Konsistenzanalyse in xlinkit

Björn Schlak
244 149

Betreut von Dipl.-Inform. Anne-Thérèse Körtgen

Zusammenfassung

Eines der Kernprobleme bei der Entwicklung großer Softwareprojekte ist die Konsistenzanalyse in den verschiedenen Phasen der Entwicklung und zwischen Dokumenten unterschiedlichen Typs. Hier bietet das xlinkit-Rahmenwerk verschiedene Möglichkeiten, eine sinnvolle, automatisierte Konsistenzanalyse durchzuführen. Dabei greift xlinkit auf XML-Technologien zurück und führt eine regelbasierte Erkennung von Inkonsistenzen durch. Darüber hinaus können auch Reparaturaktionen spezifiziert werden, die teilweise automatisch durchgeführt und teilweise vom Benutzer ausgewählt werden können.

Inhaltsverzeichnis

1 Einleitung

Das Seminar, in dessen Rahmen diese Arbeit entstanden ist, behandelt die Erstellung und Verarbeitung von Modellen in verschiedenen Domänen und Phasen der Entwicklung. Ein großes Softwareprojekt besteht im Allgemeinen aus vielen verschiedenen Phasen, in denen eine Reihe unterschiedlicher Dokumente in verschiedenen Formaten entstehen. So wird bei der Modellierung häufig UML verwendet, um die Klassenstruktur und deren Eigenschaften, die später implementiert werden sollen, graphisch darzustellen. In der Implementierungsphase wird dann Quellcode in Form von normalen Text-Dateien erzeugt. Während und nach der Implementierung ist es von großem Interesse zu wissen, ob die erstellte Funktionalität mit der gewünschten Funktionalität aus der Modellierungsphase übereinstimmt. Es soll die Konsistenz zwischen diesen Phasen, bzw. diesen Dokumenten gewährleistet sein. Die Konsistenzanalyse ist ein wichtiger Aspekt in der Entwicklung großer Softwareprojekte. Im Gegensatz zu vielen anderen Aspekten, die automatisiert ablaufen, ist dies bei der Konsistenzanalyse häufig nicht der Fall. Die meisten Werkzeuge können eine solche Analyse nicht leisten, daher werden von den Entwicklern häufig wenig effiziente Methoden verwendet.
Das Problem der Automatisierung ist darauf zurückzuführen, dass in den unterschiedlichen Phasen auch sehr unterschiedliche Dokumente entstehen. So werden in der Modellierungsphase beispielsweise UML-Dokumente generiert, die eine graphische Baumstruktur haben. Im Gegensatz dazu entstehen in der Implementierungsphase Quellcode-Dateien, die eine Text- und Absatzstruktur aufweisen. Anders als in der vorangegangenen Arbeit, die sich mit der Integration verschiedener Teilmodelle in ein Referenzmodell beschäftigt, geht es hier darum, die Dokumente der unterschiedlichen Phasen zu vergleichen und gegebenenfalls im jeweiligen Modell zu verändern. Um diese verschiedenen Dokumente automatisiert zu vergleichen ist ein Aufwand nötig, der durch die Vielfalt der Dokumente und die individuelle Verwendung noch gesteigert wird. Ein Ansatz zur Lösung dieses Problems wird im weiteren Verlauf dieser Ausarbeitung präsentiert.
Im nächsten Kapitel wird xlinkit vorgestellt. Dies ist ein Rahmenwerk, welches eine automatische Konsistenzanalyse vieler verschiedener Dokumenttypen ermöglicht. In Kapitel 3 werden zwei Erweiterungen von xlinkit vorgestellt. Zum einen werden dort Visualisierungen der Ausgabe präsentiert, zum anderen Reparaturmechanismen für inkonsistente Dokumente. In Kapitel 4 wird das Konzept von xlinkit mit alternativen Ansätzen verglichen und seine Vor- und Nachteile herausgestellt. Kapitel 5 bietet einen Überblick über den Stand der Technik und einen Ausblick darauf, was die Zielsetzung für die Zukunft betrifft. Im abschließenden Kapitel 6 wird das Thema in Form einer Zusammenfassung kompakt dargestellt.

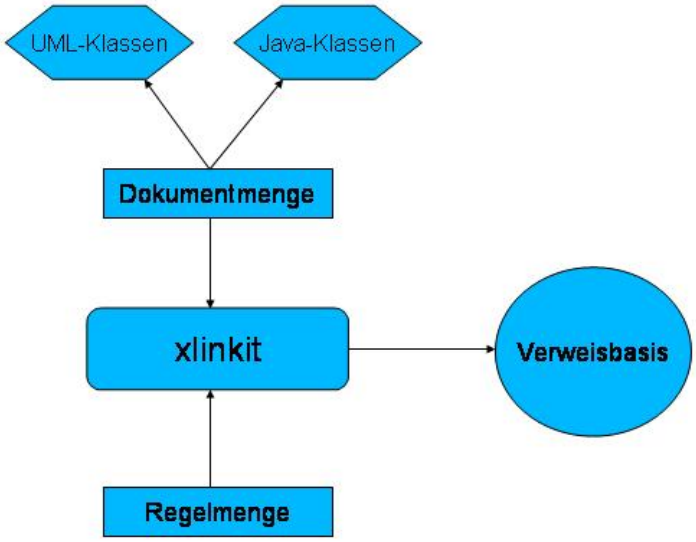

Abbildung 1: Übersicht über die Funktionsweise von xlinkit

2 xlinkit

In diesem Kapitel wird das xlinkit Rahmenwerk vorgestellt. Es unterstützt die automatisierte Konsistenzanalyse von Dokumenten mit Hilfe von XML-Technologien. Abbildung 1 skizziert die Funktionsweise von xlinkit. Xlinkit basiert dabei auf Regeln, die Konsistenzbedingungen ausdrücken. Diese Regeln werden dann auf eine ausgewählte Menge von Dokumenten angewandt, um Inkonsistenzen zwischen den Dokumenten zu ermitteln. In Abbildung 1 sind dies beispielsweise UML- und Java-Klassen. Um Konsistenzen und Inkonsistenzen zu speichern, werden mit Hilfe von XML-Technologien Verweise erstellt und gespeichert. In den Abschnitten 2.2 bis 2.4 werden die Regelsprache, die Dokument- und die Regelmenge, sowie die Verweisbasis vorgestellt. Bevor die Regelsprache von xlinkit erklärt werden kann, müssen aber erst einige Grundlagen in Abschnitt 2.1 erklärt werden.

