Consistency Management with Repair Actions

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

Comprehensive consistency management requires a strong mechanism for repair once inconsistencies have been detected. In this paper we present a repair framework for inconsistent distributed documents. The core piece of the framework is a new method for generating interactive repairs from full first order logic formulae that constrain these documents. We present a full implementation of the components in our repair framework, as well as their application to the UML and related heterogeneous documents such as EJB deployment descriptors. We describe how our approach can be used as an infrastructure for building higher-level, domain specific frameworks and provide an overview of related work in the database and software development environment community.

1 Introduction

Software engineers today make use of many informal and semi-formal notations during development, the UML being the most notable example. Large systems are increasingly developed in a distributed fashion, making it necessary to provide support for the distribution of development artifacts.

The goal of this work is to support software engineers in managing the consistency of their artifacts. Consistency management typically encompasses the specification of consistency constraints, consistency checking, and acting on inconsistency, or repair. In previous work on xlinkit [16], we have shown how a tool-independent service built on open, extensible and light-weight web technology can provide a strong consistency checking mechanism. The aim of this paper is to complete the circle and complement xlinkit with a strong, yet light-weight and tool-independent repair mechanism.

The main contribution of this paper is a tool-independent framework for repairing distributed documents. The core piece of this is a novel evaluation semantics that automatically derives a complete and correct set of repair actions from xlinkit's first order logic constraints.

The semantics has been fully implemented in the form of a repair action administration and execution tool. We discuss the design, features, and implementation of these programs. We give examples of how the approach can be used to repair inconsistent UML documents and related heterogeneous documents such as EJB deployment descriptors. Finally, we explain how the infrastructure provided by our solution supports the construction of higher-level repair frameworks such as precedence or workflow-driven repair.

The paper progresses from here as follows: we provide some background on the consistency checking mechanisms of xlinkit, which forms an essential part of the remaining discussion. We then present the architecture of our repair framework and discuss the repair action generation semantics, followed by an overview our implementation and options for higher-level repairs. The paper concludes with an account of related work and an outline of future work.

2 Background

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 [16]. The application of xlinkit to checking distributed and heterogeneous software engineering artifacts is described in [17, 18].

xlinkit builds on open technologies, most notably the eXtensible Markup Language (XML) [1] and XPath [3] to provide a lightweight and tool-independent consistency checking service for distributed documents. By combining first order logic with XPath expressions, xlinkit supports the specification of constraints over XML documents. Figure 1

is an example of an xlinkit *consistency rule* that implements a constraint from the UML [19] Core package, *"The AssociationEnds must have a unique name within the Association"*. This rule can be applied to UML models encoded in XMI [20], the standard UML interchange format.

```
<forall var="a" in="//UML:Association">
  <forall var="x" in="$a/*/UML:AssociationEnd">
    <forall var="y" in="$a/*/UML:AssociationEnd">
      <implies>
        <equal op1="$x/@name" op2="$y/@name"/>
        <same op1="$x" op2="$y"/>
      </implies>
    </forall>
  </forall>
</forall>
```

Figure 1. Sample consistency rule

One of the main contributions of xlinkit is that it provides a strong diagnostic for inconsistency: Instead of returning boolean values as the result of formula evaluation, it creates n-ary hyperlinks that connect inconsistent elements. Figure 2 shows the link generated by checking the constraint in Figure 1 against an inconsistent XMI document. It relates the association with the two association ends that have the same name, providing all the diagnostic information necessary to identify the elements causing the inconsistency.

```
<xlinkit:ConsistencyLink rule="as.xml#id('r1')">
  <xlinkit:State>inconsistent</xlinkit:State>
  <xlinkit:Locator
    xlink:href="inc.xml#//UML:Association[2]"/>
  <xlinkit:Locator
    xlink:href="inc.xml#//UML:Association[2]/*
                      /UML:AssociationEnd[1]"/>
  <xlinkit:Locator
    xlink:href="inc.xml#//UML:Association[2]/*
                      /UML:AssociationEnd[2]"/>
</xlinkit:ConsistencyLink>
```

Figure 2. Sample consistency link

Raw XML hyperlinks form a solid basis for further processing or tool integration, but they are not particularly useful to the end-user in their raw form. xlinkit provides a report generation tool, *Pulitzer*, that enables rule writers to attach diagnostic messages to inconsistent links. It processes these messages and generates HTML or XML report pages. Figure 3 shows a "report fragment" for our sample UML rule. It uses the `locator` variable to refer to the endpoints of the link in Figure 2, and retrieves information, in this case the name of the association and the name of its roles, for display. We will show later in this paper the result of processing this fragment.

```
<report:linkdescription state="inconsistent">
  The association
  <font color="red">
  <i>%@=gettext($locator[1]/@name)%</i></font>
  has two ends with the same role name,
  <font color="red">
  <i>%@=gettext($locator[2]/@name)%</i></font>.
</report:linkdescription>
```

Figure 3. Report Fragment

3 Architectural Overview

A consistency management system that performs a check and explains to the user which elements are contributing to an inconsistency is only partly complete. This section serves to introduce the architecture of the repair framework we have constructed to complement the xlinkit checker.

The aim of our work is to present to the user a simple set of options, or *repair actions*, to choose between when an inconsistency occurs. Figure 4 gives an overview of our architecture as a UML collaboration diagram.

