ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Semesterarbeit

XML-Diff-Algorithmen

Daniel Hottinger Franziska Meyer

13. Juli 2005

Betreuer: Paul E. Sevinç
Prof. Dr. David Basin

Abstract

This document describes the semester thesis of Franziska Meyer and Daniel Hottinger at the *Information Security Group*[1] of the *Department of Computer Science*[2] at the *ETH Zurich*[3].

The goal of this semester thesis was to establish a set of comparison criteria for XML-diff algorithms and compare existing algorithms by applying the criteria. The best of these algorithms was to be implemented and made available within a Java library.

[1] http://www.infsec.ethz.ch/
[2] http://www.inf.ethz.ch/
[3] http://www.ethz.ch/

Kurzfassung

Dieses Dokument beschreibt die Semesterarbeit von Franziska Meyer und Daniel Hottinger bei der *Gruppe Informationssicherheit*[4] des *Departements Informatik*[5] an der *ETH Zürich*[6].

Das Ziel dieser Semesterarbeit war es, Kriterien für den Vergleich von XML-Diff-Algorithmen aufzustellen und vorhandene Algorithmen anhand der Kriterien zu vergleichen. Der beste Algorithmus sollte implementiert und innerhalb einer Java-Bibliothek zugänglich gemacht werden.

[4] http://www.infsec.ethz.ch/
[5] http://www.inf.ethz.ch/
[6] http://www.ethz.ch/

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

In der heutigen, von Standards, Protokollen und Dateiformaten überfluteten Informatikgesellschaft stellt XML einen sehr erfolgreichen Versuch dar, Ordnung ins Chaos der Formate zur Speicherung strukturierter Daten zu bringen. Für die weitere Verbreitung dieses textbasierten Formats ist aber die Existenz guter Bearbeitungs- und Manipulationstools sehr wichtig. Das Aufspüren von Veränderungen stellt dabei ein zentrales Problem dar. Während sich bei unstrukturiertem Text seit Jahrzehnten der LCS-Algorithmus und das einfach gehaltene, zeilenbasierte Format aus Abbildung 1.1 auf Seite 10 durchgesetzt haben, gibt es für XML eine Vielzahl von Algorithmen unterschiedlicher Mächtigkeit, welche alle ihre eigenen Formate mitbringen.

XML-Dokumente können zwar auch mit einem normalen, zeilenbasierten Diff-Tool verglichen werden, wie es in Abbildung 1.2 auf Seite 10 dargestellt ist. Allerdings berücksichtigen solche Programme die speziellen Eigenschaften von XML nicht und stossen daher schnell an ihre Grenzen. Es ist beispielsweise bei XML möglich, Vorgabewerte von Attributen im Schema zu setzen, Attribute beliebig zu permutieren oder sie durch eine beliebige Menge Whitespace von einander zu trennen. Auch können ganze Teilbäume, d. h. alles zwischen einem öffnenden und einem schliessenden Tag, verschoben werden, was auch als Verschiebung erkannt werden sollte.

Das Ziel dieser Semesterarbeit war es, Kriterien für den Vergleich von XML-Diff-Algorithmen aufzustellen, diese auf vorhandene Algorithmen anzuwenden und schliesslich den besten innerhalb einer Java-Bibliothek unserem Betreuer zur Verfügung zu stellen.

Im folgenden Kapitel gehen wir näher auf die Probleme beim Finden von Änderungen in XML-Dokumenten ein. Kapitel 3 zeigt Formate, die solche Änderungen beschreiben. Der Vergleich und die Kriterien finden sich in Kapitel 4. In Kapitel 5 findet sich ein Handbuch zu unserer Implementation, welche in Kapitel 6 näher beschrieben wird. Kapitel 7 schliesslich zeigt ungelöste Probleme und Ideen für weitere Arbeiten auf. Abschliessend folgt noch Referenzmaterial wie das Inhaltsverzeichnis der CD und die Aufgabenstellung.

```
3,5c3,5
<  <person >
<   <firstname>Fransizka</firstname>
<   <lasname>Meyer</lasname>
---
>  <person female="true">
>   <firstname>Franziska</firstname>
>   <lastname>Meyer</lastname>
8c8
<   <firstname def="ghi">Daniel</firstname>
---
>   <firstname>Daniel</firstname>
12,13c12,13
<   <firstname abc="def">Paul</firstname>
<   <lastname>Sevinç</lastname>
---
>   <firstname>Paul</firstname>
>   <lastname abc="def">Sevinç</lastname>
```

Abbildung 1.1.: Ausgabe von GNU-Diff

Abbildung 1.2.: Zeilen- und zeichenbasiertes Diff des Texteditors VIM

2. Finden von Änderungen

Das Problem, Änderungen in Text-Dokumenten zu erkennen, gilt seit geraumer Zeit als gelöst [HM76]. Die Grundidee des Erkennungs-Algorithmus ist, den Text zeilenweise zu vergleichen und dabei die Longest Common Subsequence (LCS) zu berechnen. Zeilen, die darin nicht enthalten sind, wurden entweder gelöscht oder hinzugefügt.

Dieser Algorithmus hat sich in der Zwischenzeit etabliert und wird von Implementierungen wie GNU Diff oder Versionsverwaltungssystemen wie Concurrent Versions System (CVS), Subversion, Bitkeeper oder git rege genutzt.

2.1. Besonderheiten von XML

Im Gegensatz zu unstrukturiertem Text besitzen XML-Dokumente eine wohldefinierte interne Struktur [BPSM98]. So spielt die Darstellung innerhalb der Text-Datei für XML nur eine untergeordnete Rolle, und es ist bei generiertem XML durchaus üblich, alles auf eine Zeile zu schreiben. Auch bilden öffnende und schliessende Tags logische Blöcke, die beim Vergleich als Einheit betrachtet werden sollten. Dieses Problem tritt zwar auch bei Programmiersprachen auf, wo der Scope von Variablen die logische Einheit darstellt, bei XML tritt das Problem aber verstärkt auf, da sehr viele dieser logischen Einheiten vorhanden sind.

Eine Sonderrolle spielen Attribute, da ihre Reihenfolge irrelevant ist. Zwischen den Attributen kann sich beliebig viel Whitespace befinden, also auch Zeilenumbrüche oder Tabulatoren, der für XML völlig irrelevant ist. Der Zeichensatz ist nur für das Abspeichern der Daten relevant und geht bei der Internalisierung verloren. Beim Abspeichern besteht auch die Möglichkeit, für XML relevante Sonderzeichen zu escapen oder eine CDATA-Abschnitt zu verwenden.

Beim Document Object Model (DOM) werden die Daten als baumartige Struktur betrachtet. Es ist nicht möglich, einen Knoten eindeutig zu identifizieren. So nummeriert z. B. XPath alle gleichnamigen Kindknoten durch, was bei Permutationen zu Problemen führen kann. Es gibt eine Vielzahl von XPath-Ausdrücken, um den gleichen Knoten zu identifizieren, welcher dafür aber am besten geeignet ist, hängt vom Kontext ab. Dass Labels, die Strukturinformationen enthalten, sich bei Updates ändern müssen, beschreibt [CKM02, Seite 1] sehr gut.

XML-Datenbanken umgehen das Problem, indem sie eindeutige, randomisierte Labels für die Knoten generieren, die sich bei Updates nicht verändern. Dass diese Labels erhalten bleiben, kann aber nur garantiert werden, wenn die XML-Daten in der Datenbank verbleiben und nur von der Datenbank verändert werden. Sobald das Dokument exportiert wird – z. B. um es mit einem externen Programm zu bearbeiten – gehen die Labels in der Regel verloren und eine Zuordnung der Knoten ist nicht mehr möglich.

3. Darstellung von Änderungen

Um Änderungen darzustellen haben sich zwei Formate herauskristallisiert. XyDelta und XUpdate selektieren den Knoten, auf den sich eine Änderung bezieht, zuerst möglichst eindeutig. Erfüllt nur ein Knoten die Bedingung, so wird auf ihm eine Operation ausgeführt (insert, delete, ...). Dieses Format ist stark an das ursprüngliche Diff angelehnt und hat schöne mathematische Eigenschaften. So können XyDeltas aggregiert oder invertiert werden. Der Nachteil dieses Formats ist, dass der Kontext zu den Änderungen fehlt. Wir beschreiben das Format von XyDelta in Abschnitt 3.1 auf Seite 12 und das von XUpdate in Abschnitt 3.2 auf Seite 12.

Das zweite Format, das wir anhand von XOp und DeltaXML vorstellen, übernimmt die Struktur des ursprünglichen Dokuments und fügt den geänderten Knoten Attribute hinzu, die die Art der Änderung beschreiben. So können z. B. Knoten als gelöscht markiert werden, Attribute als geändert oder neue Knoten als hinzugefügt. Der Vorteil dieses Formats ist, dass der Kontext einer Änderung erhalten bleibt und dass das Format leicht weiterverarbeitet werden kann. Auf der Webseite von XOp [ep04] ist eine Demonstration mit SVG-Grafiken zu finden. Nachdem im Bild eines Tigers einige Haare entfernt wurden, wird die Differenz der beiden Dokumente berechnet. Das Resultat sind die abrasierten Haare in Form einer gültigen SVG-Grafik.

DeltaXML stellt ein XSL-Dokument zur Verfügung, das die Ausgabe in HTML umwandelt, so dass die Änderungen im Webbrowser visualisiert werden können. Um den Umfang des Diffs bei wenigen Änderungen zu reduzieren, bieten sowohl DeltaXML als auch XOp die Option an, unveränderte Teilbäume nur durch den obersten Knoten zu repräsentieren und diesen als unverändert zu markieren. Wir beschreiben das Format von DeltaXML in Abschnitt 3.3 auf Seite 14 und das von XOp in Abschnitt 3.4 auf Seite 16.

3.1. XyDelta

XyDelta [MAC+01] benutzt persistente Labels, um Knoten zu identifizieren, sogenannte XIDs. XIDs sind eindeutige Nummern. Die verwendeten Deltas sind komplett, das heisst, sie enthalten alle Informationen, die nötig sind, um sie auch wieder rückgängig zu machen. Es werden folgende Operationen definiert:

delete(n**,** k**,** T**)** Löscht den Baum T, dessen Wurzel das k-te Kind des Knotens n ist.
update(n**,** v**,** ov**)** Ändert den Wert (Text) von Knoten n auf v, wobei ov der alte Wert ist.
insert(n**,** k**,** T**)** Fügt den Baum T als k-tes Kind unterhalb des Knotens n ein.
move(n**,** k**,** m**,** p**,** q**)** Verschiebt den Teilbaum, dessen Wurzelknoten m das q-te Kind von Knoten p ist, so dass m neu das k-te Kind von Knoten n ist.
label(n**,** v**,** ov**)** Ändert die Bezeichnung von Knoten n auf v, wobei ov der alte Wert ist.