2.1 Grundlagen

xlinkit basiert auf XML (Extensible Markup Language) und verwandten Technologien wie XPath oder XLink. Hier wird nur auf die wichtigsten Aspekte dieser Technologien eingegangen. Eine genauere Beschreibung dieser Technologien kann beispielsweise in [2] für XPath und in [1] für XLink nachgelesen werden. Ein weiterer Begriff, der hier verwendet wird, ist das XML-DOM (Document Object Model), welches eine Baumstruktur aufweist. XML selbst definiert verschiedene Elemente die mit <element> beginnen und mit </element> geschlossen werden. Ein XML-Dokument besteht somit aus ineinander geschachtelten Elementen. Diese Schachtelung kann als Baumstruktur aufgefasst werden.
XLink ist eine XML-Auszeichnungssprache, die mit zusätzlichen Funktionen für Verweise ausgestattet ist. Ein *Verweis* ist eine Verbindung von einem XML-Element zu einem anderen XML-Element, wobei diese Elemente nicht zwingend im gleichen Dokument vorkommen müssen. Neben Elementen können auch ganze Dokumente auf diese Art verbunden werden. Die Verweise sind dabei bei weitem nicht so großen Einschränkungen unterworfen, wie das bei HTML der Fall ist. In HTML gibt es beispielsweise nur unidirektionale Punkt-zu-Punkt-Verweise und diese sind nur auf Datei-Ebene möglich. Eine Ausnahme ist, wenn eine spezielle Stelle in der Ziel-Datei markiert wurde. Außerdem sind in HTML alle Verweise explizit in den Ressourcen, den HTML-Dateien, eingebunden, was zu Wartungsproblemen und Fehlermeldung 404 des HTTP ”Seite nicht gefunden” führt. Jedes XML-Element kann bei XLink als Verweis agieren und der Benutzer ist damit in der Lage komplexe Verweis-Strukturen und traversierendes Verhalten zu definieren. Die wichtigste Eigenschaft von XLink für den weiteren Verlauf ist, dass Verweise in Verweisbasen zusammengefasst werden können und somit nicht starr in den Ressourcen eingebettet sind. Deshalb können sie unabhängig von den Ressourcen behandelt werden. Diese Verweisbasen können einer Menge von Ressourcen (Dateien) zugeordnet werden.
XPath-Ausdrücke beschreiben Traversierungen durch die Baumstruktur eines XML-Dokuments. Für diese Traversierungen kann aus internen Teilen wie Elementtypen, Attributwerten oder Inhalten bzw. relativen Positionen gewählt werden. Wird XPath mit XLink verbunden kann der XPath-Ausdruck nicht nur ein Dokument, sondern sogar ein spezielles Element aus diesem Dokument adressieren. Somit können Verweise zwischen Elementen aus zwei Dokumenten gezogen werden. Ein Problem ist die Navigation durch die verschiedenen XLinks, da diese nur selten durch Browser unterstützt werden. Daher müssen diese Verweise zu einfachen HTML-Verweisen gefaltet und innerhalb der Ressourcen aufgeteilt werden, in denen die Verweise dann eingefügt werden. Nur so können XLink-Verweise in den meisten Browsern angezeigt werden.

Außerdem kann eine Beschränkung auf XML-Dokumente gemacht werden, da alle Dokumente, die mit Hilfe von xlinkit überprüft werden, vorher in XML-Dokumente überführt werden können.
xlinkit nutzt standardisierte Internet-Technologien für seine Zwecke und unterstützt auch eine Aufteilung der Dokumente auf verschiedene Speicherorte.

2.2 Die Regelsprache

Konsistenzbedingungen zwischen verteilten Dokumenten werden durch Regeln in der Sprache cheXML [3] ausgedrückt. cheXML ist eine mengen-orientierte Sprache, die formal definiert ist, so dass sie von einem Werkzeug eindeutig interpretiert werden kann. Die XML-Dokumente werden wegen ihrer Baumstruktur als Menge von Knoten angesehen. Die Knoten haben Beziehungen untereinander, so dass über die Knoten und deren Eigenschaften mit XPath-Ausdrücken traversiert werden kann. Hierzu ist eine Formalisierung nötig. Zunächst wird eine Notation zur Auswertung von XPath-Ausdrücken und eine Formalisierung des DOM eingeführt. Diese Formalisierung sieht folgendermaßen aus:

- Die Großbuchstaben $A, B, C, ...$ stehen für Knotenmengen des DOM.
- Die Kleinbuchstaben $a, b, c, ...$ stehen für einzelne Knoten des DOM.
- Die Funktion $S[\![p]\!]_x$ liefert für die Auswertung eines XPath-Ausdrucks p vom Knoten x her und gibt als Ergebnis eine Menge von Knoten aus, die den XPath-Ausdruck erfüllen.
- Die Funktion $closure(x, p_1, p_2) =: C$ ermittelt die transitive Hülle, wobei x ein Knoten aus der gesamten Knotenmenge, wird im folgenden mit X bezeichnet, ist und p_1 und p_2 XPath-Ausdrücke sind. In der transitiven Hülle befindet sich anfangs nur x. Dann werden alle Knoten $n \in X$ durchlaufen und die Knoten sukzessive in die Hülle aufgenommen, für die gilt: $S[\![p_1]\!]_n = S[\![p_2]\!]_c$, wobei $c \in C$. Dies wird solange wiederholt, bis in einem Durchlauf durch X kein Knoten mehr der transitiven Hülle hinzugefügt wird.