The *repair administrator* creates repair actions by static analysis of a set of constraints. It is important that the actions are of high quality and not faulty, meaning in practice that they should be complete and correct:

- *There are no actions that can be taken to remove an inconsistency other than those generated by the generator*.
- *Any of the actions generated by the generator can remove an inconsistency*.

While our semantics can guarantee completeness and correctness, the repair administrator may still want to make use of their domain knowledge to disable actions that should not be offered to the user. For example, if an association has two equal role names, we can delete the association, delete either association end or rename either role. The repair administrator will know that it does not make sense to remove association ends – unary associations do not exists in the UML – and so can disable these two actions. The final output of the repair administrator is a parameterized set of actions that needs to be instantiated with actual inconsistent data.

When a linkbase is produced by the check engine, the user can process it with the *repair manager*. This tool will instantiate the correct set of repair actions for each inconsistency and present the available choices to the user, minus any actions disabled by the administrator. After selecting the actions for each violation, or choosing not to take action, the user can instruct the tool to execute the repair.

We deliberately restrict the scope of the architecture to the output of concrete actions. Document modification is

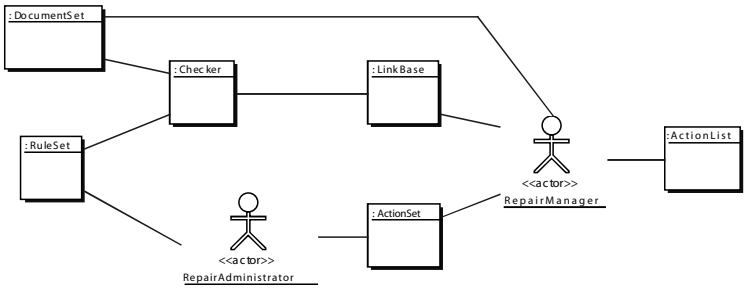

Figure 4. Repair Architecture

an implementation-specific mechanism and we did not wish to restrict our work to any particular technology. Simple implementations that can be provided include: a document update tool that makes copies of the original documents and returns modified instances – the only option when pure http is used and there is no way to modify the documents,– a tool that modifies remote documents over the web using infrastructure such as WebDAV [10], or full integration into proprietary tools, which would require a language to map from actions to component invocations.

With this brief overview in mind, we now discuss the core piece of the architecture, which is the repair action generator built into the repair administrator.

4 Repair Action Generation

The core contribution of this paper is the semantics that maps xlinkit's first order logic language to repair actions that can be used to interactively correct rule violations. In this section we provide a formal definition of this semantics, a step-by-step explanation and several examples.

$$
\begin{array}{rcl}
formula & ::= & \forall \mathbf{var} \in XPath(formula) \mid \\
& & \exists \mathbf{var} \in XPath(formula) \mid \\
& & formula \ \mathbf{and} \ formula \mid \\
& & formula \ \mathbf{or} \ formula \mid \\
& & formula \ \mathbf{implies} \ formula \mid \\
& & \mathbf{not} formula \mid \\
& & RelativePath = RelativePath \mid
\end{array}
$$

Figure 5. Rule language abstract syntax

The input to our repair action generator will be xlinkit formulae. Figure 5 shows the abstract syntax of our rule language. The meaning of the quantifiers is the same as that in standard first order logic. For a full definition see [16].

In our repair system, there are three types of modifications that can be made to a document: *Add*, *Delete* and *Change*. While change could be represented by deletion followed by addition, we retain it as an atomic action as it simplifies comprehension by the user – changing a value is an intuitive concept and the user only has to select one action.

$$
\begin{array}{rcl}
Action & ::= & Add\ Accessor \mid \\
& & Delete\ Accessor \mid \\
& & Change\ Accessor\ RelativePath \\
Accessor & ::= & Direct\ LocatorNumber \mid \\
& & Lookup\ XPath \\
XPath & ::= & AbsolutePath \mid RelativePath
\end{array}
$$

Figure 6. Repair actions

Each of the three actions are parametrized by the target of their modification. We refer to this as the action's *Accessor*. Accessors come in two forms, *Direct Accessors* and *Lookup Accessors*. Direct Accessors point to information relative to a hyperlink locator generated by xlinkit, while Lookup Accessors require the user of the repair manager to select an item interactively. The reason for this distinction will become clear from the definition of the semantics.

Figure 6 defines the abstract syntax of repair actions. We define an XPath to be either an absolute path, which starts from the root of a document, or a relative path, which starts with a variable reference. Due to space constraints, it is not possible to present a more detailed grammar of the XPath language in this paper. Change actions have an additional path parameter, which must be a path relative to the accessor: the meaning is that we wish to change a property, identified by the relative path, of some element, identified by the accessor.

We will in the following make use of the function $getvar(RelativePath)$, which retrieves the variable name from a relative path. Because the function can be implemented using simple string manipulation and formalisation would require a complete grammar for XPath, we will not go beyond this informal definition.

Our semantics maps formulae to a set of sets of repair actions. It maps to multiple sets because formulae that include the logical connectives *and*, *or* or *implies* can fail in multiple ways. Each set of actions will be an alternative, and the correct one must be chosen after a consistency check, when the hyperlink locator information becomes available. Our semantics will make use of the cartesian operator $\wp(\wp(Action)) \times \wp(\wp(Action))$ in order to prepend a set of sets of actions to each set in a set of actions. For example, abstractly, $\{\{a, b\}\} \times \{\{c\}, \{d\}\}$ becomes $\{\{a, b, c\}, \{a, b, d\}\}$. We will also make use of the function $locator(variable)$, which returns the hyperlink locator number xlinkit has assigned to a particular variable. The function is implemented in xlinkit and is not formally defined here.

