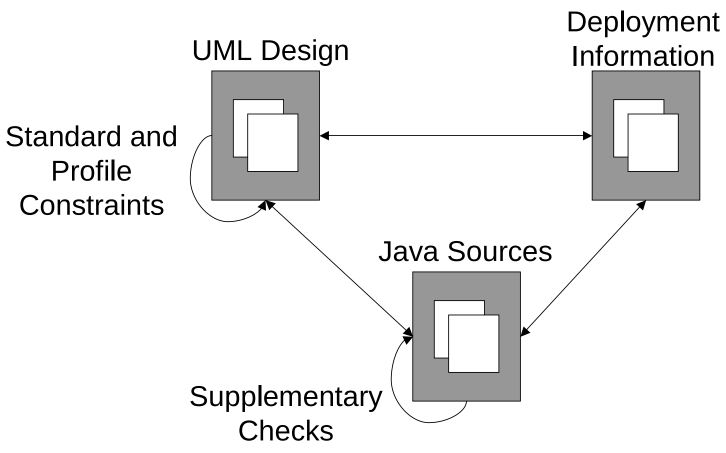


Figure 10: Consistency checks for EJB development

```
<description>
    The AssociationEnds must have a unique name within the Association
</description>
<forall var="a" in="//Foundation.Core.Association">
    <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>
```

Figure 11: Example of a UML *standard* constraint

These are all examples of *integration constraints* as they restrict the content of software development artifacts depending on the contents of additional artifacts. In the case of these checks, there is no standard to guide us. Indeed, we expect these checks to vary depending on the needs of developers or project managers. We have consequently only provided nine samples of the kinds of relationships that could be checked, they are also listed in Appendix B. Figure 13 shows a sample constraint that checks if every field in an entity bean that has been declared as a container-managed persistent field in the deployment descriptor is present as a variable in the class implementing the bean.

Finally, we have specified some additional checks for the Java implementation. These checks are internal to the implementation, but do check relationships between multiple Java files. To take one example, we have specified the constraint that for each remote interface there must be a bean implementation class that implements the interface. This

```
<description>
    The (EJB entity home) class must be tagged as persistent.
</description>
<forall var="c" in="$classes">
    <implies>
        <exists var="s" in="id($c/Foundation.Core.ModelElement.stereotype/
                               Foundation.Extension_Mechanisms.Stereotype/
                               @xmi.idref)[Foundation.Core.ModelElement.name/
                               text()='EJBEntityHomeInterface']"/>
        <exists var="t" in="$c/Foundation.Core.ModelElement.taggedValue/
                             Foundation.Extension_Mechanisms.TaggedValue[
                             Foundation.Extension_Mechanisms.TaggedValue.tag/
                             text()='persistence' and
                             Foundation.Extension_Mechanisms.TaggedValue.value/
                             text()='persistent']"/>
    </implies>
</forall>
```

Figure 12: Example of an EJB profile *extension* constraint

constraint is not checked by the Java interface implementation mechanism because EJB implementation classes do *not* implement their remote interface. Figure 14 shows this constraint expressed in xlinkit. We have expressed two sample constraints, they are given in Appendix B. These constraints are not standardised and do not depend on the contents of further artifacts, and are thus *custom constraints* – although one could imagine that they could become extension constraints in the future.

5.2 Evaluation

In order to realize the checks outlined in Figure 10, we have to map them onto the mechanisms provided by xlinkit. This mapping is straightforward, Figure 15 shows how the documents and constraints are managed using xlinkit. We will first discuss the mapping process and then present the results of checking the constraints against our case study

```
<description>
    Each attribute listed as a 'cmp-field' for an entity bean in the
    deployment descriptor must be an attribute of the bean implementation
    class
</description>
<forall var="c" in="$entitydescr">
    <forall var="b" in="$javaclasses[concat(../package/@name,'.',@name)=
                        $c/ejb-class/text()]">
        <forall var="f" in="$c/cmp-field">
            <exists var="v" in="$b/var[@name=$f/field-name/text()]"/>
        </forall>
    </forall>
</forall>
```

Figure 13: Example of a deployment descriptor – implementation *integration* constraint

```
<description>
    Every remote interface is implemented by a bean class that resides
    in the same package
</description>
<forall var="i" in="/java/interface[extends/@name='EJBObject']">
    <exists var="c" in="/java/class[../package/@name=$i/../package/@name
                        and (implements/@name='EntityBean' or
                             implements/@name='SessionBean')]"/>
</forall>
```

Figure 14: Example of a *custom* constraint

system.