Die *Regeln* für die Konsistenzüberprüfung werden nun in Form von prädikatenlogischen Formeln dargestellt. Dazu werden Quantoren über einzelnen Knoten benötigt, sowie Mengenrestriktionen für Knotenmengen. Jede Regel drückt eine Beziehung zwischen einer Menge auf der einen Seite und einer oder mehrerer Mengen auf der anderen Seite aus. Die abstrakte Syntax für die Regelsprache ist in Abbildung 2 zu sehen. Ein Beispiel aus [5] für eine solche Regel soll die Verwendung der Formalisierung verdeutlichen. Die Regel soll ausdrücken: ”Für jede

Regel :=	$\forall$ var $\in$ xpath(formel)
Formel :=	$\forall$ var $\in$ xpath(formel) |
	$\exists$ var $\in$ xpath(formel) |
	formel and formel |
	formel or formel |
	formel implies formel |
	not formel |
	xpath = xpath |
	xpath $\neq$ xpath |
	same var var

Abbildung 2: Syntax der Regelsprache

UML-Klasse u muss eine Java-Klasse j mit gleichem Namen existieren." Es gibt demnach ein Dokument, in dem die Knoten UML-Klassen repräsentieren und jede UML-Klasse hat einen Namen. Zusätzlich gibt es ein weiteres Dokument, in dem die Knoten Java-Klassen repräsentieren und jede Java-Klasse hat auch einen Namen. Dann werden beide Dokumente auf diese Regel überprüft.
Dazu werden zuerst zwei Mengen definiert:

- $U = S[\![/UML - Klassen]\!]_/$ für die Menge aller UML-Klassen und
- $J = S[\![/Java - Klassen]\!]_/$ für die Menge aller Java-Klassen

Die zugehörige Formel lautet wie folgt:

$$\forall u \in U(\exists j \in J(S[\![UML - Name]\!]_u = S[\![Java - Name]\!]_j))$$

Der nächste Schritt betrifft die Übersetzung der Regeln in XML. Bis hier handelt es sich nur um eine Formalisierung des DOM, sowie eine Notation für die Auswertung von XPath-Ausdrücken.
Eine Konsistenzregel in XML besteht aus drei Teilen: (1) einer Beschreibung der Regel in natürlicher Sprache, (2) der Festlegung von Anfangs- und Zielmenge, sowie (3) der Regel selbst. Die Übersetzung der obigen Regel ist in Abbildung 3 zu sehen. Teil (1) steht in den Zeilen 2-5. Hier wird die Regel beschrieben. In den Zeilen 7-13 werden Anfangs- und Zielmenge beschrieben (Teil (2)). Der interessanteste Teil, die Regel, ist in den Zeilen 15-19 zu finden. Die Übersetzung geschieht intuitiv. In Zeile 15 wird der Ausdruck $\forall u \in U$ übersetzt. In Zeile 16 wird dann $\exists j \in J$ übersetzt. Auch hier geschieht die Übersetzung wieder intuitiv. In Zeile 17 schließlich findet der eigentliche Vergleich statt und der Ausdruck ($S[\![UML - Name]\!]_u = S[\![Java - Name]\!]_j$)) wird übersetzt.

```
<ConsistencyRule id="r1">
    <Descriptrion>
        Zu jeder Klasse im UML-Modell muss es
        eine Java-Klasse gleichen Namens geben.
    </Description>

    <SetDefinition id="uml">
        /UML-Klassen
    </SetDefinition>

    <SetDefinition id="java">
        /Java-Klassen
    </SetDefinition>

    <ForAll var="u" in="uml">
        <Exists var="j" in="java">
            <Equal op1="u/name" op2="j/name">
        </Exists>
    </ForAll>
</ConsistencyRule>
```

Abbildung 3: Beispiel einer Konsistenzregel in XML

2.3 Dokument- und Regelmenge

Bisher ist nur die Rede von einzelnen Dokumenten gewesen, deren Konsistenz nur in Bezug auf eine Regel überprüft worden ist. In großen Softwareprojekten gibt es allerdings sehr viele Dokumente und sehr viele Bedingungen, die diese untereinander erfüllen müssen. Daher werden für die Konsistenzüberprüfung zwei Mengen benötigt: die *Dokumentmenge*, in der alle zu prüfenden Dokumente enthalten sind, und die *Regelmenge*, in der alle Konsistenzbedingungen in Form von XML-Regeln enthalten sind. Da im Normalfall viele Entwickler mit einem solchen Projekt beschäftigt sind, entstehen viele kleine Dokument- und Regelmengen. Diese müssen bei Bedarf zusammengelegt werden können. Damit dies einfach ist und nicht wieder ein Teil der Inhalte oder alle Dokumente kopiert werden müssen, dürfen Dokument- und Regelmengen auch andere Dokument- bzw. Regelmengen enthalten. Diese werden erst bei der Anwendung von xlinkit aufgelöst und zu einer einzigen Menge zusammengeführt. Danach werden nur die Regeln auf ein Dokument angewendet, die zu diesem Dokument passen. Dies bedeutet, dass die Beispielregel aus Abbildung 3 nur auf Dokumente angewendet wird, die auch UML- bzw. Java-Klassen enthalten.
Um in großen Projekten eine effiziente Überprüfung zu ermöglichen, können innerhalb der Definition einer Regel die Anfangs- und Zielmengen eingeschränkt

```
1  <SetDefinition id="java">
2      /Java-Klassen[name=$uml/name]
3  </SetDefinition>
```

Abbildung 4: Beispiel einer Zeigervariablen in einer XML-Konsistenzregel

werden. Es ist beispielsweise möglich, die Anfangsmenge auf UML-Dokumente und die Zielmenge auf Java-Dokumente zu beschränken, um eine zu aufwendige Berechnung zu verhindern. Es kann sogar noch weiter gegangen werden, indem Zeigervariablen benutzt werden, die die Mengen durch Vergleich der Attribute von Elementen der Mengen einschränken. Dadurch kann der Suchraum stark eingeschränkt werden. Ein Beispiel dazu ist in Abbildung 4 zu sehen. Dort ist in Zeile 2 eine Zeigervariable zu sehen. Die Menge der Java-Klassen wird hier auf die beschränkt, zu denen es jeweils eine UML-Klasse gleichen Namens gibt. Dies ist beispielsweise dann sinnvoll, wenn nur ein spezieller Teil des UML-Modells überprüft werden soll, d.h. auch die Anfangsmenge auf bestimmte UML-Dokumente eingeschränkt ist.