We can now discuss the denotational action generation semantics. The discussion will provide an overview of the semantics, and will be followed by several examples.

Figure 7 defines the function $\mathcal{D}_i$, the main function for mapping a formula to repair actions. It takes as a subscript parameter a set of variable bindings, $\sigma : \mathbf{var} \times XPath$, which is assumed to be empty when the function is first invoked, and a set of variables that can be referenced by locators, τ, which is also assumed to be empty in the beginning. The function subscript is short for "inconsistent", that is the function is defined to work under the assumption that the formula has been violated. Its task is to pick out and act on the violating elements.

For example, suppose a formula of the form $\forall \mathbf{var} \in XPath\ (\phi)$ has been violated. Then for some assignment of the variable, ϕ evaluates to *false*. There are two ways this inconsistency can be removed: delete the element currently assigned to the variable, or remove the inconsistency recursively by fixing ϕ. It is easy to show that there are no other ways this inconsistency can be removed: adding a new element cannot make the existing element more consistent, and neither can changing a property of the element, since the quantifier does not check for properties – although the subformula may, in which case the recursive definition will generate the appropriate repair action.

The logical connectives are defined depending on the evaluation of the subformulae necessary to make them inconsistent. If both subformulae need to evaluate to a certain result to make the result inconsistent, the sets of actions are joined using a cartesian product. Choosing any of the actions in the combined set will then be sufficient for removing the inconsistency. For example, if $a \wedge b = false$ then making either a or b true will make the overall result true.

Some connectives can fail in multiple ways. The different modes of failure are combined using the union operator and become alternatives. It is only after a check has been executed that information on how a formula has failed becomes available and the correct set of actions can be selected. For example, the definition of *and* takes three possibilities into account: the first subformula fails, the second subformula fails, or both subformulae fail and their results are joined using the cartesian product. By contrast, if *or* fails we know that both subformulae must be false and simply join their actions.

Some logical connectives invert the meaning of the repair semantics. For example, *not* requires us to look for assignments that make its subformula *consistent* for an overall inconsistent result. Similarly, in order to find a failure for *implies*, we must treat its first subformula as consistent and the second as inconsistent. The function $\mathcal{D}_c$, also defined in Figure 7, provides this inverted mapping. The goal in this function is to assume that the outcome of a formula will be *true* and to construct repair actions that prevent this.

In many cases, it is not possible to point to a definite element for modification. The simplest such case is a formula involving existential quantification, for example $\exists\ \mathbf{var} \in XPath(\phi)$. If this formula is violated, then no element exists for which the subformula ϕ evaluates to true. There are two ways of acting: add a new element that matches the XPath and for which ϕ is true, or select one of the elements on the path and make ϕ true for it. It will, however, not be possible to point to definite elements in the repair actions for ϕ anymore we cannot choose automatically for which variable assignment ϕ should be corrected. An example of this will be given in the next subsection.

As soon as an existential quantifier is encountered while in inconsistent mode, or a universal quantifier in consistent mode – in this case the subformula was consistent for all assignments and the choice which element to change can again not be made automatically –, the semantics switches to the "no information" functions in Figure 8. In these functions, no direct accessors are generated and all actions require selection.

Predicates are treated differently from logical connectives and quantifiers. We have included only equality as an example in this paper. An equality comparison in xlinkit compares elements relative to variables. In order to fix a violated equality comparison, we generate change actions. Take for example the formula $\$x/@name = \$y/@name$, which compares the name attributes of nodes x and y, which must have been bound by some outer formula. It is not possible to decide in general which side of the equality has caused the inconsistency, so we always generate two change actions. The $change$ function in Figure 7 shows how this is done: if the variable that an argument is relative to is present in τ, and can thus be accessed through a locator, we generate a direct change action. Otherwise, we generate a lookup action and the user has to select an item to change.