We load the UML model, the deployment descriptor and the Java files into xlinkit's DocumentSets. The UML model can be straightfowardly loaded as an XMI file and the deployment descriptor can also be easily loaded since it is already represented in XML markup.

In order to include the Java files into the check, we have to provide an additional *fetcher* that can parse Java classes and turn them into DOM trees. This was done by annotating the productions in a standard grammar for Java with commands that emit nodes for the tree. Figure 16 shows an XML file that was generated by parsing a small Java class, turning it into a DOM tree and serializing it to XML. We do not convert the entire Java file, including the code inside the methods, because we do not require this information in our checks. If such information was desired, recent approaches from compiler construction like XANTLR [38] could be used instead.

The different types of checks that we wish to perform map nicely onto xlinkit's RuleSets: each arc in Figure 10 becomes a rule set in Figure 15. The individual rule sets can then either be checked individually, as indicated in the figure, or assembled into one combined rule set to check all the rules.

Expressing the constraints was not straightforward at first, due to the complexity of XMI. While the logic of the constraints was straightforward, the path expressions tended to become quite verbose. Fortunately many constraints reference the same model elements in the XMI file, making it possible to reuse most path expression. Projects with undergraduate students have also shown that expressing constraints over XMI, while presenting a steep learning curve in order to get to grips with the internal format of XMI, leads to a certain familiarity over time, allowing most students to write constraints within a couple of weeks. We encountered no problem at all expressing any of the other constraints involving the deployment descriptor or Java sources, which have straightforward XML encodings. These constraints were all expressed within a matter of minutes.

We have evaluated our constraints against a complete and working EJB-based system that was built to provide a web front-end for statistical evaluations of drug trials. The system consists of 9 EJBs, 4 session beans and 5 entity beans. The design model was annotated with the required stereotypes of the EJB profile, and was 2.5 MB in size. The

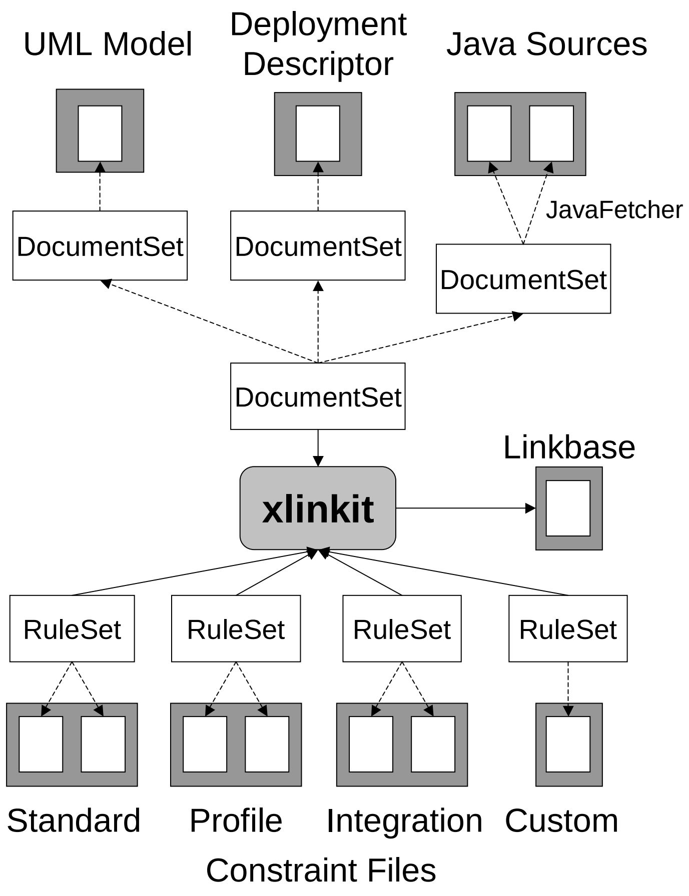


Figure 15: EJB consistency checks using xlinkit

implementation comprised 40 Java source files. The checks were performed on an Intel-based machine running at 750Mhz and using the IBM JDK 1.2.

We first checked the UML model separately before checking the remaining constraints. This check, against all 34 rules, and including loading and parsing the UML model, took 58 seconds. The check generated 324 inconsistent links. Many of those inconsistencies were caused by associations having association ends with equal names, a constraint that is often violated in design models that show high level views, by leaving association ends unnamed. Other inconsistencies were caused by features not present in the XMI files produced by Rational Rose, such as the distinction between methods and operations. No other significant inconsistencies were found.