Diese Operationen haben schöne mathematische Eigenschaften, so können sie aggregiert und invertiert werden. Allerdings enthalten sie keine Information, in welchem Kontext eine Änderung gemacht wurde.

Beispiel
Die XML-Repräsentation sieht z. B. so aus:

```
<delta>
  <unit_delta>
    ...
  </unit_delta>
  <unit_delta>
    <time from="1" to="2" />
    <delete parent="11" position="4" xid-map="(17-21)">
      <Product>
        <Name>DVD</Name>
        <Price>500</Price>
      </Product>
    </delete>
    <move xid="16" new_parent="11" new_position="2"
                   old_parent="11" old_position="1" />
    <update xid="3" new_value="50" old_value="100" />
    <update xid="8" new_value="100" old_value="150" />
  </unit_delta>
</delta>
```

3.2. XUpdate

Diese Beschreibung bezieht sich auf das Working Draft vom 14. September 2000.

XUpdate[LM00] ist ein XML-Dialekt, der darauf ausgelegt ist, Änderungen an XML-Dokumenten zu beschreiben. Für das Selektieren der Elemente wird stark von XPath [CD99] Gebrauch gemacht. XUpdate kennt im Wesentlichen die folgenden Operationen:

xupdate:insert-before Fügt ein Element vor dem über das `select`-Attribut ausgewählten ein.

xupdate:insert-after Fügt ein Element nach dem über `select` ausgewählten ein.

xupdate:append Fügt ein Element unterhalb des über `select` ausgewählten ein. Mit dem optionalen `child`-Attribut kann die Position als Integer angegeben werden (standardmässig: `last()`).

xupdate:update Ändert den Wert eines Elements oder Attributes.

xupdate:remove Löscht ein Element.

xupdate:rename Benennt ein Element oder Attribut um.

Damit können Elemente (`xupdate:element`), Attribute (`xupdate:attribute`), Text (`xupdate:text`), Meta-Angaben (`xupdate:processing-instruction`) und Kommentare (`xupdate:comment`) eingefügt, verändert oder gelöscht werden.

Das folgende XUpdate-Dokument fügt einen Datensatz hinzu und ändert den Wert von `town` im ersten Datensatz:

```
<?xml version="1.0"?>
<xupdate:modifications version="1.0"
                xmlns:xupdate="http://www.xmldb.org/xupdate">

  <xupdate:insert-after select="/addresses/address[1]" >

    <xupdate:element name="address">
      <xupdate:attribute name="id">2</xupdate:attribute>
      <fullname>Lars Martin</fullname>
      <born day='2' month='12' year='1974'/>
      <town>Leipzig</town>
      <country>Germany</country>
    </xupdate:element>
    <xupdate:update select="/addresses/address[1]/town">
      Berlin
    </xupdate:update>

  </xupdate:insert-after>
</xupdate:modifications>
```

Damit wird das Originaldokument (links) in die veränderte Kopie (recht) übergeführt:

```
<?xml version="1.0"?>
<addresses version="1.0">

  <address id="1">
    <fullname>Andreas Laux</fullname>
    <born day='1' month='12' year='1978'/>
    <town>Leipzig</town>
    <country>Germany</country>
  </address>

</addresses>
```

```
<?xml version="1.0"?>
  <addresses version="1.0">

    <address id="1">
      <fullname>Andreas Laux</fullname>
      <born day='1' month='12' year='1978'/>
      <town>Berlin</town>
      <country>Germany</country>
    </address>

    <address id="2">
      <fullname>Lars Martin</fullname>
      <born day='2' month='12' year='1974'/>
      <town>Leipzig</town>
      <country>Germany</country>
    </address>


  </addresses>
```

Die DTD von XUpdate ist im Anhang A zu finden.

3.3. DeltaXML

Das Format von DeltaXML[1] benutzt das alte Dokument als Grundlage und versieht es mit zusätzlichen Attributen, die die Art der Änderungen beschreiben. Dadurch bleibt die Struktur des Dokuments vollständig erhalten, und es ist leicht nachvollziehbar, in welchem Kontext eine Änderung gemacht wurde.

[1] `http://www.deltaxml.com/`

Jeder XML-Node bekommt ein Attribut aus dem `deltaxml`-Namespace, das angibt, ob der Node verändert, hinzugefügt oder gelöscht wurde. Im Normalfall werden unveränderte Attribute und Inhalte von Nodes nicht ausgegeben. Dies kann jedoch explizit verlangt werden (Full Context Delta).

Änderungen von Attributen werden als Attribut der Form `deltaxml:new-attributes="name=&quot;dl&quot;"` für ein hinzugefügtes Attribut `name='dl'` dargestellt. Mehrere neue Attribute werden in ein Attribut zusammengefasst.

Elemente, die einem anderen Namespace zugeordnet werden, werden als völlig verschiedene Elemente angeschaut und entsprechend wird die Änderung als Austausch des Elements und nicht als geändertes Attribut dargestellt.

Elemente im Baum, die veränderte Blattelemente haben, bekommen ein Attribut `deltaxml:delta="WFmodify"`. Ein unveränderter Teilbaum wird nur durch das oberste Element repräsentiert, welches mittels `deltaxml:delta="unchanged"` als unverändert markiert wird.

Mittels Extensible Stylesheet Language (XSL) kann die Ausgabe für einen Browser ansprechend aufbereitet werden (siehe Abbildung 3.1 auf Seite 16). DeltaXML ist kommerziell. Das Format ist patentiert und in [Fon04] genauer beschrieben.

Beispiel

Die Ausgabe für die beiden Beispieldateien (siehe Anhang B.1) sieht folgendermassen aus:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<people xmlns:deltaxml="http://www.deltaxml.com/ns/well-formed-delta-
    v1" deltaxml:delta="WFmodify">
   <person deltaxml:delta="WFmodify" deltaxml:new-attributes="female
      =&amp;quot;true&amp;quot;">
      <firstname deltaxml:delta="WFmodify">
         <deltaxml:PCDATAmodify>
            <deltaxml:PCDATAold>Fransizka </deltaxml:PCDATAold>
            <deltaxml:PCDATAnew>Franziska </deltaxml:PCDATAnew>
         </deltaxml:PCDATAmodify>
      </firstname>
      <deltaxml:exchange>
         <deltaxml:old>
            <lasname deltaxml:delta="delete">Meyer</lasname>
         </deltaxml:old>
         <deltaxml:new>
            <lastname deltaxml:delta="add">Meyer</lastname>
         </deltaxml:new>
      </deltaxml:exchange>
   </person>
   <person deltaxml:delta="WFmodify">
      <firstname deltaxml:delta="WFmodify" deltaxml:old-attributes="
         def=&amp;quot;ghi&amp;quot;">Daniel</firstname>
      <lastname deltaxml:delta="unchanged"/>
   </person>
```

```
   <person deltaxml:delta="WFmodify">
      <firstname deltaxml:delta="WFmodify" deltaxml:old-attributes="
         abc=&amp;quot;def&amp;quot;">Paul</firstname>
      <lastname deltaxml:delta="WFmodify" deltaxml:new-attributes="
         abc=&amp;quot;def&amp;quot;">Sevinc</lastname>
   </person>
</people>
```

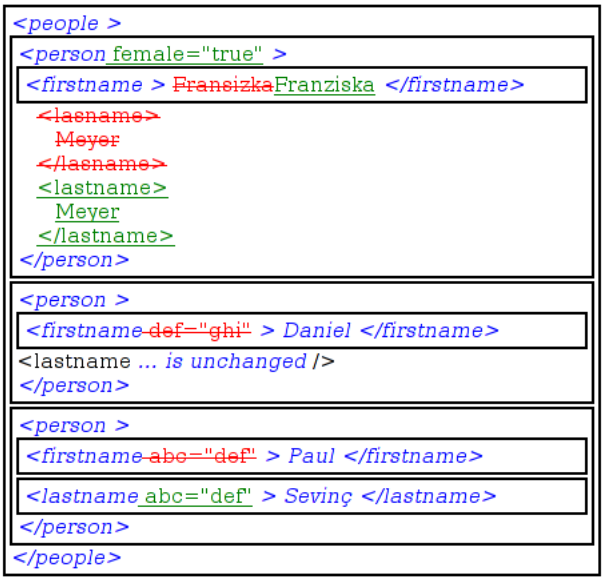


Abbildung 3.1.: Grafisch aufbereitetes Diff von DeltaXML

3.4. XOp

XOperator oder XOp [ep04] ist der Teil des Ercato-Projektes, der Unterschiede in XML-Dokumenten sucht. Es gibt vier Verwendungsmöglichkeiten:

- Command-Line-Tool (das wird auf der Seite auch mit Beispielen illustriert)
- Java-Library für die Verwendung in eigenen Projekten (vorkompiliert)
- Framework für algebraische Operationen auf XML (siehe Beispiel für das Command-Line-Tool)
- Framework, um objektorientierte Vererbung von XML-Dokumenten darzustellen

Mangels Source-Code ist die Implementation nicht bekannt, sonst sieht das Projekt aber sehr interessant aus.

Beispiel

Die Ausgabe für die beiden Beispieldateien (siehe Anhang B.1) sieht folgendermassen aus:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<people xmlns:erc="http://ercato.com/xmlns/ErcatoCore">
 <person female="true">
  <firstname>Franziska</firstname>
  <lasname erc:inherit="delete"/>
  <lastname>Meyer</lastname>
 </person>
 <person>
  <firstname def="erc:inherit=delete">Daniel</firstname>
  <lastname erc:inherit="expand"/>
 </person>
 <person>
  <firstname abc="erc:inherit=delete">Paul</firstname>
  <lastname abc="def" erc:inherit="expand"/>
 </person>
</people>
```

Es mag erstaunlich sein, doch unter Java 1.5.0 machte XOp Probleme, während es unter Java 1.4.2 ohne Probleme läuft.

4. Vergleich der Algorithmen

Dieses Kapitel diskutiert den Vergleich der verschiedenen Algorithmen. Als erstes werden Kriterien definiert, anschliessend werden die Algorithmen vorgestellt und zum Schluss folgt ein Vergleich der Algorithmen anhand der definierten Kriterien.

4.1. Kriterien

Die folgenden Änderungen sollten von den Algorithmen in der einen oder anderen Form behandelt werden. Dabei gibt es Änderungen wie A01, die von jedem Algorithmus zwingend erkannt werden müssen. Andererseits gibt es aber auch Änderungen wie A13, die nicht angezeigt werden sollten.