$$
\begin{array}{rcl}
change(RelativePath, \sigma, \tau) & = & Change\ (Direct\ locator(getvar(RelativePath)))\ RelativePath, \\
 & & if\ getvar(RelativePath) \in \tau \\
 & = & Change\ (Lookup\ \sigma(getvar(RelativePath)))\ otherwise
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{D}_i & : & formula \rightarrow \wp(\wp(Action)) \\
\mathcal{D}_i[\![\forall \mathbf{var} \in XPath\ (formula)]\!]_{\sigma,\tau} & = & \{\{Delete\ Direct\ locator(\mathbf{var})\}\} \times \\
 & & \mathcal{D}_i[\![formula]\!]_{(\{(var,XPath)\}\cup\sigma),(\{var\}\cup\tau)} \\
\mathcal{D}_i[\![\exists \mathbf{var} \in XPath\ (formula)]\!]_{\sigma,\tau} & = & \{\{Add\ Lookup\ XPath\}\} \times \mathcal{N}_i[\![formula]\!]_{\{(var,XPath)\}\cup\sigma,\tau} \\
\mathcal{D}_i[\![formula_1\ \mathbf{and}\ formula_2]\!]_{\sigma,\tau} & = & \mathcal{D}_i[\![formula_1]\!]_{\sigma,\tau} \cup \mathcal{D}_i[\![formula_2]\!]_{\sigma,\tau} \cup (\mathcal{D}_i[\![formula_1]\!]_{\sigma,\tau} \times \mathcal{D}_i[\![formula_2]\!]_{\sigma,\tau}) \\
\mathcal{D}_i[\![formula_1\ \mathbf{or}\ formula_2]\!]_{\sigma,\tau} & = & \mathcal{D}_i[\![formula_1]\!]_{\sigma,\tau} \times \mathcal{D}_i[\![formula_2]\!]_{\sigma,\tau} \\
\mathcal{D}_i[\![formula_1\ \mathbf{implies}\ formula_2]\!]_{\sigma,\tau} & = & \mathcal{D}_c[\![formula_1]\!]_{\sigma,\tau} \times \mathcal{D}_i[\![formula_2]\!]_{\sigma,\tau} \\
\mathcal{D}_i[\![\mathbf{not}\ formula]\!]_{\sigma,\tau} & = & \mathcal{D}_c[\![formula]\!]_{\sigma,\tau} \\
\mathcal{D}_i[\![RelativePath_1 = RelativePath_2]\!]_{\sigma,\tau} & = & change(RelativePath_1, \sigma, \tau) \times change(RelativePath_2, \sigma, \tau)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{D}_c & : & formula \rightarrow \wp(\wp(Action)) \\
\mathcal{D}_c[\![\forall \mathbf{var} \in XPath\ (formula)]\!]_{\sigma,\tau} & = & \{\{Add\ Lookup\ XPath\}\} \times \mathcal{N}_i[\![formula]\!]_{\{(var,XPath)\}\cup\sigma,\tau} \\
\mathcal{D}_c[\![\exists \mathbf{var} \in XPath\ (formula)]\!]_{\sigma,\tau} & = & \{\{Delete\ Direct\ locator(\mathbf{var})\}\} \times \\
 & & \mathcal{D}_c[\![formula]\!]_{(\{(var,XPath)\}\cup\sigma),(var\cup\tau)} \\
\mathcal{D}_c[\![formula_1\ \mathbf{and}\ formula_2]\!]_{\sigma,\tau} & = & \mathcal{D}_c[\![formula_1]\!]_{\sigma,\tau} \times \mathcal{D}_c[\![formula_2]\!]_{\sigma,\tau} \\
\mathcal{D}_c[\![formula_1\ \mathbf{or}\ formula_2]\!]_{\sigma,\tau} & = & \mathcal{D}_c[\![formula_1]\!]_{\sigma,\tau} \cup \mathcal{D}_c[\![formula_2]\!]_{\sigma,\tau} \cup (\mathcal{D}_c[\![formula_1]\!]_{\sigma,\tau} \times \mathcal{D}_c[\![formula_2]\!]_{\sigma,\tau}) \\
\mathcal{D}_c[\![formula_1\ \mathbf{implies}\ formula_2]\!]_{\sigma,\tau} & = & \mathcal{D}_i[\![formula_1]\!]_{\sigma,\tau} \cup \mathcal{D}_c[\![formula_2]\!]_{\sigma,\tau} \\
\mathcal{D}_c[\![\mathbf{not}\ formula]\!]_{\sigma,\tau} & = & \mathcal{D}_i[\![formula]\!]_{\sigma,\tau} \\
\mathcal{D}_c[\![RelativePath_1 = RelativePath_2]\!]_{\sigma,\tau} & = & change(RelativePath_1, \sigma, \tau) \times change(RelativePath_2, \sigma, \tau)
\end{array}
$$

Figure 7. Repair actions - Direct accessor cases

4.1 Examples

In order to clarify the semantics and show how it can work in practice, we now present some simple examples that cover the different action generation functions.

The formula $\forall\ x\ \in$ "$/a$"$(\exists\ y\ \in$ "$/b$"$(\$x/@name = \$y/@name))$ would be processed as follows: function $\mathcal{D}_i$ generates a delete action for the inconsistent x, $\{\{Delete\ 1\}\}$, where 1 is the locator number returned by xlinkit. It then calls $\mathcal{D}_i$ recursively to process the existential quantifier. We know that for some x, we cannot find any y for which the equality comparison is true, so we add the action $\{\{Add\ Lookup$ "$/b$"$\}\}$ and call $\mathcal{N}_i$ on the equals predicate. Evaluation of the $change$ function returns two actions, $\{\{Change\ Direct\ 1$ "$@name$", $Change\ Lookup$ "$/b$" "$@name$"$\}\}$, which can be read as "change the name of locator 1, or change the name of some b". We can modify the name of a directly, because it is present as a locator, but have to let the user choose which b is to be modified. The actions are all combined using the times operator to form a set of four alternative actions for the user to choose.

If we modify the formula slightly to read $\forall\ x\ \in$ "$/a$"$(not(\exists\ y\ \in$ "$/b$"$(\$x/@name = \$y/@name)))$ the evaluation proceeds differently. The same action is generated for the universal quantifier, but $\mathcal{D}_c$ is called on the existential quantifier instead of $\mathcal{D}_i$. In this case there is no uncertainty: if a violation occurs, it is because an element that should not be there, is present anyway. Thus $\mathcal{D}_c$ creates the action $\{\{Delete\ 2\}\}$ where 2 is the locator number returned by xlinkit. The change function also returns the slightly different set of actions, $\{\{Change\ Direct\ 1$ "$@name$", $Change\ Direct\ 2$ "$@name$"$\}\}$. Since we know the element that should not be there is referenced by a locator we do not have to offer a choice to the user.