2.4 Die Verweisbasis

Aus der Dokument- und Regelmenge erstellt xlinkit eine Menge von XLinks. Diese Menge stellt die *Verweisbasis* dar. Es gibt verschiedene Möglichkeiten XLinks in der Verweisbasis zu erstellen. Es ist meist nicht erwünscht Verweise für Konsistenzen zu erzeugen, da diese nicht problematisch sind. Ein Beispiel dafür ist, dass es im Normalfall zu den meisten UML-Klassen auch jeweils eine Java-Klasse gibt. Es ist nicht sinnvoll diese vielen Verweise zu speichern, sondern eher die wenigen zu denen es keine Java-Klassen gibt. Beispielsweise können folgende Verweise in der Verweisbasis gespeichert werden:

- Die Konsistenzbedingung wird erfüllt und es wird ein Verweis von einem Element der Anfangsmenge zum ersten oder zu allen Elementen der Filtermenge erstellt. Die *Filtermenge* ist dabei die Menge der Elemente, die mit dem Element der Anfangsmenge unter der verwendeten Regel konsistent sind.
- Es liegt eine Inkonsistenz vor und es wird ein Verweis zwischen dem Element der Anfangsmenge und der verletzten Regel erstellt.

Insgesamt gibt es mehrere Möglichkeiten, Verweise zu erzeugen und diese lassen sich wie folgt für den Konsistenz- bzw. Inkonsistenzfall konfigurieren:

Verweisart	Konsistenz	Inkonsistenz
Anfangsmenge/Filtermenge	C	I
Anfangsmenge/Regel	CX	IX
Keine Verweise		

Zur Konfiguration werden die jeweiligen Attribute 'imode' und 'cmode' in die Konsistenzregel eingefügt, die die oben beschriebenen Werte enthalten. Als Beispiel siehe Abbildung 5. Hier werden im konsistenten Fall Verweise zwischen Anfangs- und Filtermenge gezogen und im inkonsistenten Fall zwischen Anfangsmenge und der verletzten Regel. Dass xlinkit Verweise erzeugt und die Inkonsistenzen zunächst unverändert lässt, hat einen einfachen Hintergrund. Teilweise sind Inkonsistenzen nicht negativ und bedürfen deshalb auch nicht immer der sofortigen Auflösung. Es kann beispielsweise der Fall auftreten, dass im Verlauf des Entwicklungsprozesses auch Änderungen am Modell und nicht nur an der Implementierung vorgenommen werden. In einem solchen Fall ist eine Inkonsistenz sogar erwünscht, da dann das Modell angepasst werden muss. Die Verweise sollen nur dazu dienen, eine Diagnose zu ermöglichen, um unerwünschte Inkonsistenzen aufspüren zu können.
Eine Ausgabe von xlinkit ist in Abbildung 6 dargestellt. Die Namen der XML-Dokumente für die Dokumentmenge und die Regelmenge werden in den Attributen *docset* und *ruleset* angegeben. Hier handelt es sich um die Dateien *DokumentMenge.xml* und *RegelMenge.xml*. In *DokumentMenge.xml* befinden sich alle Dokumente. In unserem Fall sind das die UML-Dokumente *klasse1.uml*, *klasse2.uml* und das Java-Dokument *klasse1.java*. Im Dokument *RegelMenge.xml* ist die Datei *uml_java_regeln.xml* enthalten, die die Namensregel aus Abbildung 3 enthält. In der Verweisbasis gibt es zwei Einträge: (1) einen konsistenten und (2) einen inkonsistenten. Der konsistente Verweis (Abbildung 6 Zeile 2-8) wird zwischen *klasse1.uml* und *klasse1.java* gezogen, da diese die Regel in *uml_java_regeln.xml* erfüllen. Im Fall des inkonsistenten Verweises (Zeile 9-15) wird dieser zwischen *klasse2.uml* (Zeile 11) und der Regel (Zeile 13) gezogen, da diese für diese UML-Klasse nicht gilt. Für die nähere Erklärung der XLinks wird erneut auf [1] verwiesen.

```
1     <ForAll var="u" in="uml">
2         <Exists var="j" in="java">
3             <Equal op1="u/name" op2="j/name" cmode="C" imode="IX">
4         </Exists>
5     </ForAll>
```

Abbildung 5: Beispiel einer Konsistenzregel in XML

```
<LinkBase docset="DokumentMenge.xml" ruleset="RegelMenge.xml">
    <ConsistencyLink ruleid="uml_java_regeln.xml#id('r1')">
        <State>consistent</State>
        <Locator xlink:href="klasse1.uml#/name"
                 xlink:label="source" xlink:title=" "/>
        <Locator xlink:href="klasse1.java#/name"
                 xlink:label="dest" xlink:title=" "/>
    </ConsistencyLink>
    <ConsistencyLink ruleid="uml_java_regeln.xml#id('r1')">
        <State>inconsistent</State>
        <Locator xlink:href="klasse2.uml#/name"
                 xlink:label="source" xlink:title=" "/>
        <Locator xlink:href="uml_java_regeln.xml#id('r1')"
                 xlink:label="dest" xlink:title=" "/>
    </ConsistencyLink>
</LinkBase>
```

Abbildung 6: Beispiel eines Verweises in der Verweisbasis von xlinkit

3 Erweiterungen von xlinkit

In diesem Kapitel werden Erweiterungen von xlinkit vorgestellt, die das Arbeiten mit dem Rahmenwerk erleichtern und es sinnvoll ergänzen. Dazu wird zuerst auf die Visualisierung der Ausgabe (Verweisbasis) eingegangen. Der zweite Abschnitt behandelt die Möglichkeiten, Inkonsistenzen aufzulösen. Dazu werden Reparaturaktionen benötigt, die dort beschrieben werden.