4.1.1. Einfache Änderungen

Operationen auf Tags gelten auch für Kommentare, die eine spezielle Form von Tags sind. Die Ergebnisse des Vergleiches stehen in der Tabelle 4.1 auf Seite 19, die Legende dazu ist Tabelle 4.2 auf Seite 19.

A01 Einfügen eines Elementes
A02 Einfügen von mehreren Tags unter dem selben Parent-Element
A03 Löschen eines Elementes
A04 Löschen von mehreren Tags unter dem selben Parent-Element
A05 Umbenennen eines Elementes
A06 Umbenennen von mehreren Tags unter dem selben Parent-Element
A07 Einfügen eines Attributs
A08 Einfügen von mehreren Attributen im selben Element
A09 Löschen eines Attributs
A10 Löschen von mehreren Attributen im selben Element
A11 Umbenennen eines Attributs
A12 Umbenennen von mehreren Attributen im selben Element
A13 Ändern der Reihenfolge von mehreren Attributen im selben Element (Da die Reihenfolge der Attribute nicht relevant ist, sollten solche Änderungen nicht angezeigt werden. Ein Häkchen bedeutet, dass die Änderung nicht angezeigt wird. Dieser Punkt zeigt deutlich die Überlegenheit der spezialisierten Algorithmen gegenüber dem zeilenbasierten Diff.)
A14 Ändern des Wertes eines Attributes
A15 Ändern des Wertes von mehreren Attributen im selben Element
A16 Ändern von Text
A17 Ändern von Text in Kommentaren

A18 Ändern von CDATA-Daten
A19 Normales Diff für nicht strukturierte Daten (Text, CDATA, Kommentare)
A20 Ändern des Zeichensatzes bei unveränderten Daten (dies sollte nicht als Änderung erkannt werden)
A23 Schema/DTD herbeigezogen (Whitespace, ...)?
A24 Namespaces
A25 Verschiedene Schreibweisen für leere Tags (`<abc></abc>` sollte das Gleiche sein wie `<abc/>`, das wird oft vom Parser übernommen)

Die Kriterien A23 bis A25 sind mehr vom Parser als vom Algorithmus abhängig.

	A01: Tag einfügen	A02: Tags einfügen	A03: Tag löschen	A04: Tags löschen	A05: Tag umbenennen	A06: Tags umbenennen	A07: Attribut einfügen	A08: Attribute einfügen	A09: Attribut löschen	A10: Attribute löschen	A11: Attribut umbenennen	A12: Attribute umbenennen	A13: Reihenfolge Attribute	A14: Attributwert ändern	A15: Attributwerte ändern	A16: Text ändern	A17: Kommentar ändern	A18: CDATA ändrn	A19: normales Diff	A20: Zeichensatz ändern	A23: Schema/DTD	A24: Namespaces	A25: leere Tags	Beschreibung
FMES (Python)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	(✓)	✓	✓	✓	✓	✓	✓	–	–	–		–	[CRGMW96]
FMES (Java)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–	–	–	[CRGMW96]
X-Diff	✓	✓	✓	✓	±	±	✓	✓	✓	✓	±	±	✓	✓	✓	✓		✓	±		–		–	[WDC]
XyDiff	✓	✓	✓	✓			✓	✓	✓	✓			✓	✓	✓	✓		✓			✓		–	[CAM02]
XOp	✓	✓	✓	✓	±	±	✓	✓	✓	✓	±	±	✓	✓	✓	✓	–	⊕	⊕	–		✓	–	
DeltaXML	✓	✓	✓	✓	±	±	✓	✓	✓	✓	±	±	✓	✓	✓	✓	–	✓	–	–		✓	–	

Tabelle 4.1.: Vergleich der Algorithmen anhand der allgemeinen Kriterien

✓ Änderungen erkannt
⊕ Unveränderte Tags werden auch als geändert erkannt (zusätzliche Updates für Elternknoten der geänderten Knoten)
± Änderungen nur als Folgen von Insert/Delete erkannt
– Änderungen nicht erkannt
✗ nicht unterstützt/Exception
keine Angabe: implementationsabhängig oder aus anderen Gründen unbekannt

Tabelle 4.2.: Legende für die Vergleichstabellen

4.1.2. Umgang mit HTML

HTML haben wir nicht weiter betrachtet, da die Tools, die HTML überhaupt akzeptieren, jeweils versuchen, das Dokument zuerst in XML zu transformieren. Falls sie dabei scheitern, geben sie auf.

4.1.3. Verschiebungen

Verschiebungen zu entdecken ist möglicherweise NP-hart (siehe Abschnitt 7.2 auf Seite 35), weshalb viele Algorithmen darauf verzichten. Trotzdem wäre es sehr sinnvoll, wenn Verschiebungen bei einem Diff als solche zu erkennen wären, da sie sich auch eignen, um Änderungen im verschobenen Teilbaum zu verschleiern. Sie sollten zumindest approximiert werden. Dadurch wird das Diff kleiner, wenn ganze Teilbäume verschoben werden. Das Ergebnis des Vergleiches steht in der Tabelle 4.3 auf Seite 20.

M01 Verschiebung eines unveränderten Teilbaumes

M02 Verschiebung eines veränderten Teilbaumes erkennen und Änderungen an diesem anzeigen

M03 Kopieren eines Teilbaumes

M04 Optionales Ignorieren von Verschiebungen innerhalb einer Sequenz, z. B. bei mehreren `<person>`-Elementen:

```
<addressList deltaxml:ordered="false">
  <person id="1">
    ...
  </person>
  <person id="2">
    ...
  </person>
</addressList>
```

M05 Optionales Ignorieren von Verschiebungen innerhalb einer „choice“ (orderless comparison)

	M01: unveränderter Teilbaum	M02: veränderter Teilbaum	M03: Kopieren	M04: Ignorieren in Sequenz	M05: Ignorieren in Choice
FMES (Python)	±	±	±	±	±
FMES (Java)	✓	✓	±	✓	✓
X-Diff	−	−	±	✓	✓
XyDiff	✓	✓			
XOp	±	±	±	⊕	±
DeltaXML	±	±	±	±	±

Tabelle 4.3.: Vergleich der Algorithmen in Bezug auf Moves

4.2. Algorithmen

4.2.1. FMES

Die von Fast Match/Edit Script (FMES) benutzten Operationen sind die folgenden:

INS((x, l, v), y, k) fügt den neuen Knoten x mit Label l und Wert v als das k-te Kind des Knotens y ein.
DEL(x) löscht den Knoten x.
UPD(x, val) ändert den Wert von x auf den Wert val.
MOV(x, y, k) verschiebt den Knoten x, so dass x das k-te Kind von Knoten y wird.

Python

Das in Python geschriebene Programm `xmldiff` implementiert [CRGMW96] und kann somit Moves entdecken – zumindest theoretisch. Bei der genaueren Analyse stellten wir fest, dass der dafür verantwortliche Teil im Schritt „Fast Match“ nicht implementiert wurde. Der einzige Ort, wo Moves erkannt werden, ist im „AlignChildren“ Schritt, was bei der Auswertung jedoch nie vorkam. In Test A12 wird das Umbenennen von zwei Attributen einmal erkannt und einmal als Insert/Delete ausgegeben. Dies liegt an den Voreinstellungen des Algorithmus und lässt sich durch Ändern der Parameter F und T beheben, wie dies bei der Java-Implementation geschehen ist. Eine detailierte Diskussion der Unterschiede zur Java-Variante ist in Abschnitt 6.3.1 auf Seite 32 zu finden.

Java

Unsere Implementation von [CRGMW96] in Java.

Ein zeilenweises Diff für Test A19 liesse sich leicht implementieren, insbesondere, da der dafür benötigte LCS-Algorithmus bereits in generischer Form vorhanden ist.

4.2.2. X-Diff

X-Diff arbeitet mit ungeordneten Bäumen und begründet das so, dass die Reihenfolge für viele XML-Anwendungen irrelevant ist[1]. Als wichtigste Eigenschaft des Algorithmus wird neben den ungeordneten Bäumen die Effizienz genannt, da das laut [ZSS92] im Allgemeinen NP-komplette Problem der Änderungsfindung auf ungeordneten Bäumen mit Hilfe von starken Annahmen über XML in polynomieller Zeit gelöst wird.

Die von X-Diff benutzten Operationen sind die folgenden:

Insert(x(name, value), y) fügt den neuen Knoten x mit den angegebenen Eigenschaften als Kind von y ein.
Delete(x) löscht den Knoten x.

[1] Als Beispiel wird eine Bücherliste genannt, bei der nur das Vorkommen eines Buches, nicht aber seine Position relevant ist

Update(x, new_value) ändert den Wert von x auf new_value.

Diese Operationen sind nur auf Blättern definiert, wobei für Insert und Delete die Operationen auf Subbäumen relativ einfach mit Hilfe obiger Operationen definiert werden können. Moves werden nicht unterstützt, da die Ordnung zwischen den Kindern irrelevant ist und Moves allgemein durch Insert und Delete dargestellt werden können.

Der Algorithmus ist in [WDC] beschrieben. Allerdings wird beispielsweise die Vorgehensweise fürs Hashing nicht weiter ausgeführt[2].

Die Laufzeit wird für die drei Phasen getrennt angegeben:

Hashing $\mathcal{O}(|T_1| \cdot \log|T_1| + |T_2| \cdot \log|T_2|)$ (Der Logarithmus entsteht, weil die Hashes der Kinder jeweils sortiert werden.)

Matching $\mathcal{O}(|T_1| \cdot |T_2| \cdot \max\{\deg(T_1), \deg(T_2)\} \cdot \log_2(\max\{\deg(T_1), \deg(T_2)\}))$, wobei $\deg(T)$ der maximale Ausgangsgrad eines Knotens im Baum T ist.

EditScript $\mathcal{O}(|T_1| + |T_2|)$

4.2.3. XyDiff

[CAM02] beschreibt den Algorithmus XyDiff, der auch spezifische XML-Eigenschaften wie ID-Attribute auswertet.
Die folgenden Operationen werden unterstützt:

- Insert
- Delete
- Update
- Move (verschiebt grosse Teilbäume)

Laufzeit: $\mathcal{O}(n \cdot \log n)$.

4.2.4. XOp

Wenn grössere Teilbäume, Text-Passagen oder CDATA-Abschnitte geändert wurden, kann es vorkommen, dass der Parent-Node als geändert erkannt wird und sein gesamter Inhalt (auch die unveränderten Kinder) als ein grosses Update ausgegeben wird.

HTML wird falls möglich nach XHTML beziehungsweise XML konvertiert, andernfalls bricht XOp ab.