4.2 Additional Considerations

The most important question to ask of our semantics is whether its results are correct and complete. The size restrictions for this paper do not permit us to answer this question, but we note that both properties can be shown to hold by structural induction over the formulae. A simple example was given earlier in this section in the explanation of action generation for the universal quantifier.

Clearly, additional actions could be generated for equal-

$$
\begin{array}{rcl}
\mathcal{N}_i & : & formula \rightarrow \wp(\wp(Action)) \\
\mathcal{N}_i[\![\forall \textbf{var} \in XPath\ (formula)]\!]_{\sigma,\tau} & = & \{\{Delete\ Lookup\ XPath\}\} \times \mathcal{N}_i[\![formula]\!]_{\{(var,XPath)\}\cup\sigma,\tau} \\
\mathcal{N}_i[\![\exists \textbf{var} \in XPath\ (formula)]\!]_{\sigma,\tau} & = & \{\{Add\ Lookup\ XPath\}\} \times \mathcal{N}_i[\![formula]\!]_{\{(var,XPath)\}\cup\sigma,\tau} \\
\mathcal{N}_i[\![formula_1\ \textbf{and}\ formula_2]\!]_{\sigma,\tau} & = & \mathcal{N}_i[\![formula_1]\!]_{\sigma,\tau} \times \mathcal{N}_i[\![formula_2]\!]_{\sigma,\tau} \\
\mathcal{N}_i[\![formula_1\ \textbf{or}\ formula_2]\!]_{\sigma,\tau} & = & \mathcal{N}_i[\![formula_1]\!]_{\sigma,\tau} \times \mathcal{N}_i[\![formula_2]\!]_{\sigma,\tau} \\
\mathcal{N}_i[\![formula_1\ \textbf{implies}\ formula_2]\!]_{\sigma,\tau} & = & \mathcal{N}_c[\![formula_1]\!]_{\sigma,\tau} \times \mathcal{N}_i[\![formula_2]\!]_{\sigma,\tau} \\
\mathcal{N}_i[\![\textbf{not}\ formula]\!]_{\sigma,\tau} & = & \mathcal{N}_c[\![formula]\!]_{\sigma,\tau} \\
\mathcal{N}_i[\![RelativePath_1 = RelativePath_2]\!]_{\sigma,\tau} & = & change(RelativePath_1,\sigma,\tau) \times change(RelativePath_2,\sigma,\tau) \\
 & & \\
\mathcal{N}_c & : & formula \rightarrow \wp(\wp(Action)) \\
\mathcal{N}_c[\![\forall \textbf{var} \in XPath\ (formula)]\!]_{\sigma,\tau} & = & \{\{Add\ Lookup\ XPath\}\} \times \mathcal{N}_c[\![formula]\!]_{\{(var,XPath)\}\cup\sigma,\tau} \\
\mathcal{N}_c[\![\exists \textbf{var} \in XPath\ (formula)]\!]_{\sigma,\tau} & = & \{\{Delete\ Lookup\ XPath\}\} \times \mathcal{N}_c[\![formula]\!]_{\{(var,XPath)\}\cup\sigma,\tau} \\
\mathcal{N}_c[\![formula_1\ \textbf{and}\ formula_2]\!]_{\sigma,\tau} & = & \mathcal{N}_c[\![formula_1]\!]_{\sigma,\tau} \times \mathcal{N}_c[\![formula_2]\!]_{\sigma,\tau} \\
\mathcal{N}_c[\![formula_1\ \textbf{or}\ formula_2]\!]_{\sigma,\tau} & = & \mathcal{N}_c[\![formula_1]\!]_{\sigma,\tau} \times \mathcal{N}_c[\![formula_2]\!]_{\sigma,\tau} \\
\mathcal{N}_c[\![formula_1\ \textbf{implies}\ formula_2]\!]_{\sigma,\tau} & = & \mathcal{N}_i[\![formula_1]\!]_{\sigma,\tau} \times \mathcal{N}_c[\![formula_2]\!]_{\sigma,\tau} \\
\mathcal{N}_c[\![\textbf{not}\ formula]\!]_{\sigma,\tau} & = & \mathcal{N}_i[\![formula]\!]_{\sigma,\tau} \\
\mathcal{N}_c[\![RelativePath_1 = RelativePath_2]\!]_{\sigma,\tau} & = & change(RelativePath_1,\sigma,\tau) \times change(RelativePath_2,\sigma,\tau)
\end{array}
$$

Figure 8. Repair actions - no information cases

ity comparisons: we could delete the properties referenced by the predicate, e.g. delete the $name$ attributes referenced in the example. We have deliberately chosen not to include these actions in the semantics because we found that in practice these properties are in almost all cases mandatory. Deleting them would thus simply lead to a violation of document structure.

It is possible to provide some support for default values for node changes in the case of equality: if $a = b$ is violated, then the only way to remove the inconsistency is to set a equal to b and vice-versa. We can thus offer a default value of b for changes to a and the other way round. This approach will not work once the equality predicate occurs in a negative formula: if $not(a = b)$ is violated then we cannot determine a default value – the only criterion is that after the change the two values must not be equal.

5 Implementation

We have implemented a framework for repair action generation and execution and tested it in several application domains. This section presents the implementation of the repair administrator and repair manager tool and its application to UML models.