We then proceeded to check the extension, integration and custom constraints between the UML model, the deployment descriptor and the Java files. Checking all 26 rules against all 42 files tooks 56 seconds, including parsing the Java files and loading the UML models. By far the greatest amount of time was spent on checking rules that related to the UML model, due to the complexity of the XPath expressions needed to get information from XMI. None of the rules involving only Java sources or the deployment descriptor took

```
<?xml version="1.0" encoding="UTF-8"?>
<java>
    <package name="uk.ac.ucl.aivetac.session.formjob"/>
    <class name="FormJobWorkFlowBean">
        <implements name="SessionBean"/>
        <method name="findHome"/>
        <method name="allProjects"/>
        <method name="allProtocols"/>
        <method name="allReports"/>
        <method name="setSessionContext"/>
        <method name="ejbActivate"/>
        <method name="ejbPassivate"/>
        <method name="ejbCreate"/>
        <method name="ejbRemove"/>
    </class>
</java>
```

Figure 16: Java structure representation in XML

longer than 380 milliseconds each to check.

Checking the extension, integration and custom constraints generated 94 inconsistent links. 92 of those were caused by the model not being entirely EJB-profile compliant: the methods in the home and remote interfaces did not carry any EJB method stereotypes. One other inconsistency occured because the primary key for one entity bean was specified to be a primitive type, rather than a class type, as required by the EJB specification. And finally, a variable that had been declared in the deployment descriptor as a persistent field for an entity bean was missing in the bean implementation.

The quality of the links produced by xlinkit's link generation semantics can be seen by looking at what kinds of links are produced by the sample rules we have shown in the previous subsection:

- *The AssociationEnds must have a unique name within the Association* (Figure 11). For associations that violated that rule, xlinkit produced a ternary link that connects the association and the two association ends with equal names.

```
<xlinkit:ConsistencyLink ruleid="javadeploy_inter.xml#//consistencyrule[@id='r2']">
    <xlinkit:State>inconsistent</xlinkit:State>
    <xlinkit:Locator
        xlink:href="/config/ejb-jar.xml#/ejb-jar/enterprise-beans[1]/entity[1]"/>
    <xlinkit:Locator
        xlink:href="/entity/protocol/ProtocolBean.java#/java/class[1]"/>
    <xlinkit:Locator
        xlink:href="/config/ejb-jar.xml#/ejb-jar/enterprise-beans[1]/
                                       entity[1]/cmp-field[3]"/>
</xlinkit:ConsistencyLink>
```

Figure 17: Sample inconsistency - field from deployment descriptor not implemented

- *The (EJB entity home) class must be tagged as persistent* (Figure 12). For classes that violated that constraint, xlinkit produced a link from the class to the stereotype declaring the class as an EJB entity home interface. This makes sense because in the absence of persistence tag, the inconsistency is caused by the stereotype being present.
- *Every remote interface is implemented by a bean class that resides in the same package* (Figure 14). If no such bean class exists for a particular remote interface, an inconsistent link that points to the remote interface is generated.

As an example of what these links look like in their XLink form, Figure 17 shows the link xlinkit generated when the constraint in Figure 13 was violated: The `entity` declaration in the deployment descriptor is linked to the `ProtocolBean` class implementation, and the `cmp-field` (Container-Managed Persistence field) in the descriptor. The cause of the inconsistency is thus immediately obvious by looking at the rule description text and this link: the linked field in the deployment descritor should have been added to the linked class, or removed from the descriptor.