4.2.5. DeltaXML

DeltaXML überzeugt vor allem durch die lesbare Ausgabe, wenn das HTML-Format gewählt wird. Moves werden nicht erkannt und als Folge von Insert/Delete-Operationen ausgegeben. Von XOp unterscheidet sich DeltaXML in manchen Punkten, wo die Änderungen feinkörniger ausgegeben werden.

[2] Yuan Wang hat uns den Source-Code per Mail zugeschickt, es existiert also eine zugängliche Dokumentation in Form von Code.

5. Handbuch

Kein Weltraum links auf dem Gerät.
-- Quelle unbekannt

5.1. Übersetzen des Quelltextes

Als Build-Tool wird Ant verwendet. Um die Quelltexte zu übersetzen, reicht der Aufruf von `ant` im Verzeichnis `src/trunk/project`. Dies erstellt das Java Archive (JAR) `XMLDiff.jar` im aktuellen Verzeichnis.

5.2. Aufruf auf der Kommandozeile

Um zwei XML-Dateien mit FMES zu vergleichen, kann direkt das JAR-File ausgeführt werden. Dabei werden die beiden XML-Dateien als Kommandozeilenparameter übergeben.

```
java -jar XMLDiff.jar alt.xml neu.xml
```

Die Ausgabe erfolgt im XUpdate-Format über den Standard-Ausgabekanal (stdout). Welche Dateien verglichen werden, gibt das Programm auf dem Standard-Fehlerkanal (stderr) aus. Erweiterte Informationen können für den FMES-Algorithmus eingeschaltet werden, indem die Variable `debug` in der Klasse `FMES`[1] auf `true` gesetzt wird. Sind die zusätzlichen Informationen unerwünscht, so können sie unter UNIX durch Anhängen von `2>/dev/null` beim Aufruf unterdrückt werden. Der Rückgabewert ist 0 genau dann, wenn beide Dokumente gleich sind und 1 sonst. Er kann unter UNIX mittels `echo $?` angezeigt oder in Shell-Ausdrücken verwendet werden:

```
if java -jar XMLDiff.jar alt.xml neu.xml >/dev/null 2>&1; then
  echo Dateien gleich
else
  echo Dateien nicht gleich
fi
```

Um grössere Dateien zu vergleichen, ist es empfehlenswert, der Java Virtual Machine (JVM) mehr Speicher zur Verfügung zu stellen, z. B. mittels:

```
java -Xms64m -Xmx128m -jar XMLDiff.jar alt.xml neu.xml
```

Eine entsprechende Warnung mit angemessenen Werten wird ausgegeben, wenn zur Laufzeit der für das Programm allozierte Speicher knapp wird. Allerdings kann die

[1] `ch.ethz.xmldiff.algorithms.fmes.FMES`

Laufzeit deutlich verbessert werden, wenn die JVM bereits zu Beginn genügend Heap-Speicher allozieren kann.

5.3. Verwendung als Bibliothek

Mehr Möglichkeiten bietet die Implementation, wenn sie als Library verwendet wird. Das gleiche Ergebnis wie oben erhält man mit folgendem Code:

```
import ch.ethz.xmldiff.XMLDiff;

public class First {
  public static void main(String[] args) {
    if (args.length != 2) {
      System.err.print("Usage: java -cp .:XMLDiff.jar ");
      System.err.println("First <file1.xml> <file2.xml>");
      System.exit(1);
    }
    XMLDiff xmldiff = new XMLDiff();
    try {
      StringBuffer output
        = (StringBuffer) xmldiff.run(args[0],args[1]);
      System.out.println(output);
    } catch (Exception e) {
      System.err.println(e.getMessage());
    }
  }
}
```

`XMLDiff.run` gibt in diesem Fall einen `StringBuffer` zurück, der nicht nur direkt ausgegeben, sondern auch beliebig weiterverarbeitet werden kann.

Mehr Möglichkeiten bietet der zweite Konstruktor von `XMLDiff`, der einen Algorithmus, einen Emitter und die Information, ob zusätzlich Move Detection gewünscht ist, entgegen nimmt. Folgender Code macht das Gleiche wie das erste Beispiel, benutzt dazu aber den anderen Konstruktor:

```
import ch.ethz.xmldiff.XMLDiff;
import ch.ethz.xmldiff.algorithms.fmes.FMES;
import ch.ethz.xmldiff.emitter.XUpdateOutput;

public class Second {
  public static void main(String[] args) {
    if (args.length != 2) {
      System.err.print("Usage: java -cp .:XMLDiff.jar ");
      System.err.println("Second <file1.xml> <file2.xml>");
      System.exit(1);
    }
    XMLDiff xmldiff = new XMLDiff(new FMES(),
        new XUpdateOutput(),
        false);
```

```
    try {
      StringBuffer output
        = (StringBuffer) xmldiff.run(args[0],args[1]);
      System.out.println(output);
    } catch (Exception e) {
      System.err.println(e.getMessage());
    }
  }
}
```

Das nächste Beispiel warnt zusätzlich, wenn der Speicher knapp wird. Ausserdem zeigt es, dass `XMLDiff.run` auch ohne `Emitter`[2] funktioniert und in diesem Fall ein `EditScript`[3] zurückgibt:

```
MemoryWarningSystem.init(); // warnt, wenn der Speicher knapp wird

XMLDiff xd = new XMLDiff(new FMES(), null, false);
XUpdateOutput xo = new XUpdateOutput();

EditScript es = (EditScript) xd.run(argv[0], argv[1]);

System.err.println("Result:");
System.out.println(xo.emit(es));

if (es.size() > 0)
  System.exit(1);
```

Der Code ist aus der main-Methode von `XMLDiff`[4] übernommen, welche auch die Aufrufe von der Kommandozeile bearbeitet.

5.4. Eigene Algorithmen

FMES kann ohne Weiteres durch eigene Algorithmen ersetzt werden. Dafür muss der Algorithmus das Interface `DiffAlgorithm`[5] implementieren:

```
public interface DiffAlgorithm {
  public EditScript diff(Document doc1, Document doc2);
  public EnumSet<DiffFeature> getFeatures();
  public String getName();
}
```

Benutzt wird der Algorithmus, indem er im zweiten Beispiel anstelle von FMES übergeben wird. Da der Algorithmus einen Zustand speichern kann, sollte jede Instanz nur einmal verwendet werden.

[2] `ch.ethz.xmldiff.emitter.Emitter`
[3] `ch.ethz.xmldiff.algorithms.EditScript`
[4] `ch.ethz.xmldiff.XMLDiff`
[5] `ch.ethz.xmldiff.algorithms.DiffAlgorithm`

5.4.1. Implementationsdetails

Die drei Methoden, die ein Algorithmus zur Verfügung stellen muss, sind:

- `getName` gibt den Namen des Algorithmus zurück
- `getFeatures` gibt ein paar Metadaten zum Algorithmus zurück
- `diff` vergleicht die beiden angegebenen Dokumente

Die von `getFeatures` zurückgegebenen Informationen sagen aus, ob der Algorithmus Move Detection hat und ob er selbst einen vollständigen Algorithmus implementiert oder nur die Ausgabe eines Algorithmus weiterverarbeitet.

`diff` gibt ein `EditScript`[6] zurück, welches folgende Informationen enthält:

- `public Document getDocument()`
  Liefert das als erstes übergebene `Document` zurück. Bei diesem wurden die neu hinzugefügten Knoten eingefügt und bei Verschiebungen findet sich am Zielort ein Platzhalter.
- `public Change get(Node node)`
  Liefert für den gegebenen Knoten den `Change`[7] zurück. Ein `Change` enthält eine Referenz auf den Knoten (`getNode()`), änderungsspezifische Zusatzinformationen und natürlich die Art der Änderung (`getType()`), welche eine der in der Enumeration `ChangeType` definierten Operationen ist: `ADD`, `REMOVE`, `RENAME`, `UPDATE`, `MOVE` oder `MOVE_DESTINATION`. `MOVE_DESTINATION` ist keine eigentliche Operation, sondern markiert lediglich den Platzhalter am Ziel einer Verschiebung.
- `... implements Iterable<Change>`
  Über die `Changes` im `EditScript` kann auch iteriert werden, was sich der Emitter `XUpdateOutput`[8] zu Nutze macht und es so vermeidet, den ganzen Baum nach Änderungen abzusuchen.

5.4.2. Algorithmen testen

Die Algorithmen, die getestet werden, stehen in `algorithms.properties`, wobei jeweils nur der Key relevant ist, der dem Paket-Pfad inklusive Klassennamen des Algorithmus entsprechen muss.

Um die Tests aufzurufen, muss die Klasse `AllTests`[9] als JUnit-TestCase ausgeführt werden.

5.5. Eigene Emitter

Wie der Algorithmus kann auch die Ausgabe durch eine eigene Implementation ersetzt werden. Eigene Emitter müssen das Interface `Emitter`[10] implementieren:

[6] `ch.ethz.xmldiff.algorithms.EditScript`
[7] `ch.ethz.xmldiff.algorithms.Change`
[8] `ch.ethz.xmldiff.emitter.XUpdateOutput`
[9] `ch.ethz.xmldiff.AllTests`
[10] `ch.ethz.xmldiff.emitter.Emitter`

```
public interface Emitter {
  public StringBuffer emit(EditScript es) throws IOException;
}
```

Was ein Emitter genau mit dem `EditScript` macht bleibt der Implementation überlassen. Es wäre beispielsweise auch möglich, ein Fenster zu öffnen, indem die Änderungen visualisiert werden.

6. Implementation

6.1. Design

Wir wollten weder die Algorithmen noch das Ausgabeformat fix vorgeben. Daher entschieden wir uns für einen modularen Aufbau, der aus den Komponenten Parser, Algorithmus, optionale Move Detection und Emitter besteht. Abbildung 6.1 auf Seite 28 beschreibt, wie die Daten von der Ein- bis zur Ausgabe weitergegeben werden.

Abbildung 6.1.: Zusammenspiel der Komponenten

Der Parser ist auswechselbar, er muss einfach das DOM-Modell implementieren. Algorithmen können beliebig ausgetauscht werden, so lange sie das vorgegebene Interface implementieren. Die Move Detection könnte wie in Abschnitt 7.2.3 auf Seite 37 beschrieben implementiert werden und für Algorithmen, die keine Moves erkennen, zusätzlich hinzugenommen werden. Für die Emitter gilt das Gleiche wie für die Algorithmen.

Um XML-Dokumente intern zu repräsentieren, haben wir als minimale Anforderungen die in Abbildung 6.2 auf Seite 29 aufgezeichnete Datenstruktur definiert, die das gesamte Dokument im Speicher repräsentiert.