The first step in the architecture we outlined in Section 3 was the generation of repair actions from constraints. Figure 9 shows a snapshot of our repair administrator tool. The administrator has selected the example rule from Figure 1, and the tool has generated the repair actions. For each action, the administrator can enter a natural language description that is comprehensible to the user, or leave the default description generated by the action generator.

The administrator has decided that deleting the ends of the association should not be an option available to the user, but they should instead rename either of the role names or delete the whole association. The delete actions for the association ends have thus been disabled and have turned red. In the *diagnostic message* section, the administrator can enter a diagnostic message that describes the inconsistency as a whole. When everything has been entered, the set of actions and messages is saved in a report fragment in XML format for use by the repair execution tool.

Figure 10 shows how default value generation is handled in the case of add-actions. The rule displayed at the top says that for every class stereotyped "EJBImplementation" in the UML model, there must be a Session Bean or Entity Bean declaration in the Enterprise JavaBeans [12] deployment descriptor. If such an entry does not exists, one option is to add one. In this case, we have selected the "From File" option and selected an existing entry in the XML browser (not shown). All text elements of the tree fragment have been parametrized by the tool and will be offered as text fields for the user to fill in when they execute the repair manager. If no sample file is available, the administrator can automatically construct the default value to add from an XML Schema [6] instead.

Figure 11 shows a snapshot of the repair manager tool. We have fed into the tool a linkbase generated by checking

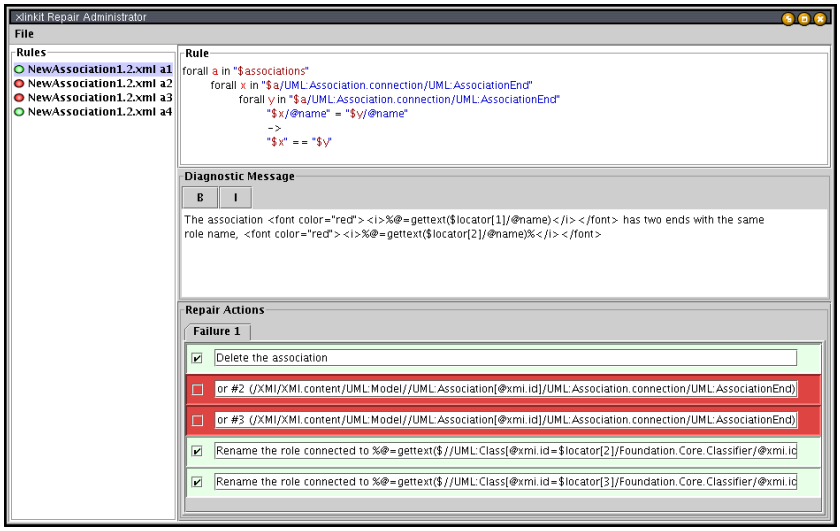

Figure 9. Repair Administrator Tool

an inconsistent UML model. On the left-hand side, the user can see which rules have been violated. Rules are flagged red if no repair actions have yet been chosen for them and turn green as soon as one or more actions are chosen. Notice that only the two types of actions permitted by the administrator are present, and that the role and class names have been substituted from the UML model. Changing either name will fix the inconsistency, so we have entered a new value for the role connected to the "SalesAssistant" class. We can now click the "Execute Actions" button and a suitable back-end can modify the files, as explained in Section 3.

The repair execution tool can also easily check for contradictory repair actions. The way this is done depends on the action: Two change actions cannot modify the same property of the same element to hold two different values; and no action of any type must access a path that is logically beneath the path of a delete action. If either of these scenarios occurs, the user must deselect one of the actions.

6 Advanced Repair

Using our repair action generator as an infrastructure tool we can build higher level repair frameworks.

An advanced repair framework in our sense is one that restricts the set of repair actions automatically in some way, so as to minimize or even eliminate user interaction. The simplest type is a *static precedence relationship* where certain types of artifacts take precedence over others. For example, if a UML model is inconsistent with a Java class we may dictate that the model should override the class. This can be implemented by removing those repair actions that refer to elements in the UML model and only allowing those that modify Java classes.

A more sophisticated approach would be to construct *dynamic precedence relationships*. In this approach, the override relationship would change over time. For example, during design it may be appropriate for the UML model to override Java classes but during an implementation phase, when source code evolves more quickly and models become outdated, we may wish to invert this relationship. We could use an off-the-shelf workflow system or a simple event notification mechanism such as Emu [8] to implement such a system.

7 Evaluation

Since the repair action semantics can be proven to generate correct and complete repair actions, this evaluation section concentrates mainly on practical issues we encountered during testing.

We have applied our repair generator the the full set of rules in the UML Foundation/Core package. On the whole, the action generator performed very well and it is possible for somebody with knowledge of the documents to intuitively grasp the meaning and consequence of actions. We have however identified several problems that must be addressed before this framework can become a practical proposition.

The biggest problem is the interaction of repair actions with the grammar of documents and with actions generated by other constraints. These cases are currently not addressed at all and are investigated further in the section on future work.