3.1 Visualisierung der Verweisbasis

Die Ausgabe von xlinkit, wie im letzten Kapitel beschrieben, besteht aus der Verweisbasis in Form einer XML-Datei. Im Beispiel von Abbildung 6 ist zu sehen, dass dort schon eine unübersichtliche Ausgabe zustande kommt. Dadurch wird schnell klar, dass die Ausgabe sowohl interpretiert, als auch aufbereitet werden muss. Wesentlich unübersichtlicher und damit auch weniger nützlich wird es bei größeren Projekten. Um diesem Problem vorzubeugen, gibt es verschiedene Arten der Visualisierung. Eine Möglichkeit bietet ein Webportal, das von den Entwicklern von xlinkit selbst eingerichtet wurde.[1] Dort werden die einzelnen Verweise mittels eines Stylesheets in das HTML-Format transformiert. Die Darstellung im Browser teilt sich in drei Bereiche: (1) Im oberen Bereich sind alle in der Verweisbasis vorhandenen Verweise zu sehen. Dabei steht bereits in der Überschrift, ob

[1] Leider ist dieses Portal unter www.xlinkit.com nicht mehr erreichbar

es ein konsistenter Verweis ist, oder nicht. Der untere Bereich wird in der Mitte geteilt und (2) links erscheint das Anfangsdokument und (3) rechts das Zieldokument. Es sind dabei jeweils die Stellen der Dokumente sichtbar, zwischen denen die Inkonsistenz auftritt. Da HTML eng verwandt mit XML ist, ist der Aufwand der Transformation nicht allzu groß und die Verweisbasis lässt sich so bereits viel besser handhaben.
Falls die zu prüfenden Objekte UML-Dokumente sind, gibt es auch die Möglichkeit, diese mit Hilfe des Werkzeugs BOX [5] im Browser zu visualisieren. Dort verweisen Pfeile in bestimmten Farben auf inkonsistente Elemente (UML-Klassen oder UML-Elemente). Eine solche graphische Visualisierung ist meist leichter zugänglich, als eine rein textuelle. Kommen allerdings noch andere Dokumenttypen dazu, wird die Visualisierung auf diese Art schon weit schwieriger. Das ist darauf zurückzuführen, dass ein Dokumenttypanpassung stattfinden muss, da nur UML unterstützt wird.
Das Ziel muss aber sein, dass die Konsistenzanalyse in CASE-Werkzeuge integriert wird. Auf diese Art wird es für den industriellen Einsatz erst wirklich interessant. Denn wird ein Konsistenzerhaltungswerkzeug direkt im CASE-Werkzeug integriert, erleichtert dies den Entwicklern die Navigation durch inkonsistente Stellen. Eine Möglichkeit ist die Verwendung von Skript-Sprachen, da sie von vielen CASE-Werkzeugen unterstützt werden. Dazu müssen auch Inkonsistenzen zwischen verschiedenen Typen von Dokumenten verständlich dargestellt werden, was über ein einheitliches Rahmenwerk zur Visualisierung realisiert werden kann. Einen ersten akzeptablen Versuch stellt Chimera [5] dar.

3.2 Beseitigung von Inkonsistenzen

Die Auflösung von Inkonsistenzen ist eine wichtige Erweiterung von xlinkit. Wie jedoch oben bereits erwähnt, muss nicht jede Inkonsistenz zwangsläufig schlecht sein und aufgelöst werden. Oftmals spielen kleinere Inkonsistenzen keine Rolle. Somit ist eine automatische Auflösung dieser nicht zwingend nötig. Wenn eine Inkonsistenz aufgelöst werden muss, ist dies schwer realisierbar, da es oft mehrere Möglichkeiten dazu gibt. Außerdem ist es möglich, dass in einer Phase des Projekts Inkonsistenzen auf die eine Art beseitigt werden, in einer anderen Phase aber auf eine andere. Als Beispiel sei wieder das UML-Java-Beispiel genannt. So ist es zu Beginn der Implementierungsphase wünschenswert, dem UML-Modell exakt zu entsprechen. Also werden hier die Java-Dokumente den UML-Modellen angepasst und somit wird die Inkonsistenz im Quellcode aufgelöst. Es kann aber im Verlauf der Implementierungsphase auch passieren, dass es wegen verwendeter Techniken zu einer Inkonsistenz zu dem Modell kommt. Dann soll nicht die Implementierung überdacht und abgeändert werden, nur um dem Modell wieder

```
Aktion :=           Hinzufuegen Zugriffspunkt |
                    Loeschen Zugriffspunkt |
                    Aendern Zugriffspunkt relativerPfad
Zugriffspunkt :=    Direkt LocatorNummer |
                    Lookup XPath
XPath :=            AbsoluterPfad | relativerPfad
```

