der Bundeswehr
Universität München

Efficient Change Management of XML Documents

Dissertation
zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

vorgelegt von
Dipl.-Inf. Sebastian Rönnau
im August 2010

Tag der mündlichen Prüfung:	09. Dezemver 2010
Vorsitzender der Kommission:	Prof. Dr. Peter Hertling
1. Berichterstatter:	Prof. Dr. Uwe M. Borghoff
2. Berichterstatter:	Prof. Dr. Michael Koch
1. Prüfer:	Prof. Dr. Ulrike Lechner
2. Prüfer:	Prof. Klaus Buchenrieder, Ph.D.

Universität der Bundeswehr München
Fakultät für Informatik

Abstract

XML-based documents play a major role in modern information architectures and their corresponding work-flows. In this context, the ability to identify and represent differences between two versions of a document is essential. A second important aspect is the merging of document versions, which becomes crucial in parallel editing processes.

Many different approaches exist that meet these challenges. Most rely on operational transformation or document annotation. In both approaches, the operations leading to changes are tracked, which requires corresponding editing applications. In the context of software development, however, a state-based approach is common. Here, document versions are compared and merged using external tools, called *diff* and *patch*. This allows users for freely editing documents without being tightened to special tools. Approaches exist that are able to compare XML documents. A corresponding merge capability is still not available.

In this thesis, I present a comprehensive framework that allows for comparing and merging of XML documents using a state-based approach. Its design is based on an analysis of XML documents and their modification patterns. The heart of the framework is a context-oriented delta model. I present a diff algorithm that appears to be highly efficient in terms of speed and delta quality. The patch algorithm is able to merge document versions efficiently and reliably. The efficiency and the reliability of my approach are verified using a competitive test scenario.

Acknowledgements

Many people have supported me in my work. First of all, I have to thank to my supervisor, Uwe M. Borghoff, for giving me the opportunity to do my research. He allowed me to follow my own research interests, with broad but precise constraints. He has always trusted in me and my work and encouraged me to take on the responsibility for the organization of the ACM DocEng 2009. I also have to thank my second advisor, Michael Koch. He provided me with valuable comments, helping me to sharpen this thesis up.

Almost all of the time, I did really enjoy being member of the Institute of Software Technology. Apart from the excellent research conditions, the staff played a major role. Florian Brieler, Nico Krebs, Sonja Maier, Steffen Mazanek, Mark Minas, Peter Rödig, Arne Seifert, Thomas Triebsees, and Daniel Volk helped me elaborating my ideas in countless discussions. Susanna Defendi, Harald Hagel, and Michael Minkus allowed me for discussing topics not related to the domain of computer science. It was a pleasure and a honor to me to work with my colleagues. Some of them became even friends.

Other people have contributed to my thesis. First of all, Jan Scheffczyk helped me writing my first paper, opening me the way to an academic career. Many students supported my research. Christian Pauli wrote the first prototype of XCC Patch. Geraint Philipp, a highly skillful programmer, made a complete re-design of XCC Patch and programmed XCC Diff. Maik Teupel designed a user interface and wrote converting tools. Arthur Müller developed further applications of the XCC framework on the file system level. Manfred Pohlemann supported my work by developing a test framework.

Finally, I have to thank my family. My wife Silvia and my children Luca and Lilith have been very patient with me, accepting when I was coming home late or lost in thought. My youngest girl, Lene, was being born just a few hours after the submission of my final draft. Thanks for her patience, too.

Contents

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interface
CMS	Content Management System
CPU	Central Processing Unit
CSCW	Computer-Supported Cooperative Work
CVS	Concurrent Versions System
DOM	Document Object Model
DTD	Document Type Definition
DUL	Delta Update Language
EBNF	Extended Backus-Naur Form
ECMA	European Computer Manufacturers Association
FMES	FastMatch EditScript
GNU	GNU's not Unix!
GUI	Graphical User Interface
HTML	Hypertext Markup Language
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
JAR	Java Archive
JDK	Java Development Kit
JRE	Java Runtime Environment
LCS	Longest Common Subsequence
MathML	Mathematical Markup Language