It is also not clear at the moment whether the localized view of one single inconsistency at a time may cause prob-

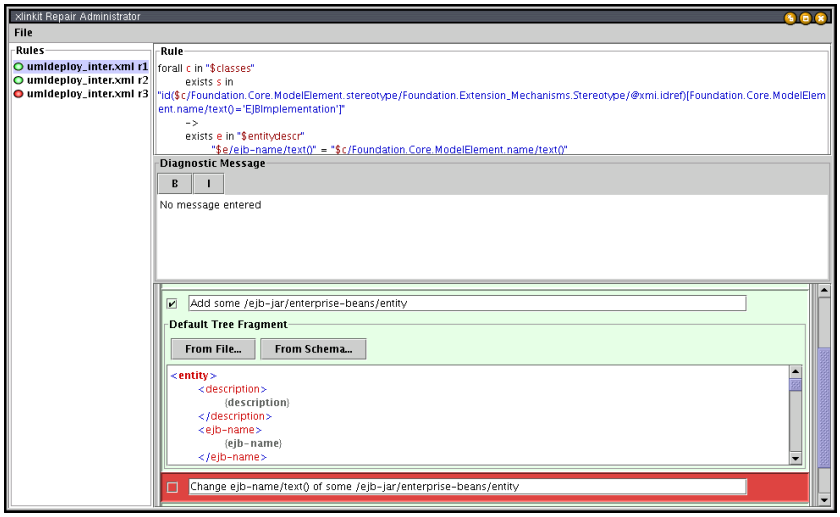

Figure 10. Default Tree Fragment Generation

lems with multiple interconnected inconsistencies. For example, a single class may violate several constraints, but deleting it could automatically fix all of them. One possible way of approaching this problem would be to provide a more "global" overview of problems where a single element causes multiple inconsistencies. This could help to reduce user disorientation. Another possibility is the automatic selection of actions for elements that contribute to multiple inconsistencies as soon as the user chooses to fix the element once.

Finally, another common problem we have identified is the apparent complexity of the paths the action generator prints out for the administrator, for example the paths in the second and third action in Figure 9. This complexity is mainly due to the complexity of the XMI notation. In the case of our evaluation, the repair administrator also wrote the constraints, so it was easy to identify these paths as pointing to association ends. If the administrator was a pure UML domain expert, we would have to provide additional support such as intelligent abbreviation or translation.

8 Related Work

The problem of repairing integrity violations has received attention in the areas of software development environments and databases. In this section we give a brief account of work most closely related to ours.

The software engineering community has produced a substantial body of work on Software Development Environments. The goal of systems like ESF [21], GoodStep [5] or IPSEN [15], to name but a few, is to integrate different tools and to ensure consistency across heterogeneous artifacts. These environments include change propagation mechanisms that can restore consistency in a dependency graph when changes are made. GoodStep includes a scripting language for adding domain-specific repair mechanisms, for example to add a variable declaration at the right level of scope when a variable reference is invalid. To the best of our knowledge, none of them include mechanisms for automatically transforming a declarative constraint language into repair actions without user intervention or hand-coding. In addition, these environments typically require a central repository and cannot support the loose style of fully distributed development.

Active integrity maintenance through triggering in a software development environment is discussed in [22]. The paper gives examples of an algorithm that can transform a constraint into condition-action repair actions that are executed when a constraint is violated. The approach is powerful in that it works automatically from a declarative specification. It also relies on a fast triggering mechanism rather than a complete consistency check, which is expensive, but it can only handle simple boolean constraints and no support for distribution is indicated.

The process of making changes to transactions or databases to recreate a state of integrity is referred to in the literature as *integrity constraint maintenance*. A survey of the area can be found in [13]. The common goal of work in this area is to prevent expensive transaction roll-back by making amendments that prevent constraint violation before committing. Approaches typically differ by the expressiveness of the constraints they permit – though none of them support unrestricted first order logic formulae, – the underlying data model they are trying to support, the level of user interaction, and the guarantees that can be made about repairs. In the following, we present some of the most relevant work in the area.

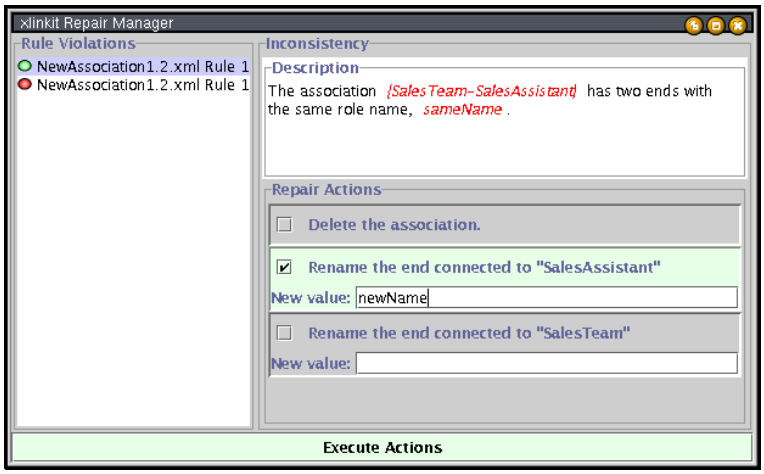

Figure 11. Repair Manager Tool

The rule generator described in [2] takes restricted constraints over relational databases of the form $\forall\exists$ and generates set of repair actions. The actions and constraints are ordered so as to generate a minimal set of actions that are self-contradictory and cannot be executed automatically. Because the approach builds on restricted formulae, constraint checking can be performed quickly through "violation queries". User interaction is treated using a low-level text mechanism but no tool support is discussed.

A similar class of constraints is analyzed in [9]. The paper includes an additional section on repair strategies, which discusses mechanisms for reducing the set of repair actions made available to the user. These include optimisations on the number of changes necessary to the database or to the transaction that triggered the repairs. These strategies differ from the higher-level repairs proposed here as they are analytical and do not build on domain knowledge.