Abbildung 7: Syntax der Reparaturaktionen

gerecht zu werden. Hier wird das UML-Modell an den Quellcode angepasst. Der Übergang zwischen solchen Phasen ist nicht automatisiert feststellbar. Somit ist die voll-automatisierte Beseitigung von Inkonsistenzen zwar wünschneswert, aber nicht das einzige Ziel. Stattdessen sollen Inkonsistenzen aufgespürt werden und automatisch (sinnvolle) Vorschläge zu deren Beseitigung gemacht werden.
Die so erzeugten Vorschläge heißen *Reparaturaktionen* und werden von einem Reparatur-Werkzeug automatisch erzeugt. Dabei soll die Menge der Reaparturaktionen korrekt und vollständig sein, d.h. jede Reparaturaktion kann die Inkonsistenz beseitigen und es gibt keine Aktion, welche die Inkonsistenz auflöst und nicht in dieser Menge ist. Neben der Bedingung, dass genau alle Reparaturaktionen angeboten werden, sollen diese ebenfalls sinnvoll sein. Es macht beispielsweise keinen Sinn durch das Auflösen einer Inkonsistenz eine oder mehrere andere Inkonsistenzen zu erzeugen. Somit erfüllt das Reparatur-Werkzeug nicht nur die Funktion, alle Reparaturmöglichkeiten zu finden, sondern bietet auch eine Möglichkeit, die in diesem Fall nicht sinnvollen zu deaktivieren. Dies ist durch das spezifische Wissen über die Eigenschaft bestimmter Dokumenttypen möglich. Dieses Wissen muss durch einen Administrator eingegeben werden.
Das Reparatursystem kann drei Arten von Änderungen an einem Dokument vornehmen: (1) Hinzufügen, (2) Löschen und (3) Ändern. Die Syntax dieser Befehle ist in Abbildung 7 zu sehen. Die *Zugriffspunkte* sind dabei in zwei Gruppen aufgeteilt. Zum einen gibt es die *direkten* Zugriffspunkte, die auf Informationen relativ zu einem xlinkit *Locator* (Abbildung 6) zeigen und zum anderen die *Lookup* Zugriffspunkte, aus denen der Benutzer interaktiv wählen muss. Letztere treten dann auf, wenn mehrere Belegungen einer Variablen möglich sind.
Im Gegensatz zur einfachen Syntax, ist die Semantik für das Erzeugen von Reparaturaktionen vergleichsweise kompliziert und würde den Rahmen dieser Ausarbeitung sprengen. Es soll an dieser Stelle nur erwähnt werden, dass es für die Semantik vier Funktionen gibt. Zwei davon gibt es für den Fall von direkten Zugrffspunkten (D_i und D_c) und zwei für den Fall, dass es keine direkten Zugriffspunkte gibt (N_i und N_c). Die Buchstaben *i* und *c* stehen dabei jeweils für *inconsistent* und *consistent*. Im zweiten Fall, muss der Benutzer die Wahl zwischen mehreren mög-

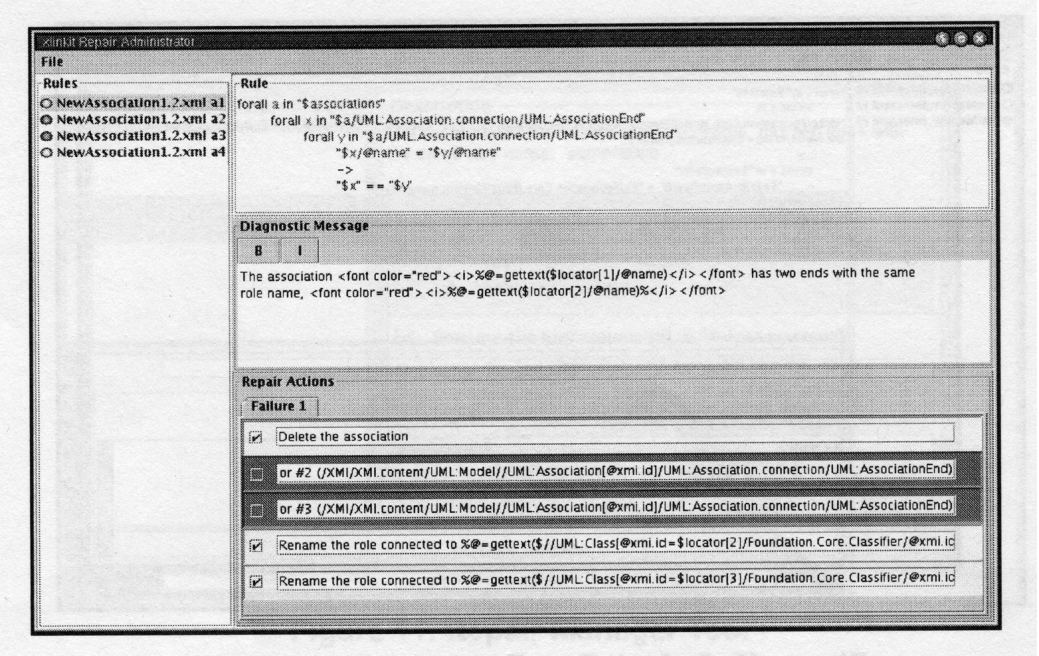

Abbildung 8: Reparatur-Administrator Werkzeug

lichen Variablenbelegungen treffen. Die genaue Definition der Semantik kann in [4] nachgelesen werden.
Zum Abschluß des Kapitels wird jetzt noch eine Visualisierung der Reparaturmöglichkeiten gegeben. Dazu werden die in [4] beschriebenen Werkzeuge ”xlinkit Repair Administrator” und ”xlinkit Repair Manager” verwendet, die jeweils in Abbildung 8 und Abbildung 9 zu sehen sind.[2] Als Beispiel dient folgende Regel für XML-Dokumente: *"Die Assoziationen im UML-Modell müssen einen eindeutigen Namen innerhalb der Verweise aufweisen"*. Wie in der Abbildung zu sehen, sind diverse Reparaturmöglichkeiten vorhanden. Dabei können diese entweder in natürlicher Sprache beschrieben werden, oder aber die Standard-Beschreibung enthalten, die bei der Erzeugung der Möglichkeiten erstellt wird. Die Möglichkeiten zwei und drei sind dunkel eingefärbt und somit gesperrt. In diesem Fall sind das die Optionen zur Löschung der Assoziationen; dies ist nicht wünschenswert, da dadurch die Struktur der Dokumente untereinander geändert würde. Die An- und Abwahl muss durch einen Administrator geschehen, der die Möglichkeiten des Benutzers einschränken soll.
Das Fenster über den Reparaturmöglichkeiten beinhaltet eine Diagnose-Nachricht. Hier wird die Inkonsistenz als ganzes beschrieben und nicht so speziell wie darunter. Es soll festgehalten werden, welches Problem im Allgemeinen in Hinblick