OASIS Organization for the Advancement of Structured Information Standards

ODF OpenDocument Format

OOXML Office Open XML

PI Processing Instruction

RCS Revision Control System

SGML Standard Generalized Markup Language

SVG Scalable Vector Graphics

SVN Subversion

USB Universal Serial Bus

W3C World Wide Web Consortium

WYSIWYG What you see is what you get

XCC XML Change Control

XHTML Extensible Hypertext Markup Language

XOM XML Object Model

XSLT XML Stylesheet Language Transformations

XML Extensible Markup Language

1 Introduction

This thesis deals with the efficient and reliable change control of XML documents, which build the backbone of today's business processes. First, I explain the motivation of my work which shall support every-day ad-hoc collaboration without tightening the user to a strict work-flow or editing tool. Afterwards, I discuss the term document change control. Finally, I briefly sketch my approach, followed by the outline of this thesis.

1.1 Motivation

Nowadays, electronic documents are a major information exchange carrier. The creation and editing of texts, spreadsheets, and presentations is an every-day task. Usually, these documents are edited incrementally. A first draft is written, revisions are made, and so on. Especially in the context of office work, documents are often edited collaboratively. Large documents consist of different parts, contributed by different people from different organizational entities. Even small documents are usually proof-read internally for quality control. In the past, these processes have mostly been serial ones. The document was either printed and therefore tighten to paper, or bound to its physical representation in a data carrier like a USB pen-drive. Since ubiquitous network access has become the standard, the collaboration process became more and more a parallel one. The document is either stored on a widely available network space or distributed via e-mail.

Parallel editing of a document leads to the important question of document integrity. In case that two users modify a document in parallel, the concurrent (write) access may lead to severe problems. Either the last write access tampers with the document format, leading to an unreadable document. Or the changes performed by the first access are discarded, overwritten by the second version. To prevent this unacceptable drawback, several approaches exist. First, locking can be used to prevent a parallel write access. If a user opens the document for writing, no other user is granted write access until the first user has written his version [Borghoff and Schlichter 2000]. Another approach is copy-modify-merge [Tichy 1982]. Here, each user works on a local copy, and commits his changes to a centralized repository. The consolidation of the modified versions is called *merge*. Alternatively, the document could be modified using a collaborative editor [Koch 1997]. Here, all users work on virtually the same workplace. Each change is displayed to all users as soon as possible.

All these approaches are common methods in the domain of computer-supported cooperative work (CSCW). However, most of the software used in offices do not support these methods. Only file-based locking is provided. Especially for large documents, this approach is not

very suitable, as it makes parallel editing of documents impossible. Here, more fine-grained locking mechanisms would be helpful [Borghoff and Teege 1993b]. Nevertheless, locking is only a solution if all participants have access to the same repository. In case that documents are exchanged via e-mail, however, locking does not provide any advantage. The changes have to be merged anyway. Collaborative editors have a similar drawback by requiring a permanent connection of the participants. Although editors exist that allow for asynchronous collaboration, all participants have to use the same editor. On the other hand, different applications may be used to display and/or edit a document, due to the limited capabilities of the displaying device (e.g. netbook or Blackberry), or due to the low user experience. As an example for latter case, complex calculations are performed within a spreadsheet application by an expert user. Other users may only see the final table displayed within a text. Different applications are needed to handle these highly complex and user-specific issues. Copy-modify-merge is a well-established approach in the domain of software source code. However, corresponding merging applications for office documents are not mature yet.

Tool support for the described collaboration model is still in its infancy. Therefore, documents or parts of them are usually exchanged completely. Document versions are merged manually which is both time-consuming and error-prone. In the context of office documents, the "change tracking" feature of the corresponding applications has become a de-facto standard for displaying the changes made. Here, each change is tracked on a character-level and in-lined within the document. However, these changes are hard to understand [Neuwirth et al. 1992]. Especially if different versions ought to be merged, the user interface is confusing. Additionally, the tracked changes contain a lot of meta-data that offer a deep insight into the editing process. In many cases, this should be hidden from the other participants. At last, the tracked changes may even miss some changes. In some cases, format changes are not tracked and cannot be merged for that reason [Rönnau and Borghoff 2009].