Die Knoten repräsentieren die XML-Elemente und machen die Eigenschaften dieser wie Attribute oder enthaltener Text für das Programm zugänglich. Die roten Pfeile bilden einen Baum, allerdings sind nur die durchgezogenen Pfeile als tatsächliche Referenzen vorhanden. Von einem Vaterknoten aus sind alle Kindknoten erreichbar, indem man zuerst dem roten Pfeil zum ersten Kindknoten folgt und dann der verlinkten Liste (dunkelblau) mit allen Kindern folgt.

Um die Navigation zu erleichtern ist es zudem möglich, von jedem Knoten aus zu seinem Vaterknoten zu gelangen, indem man dem hellblauen Parentlink folgt.

6.2. Parser

Da der Parser einen sehr starken Einfluss auf die Internal Representation (IR) haben kann, überlegten wir uns, welche Parsermodelle in Frage kämen. Einen bereits existierenden Parser zu verwenden bietet zahlreiche Vorteile:

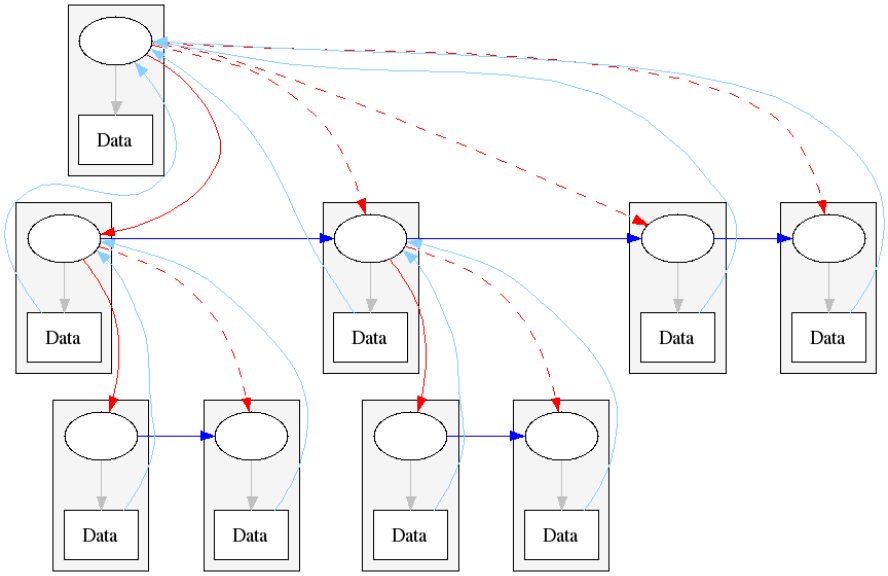


Abbildung 6.2.: Datenstruktur für die interne Repräsentation von XML-Dokumenten

- Test, ob das Dokument well-formed ist
- Automatische Umwandlung des Zeichensatzes
- Behandlung von CDATA-Abschnitten
- Verwaltung von Namespaces für jedes Element
- Validierung gegen ein Schema
- Typisierte Attribute

6.2.1. SAX

Das Simple API for XML (SAX) ist ein Event-basiertes, leichtgewichtiges API für XML. Da SAX einen Parser modelliert und lediglich für jede Klasse von Elementen eine eigene Callback-Methode aufruft, kommt die Baumstruktur von XML nicht zur Geltung. Das Aufbauen einer eigenen, möglicherweise sehr spezialisierten IR bleibt dem Programm selbst überlassen.

Durch das Push-Modell wird SAX sehr effizient, und der Speicherverbrauch ist moderat, was es für die Bearbeitung von grosse Dokumente oft zur einzig gangbaren Lösung macht. Auch für Streaming-Anwendungen wie Filter, die nur sehr lokalen Zugriff und keine Vorwärtsreferenzen brauchen, ist SAX interessant.

Im Vergleich zu auf SAX-Parsern aufsetzenden Modellen wie DOM oder JDOM ist das Fehlen einer IR für unsere Anwendung von Nachteil.

6.2.2. DOM

Das Document Object Model (DOM) modelliert die Baumstruktur von XML, welche beliebig manipuliert werden kann, indem Methoden auf den Klassen, die die Knoten repräsentieren, aufgerufen werden. Der Speicherverbrauch ist daher zwar grösser als bei SAX, dafür ist der gleichzeitige Zugriff auf mehrere Teilbäume oder gar mehrere Dokumente ohne Weiteres möglich. Die IR von DOM bildet XML in seiner ganzen Komplexität ab und begünstigt die Entwicklung eher monolithischer Anwendungen.

Wir haben uns wegen seiner Flexibilität und der grossen Ähnlichkeit zu den von uns aufgestellten Anforderungen an eine IR und wegen der internen Konsistenzprüfungen für DOM entschieden.

Als konkretes Application Programming Interface (API) haben wir das Java API for XML Processing (JAXP) genommen, da es so möglich ist, den Parser auszutauschen. Andere Kapselungen von DOM wie JDOM oder dom4j erachten wir als weniger geeignet, da sie dem Programmierer zwar ein bequemeres API für spezielle XML-Daten anbieten, die Unterstützung für allgemeine XML-Daten aber nicht grösser ist als bei JAXP.

6.3. Algorithmen

Ein Diff-Algorithmus hat die Aufgabe, die Unterschiede zwischen zwei DOM-Bäumen festzustellen. Die Ausgabe dieser Änderungen in einem bestimmten Format ist die Aufgabe des Emitters. In der Literatur wird für die Repräsentation der Änderungen ein Edit-Script, also eine Auflistung aller Änderungen, benutzt. Dies ist jedoch dann unvorteilhaft, wenn die Änderung im Kontext der ursprünglichen Daten dargestellt werden soll. Man stelle sich z. B. ein Programm vor, das dem Benutzer erlaubt, durch den XML-Baum zu navigieren und ihm dabei die veränderten Knoten farblich hervorhebt.

Wir haben uns daher für eine zweigleisige Repräsentation der Änderungen entschieden:

- Einerseits soll es möglich sein, einen DOM-Baum zu traversieren und für jeden Knoten feststellen zu können, ob er eingefügt, umbenannt, gelöscht, verschoben oder sein Inhalt geändert wurde.
- Andererseits soll es möglich sein, einfach über die Liste aller Änderungen zu iterieren.

Wir benutzen den ersten DOM-Baum als Grundlage. Die `get`-Methode der Klasse `EditScript` liefert für jeden Knoten die ihm zugeordnete Änderung. Um auch neu eingefügte Knoten erfassen zu können, wird eine Kopie aus dem zweiten DOM-Baum an entsprechender Stelle eingefügt. Verschobene Knoten werden als verschoben markiert, aber nicht entfernt. An ihrem Bestimmungsort wird ein Platzhalter eingefügt und als Ziel einer Verschiebung markiert.

Somit ist es möglich, den DOM-Baum zu traversieren und für jeden Knoten festzustellen, ob er eingefügt, umbenannt, gelöscht, verschoben oder sein Inhalt verändert wurde.

Zusätzlich implementiert das EditScript das Iterable-Interface, wodurch es möglich wird, über die Änderungen in der richtigen Reihenfolge zu iterieren.

6.3.1. Fast Match / Edit Script

Als einer der wenigen Algorithmen, die wir gefunden haben, implementiert der in [CRGMW96] beschriebene Algorithmus das Erkennen von Verschiebungen.

Fast Match

In einer ersten Phase werden die Blätter der beiden Bäume in zwei Listen gesammelt und die LCS berechnet. Dazu implementierten wir den Algorithmus aus [Mye86] als Template-Methode. Die Methode nimmt neben den beiden Listen beim Aufruf noch eine funktionale Klasse für den Vergleich der Listenelemente entgegen.

Der Vergleich der Blätter wird im Wesentlichen durch die Funktion `quickRatio` durchgeführt, die den Wert (Text) zweier Blätter auf das Intervall $[0, 1]$ abbildet und damit angibt, wie ähnlich die beiden Blätter sind. `quickRatio` liefert recht brauchbare Ergebnisse, könnte aber bei grösseren Texten schnell schlechter werden, da die Häufigkeit der einzelnen Buchstaben in den beiden Texten miteinander verglichen wird. Beliebige Permutationen haben somit die Ähnlichkeit 1.0.

Für den Vergleich der inneren Knoten wird eine andere funktionale Klasse verwendet: `InnerNodeEqual`. Zwei Knoten werden als hinreichend ähnlich betrachtet, wenn ein gewisser Prozentsatz ihrer Kinder einander zugeordnet werden konnten. Während die Python-Implementation für die Berechnung dieses Wertes jedes mal über das gesamte Matching iteriert, betrachten wir nur die Kinder in einem der beiden Teilbäume und zählen die Anzahl der Partner, die sich im anderen Teilbaum befinden. Da für jeden Vergleich ein (potentiell grosser) Teilbaum traversiert werden muss, besteht hier sicher noch genügend Potential für Optimierungen.

Edit Script

Für das Generieren des Edit-Scripts wird der zweite Baum mittels Breitensuche durchlaufen. Für jeden Knoten wird zuerst überprüft, ob er im Matching enthalten ist und nötigenfalls eine Insert-Operation gestartet. Hier stiessen wir auf ein Problem mit der verwendeten DOM-Implementierung: ein `Document` darf nicht mehr als ein Kind haben. Genau das ist aber vorübergehend nötig, wenn die Dokumente zu verschieden sind und somit der Wurzelknoten ausgetauscht wird. Wir geben in diesem Fall eine Warnung aus und fügen den neuen Knoten nur ins `EditScript`, nicht aber in den DOM-Baum ein – die Ausgabe durch das Iterieren über das `EditScript` bleibt dadurch unverändert korrekt.

Die Ausgabe von Updates, Verschiebungen und Umbenennungen funktioniert wie im Paper beschrieben, allerdings mit dem Unterschied, dass wir den Baum nur wenn unbedingt nötig verändern.

Für das Ausgeben der gelöschten Knoten verwenden wir eine Preorder-Traversierung und nicht Postorder, wie im Paper vorgeschlagen. Dies hat den Vorteil, dass ganze Teilbäume mit einer Lösch-Operation gelöscht werden können und nicht für jeden Knoten unterhalb eines Gelöschten eine einzelne Operation ausgegeben wird.

Vergleich mit der Python-Implementierung

Unter `http://www.logilab.org/projects/xmldiff/` kann die neueste Version der Python-Implementierung FMES von Logilab gefunden werden. Wir beziehen uns im Folgenden auf die Version 0.6.7 vom 4. Mai 2005.