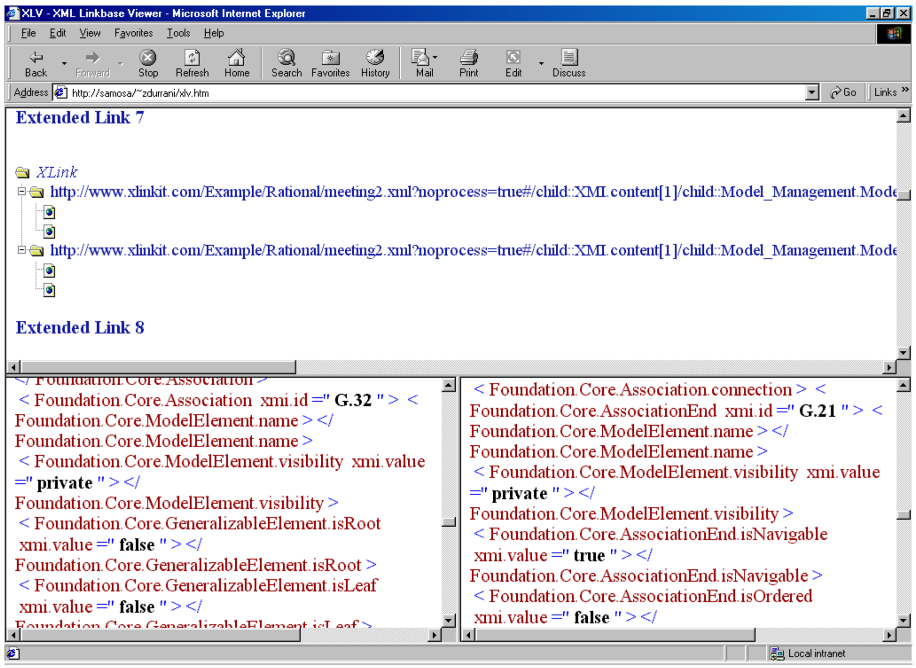
This case study demonstrates that xlinkit can be used to check all four types of constraints we have identified; that it can cope with heterogeneous notations; that the constraints expressed in xlinkit's language are transparent with respect to distribution, and that distribution of specifications can be addressed through a structured document management mechanism. It shows how xlinkit's novel semantics for first order logic, which produces hyperlinks instead of boolean results, generates powerful diagnostics that connect inconsistent elements across specifications. And, importantly, it demonstrates that inconsistencies in the design have not led to any problems in the working EJB system, which justifies our tolerant approach to inconsistency.

6 Visualization

We have shown how the linking semantics of xlinkit is used to automatically generate hyperlinks between inconsistent elements in software engineering specifications. These links will be stored in a linkbase that is returned to the user as diagnostic information. While these links pinpoint all the information required to make a detailed judgement on which elements cause inconsistency, we appreciate that developers may not want to look at an XML linkbase to get feedback.

This problem is part of the wider problem of integrating CASE tools over a hypertext infrastructure, which has been addressed in [1]. Nevertheless, we provide some inexpensive solutions for making consistency information more accessible.

There are several ways in which the diagnostic information can be made more accessible: we can provide improved visualisation of the XML linkbase to make the task of comparing related elements easier. Figure 18 shows a screenshot of a linkbase rendered into an interactive HTML page by a servlet. The user can click on a link, and upon doing so,


XLV - XML Linkbase Viewer - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Address http://samosa/~zdurrani/xlv.htm
Go Links
Local intranet

Extended Link 7

XLink

http://www.xlinkit.com/Example/Rational/meeting2.xml?noprocess=true#/child::XMI.content[1]/child::Model_Management.Mode

http://www.xlinkit.com/Example/Rational/meeting2.xml?noprocess=true#/child::XMI.content[1]/child::Model_Management.Mode

Extended Link 8

```
</ Foundation.Core.Association >
 < Foundation.Core.Association  xmi.id =" G.32 " > <
Foundation.Core.ModelElement.name > </
Foundation.Core.ModelElement.name >
 < Foundation.Core.ModelElement.visibility  xmi.value
=" private " > </
Foundation.Core.ModelElement.visibility >
 < Foundation.Core.GeneralizableElement.isRoot
 xmi.value =" false " > </
Foundation.Core.GeneralizableElement.isRoot >
 < Foundation.Core.GeneralizableElement.isLeaf
 xmi.value =" false " > </
Foundation.Core.GeneralizableElement.isLeaf >
```

```
 < Foundation.Core.Association.connection > <
Foundation.Core.AssociationEnd  xmi.id =" G.21 " > <
Foundation.Core.ModelElement.name > </
Foundation.Core.ModelElement.name >
 < Foundation.Core.ModelElement.visibility  xmi.value
=" private " > </
Foundation.Core.ModelElement.visibility >
 < Foundation.Core.AssociationEnd.isNavigable
 xmi.value =" true " > </
Foundation.Core.AssociationEnd.isNavigable >
 < Foundation.Core.AssociationEnd.isOrdered
 xmi.value =" false " > </
```