One of the basics of the success of the Internet was the availability of the Hypertext Markup Language (HTML). This language is used to describe different kinds of documents, trying to separate the content from the markup information. HTML, however, is closely tightened to text documents; content and markup separation is not straightforward. Therefore, XML has been designed as a meta-language to allow for a broader class of documents. Today, XML has emerged as lingua franca in many domains. A huge variety of XML dialects is used to encode documents. Agreed-upon XML-based document standards exist for texts, spreadsheets, presentations, and many more kinds of documents. Therefore, I focus on the issues of XML documents, without restraining to a specific dialect of document format.

In this thesis, I address the issue of collaborative editing of XML documents through arbitrary exchange of document versions. The motivation is to enable users to compare, reconstruct, and merge document versions efficiently and reliably.

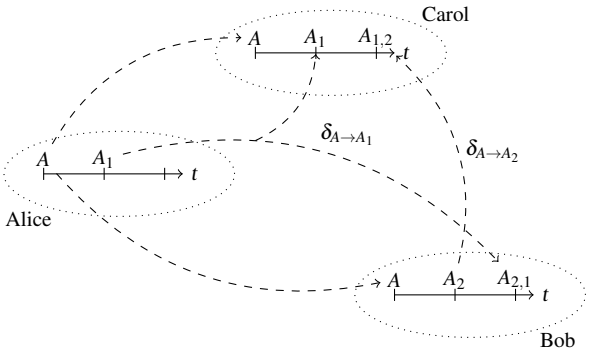


Figure 1.1: In a loosely coupled collaborative environment, changes are exchanged arbitrarily, leading to many merges.

1.2 Document Change Control

Figure 1.1 shows an example of loosely coupled ad-hoc collaboration groups, working on document A. Alice initiates the editing by distributing the first version to Bob and Carol. She continues working on her draft, while Bob starts to perform his changes. Alice sends her changes to Carol, requesting her comments. After that, Bob sends his changes to Carol. Carol merges both changes to $A_{1,2}$. After Carol's approval, Alice sends her changes (indicated by $\delta_{A\to A_1}$) to Bob. Now, Bob has to merge Alice's changes into his version, leading to $A_{2,1}$. If the changes are merged manually, the probability that all participants have the same version in the end is quite low. Document change control tools shall ensure a deterministic merge result. In case of non-conflicting changes, the merged documents must be identical across all copies, independent from the merge order.

As already pointed out, users shall not be tightened to a specific application to edit their documents. This raises the questions how users can exchange their changes. In a naive approach, the changes could be described in another document which is attached to the modified document. Although this is a substantial overhead and error-prone, it is a common practice in office collaboration[1]. Automatic finding and encoding of the changes is more efficient (in terms of labor time) and effective. Additionally, the description of changes cannot be forgotten by the authors anymore. A *diff* tool compares two document versions (aka states) and stores the changes in a *delta*. The differencing tool is also called *diff*, the delta is also referred to as *edit script*[2]. The application of a delta to a file to create the new version is called *patch*. If a delta is applied to another version of the document as is has been computed for, a *merge* is performed.

Diff, patch, and merge lay the basis of state-based document change control. In this context, the encoding of the changes within a delta is essential. As the delta is the information carrier

[1]Here, I have to distinct between the description of the changes themselves or their intention. Obviously, comments on the changes as additional notes may be important and helpful.

[2]The terms delta and edit script are not fully equivalent. I will discuss the differences in Section 6.4.