Wir haben neben den normalen Tests für den Vergleich in Abschnitt 4 auf Seite 18 auch die persönlichen Bookmarks von Daniel Hottinger verwendet. Das Format ist die vom Browser Galeon verwendete XML Bookmarks Exchange Language in der Version 1.0. Die Document Type Definition (DTD) kann unter `http://pyxml.sourceforge.net/topics/dtds/xbel-1.0.dtd` gefunden werden. Die beiden Dateien repräsentieren die Änderungen von ca. 2 Wochen, haben je eine Grösse von ca. 820 Kilobytes, enthalten ca. 37500 Tags (öffnende und schliessende zusammen) und insgesamt 6400 Attribute.

Zuerst untersuchten wir, ob – und wie – sich die Ausgabe der beiden FMES-Implementierungen unterscheidet. Erfreulicherweise sind die Unterschiede gering. Die Unterschiede liegen hauptsächlich in der Formatierung, die bei der Java-Implementierung lesbarer ausfällt. Weitere Unterschiede sind verschiedene Formulierungen für die selbe Änderung: ein „append“ vor dem ersten Kind ist das gleiche wie ein „insert-before“ mit einer Referenz auf das erste Kind.

Unangenehm fällt auf, dass die Python-Implementierungen keine Verschiebungen entdeckt hat. Das liegt daran, dass der dafür zuständige Schritt 2(e) von *FastMatch* nicht implementiert wurde. Stattdessen werden Knoten eingefügt und andere gelöscht.

Drastisch fällt der Laufzeitunterschied der beiden Implementierungen aus. Auf einem AMD Athlon 64 3200+ Prozessor mit 1 Gigabyte RAM rechnet die Python-Variante ganze 9 Minuten und 50 Sekunden (davon 0.5 Sekunden für Systemcalls), während die Java-Implementation nur gerade 20 Sekunden braucht (davon 0.3 Sekunden für Systemcalls). Der Hauptgrund für dieses frappante Ergebnis dürfte wohl sein, dass die Java-Implementation wo immer möglich `IdentityHashMap`s einsetzt. Diese reduzieren deutlich die Zeit, die für einen Lookup nötig ist. Auch können die Hash-Werte aus 4 bis 8 Bytes berechnet werden (`IdentityHashMap`s vergleichen Objektidentitäten und benutzen nicht `equals()`). Die Python-Implementation hingegen verwendet für das Mapping eine Liste und muss diese bei jedem Lookup durchsuchen. Um dies zu beschleunigen, wurden die entsprechenden Funktionen in C geschrieben, was das Problem aber grundsätzlich nicht löst. Im Gegenteil, C-Funktionen bringen eine für eine Scriptsprache unnötige Abhängigkeit von der verwendeten Architektur mit sich. So lief die Python-Variante anfänglich auf AMD 64 überhaupt nicht. Im Laufe der Implementierung wurde der Fehler offensichtlich: in `maplookup.c` wurden Objektpointer nach `int` gecastet, was auf 64-Bit Architekturen schief geht, da `int`s 4 Bytes, Pointer jedoch 8 Bytes gross sind. Wir haben den Fehler behoben und Logilab einen Patch ge-

schickt, welcher auch in Anhang B.2 auf Seite 43 zu finden ist und in die nächste Version aufgenommen werden wird.

Fairerweise muss gesagt werden, dass für diesen Test bei der Java-Implementierung die anfängliche und maximale Grösse des Heaps auf 128 MB vergrössert werden musste. Wird anfänglich ein Heap der Grösse 64 MB verwendet, so verlängert sich die Laufzeit durch die nachträglich nötige Vergrösserung des Heaps auf ca. 40 Sekunden. Hier besteht also noch Optimierungsbedarf.

6.3.2. Move-Detection als Decorator

Um Algorithmen, die keine Verschiebungen erkennen können, um diese Fähigkeit zu erweitern, kann das EditScript nachbearbeitet werden. Dazu kann der in Abschnitt 7.2.3 auf Seite 37 beschriebe Algorithmus verwendet werden. Er kapselt als Decorator den primitiven Algorithmus, fängt den Aufruf der Diff-Methode ab und bereitet das Ergebnis so auf, dass Verschiebungen erkannt werden.

6.4. Emitter

Aufgabe des Emitters ist es, das EditScript für die Ausgabe aufzubereiten. Mögliche textuelle Darstellungsformen haben wir bereits in Abschnitt 3 auf Seite 12 kennen gelernt.

Wir haben bei unserer Implementation einen Emitter für XUpdate geschrieben (siehe Abschnitt 3.2 auf Seite 13), da das Format einerseits für Menschen gut lesbar ist und andererseits als vollwertiges XML weiterverarbeitet werden kann. Die Implementation iteriert über das EditScript und gibt jede Änderung aus. Da XUpdate den Ort der Änderung als XPath-Ausdruck beschreibt, war es nötig, diesen für einen gegebenen Knoten berechnen zu können. Dabei erwies sich der Parent-Link im DOM-Baum als überaus nützlich. Bei der Berechnung der Indizes für den XPath galt es zudem zu beachten, dass Knoten hinzugefügt, umbenannt oder gelöscht wurden. Die korrekte Berechnung dieser Indizes war daher nicht ganz trivial, funktioniert aber mittlerweile zu unserer vollsten Zufriedenheit.

Mit der rein textuellen Ausgabe sind die Möglichkeiten unseres Frameworks aber bei weitem nicht erschöpft. So ist es leicht möglich, den DOM-Baum in einem GUI darzustellen. Knoten könnten beliebig auf und wieder zu geklappt werden und eine farbliche Hervorhebung könnte die Änderungen andeuten. Es wäre auch möglich, ein PDF- oder HTML-Output zu generieren, das das XML und die Änderungen schön aufbereitet darstellt, so wie in Abbildung 3.1 auf Seite 16 am Beispiel von DeltaXML dargestellt.

Auch könnte die Ausgabe in XSL erfolgen, womit es möglich würde, einen beliebigen XSL-Prozessor zum Patchen einzusetzen (siehe Abschnitt 7.1 auf Seite 35).

6.5. Tests

Unittests stellen in komplexen Projekten sicher, dass durch Änderungen keine neuen Fehler eingeführt werden. Neben einfachen Tests wie denjenigen für den LCS-Algorithmus, wo einfache Zeichenketten verwendet werden konnten, galt es auch, komplexe Tests für FMES und die Emitter zu schreiben. Da die Schnittstelle zwischen Algorithmen und Emittern eine recht komplexe Datenstruktur ist, beschlossen wir, die beiden Module zusammen zu testen.

In `src/testdata/` befinden sich die XML-Dateien, die wir für die Evaluation der Algorithmen verwendet haben. Die Namen sind nach folgendem Schema aufgebaut:

```
Kriterium-BeschreibungVersion.xml
```

Kriterium ist die in Abschnitt 4.1 auf Seite 18 angegebene Nummer. Die Beschreibung drückt aus, was getestet wird und die Version ist 1 für das Original und 2 für die veränderte Kopie. Um die Tests zu automatisieren, existiert noch eine dritte Variante mit Version 3; sie enthält die gewünschte Ausgabe im XUpdate-Format. Mittels der Klasse `AssertEqual` wird dann die Ausgabe des Emitters mit dieser Datei auf DOM-Ebene verglichen.

Da nicht alle Kriterien auch Testcases sein sollten, muss jeder Testcase in den `enum TestFile` im Package `ch.ethz.xmldiff.algorithms` eingetragen werden. Das Test-Framework übernimmt dann das automatische Erzeugen einer Instanz mit dem Namen

```
Methodenname<Kriterium-Beschreibung>
```

unterhalb des entsprechenden Algorithmus in der Testhierarchie. Um alle Tests zu starten, muss die Klasse `AllTests` im Package `ch.ethz.xmldiff` als Testcase ausgeführt werden.

7. Überlegungen zu weiteren Themen

7.1. XSL – ist der Einsatz sinnvoll?

An einer Sitzung tauchte die Idee auf, XSL einzusetzen. Die möglichen Einsatzgebiete wären die Implementation eines Diff-Algorithmus in XSL, XSL als Ausgabeformat sowie die Verwendung von XSL, um die Ausgabe lesbarer zu gestalten (so wie das DeltaXML macht, siehe Abschnitt 3.3 auf Seite 14).

7.1.1. Diff-Algorithmus in XSL

Diese Idee verfolgten wir nicht weiter, weil XSL grundsätzlich auf einem XML-Dokument arbeitet. Es ist zwar möglich, ein weiteres Dokument zu laden, aber da die Sprache doch eher auf Iteration ausgelegt ist, zogen wir die bereits vertraute Sprache Java vor.

7.1.2. XSL als Ausgabeformat

Die Option, XSL als Ausgabeformat zu benutzen, steht noch offen. Eine Möglichkeit wäre beispielsweise, einen speziellen Emitter für XSL zu schreiben, so wie wir auch XUpdate ausgeben können. Dadurch wird es möglich, die Änderungen mittels eines beliebigen XSL-Prozessors auf das Originaldokument anzuwenden und dadurch das veränderte Dokument zu erhalten, ähnlich wie das bereits mit dem UNIX-Programm patch(1) für normale Diffs möglich ist.

7.1.3. XSL für die Transformation der Ausgabe

Die Umsetzung von XUpdate in XSL bedeutet wieder, dass mindestens zwei Dokumente parallel verarbeitet werden. Grundsätzlich ist es denkbar, das Ursprungsdokument zu verarbeiten und bei jedem Knoten zu schauen, ob im XUpdate-Dokument eine Änderung dazu vorhanden ist. Doch dazu wären genauere Kenntnisse von XSL nötig.

7.2. Sind Moves NP-hart?

7.2.1. Ausgangslage: Die Quellen

Bei der Recherche stiessen wir in [CAH02, Seiten 3 und 9] auf die These, dass die Erkennung von Moves das Generieren eines minimalen Edit-Scripts NP-hart macht:

> We recall that most formulations of the change detection problem with *move* operations are NP-hard [ZWS95]. So the drawback of detecting *moves* is that such algorithms will only approximate the minimum edit script, whereas most algorithms on ordered trees provide a minimal editing script in quadratic time.
>
> [...]
>
> The main reason why few diff algorithm supporting *move* operations have been developed earlier is that most formulations of the tree diff problem are NP-hard [ZWS95], [CGM97] (by reduction from the „exact cover by three-sets“). One may want to convert in editing scripts a pair of *insert* and *delete* operations into a single *move*. The result obtained is in general not minimal, unless the cost of *move* operations is strictly identical to the total cost of *insert* and *delete*.

Das heisst, dass die Erkennung von *insert* und *delete* in XML nicht NP-hart ist, jedoch die Hinzunahme der neuen Operation *move* das Problem NP-hart macht.