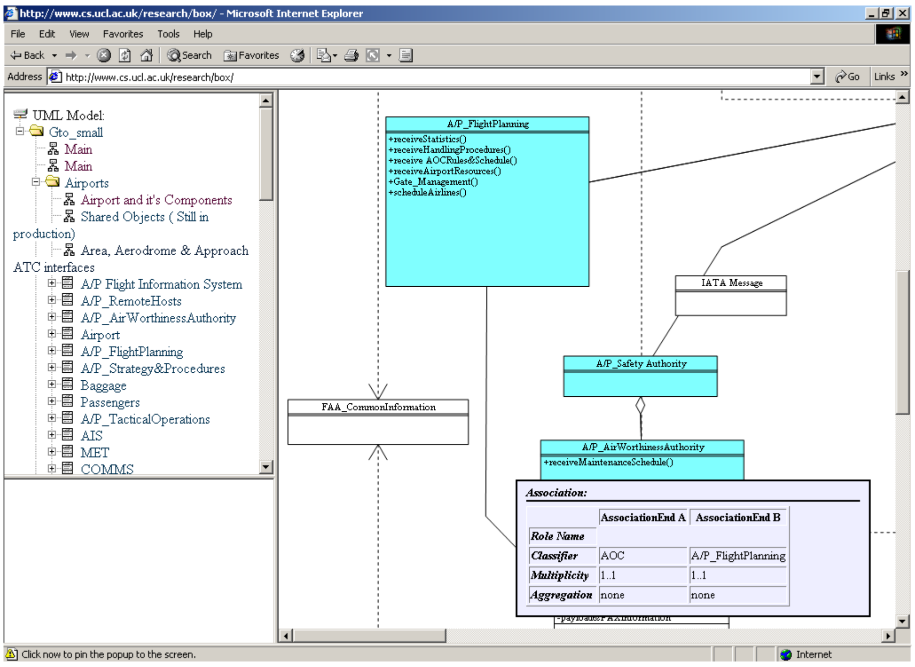
Figure 18: Dynamic linkbase servlet

the documents at two endpoints of the link are juxtaposed at the bottom. The linked elements are centered and can be easily compared.

The next option is to provide web-based visualisation of the models involved in a check. We have built a tool, BOX [27], that transforms UML models from their XMI representation into SVG, a vector graphic markup language. Figure 19 shows a fragment of a UML model rendered in a browser, with a model tree on the left hand side and a diagram rendered on the right. BOX can be used to display inconsistency information graphically, for example by drawing coloured arcs between inconsistent elements, and by providing textual annotations for UML elements, such as data types, that have no diagrammatic representation. A challenge arises when inconsistency links between UML models and other types of specifications are to be displayed. We have produced a prototype that shows which UML elements are inconsistent and are working on the remaining functionality.

We have also implemented our own "linkbase processor", XTooX. A linkbase processor takes a linkbase of out-of-line links, and inserts the links into the files that they are pointing to. The files can then be rendered in a standard manner using stylesheets. By these means a simple web portal can be constructed.

The final, and probably most interesting option from an industrial point of view, is to let the CASE tools of the individual specifications display the inconsistency information. Many CASE tools support scripting languages, which could be used to load linkbase information and point to inconsistent elements. If this approach is to work with heterogeneous CASE tools, a common visualisation framework will have to be established. Such a generic framework could then be used to display hyperlinks between elements in any of the CASE


http://www.cs.ucl.ac.uk/research/box/ - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://www.cs.ucl.ac.uk/research/box/
Go Links
UML Model:
Gto_small
Main
Main
Airports
Airport and it's Components
Shared Objects (Still in production)
Area, Aerodrome & Approach ATC interfaces
A/P Flight Information System
A/P_RemoteHosts
A/P_AirWorthinessAuthority
Airport
A/P_FlightPlanning
A/P_Strategy&Procedures
Baggage
Passengers
A/P_TacticalOperations
AIS
MET
COMMS
A/P_FlightPlanning
+receiveStatistics()
+receiveHandlingProcedures()
+receive AOCRules&Schedule()
+receiveAirportResources()
+Gate_Management()
+scheduleAirlines()
IATA Message
A/P_Safety Authority
FAA_CommonInformation
A/P_AirWorthinessAuthority
+receiveMaintenanceSchedule()
Click now to pin the popup to the screen.
Internet

Association:

	AssociationEnd A	AssociationEnd B
Role Name		
Classifier	AOC	A/P_FlightPlanning
Multiplicity	1..1	1..1
Aggregation	none	none

Figure 19: SVG view of a UML model

tools. There is evidence that this approach is feasible in the work on Chimera [1].

As an inexpensive alternative, it is possible to make use of the native notation of a CASE tool to “trick” it into displaying inconsistency information. We could, for example, insert stereotyped dependencies into an XMI file of a UML model that connect inconsistent elements. The CASE tools would then display these links at no extra cost.

7 Related Work

Below we give an account of related work in the area of software engineering. The general problem of consistency management is of course much broader and spans a large number of disciplines. For a survey of how xlinkit relates to work in databases and hypermedia please refer to [25].