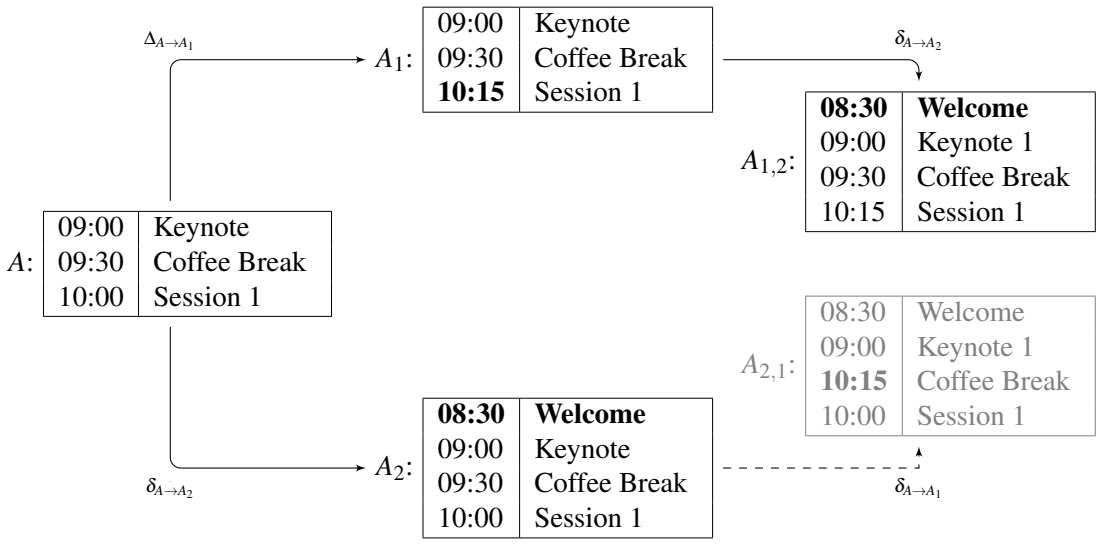


Figure 1.2: Absolute paths may lead to unwanted merge results. Performing the delta marked with a dashed line would affect the wrong node, leading to a wrong result.

between the diff and the patch tool, only the information contained within the delta can be used to patch and merge document versions. Merging documents by applying deltas from other versions can be ambiguous. Changes may affect the structure of the document, thus affecting the addressing of the subsequent parts of the document. Figure 1.2 exemplifies this problem using a schedule. In version A_1, a time is modified, in version A_2, a new entry is inserted. If delta $\delta_{A\to A_1}$ encodes the change like "change column 1, row 3 into 10:15", the application of $\delta_{A\to A_1}$ would address the wrong cell. Therefore, the delta should contain additional information to ensure a reliable merge.

As a side-note, the mentioned techniques only cover the technical aspects of document change control. Organizational and psychological aspects play an important role, too. However, they are not part of my research and are not discussed further. Nevertheless, I may point out that diff and patch have been enjoying great popularity in the domain of software source control for nearly 35 years [Rochkind 1975; MacKenzie et al. 2002].

1.3 Goal and Approach

In this thesis, I develop a framework for differencing, patching, and merging XML documents. The heart of the framework is a novel delta model that stores the syntactic context of an edit operation. This syntactic context is used as additional source of information during merging. This way, a reliable merge can be ensured.

The use of the syntactic context for merging is not new. Davison [1990] has proposed this approach for the domain of line-based documents. Today, this context-oriented delta format is

still the state-of-the-art in source code control [MacKenzie et al. 2002]. However, the mapping of the one-dimensional line-based view onto two-dimensional trees as used in XML is not self-evident. A first approach has been presented by Mouat [2002], but has not reached maturity[3]. My approach is the first comprehensive context-oriented delta model. Using the syntactic context, the patch procedure is enabled to reliably merge document versions including conflict detection.

Complexity is another important aspect of change control. The comparison of XML trees is a difficult task. Evaluations, e.g. by Rönnau et al. [2005], have shown that even for quite simple documents, existing XML diff tools may take more than a minute to compare two documents. Apparently, this is not acceptable in every-day use. Therefore, I also focus on the question of efficient differencing and patching. Efficient XML diff algorithms have been presented before, e.g., by Cobéna [2003] and Lindholm et al. [2006]. However, none of these approaches is able to merge document versions.

In this thesis, I present an efficient differencing algorithm especially designed for XML documents. It bases on a computation of the longest common subsequence of leaf nodes, combined with a bottom-up comparison of the non-leaf nodes. As the content of the documents lay within the leaves, my algorithm focuses on content changes, before investigating markup and structure changes.

The corresponding patch algorithm is efficient and merge-capable. It uses the syntactic context for reliably identifying the correct path to an edit operation. The efficiency is gained by restraining the search space to the neighborhood of the expected path.

1.4 Outline

This thesis is divided into three parts. Part I deals with the basics of document change control. Here, I define the problem statement and present the related work in this research domain. In Part II, I present my approach to change control of XML documents. Part III concludes the thesis.