Als erstes verfolgten wir die Referenz [CGM97], wo auf den Seiten 27 und 36 die folgenden Aussagen zu finden sind:

> There is a good reason why difference algorithms with the features we have described here have not been developed earlier, even though they are clearly desirable. The reason is the inherent complexity of the problem; one can show that the problem is NP-hard (By [sic!] reduction from the „exact cover by three-sets“ problem).
>
> [...]
>
> A proof of the NP-hardness of a similar change detection problem (using insertion, deletion and label-update) for unsorted trees is presented in [ZWS95], which als presents an algorithm for a restricted version of the change detection problem.

Dabei umfassen die Features die Operationen *insert*, *delete*, *update*, *move* und *copy*.

Der erwähnte Beweis in [ZWS95] geht allerdings von einem etwas anderen Problem aus. Die Aussage ist, dass die Bestimmung der Grösse des minimalen Edit-Scripts für CUAL-Graphen mit den Operationen *relabel*, *delete* und *insert* NP-hart ist.

7.2.2. Unklarheiten

Nun stellen sich ein paar Fragen:

- Was haben die im Beweis behandelten Operationen mit *move* zu tun? Der Beweis behandelt *insert*, *delete* und *relabeling*, *moves* werden da gar nicht betrachtet.
- Kann die Tatsache, dass es sich bei XML um gerichtete Bäume mit bekannter Wurzel handelt, für die Laufzeit nicht ausgenutzt werden? Beziehungsweise wie kann man aus dem Beweis folgern, dass es für XML mit nur *delete* und *insert* nicht NP-hart ist und mit *move* plötzlich schon?

- Kann mit der Zusammenfassung von *insert* und *delete* zu *move* überhaupt ein minimales Edit-Script erreicht werden? Möglich sein sollte es, da ein *move* als einzige Änderung als solches erkannt werden kann und somit aus einem *insert* und *delete* ein minimales Edit-Script bestehend aus einem *move* generiert werden kann.

Weitere Bemerkungen:

- Die Idee, den Graphen im Beweis einfach eine willkürliche Wurzel zu geben, funktioniert nicht, da die Wurzel für beide Graphen bereits ein sinnvolles Mapping der Knoten voraussetzt.
- Das *relabeling* entspricht nicht einem *move*, sondern der Umbenennung eines XML-Tags.

7.2.3. Einfache Move Detection

Die naheliegendste Variante ist, alle Inserts mit allen Deletes zu vergleichen und identische einander zuzuordnen. Dabei können ähnliche rekursiv weiterverglichen werden, um die Änderungen möglichst klein zu halten.

Als Erweiterung könnte zuerst die LCS zwischen Inserts und Deletes gesucht werden, dann müssen nur noch die übrigbleibenden Änderungen genauer angeschaut werden. Im stark vereinfachten Beispiel mit zwei Dokumenten (links und rechts) in Abbildung 7.1 repräsentieren die leeren Tags jeweils ganze Subbäume. Im ersten Schritt wurde die LCS bestimmt und blau markiert. Im zweiten Schritt wurden die restlichen Elemente miteinander verglichen. Dabei wurde festgestellt, dass das rot markierte Element zwar nicht zur LCS gehört, aber trotzdem in beiden Dokumenten vorkommt. Das heisst, dass es verschoben wurde.

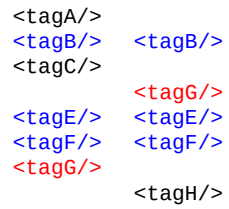


Abbildung 7.1.: Beispiel für eine einfache Move Detection

Möglich (aber auch aufwendig) wäre nun ein rekursives Vorgehen, bei dem die Kinder der noch schwarzen Knoten miteinander verglichen werden.

8. Schlussfolgerungen

Änderungen in strukturierten Daten zu finden ist ein nicht triviales Problem. Wo XML-Datenbanken mittels einer eindeutigen ID Knoten einander zuordnen, haben die allgemeineren Algorithmen für XML-Diff keine solchen Anhaltspunkte. Dementsprechend unterschiedlich sind die Lösungsansätze: Während FMES aus [CRGMW96] die hierarchischen Daten für das Matching in flache Sequenzen transformiert um darauf die LCS zu berechnen, arbeitet X-Diff aus [WDC] generell auf ungeordneten Bäumen und muss starke Annahmen treffen, um eine polynomielle Laufzeit zu erreichen. Es gibt auch kommerzielle Projekte wie DeltaXML, bei denen die eingesetzten Algorithmen nicht oder nur eingeschränkt einsehbar sind, welche sich aber sehr intensiv um eine akademische Nutzung bemühen. Nur sehr wenige Algorithmen unterstützen das Erkennen von Verschiebungen, da es einen Mehraufwand mit sich bringt und als NP-hart vermutet wird. Dies können wir jedoch nicht ganz nachvollziehen, da das in [CAH02] zitierte Paper ein allgemeineres Problem für eine Anwendung aus der Biologie behandelt (siehe Abschnitt 7.2 auf Seite 35). Einige Algorithmen beschränken sich zudem auf allgemeine, baumartige Strukturen und überlassen die Anpassung an XML der Implementation, was dazu führt, dass diese sich nicht immer identisch verhalten.

Unsere Implementation stellt neben dem Algorithmus FMES ein Framework zur Verfügung, mit dem eigene Algorithmen einfach getestet werden können. Durch die Trennung von Eingabe-, Diff- und Ausgabeteil ist es möglich, die einzelnen Komponenten beliebig auszutauschen. Als Library kann das JAR auch in eigenen Projekten verwendet werden. Die XML-Dateien für den Algorithmenvergleich können zum Testen von FMES als auch von eigenen Implementationen verwendet werden. Die dafür nötigen Unit-Tests sind leicht anzupassen.

8.1. Ausblick

Wir haben einen Algorithmus (nämlich FMES, siehe Abschnitt 4.2.1 auf Seite 21) implementiert. Als Basis für die Implementation weiterer Algorithmen bieten wir ein Framework, mit dem die Algorithmen austauschbar eingesetzt sowie getestet werden können.

Auch weitere Ausgabeformate, wie sie in Kapitel 3 auf Seite 12 beschrieben werden, können ohne Probleme hinzugefügt werden.

Eher theoretisch orientierte Leute möchten sich möglicherweise mit der im Abschnitt 7.2 auf Seite 35 vorgestellten Frage beschäftigen, ob die Generierung von minimalen Editscripts, die Moves enthalten, tatsächlich NP-hart ist.

A. XUpdate DTD

```
<!ENTITY % commands "
    xupdate:variable
  | xupdate:insert-before
  | xupdate:insert-after
  | xupdate:append
  | xupdate:update
  | xupdate:remove
  | xupdate:rename
">

<!ENTITY % instructions "
    xupdate:element
  | xupdate:attribute
  | xupdate:text
  | xupdate:processing-instruction
  | xupdate:comment
">

<!ENTITY % qname "NMTOKEN">

<!ENTITY % template "
 (#PCDATA
  | %instructions;)*
">

<!ELEMENT xupdate:modifications (%commands;)*>
<!ATTLIST xupdate:modifications
   id          ID      #IMPLIED
   version     NMTOKEN #REQUIRED
   xmlns:xupdate CDATA   #FIXED "http://www.xmldb.org/xupdate"
>

<!ELEMENT xupdate:insert-before (%instructions;)*>
<!ATTLIST xupdate:insert
   select      CDATA   #REQUIRED
>

<!ELEMENT xupdate:insert-after (%instructions;)*>
<!ATTLIST xupdate:insert
   select      CDATA   #REQUIRED
>
```

```
<!ELEMENT xupdate:append (%instructions;)*>
<!ATTLIST xupdate:insert
   select      CDATA   #REQUIRED
   child       CDATA   #IMPLIED
>

<!ELEMENT xupdate:element %template;>
<!ATTLIST xupdate:element
   name        %qname; #REQUIRED
   namespace   CDATA   #IMPLIED
>

<!ELEMENT xupdate:attribute (#PCDATA)>
<!ATTLIST xupdate:attribute
   name        %qname; #REQUIRED
   namespace   CDATA   #IMPLIED
>

<!ELEMENT xupdate:text (#PCDATA)>

<!ELEMENT xupdate:processing-instruction (#PCDATA)>
<!ATTLIST xupdate:processing-instruction
   name        NMTOKEN #REQUIRED
>

<!ELEMENT xupdate:update (#PCDATA)>
<!ATTLIST xupdate:update
   select      CDATA   #REQUIRED
>

<!ELEMENT xupdate:remove EMPTY>
<!ATTLIST xupdate:remove
   select      CDATA   #REQUIRED
>

<!ELEMENT xupdate:rename (#PCDATA)>
<!ATTLIST xupdate:rename
   select      CDATA   #REQUIRED
>

<!ELEMENT xupdate:variable (#PCDATA)*>
<!ATTLIST xupdate:variable
   name        NMTOKEN #REQUIRED
   select      CDATA   #IMPLIED
>

<!ELEMENT xupdate:value-of EMPTY>
<!ATTLIST xupdate:value-of
   select      CDATA   #REQUIRED
```

```
>

<!ELEMENT xupdate:if %template;>
<!ATTLIST xupdate:if
   test        CDATA   #REQUIRED
>
```

B. Listings

B.1. Beispiel-Dateien

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<people>
 <person >
  <firstname>Fransizka</firstname>
  <lasname>Meyer</lasname>
 </person>
 <person>
  <firstname def="ghi">Daniel</firstname>
  <lastname>Hottinger</lastname>
 </person>
 <person>
  <firstname abc="def">Paul</firstname>
  <lastname>Sevinç</lastname>
 </person>
</people>
```

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<people>
 <person female="true">
  <firstname>Franziska</firstname>
  <lastname>Meyer</lastname>
 </person>
 <person>
  <firstname>Daniel</firstname>
  <lastname>Hottinger</lastname>
 </person>
 <person>
  <firstname>Paul</firstname>
  <lastname abc="def">Sevinç</lastname>
 </person>
</people>
```