Consistency management has been recognised 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 [35], Gandalf [22] or Centaur [6]. These environments typically provide mechanisms for automatically creating syntax-directed editors from grammars. When the user has finished entering a construct, incremental consistency checks related to the static programming language semantics 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 [24], Arcadia [37], ESF [36], ATMO-

SPHERE [5] and GOODSTEP [14] raised the complexity by integrating tools for different languages. The latter in particular allowed the specification of *semantic rules* [14]. Checks for semantic integrity between documents could be triggered by user actions.

Our approach represents a generalisation in that it builds on the open model of XML rather than specific programming formalisms. While most SDEs are based on centralised repositories such as PCTE [7] or an object database, we allow for the distribution of the documents across Web servers and provide appropriate diagnostics in the form of links. The distribution of documents combined with our "tolerant" view of inconsistency led us to a much more loosely integrated approach, where consistency checks will not be triggered by modifications to documents but invoked only at certain points in time – for example at predefined points in a process or when specifications are baselined.

Consistency management in software engineering has itself become a topic, for a survey we refer to [28] and for a research agenda to [17]. The "tolerant" approach to inconsistency, which asserts that inconsistency cannot be eradicated in any sufficiently complex system, but must be monitored nonetheless, was first seen in the area of databases in [4]. This tolerant approach gains special significance in our scenario of distributed specifications, as enforcing total consistency becomes even less feasible.

A viewpoint [19] 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 multi-perspective specifications are outlined in [18]. Research in the viewpoints area also introduces the idea of *consistency rules* [11] 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 [17]. 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 realises these ideas by providing a concrete implementation on top of which a viewpoint framework can be built.

Chimera [1] demonstrates how heterogeneous software engineering tools can be integrated over a hypertext infrastructure. It shows how n-ary links can be established and visualised between distributed CASE tools. We believe that our work complements Chimera, as it could be used as a front-end to establish a "web of inconsistency" – or a "web of consistency" – across tools, while Chimera takes care of visualisation.

Finally, the link generation semantics for xlinkit was formally specified and evaluated in [25]. xlinkit's UML constraints have also been tested against a series of industrial models [26]. xlinkit is based on previous work on consistency checking using XML technologies [13]. The expressive power of xlinkit has since been greatly increased, permitting first order logic constraints that relate any number of documents, rather than just two documents in a pair-wise comparison. xlinkit's link generation semantics has also removed the need for linking annotations that were previously used to guide the checker, the document and rule sets allow for more structured input, and the fetcher subsystem enables checking of legacy data that is not in XML format. Scalability of consistency checks was addressed in [20], which contains an evaluation of a mobile agent architecture for incremental consistency checks.

8 Conclusion and Future Work

We have identified in this paper a set of challenges for consistency management that arise during the development of large and complex systems. Based on a categorisation of the types of consistency constraints that occur in and between specifications at various stages of the lifecycle, we have identified a set of requirements that consistency management mechanisms have to address in order to provide proper support. Those are: flexibility in constraint application and a tolerant approach to consistency; support for distributed specifications; a mechanism for bridging the heterogeneity gaps between different specification languages, without resorting to a "common" vocabulary; and strong diagnostics that show which parts of specifications contribute to inconsistency.

Using the xlinkit framework, we have demonstrated that it is possible to address these problems in a very light-weight manner, without requiring tight integration, complex translation of specifications or bulky tools. xlinkit's hyperlink semantics has been shown to be useful in the Enterprise JavaBeans case study, where it was used to link inconsistent specification elements, irrespective of whether they were part of the design, implementation or deployment descriptor.

Our work with xlinkit and our case studies have left some questions unanswered and have also left us with new insights. Firstly, the different rule sets in the study were not equal citizens. The Java-UML constraints, for example, cannot be checked before the profile constraints have been checked and the model has been made consistent. The reason is that the Java constraints rely on the correct stereotypes and tagged values being in place so as to relate to various bean properties in the model. It thus seems that some lightweight process is required: In the initial stages, the design model will have to be annotated with the correct stereotypes from the EJB profile. The first rule set can then be checked to find any errors in the annotation. Once the correct annotation is in place, the remaining checks can be performed. We do not provide an automated dependency mechanism between constraints because that would violate our view of consistency: a developer may want to incompletely annotate a UML model and still perform some checks against Java source files. Instead, we leave it to developers to chose the order in which they make their changes. It would also be possible to connect xlinkit with a process or workflow engine, although this would have to be done carefully if the light-weight and unintrusive characteristics of xlinkit were to be maintained.