The preliminaries of this thesis are discussed in Chapter 2. Here, I define the term document as well as the XML model used. In the following, I present different XML document formats. Using an empiric analysis, I investigate the properties of common XML documents, as well as the modification patterns which arise during the editing process. In Chapter 3, I present the basics of document differencing, including the common solutions for string and tree differencing. Existing approaches to XML differencing are presented last. Chapter 4 deals with the problem of document merging. The most common approaches are presented and I discuss their applicability to the domain of XML documents.

In Chapter 5, I sketch the architecture of my change control framework, including basic implementation details. In Chapter 6, I present my XML context-oriented delta model that builds the heart of the framework. An efficient differencing algorithm is presented in Chapter 7. I

[3] I will discuss this approach in Section 6.5.

present a corresponding merge-capable patch algorithm in Chapter 8. Both algorithms have been implemented and are evaluated in terms of efficiency and reliability in Chapter 9.

Finally, I compare the proposed differencing algorithm and the resulting deltas in an evaluation against the state-of-the-art approaches in this domain. After this evaluation in Chapter 10, I give a summary of the thesis in Chapter 11. Here, I also present different applications of my framework and give an outlook on future research.

Appendix A lists the documents that have been analyzed in the evaluation of the XML document properties. In Appendix B, I present the XML hashing model used in my framework. Appendix C defines the XML representation of the context-oriented deltas. Finally, Appendix D shows a running example of a whole document editing process using the proposed framework.

Part I

Concepts and Notations

2 XML Documents

The Extensible Markup Language (XML) has emerged as lingua franca in many domains, including the Web and office documents. The simplicity and the clear design, in addition with powerful and easy-to-use tools have built the basis of the success of XML. Many different XML dialects exist. Each of them has its own characteristics and design goals. However, there are many similarities.

This chapter contains the preliminaries in the context of XML documents. It starts with a brief discussion of the term "document". It is followed by an overview of XML in general, introducing some basic notations. Afterwards, two major XML document formats, ODF and XHTML, are presented. I explain the background of these formats first, before performing an empiric analysis of real-world documents. The main goal of this evaluation is to gain knowledge about the properties of XML document trees. Finally, I explore the modification patterns in the course of the evolution of the documents. These findings aim to lay the basis for the design of an efficient differencing algorithm later on.

2.1 What is a Document?

The question, which properties specify an artifact to define it as "document" has been extensively discussed in the first half of the 20th century, mostly in the context of bibliography. Documents were commonly seen as physical artifacts, created with an intention. For an overview on this discussion, please refer to Buckland [1997]. Beginning with the 1980's, the applicability of the term "document" to digital artifacts was discussed. In context of my work, I define the term *document* as follows:

Definition 2.1 *A document is the digital representation of knowledge. It must exist an unambiguous way of converting it into a physical representation (e.g. by printing). A document consists of content and markup and must have an inner ordering (i.e., the reading direction). A document may evolve over time, but must have a definite state at any time.*

Some implications arise from this definition. First, it is arguable whether a graphic may be seen as document, as a graphic does not provide a common inner ordering. Some graphics, e.g. diagrams, may have a reading direction, but this is a domain-specific one. Therefore, graphics are not part of my research. A second implication affects the field of active documents. Recently, approaches have been presented that add application-logic to documents, e.g. by Boyer et al. [2008]. In these active documents, the current state of the document is not determinable

at any time. Thus, the document property defined earlier is not satisfied. Therefore, active documents are not part of my research as well.

Even if I do not aim to research on graphics and active documents, some of my findings may be applicable to these domains. In Section 11.3.4, I will present research directions that could ensure the applicability of my approach to a broader definition of the term "document".

2.2 XML in General

XML is a universal metalanguage for a wide variety of applications. It is an application-independent metalanguage with a clear and precise definition that allows for defining human-legible and reasonably clear documents [Bray et al. 2008]. One of the main goals during the design of XML was to separate the content from the markup of a document. This was mainly intended by the experienced gained from the prevailing binary-based document formats that could only be interpreted using specific tools. Therefore, Borghoff et al. [2006] recommend to use XML to ensure the long-term readability of documents.

An XML document is a tree that can be represented either by a text file or a corresponding object model like the Document Object Model (DOM) [Hors et al. 2004]. Usually, the text representation is used for serialization; the object model is mostly used for processing the tree within an application.