B.2. Patch für xmldiff

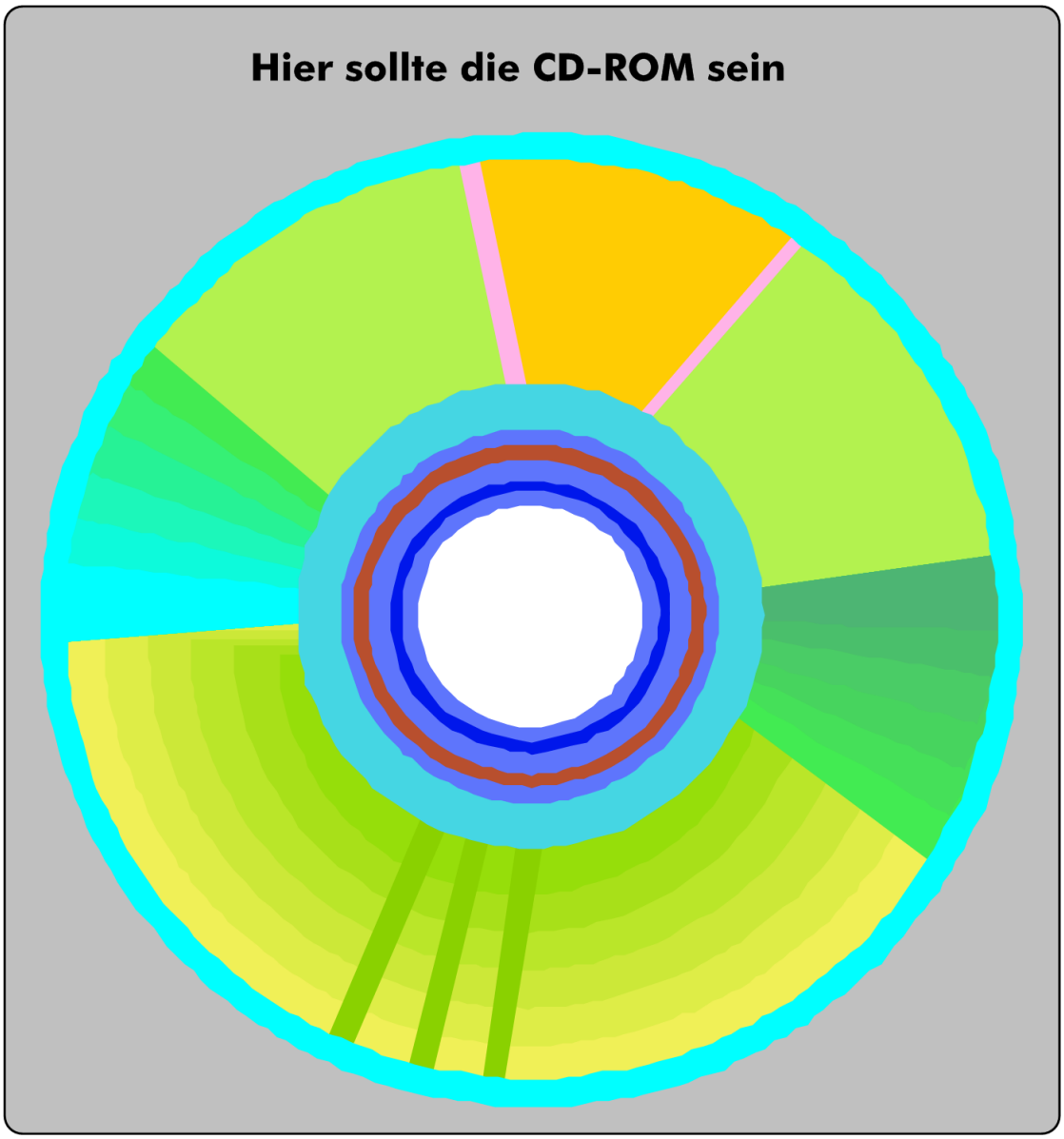
```
diff -Nru xmldiff-0.6.7-orig/extensions/maplookup.c xmldiff-0.6.7/extensions/maplookup.c
--- xmldiff-0.6.7-orig/extensions/maplookup.c    2005-05-03 17:28:45.000000000 +0200
+++ xmldiff-0.6.7/extensions/maplookup.c         2005-06-24 18:08:39.327922316 +0200
@@ -1,5 +1,6 @@
 #include <Python.h>
 #include <stdio.h>
+#include <inttypes.h>

 char * __revision__ = "$Id: xmldiff-0.6.7_amd64.diff 154 2005-06-24 19:42:14Z hotti $";

@@ -156,11 +157,19 @@
     {
         PyObject *key ;
         couple = PyList_GET_ITEM(_mapping, i) ;
-        key = Py_BuildValue("(i,i)", (int)node1, (int)PyTuple_GET_ITEM(couple, 0)) ;
+#if __WORDSIZE == 64
+        key = Py_BuildValue("(l,l)", (size_t)node1, (size_t)PyTuple_GET_ITEM(couple, 0)) ;
+#else
+        key = Py_BuildValue("(i,i)", (size_t)node1, (size_t)PyTuple_GET_ITEM(couple, 0)) ;
+#endif
         if (PyDict_GetItem(_dict1, key) != NULL)
             {
                 Py_DECREF(key) ;
-                key = Py_BuildValue("(i,i)", (int)node2, (int)PyTuple_GET_ITEM(couple, 1)) ;
+#if __WORDSIZE == 64
+                key = Py_BuildValue("(l,l)", (size_t)node2, (size_t)PyTuple_GET_ITEM(couple, 1)) ;
+#else
+                key = Py_BuildValue("(i,i)", (size_t)node2, (size_t)PyTuple_GET_ITEM(couple, 1)) ;
+#endif
                 if (PyDict_GetItem(_dict2, key) != NULL)
                     {
                         seq_num += 1 ;
```

C. CD-ROM

C.1. Die CD-ROM



C.2. Inhalt der CD-ROM

Die CD-ROM enthält folgende Verzeichnisse:

doc Der Bericht, die Präsentation und die Handouts.

jar Das JAR unserer Arbeit.

papers Die Papers aus den Referenzen als PDF oder PS.

src Der Quellcode:

- **project** für das Projekt,
- **tests** für die Tests und
- **scripts** weitere Scripts.

svn Subversion-spezifische Dateien:

- **checkout** Kompletter Checkout des Repositories.
- **repository** Das Repository selbst.

testdata Die XML-Testdateien, die wir bei der Evaluation benutzt haben.

D. Aufgabenstellung

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Semesterarbeit for one student

Diff and XML-Diff Algorithms

Contact: paul.sevinc@inf.ethz.ch

Introduction & Project Objectives

Tracking changes in XML documents can be of relevance to security. For example, when authors cannot repudiate their changes, they may be more reluctant to mount a social-engineering attack such as phishing by changing the URL of the log-in page of a banking application in an otherwise authentic electronic brochure:

- <li href="http://www.niceandh**o**nestbank.ch/">N & H Bank
- <li href="http://www.niceandh**o**nestbank.ch/">Nice & Honest Bank

XML documents are text based, so one could compare XML documents by comparing them line by line. XML documents are also tree structured, so one could compare them node by node. One approach may be better in terms of execution speed of the computer while another may be better in terms of ease of spotting (security!) relevant differences for a human user.
In the example above, one might think that the lines differ only in the text if the line as a whole was flagged as having changed when in fact both the attribute node (lower-case letter o to digit zero) and the text node ("N & H" to "Nice & Honest") have been changed.
The objectives of this project are to survey diff and XML-diff algorithms and to discuss their differences with the help of insightful examples.

Work Plan

1. Search, collect, and study relevant publications (cf. References).
2. Define a set of abstract comparison criteria and a set of concrete examples.
3. Apply the criteria to diff and XML-diff algorithms and illustrate their differences with the examples.
4. Present the findings in a talk (in German or English) and in a report (in German or in English).

Prerequisites

Working knowledge of Java development and the Extensible Markup Language (XML).

Supervision

Prof. Dr. David Basin and Paul E. Sevinç (paul.sevinc@inf.ethz.ch).

E. Abkürzungsverzeichnis

API Application Programming Interface

CUAL Connected, Undirected, Acyclic, with Labeled nodes

CVS Concurrent Versions System

DOM Document Object Model

DTD Document Type Definition

ETH Eidgenössische Technische Hochschule

FMES Fast Match/Edit Script

GNU GNU's Not Unix

GUI Graphical User Interface

HTML HyperText Markup Language

ID Identification

IR Internal Representation

JAR Java Archive

JVM Java Virtual Machine

JAXP Java API for XML Processing

LCS Longest Common Subsequence

PDF Portable Document Format

PS PostScript

SAX Simple API for XML

SVG Scalable Vector Graphics

XHTML Extensible HyperText Markup Language

XID XML Identifier

XML Extensible Markup Language

XSL Extensible Stylesheet Language

Literaturverzeichnis

[BPSM98] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. XML 1.0. `http://www.w3.org/TR/1998/REC-xml-19980210`, 1998.

[CAH02] Grégory Cobéna, Talel Abdessalem, and Yassine Hinnach. A comparative study for XML change detection. `ftp://ftp.inria.fr/INRIA/Projects/verso/VersoReport-221.pdf`, 2002. Überblick über verschiedene XML-Diffs.

[CAM02] Grégory Cobéna, Serve Abiteboul, and Amélie Marian. Detecting Changes in XML Documents. In *International Conference on Data Engineering*, volume 18, San Jose, February 2002.

[CD99] James Clark and Steve DeRose. XML Path Language (XPath) 1.0. `http://www.w3.org/TR/xpath`, 1999.

[CGM97] Sudarshan S. Chawathe and Hector Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 26–37, Tuscon, Arizona, May 1997.

[CKM02] E. Cohen, H. Kaplan, and T. Milo. Labeling dynamic XML trees. In PODS, 2002.

[CRGMW96] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 493–504, Montréal, Québec, June 1996.

[ep04] The ercato project. XOp. `http://www.living-pages.de/de/projects/xop/`, 2004.

[Fon04] Robin La Fontaine. How DeltaXML Represents Changes to XML Files. `http://www.deltaxml.com/core/deltaxml-changes-markup.html`, 2004.

[HM76] J. W. Hunt and M. D. McIlroy. An Algorithm for Differential File Comparison. `http://www.cs.dartmouth.edu/~doug/diff.ps`, 1976.

[LM00] Andreas Laux and Lars Martin. XML Update Language. `http://xmldb-org.sourceforge.net/xupdate/`, 2000.

[MAC+01] A. Marian, S. Abiteboul, G. Cobena, , and L. Mignet. Change-centric management of versions in an XML warehouse. In VLDB, 2001.

[Mye86] Eugene W. Myers. An O(ND) difference algorithm and its variations. Algorithmica, 1(2): 251-266, 1986.

[WDC] Yuan Wang, David J. DeWitt, and Jin-Yi Cai. X-Diff: An Effective Change Detection Algorithm for XML Documents. `http://www.cs.wisc.edu/~yuanwang/xdiff.html`.

[ZSS92] Kaizhong Zang, R. Statman, and Dennis Shasha. On the Editing Distance between Unordered Labeled Trees. In *Information Processing Letters*, volume 42, pages 133–139, 1992.

[ZWS95] Kaizhong Zhang, Jason T. L. Wang, and Dennis Shasha. On the editing Distance between undirected acyclic Graphs. In *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, pages 395–407, 1995.