We have not addressed the problem of what happens once inconsistencies have been detected. While we have outlined some ideas for visualising consistency links, the problem of acting on inconsistency is complex and cannot be addressed here. We are planning to make some low-level repair facilities available based on the links in our linkbases. How such repair mechanisms would be integrated in the overall development process is a matter for further research.

The checking times in the study, while reasonable, may not be adequate for interactive applications, such as the edit-compile-debug cycle common in software engineering. We have already made inroads into this problem by specifying an incremental checking scheme that only checks those constraints that are affected by changes to documents. We are currently evaluating whether this scheme will provide sufficient time savings.

Finally, we are investigating the application of xlinkit in a variety of different domains, ranging from finance to bio-informatics. In the latter case, we will attempt to check the consistency of large (gigabytes) database containing information about proteins. xlinkit's checking mechanism is fairly architecture-independent, which may be to our benefit since early investigation shows that the requirements on consistency checking architectures vary widely between different application domains.

Acknowledgements

We would like to thank Anthony Ivetac, who not only pointed us to the EJB profile for the UML but also made the case study implementation available. We thank Zeeshawn Durrani for implementing the linkbase servlet and Danila Smolko and Andrea Zisman for early work on using XML for consistency management. We gratefully acknowledge financial support from Zuhlke Engineering for Christian Nentwich.

References

[1] K. M. Anderson, R. N. Taylor, and E. J. Whitehead. Chimera: hypermedia for heterogeneous software development enviroments. *ACM Transactions on Information Systems*, 18(3):211–245, July 2000.

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

[3] V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. Le Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, and L. Wood. Document Object Model (DOM) Level 1 Specification. W3C Recommendation http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001, World Wide Web Consortium, October 1998.

[4] R. Balzer. Tolerating Inconsistency. In *Proceedings of the 13th International Conference on Software Engineering*, pages 158–165, Austin, TX USA, May 1991. IEEE Computer Society Press.

[5] J. Boarder, H. Obbink, M. Schmidt, and A. Völker. Advanced techniques and methods of system production in a heterogeneous, extensible, and rigorous environment. In N. Madhavji, W. Schäfer, and H. Weber, editors, *Proc. of the 1st Int. Conf. on System Development Environments and Factories*, pages 199–206, Berlin, Germany, 1989. Pitman Publishing.

[6] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The System. *ACM SIGSOFT Software Engineering Notes*, 13(5):14–24, 1988. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston, MA, USA.

[7] G. Boudier, F. Gallo, R. Minot, and I. Thomas. An Overview of PCTE and PCTE+. *ACM SIGSOFT Software Engineering Notes*, 13(2):248–257, 1989. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston, MA, USA.

[8] 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.

[9] 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.

[10] S. DeRose, E. Maler, and D. Orchard. XML Linking Language (XLink) Version 1.0. W3C Recommendation http://www.w3.org/TR/xlink/, World Wide Web Consortium, June 2001.

[11] 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.

[12] Jack Greenfield (Editor). UML/EJB Mapping Specification 1.0. Technical Report JSR-000026, Java Community Process, May 2001.

[13] 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.

[14] 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.

[15] J. Ernst. *CDIF – XML-based Transfer Format*. Electronic Industries Association, Engineering Dept., http://www.cdif.org, June 1998.

[16] D. C. Fallside. XML Schema Part 0: Primer. Recommendation http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/, World Wide Web Consortium, MAY 2001.

[17] 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.

[18] 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.

[19] 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.

[20] A. Finkelstein and D. Smolko. Software Agent Architecture for Consistency Management in Distributed Documents. In *Proceedings of SCI 2000 – World Multiconference on Systemics, Cybernetics and Informatics and ISAS 2000 – 6th Internatiomal Conference on Information Systems Analysis and Synthesis*, pages 715–719. International Institute of Informatics and Systemics, 2000.

[21] 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.

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

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

[24] M. Nagl, editor. *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, volume 1170 of *Lecture Notes in Computer Science*. Springer Verlag, 1996.

[25] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: a Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2001. To appear.

[26] C. Nentwich, W. Emmerich, and A. Finkelstein. Static Consistency Checking for Distributed Specifications. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE)*, San Diego, CA, November 2001. IEEE Computer Science Press. To appear.

[27] C. Nentwich, W. Emmerich, A. Finkelstein, and A. Zisman. BOX: Browsing Objects in XML. *Software Practice and Experience*, 30(15):1661–1676, 2000.