[2]Leider ist auch dieses Werkzeug nicht verfügbar

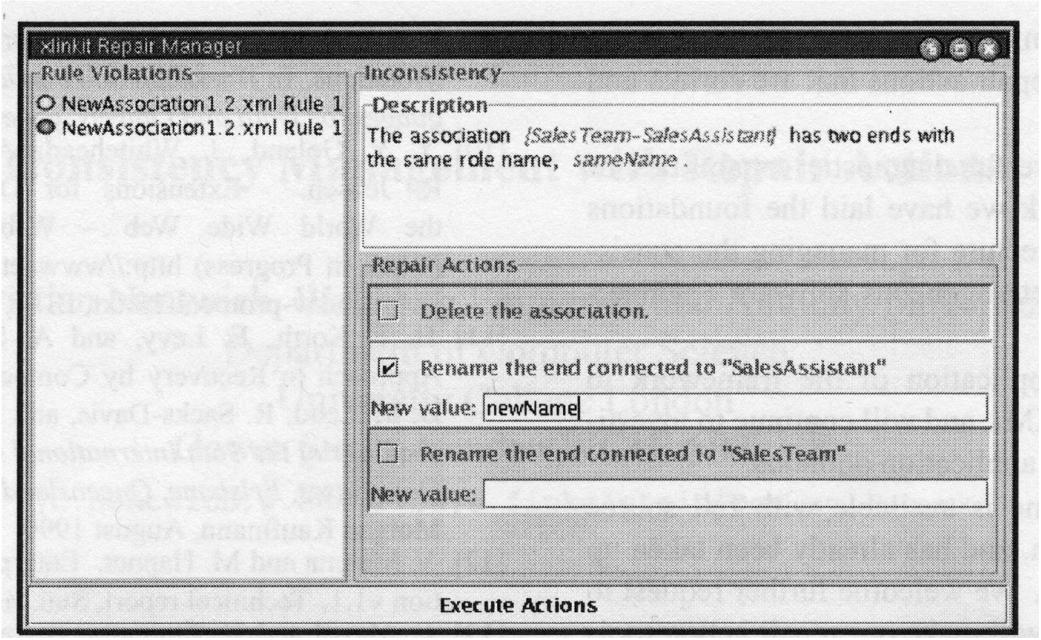

Abbildung 9: Reparatur-Manager

auf diese Regel aufgetreten ist. Auch dieser Eintrag kann wiederum durch den Administrator im Administrator-Werkzeug geändert werden.
Sind diese Entscheidungen getroffen worden, werden die Optionen als Teil eines Reports (in XML) gespeichert, damit diese später im eigentlichen Reparatur-Werkzeug benutzt werden können. Diese Entscheidungen müssen für alle aufgelisteten Regeln getroffen werden.
Der Reparatur-Manager ist in Abbildung 9 zu sehen. Dort werden auf der linken Seite die Regeln angezeigt, für die Reparaturaktionen ausgeführt werden müssen. Die rechte Seite ist in die Beschreibung (oben) und die möglichen Reparaturaktionen (unten) unterteilt. Hieraus kann der Benutzer nun die gewünschten Aktionen auswählen. Wichtig ist, dass hier nur noch bei den Regeln gewählt werden kann, für die der Administrator im anderen Werkzeug mehr als eine Option gelassen hat. Hat dieser nur eine gelassen, wird diese Aktion automatisch ausgeführt, lässt er keine übrig, wird keine Aktion ausgeführt. Der Reparatur-Manager ist ebenfalls in der Lage, widersprüchliche Reparaturaktionen zu erkennen und dort zu einer erneuten Änderung aufzufordern. Der *Execute Actions*-Knopf am unteren Bildschirmrand führt dann alle gewählten Änderungen aus.

4 Bewertung und Ausblick

Die in den letzten Kapiteln beschriebene Vorgehensweise von xlinkit wirkt auf den ersten Blick wie ein guter, einfacher Ansatz zur Konsistenzanalyse. Laut der Autoren gibt es bereits einige erfolgreiche Ansätze zur Konsistenzanalyse mit xlinkit, sowie mit einer darauf aufbauenden Reparatur von Inkonsistenzen. Neben den theoretischen und formalen Grundlagen sind demnach auch bereits Implementierungen vorhanden. Problematisch ist jedoch, dass keine der Implementierungen, in denen das Rahmenwerk verwendet wird, mehr zugänglich ist. Ebenso sieht es im Bereich der automatisierten Beseitigung von Inkonsistenzen aus. Die hierzu vorgestellten Werkzeuge aus Abschnitt 3.2 sind leider auch nicht mehr zu finden. Doch nicht nur hier werden Zweifel an diesem Ansatz deutlich. Ebenso ist es ein großes Problem, dass die vorgestellten Beispiele sehr trivial sind und somit über die Mächtigkeit des xlinkit-Ansatzes wenig aussagen. Durch diese Probleme kann nur wenig über die tatsächliche Leistungsfähigkeit des Ansatzes gesagt werden. Leider sind auch keine aktuelleren Veröffentlichungen zu dem Thema mehr zu finden.
Ein Vorteil des xlinkit-Ansatzes ist jedoch das intuitive Verständnis des Aufbaus der Dokumente und Regeln, was auf die Verwendung von XML-Technologien zurückzuführen ist. Eine kurze Einarbeitung genügt, um eine erste sinnvolle Nutzung zu ermöglichen.
Die Autoren haben allerdings auch klar gesteckte Ziele für die Zukunft. Bei der reinen Konsistenzanalyse soll es primär darum gehen, diese Technik in CASE-Werkzeuge zu integrieren, um damit auch eine industrielle Nutzung voranzutreiben. Außerdem ist es ein Ziel, bei Änderungen nur noch die betroffenen Regeln und Dokumente zu prüfen und nicht wieder eine komplette Analyse durchzuführen. Dies ist ein wichtiger Schritt auf dem Weg zur industriellen Nutzung.
Im Bereich der Reparatur-Werkzeuge stellt die weitere Automatisierung eine große Herausforderung dar. Aus den beiden oben vorgestellten Werkzeugen, Reparatur-Administrator und Reparatur-Manager soll dazu ein kombiniertes Werkzeug werden. Durch die Zusammenfassung der beiden Werkzeuge fällt die Aufgabe des Administrators heraus und somit ein manueller Schritt. Dies führt zu einer größeren Automatisierung. Dazu müssen jedoch Präzedenzregeln nicht nur statisch eingebunden werden, sondern, je nach Situation, auch automatisch änderbar sein. Dazu ist ein dynamisches Verfahren von Nöten. Auch dazu gibt es bereits Ideen, die sich auf ein System namens Emu [4] beziehen.
Das endgültige Ziel für die reine Konsistenzanalyse und die Konsistenzanalyse mit Reparaturaktionen ist die Integration in gängige Entwicklungsumgebungen, wie z.B. Eclipse, so dass die automatische Konsistenzanalyse ein Bestandteil der Softwareentwicklung wird und sich in den gesamten Prozess homogen einfügt