Repair actions have also been a topic of interest in deductive databases. In [14], a system is described that generates actions from closed, range-restricted first order logic formulae. The repair algorithm can automatically find repairs for violated existential formulae without user intervention. For this, the algorithm relies on the rules of the database and the closed-world assumption. These mechanisms are not available in the case of distributed, semi-structured documents, hence the need for a lookup mechanism in our generator.

Compensating transactions [11] are similar in approach but make use of transaction operations as well as database data to undo dirty reads caused by lack of isolation between multiple transactions. This is not possible in our scenario as we treat the data "as-is", and have no information on modification by the user.

9 Future Work

There are many theoretical and practical questions that we cannot address in this paper.

The first concerns the interaction of repair actions with the syntax of documents. Repair actions that make a document consistent with respect to some xlinkit constraint, but at the same time violate the document's grammar should not be permitted – the grammar of a document is a lower-level mechanism and takes precedence. It is not possible to statically compare our constraints against a DTD or Schema to predict whether this behaviour must necessarily occur [7], but it should be possible to incrementally revalidate a document after executing each action and to retract the action if it violates the grammar.

A more complex problem arises when the repair action for one constraint forces the user to violate another. Since the equivalence problem for first order logic formulae is undecidable, it is not possible to predict in advance whether two formulae are contradictory. An iterative process that is capable of detecting repair cycles, similar to that in [9], could be used to address this problem.

We are also interested in comprehensive architectures for consistency management in various application domains. Tighter integration of consistency management mechanisms invariably raises domain specific problems. We have already applied xlinkit to checking the integrity of financial derivative trade data [4] and will extend our work in this area to include repair.

10 Conclusion

We have presented in this paper a novel framework for repairing inconsistent distributed documents. The framework

is supported by a strong semantics that can translate first order logic constraints into repair actions that are correct and complete.

By combining the powerful diagnostic capabilities of xlinkit with this framework we have laid the foundations for a comprehensive architecture for managing the consistency of distributed and heterogeneous software engineering specifications.

We have shown the application of the framework to UML models encoded in XMI, and will continue to investigate its usefulness in other application domains.

The xlinkit check engine is available with full source code for academic research, and has already been taken up by several research groups. We welcome further request to use the check engine, and will make our repair action tools available as soon as the implementation stabilises. For more information please visit `http://www.xlinkit.com`.

Acknowledgements

We gratefully acknowledge financial support from Zuhlke Engineering for Christian Nentwich.

References

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

[2] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Automatic Generation of Production Rules for Integrity Maintenance. *ACM Transactions on Database Systems*, 19(3):367–422, September 1994.

[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] D. Dui, W. Emmerich, C. Nentwich, and B. Thal. Consistency Checking of Financial Derivatives Transactions. In *Proceedings of Net.ObjectDays 2002*, October 2002. To Appear.

[5] 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.

[6] 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.

[7] W. Fan and L. Libkin. On XML Integrity Constraints in the Presence of DTDs. *Journal of the ACM*, 49(3):368–406, 2002.

[8] S. Fickas, T. Beauchamp, and A. R. Mamy. Monitoring Requirements: A Case Study. In *Proceedings of the 17th IEEE International Conference Automated Software Engineering*, 2002. To appear.

[9] M. Gertz. An Extensible Framework for Repairing Constraint Violations. In *Workshop on Foundations of Models and Languages for Data and Objects*, pages 41–56, 1996.

[10] Y. Y. Goland, J. Whitehead, A. Faizi, S. Carter, and R. Jensen. Extensions for Distributed Authoring on the World Wide Web – WebDAV. Internet Draft (Work in Progress) http://www.ietf.org/internet-drafts/draft-ietf-webdav-protocol-10.txt, IETF, Nov. 1998.

[11] H. F. Korth, E. Levy, and A. Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In D. McLeod, R. Sacks-Davis, and H. J. Schek, editors, *Proceedings of the 16th International Conference on Very Large Data Bases, Brisbane, Queensland, Australia*, pages 95–106. Morgan Kaufmann, August 1990.

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

[13] E. Mayol and E. Teniente. A Survey of Current Methods for Integrity Constraint Maintenance and View Updating. In *Advances in Conceptual Modeling: ER '99 Workshops on Evolution and Change in Data Management, Reverse Engineering in Information Systems, and the World Wide Web and Conceptual Modeling, Paris, France, November 15-18, 1999*, volume 1727 of *Lecture Notes in Computer Science*, pages 62–73. Springer, 1999.

[14] G. Moerkotte and P. C. Lockemann. Reactive Consistency Control in Deductive Databases. *ACM Transactions on Database Systems*, 16(4):670–702, December 1991.

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

[16] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: a Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2(2):151–185, May 2002.

[17] 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)*, pages 115–124, Coronado Island, CA, Nov. 2001. IEEE Computer Science Press.

[18] C. Nentwich, W. Emmerich, A. Finkelstein, and E. Ellmer. Flexible Consistency Checking. Research note, University College London, Dept. of Computer Science, 2001. Submitted for Publication.

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

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

[21] 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.

[22] S. D. Urban, A. P. Karadimce, and R. B. Nannapaneni. The Implementation and Evaluation of Integrity Maintenance Rules in an Object-Oriented Database. In *Proceedings of the Eighth International Conference on Data Engineering*, pages 565–572, Los Alamitos, 1992. IEEE Computer Society Press.