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

[29] Object Management Group. *The Meta Object Facility 1.3*. Object Management Group, 492 Old Connecticut Path, Framingham, MA 01701, USA, March 2000.

[30] Object Management Group. *UML Profile for CORBA Specification*, October 2000.

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

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

[33] Object Management Group. *The Common Object Request Broker: Architecture and Specification Revision 2.5.* 492 Old Connecticut Path, Framingham, MA 01701, USA, September 2001.

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

[35] 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.

[36] 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.

[37] R. N. Taylor, R. W. Selby, M. Young, F. C. Belz, L. A. Clarce, J. C. Wileden, L. Osterweil, and A. L. Wolf. Foundations of the Arcadia Environment Architecture. *ACM SIGSOFT Software Engineering Notes*, 13(5):1–13, 1988. Proc. of the 4th ACM SIGSOFT Symposium on Software Development Environments, Irvine, Cal.

[38] B. Trancon y Widemann, M. Lepper, J. Wieland, and P. Pepper. Automized Generation of Typed Syntax Trees Via XML. In *Proc. of the XSE Workshop*, pages 20–23, 2001.

A UML Foundation/Core Constraints

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

B EJB Constraints

B.1 Design Model - External View Constraints

B.1.1 EJB Remote Interface

[1] The Class must specialize a model elements that represents the Java Interface, javax.ejb.EJBObject
[2] All of the Operations contained by the Class must represent EJB Remote Methods
[4] The Class must be the supplier of a UML Usage, stereotyped as <<instantiate>>, whose client represents the EJB Home Interface of the same EJB Enterprise Bean

B.1.2 EJB Home Interface

[1] The Class must specialize a model elements that represents the Java Interface, javax.ejb.EJBHome
[2] All of the Operations contained by the Class must represent EJB Home Methods
[4] The Class must be the client of a UML Usage, stereotyped as <<instantiate>>, whose supplier represents the EJB Remote Interface of the same EJB Enterprise Bean

B.1.3 EJB Session Home

[1] The Class must not be tagged as persistent
[2] The value of the EJBSessionType tag must be either Stateful or Stateless
[3] The Class may not contain any Operations that represent EJB Finder Methods
[4] The Class must contain at least one Operation that represents an EJB Create Method
[5] If the value of the EJBSessionType tag is Stateless, then the Class must contain exactly one Operation that represents an EJB Create Method. The type of its return Parameter must be the supplier of a UML Usage, stereotyped as <<instantiate>>, whose client is the Class

B.1.4 EJB Entity Home

[1] The Class must be tagged as persistent
[2] The Class must contain exactly one Operation that represents an EJB Primary Key Finder Method. The type of its return Parameter must be the supplier of a UML Usage, stereotyped as <<instantiate>>, whose client is the Class
[3] The Class must be the client of a UML Usage, stereotyped as <<EJBPrimaryKey>>, whose supplier represents an EJB Primary Key Class. The supplier must be the type of the in Parameter of the Operation that represents the EJB Primary Key Finder Method

B.1.5 EJB Primary Key Class

[2] The Class must contain implementations for Operations named hashCode and equals
[3] The Class must be the supplier of a UML Usage, stereotyped <<EJBPrimaryKey>>, whose client represents an EJB Entity Home

B.2 Design - Implementation Sample Checks

[1] Each remote interface declared in the UML model is implemented as a Java interface extending EJBObject
[2] Each home interface declared in the UML model is implemented as a Java interface extending EJBHome
[3] Each bean class realizing a session bean home interface is implemented as a Java class implementing the SessionBean interface
[4] Each bean class realizing an entity bean home interface is implemented as a Java class implementing the EntityBean interface

B.3 Design - Deployment Information Sample Checks

[1] Each EJB Implementation Class declared in the UML model corresponds to an entry in the deployment descriptor
[2] If the name of an entity bean in the model matches a bean in the deployment descriptor, the bean in the deployment descriptor must be declared as an entity bean
[3] If the name of an session bean in the model matches a bean in the deployment descriptor, the bean in the deployment descriptor must be declared as an session bean

B.4 Implementation - Deployment Information Sample Checks

[1] Each bean listed in the deployment descriptor has an implementation for the given bean class, home interface and remote interface
[2] Each attribute listed as a container managed persistence field ('cmp-field') for an entity bean in the deployment descriptor must be an attribute of the bean implementation class

B.5 Implementation - Internal Checks

[1] For every remote interface there is a bean class that resides in the same package
[2] For every remote interface, there is a bean class in the same package that implements all the methods declared by the interface.