und nicht mehr per Hand oder durch andere ineffektive Verfahren durchgeführt werden muss.

5 Vergleich mit anderen Ansätzen

Neben xlinkit gibt es noch viele andere Werkzeuge, die eine Konsistenzanalyse durchführen können. Jedoch können viele dieser Werkzeuge, z.B. GoodStep [5], keine beliebig verteilten Dokumente analysieren. Sie benötigen immer ein zentrales Archiv, in dem die Dokumente vorhanden sind. Außerdem sind diese Ansätze oft auch Programmiersprachen-spezifisch und nicht für ein so weites Feld an Sprachen geeignet wie xlinkit. Hier hat xlinkit durch die Vielseitigkeit von XML einen klaren Vorteil. Jedoch muss auch hier bedacht werden, dass nicht nachgewiesen worden ist, dass der xlinkit-Ansatz zu einer komplexen Konsistenzanalyse in der Lage ist.
Ein anderer Ansatz zur Konsistenzanalyse ist die Verwendung von Graph-Grammatiken [6], um die Konsistenz prüfen zu können. Angeblich ist allerdings die Aussagekraft der Konsistenzregeln nicht so groß, wie durch die prädikatenlogischen Formeln [3]. Die Ansätze gehen auch grundsätzlich dadurch auseinander, da xlinkit Standards nutzt und die Verteiltheit von Dokumenten zum zentralen Bestandteil macht, während graph-grammatische Ansätze eigene Formalismen und zentrale Datenstrukturen verwenden.
Auch ist es, soweit bekannt, bislang in keinem anderen Ansatz möglich, aus der Konsistenzanalyse heraus direkt Reparaturaktionen anzubieten. Dies ist ein weiterer Vorteil von xlinkit. Es gibt auch andere Ansätze, wie aktive Integritätserhaltung per Auslöser [4], jedoch ist dieser nicht in der Lage, mehr als einfache boolesche Ausdrücke auszuwerten und das auch nicht auf verteilten Dokumenten.

6 Zusammenfassung

In dieser Ausarbeitung wurde das Rahmenwerk xlinkit zur Konsistenzanalyse vorgestellt. Dazu wurde ein Blick auf die Wurzeln in XML geworfen, sowie auf die formale Definition der Regeln, bis hin zu den Regeln in XML-Schreibweise und die Form der Ausgabe. Außerdem sind noch einige Erweiterungen von xlinkit präsentiert worden. Dazu gehörte neben der Visualisierung des Verfahrens, auch die Beseitigung von Inkonsistenzen durch Reparaturaktionen. Hier wurde dann auf die Schwierigkeiten der Automatisierung hingewiesen und ein erstes Werkzeug zur Durchführung von Reparaturen vorgestellt. Nachdem diese Vorstellung abge-

schlossen war, wurde eine Bewertung gemacht, sowie auf die Ziele für die Zukunft verwiesen. Den Abschluß bildet der Vergleich mit anderen ähnlichen Verfahren unter besonderer Berücksichtigung der Vorteile, die xlinkit bietet.

Literatur

[1] DEROSE, STEVE, EVE MALER und DAVID ORCHARD: *XML Linking Language (XLink) Version 1.0*. Candidate Recommendation http://www.w3.org/TR/xlink/, World Wide Web Consortium, 2001.

[2] KEMPER, ALFONS und ANDRÉ EICKLER: *Datenbanksysteme: Eine Einführung*. Oldenbourg Wissenschaftsverlag, München, 5. Auflage, 2004.

[3] NENTWICH, CHRISTIAN, LICIA CAPRA, WOLFGANG EMMERICH und ANTHONY FINKELSTEIN: *xlinkit: A Consistency Checking and Smart Link Generation Service*. ACM Transactions on Internet Technologie, 2(2):151–185, 2002.

[4] NENTWICH, CHRISTIAN, WOLFGANG EMMERICH und ANTHONY FINKELSTEIN: *Consistency Management with Repair Actions*. In Proceedings of the 25th International Conference on Software Engineering, 2003.

[5] NENTWICH, CHRISTIAN, WOLFGANG EMMERICH, ANTHONY FINKELSTEIN und ERNST ELLMER: *Flexible Consistency Checking*. ACM Transactions on Software Engineering and Methodology, 12(1):28–63, 2003.

[6] ZÜNDORF, ALBERT: *PROgrammierte GRaphErsetzungsSysteme - Spezifikation, Implementierung und Anwendung einer integrierten Entwicklungsumgebung*. PhD thesis, University of Aachen, 1996.