In this section, I briefly present my notation regarding XML documents first, before describing the node types of XML. Aspects of parsing the text representation into an object model are discussed afterwards, as well as validity constraints of documents. I present an elementary addressing scheme for XML nodes that is used throughout this thesis. Finally, I differentiate the ordered tree model of XML documents from the unordered model used in dataset-representations.

2.2.1 Notations

From a theoretical point of view, an XML tree is an undirected acyclic graph. Each node may have an arbitrary number of *children*, where each child has only one *parent* node. The node without a parent is called *root*, all nodes without children are *leaves*. All parent nodes on the path up to root are the *ancestors* of a node. All child nodes on the possible paths to the leaves are called *descendants*. The descendants of a node n build a *subtree*, with n being their root. Nodes with the same parent node are called *siblings*.

The *depth* of a node conforms to the length of the path from the root to the node. The root node has a depth of 0. The *height* of a node is the length of the longest path from the node to the leaves. A leaf has a height of 0, the height of the root node corresponds to the highest depth. A *top-down level* of the tree is defined as all nodes with the same depth. Additionally, the *bottom-up level* comprises all nodes with the same height[1]. The top-down level and the bottom-up level can be equal, which is unlikely in complex documents.

[1] In literature, the term *level* is mostly used according to my definition of the top-down level.

Definition 2.2 *The basic set of all XML documents shall be denoted as* $\mathbb{A}$*. An XML document is denoted as* $A \in \mathbb{A}$*. Different versions of A are denoted as* $A_1, \ldots, A_n$*, with* $n \in \mathbb{N}$*.*

In this context, I define a document version as follows:

Definition 2.3 *A document* A_1 *is a version of another document A, if and only if it has been transformed from this document or another version of that document using any editing action.*

Two versions of one document essentially share a common content and markup as far as possible. However, this is not a requisite.

2.2.2 Node Types

XML distinguishes different types of nodes. A node can either be an *element* node, a *text* node, a *processing instruction*, or a *comment* [Bray et al. 2008]. Comments may be omitted during parsing of the document and do not affect the content or markup of the document. Therefore, this node type is mostly neglected. Processing instructions (PI) are intended to pass non-XML-coded commands to the corresponding application. They are a relict from the preceding SGML format that remained for compatibility reasons. Goldfarb [1990] stressed that even for SGML, the use of processing instructions is a bad design that should be used sparingly. They are often neglected, too. I follow this decision for my XML model.

Text nodes (also known as CDATA nodes) store the content of the XML document. They are not parsed but directly passed to the application. In this context, the term "text node" is somewhat misleading. Basically, any kind of data, including binary representations, can be stored within a text node. As I focus on document representations where the content is mostly stored as text, I use the term text node throughout this thesis. Deriving from its definition as non-parsed entity, text nodes are leaf nodes on principle.

Element nodes contain the markup of the document. Basically, all non-leaf nodes within the XML tree are element nodes. An element node consists of its *label* and an arbitrary number of *attributes*. Attributes are name-value assignments that are stored as an unordered list attached to the element. An element node can be encoded in different ways. Due to different character encodings, name-space resolutions, or attribute orderings, the syntactic appearance of one and the same node may differ. To resolve these ambiguities, a node has to be normalized.

Definition 2.4 *The node value is the normalized representation of the node label, including a normalized representation of the attribute list.*

A normalized node label means a representation where character encodings and name-space prefixes are used in a uniform way. In a normalized list of attributes, all entries are ordered unambiguously. A normalization scheme is presented in Appendix B. It is based on Canonical XML by Boyer [2001] and DOMHash by Maruyama et al. [2000]. Using the node value, I define the equivalence of nodes as follows:

Definition 2.5 *For two nodes* $n_1, n_2 \in A : n_1 = n_2$ *is true, if and only if* $\mathrm{value}(n_1) = \mathrm{value}(n_1)$

That means that the two nodes are equal if their normalized node values are equal.

2.2.3 Parsing XML

XML documents are usually exchanged in their text-based representation. Basically, this is a simple text file encoded in Unicode, a text encoding format designed for handling all kinds of characters [Davis and Collins 1990]. By parsing this text representation, the corresponding object model is built. There exist common object models like the Document Object Model (DOM) [Hors et al. 2004] and application-specific object models as well. However, most application-specific object models are only a further abstraction of DOM, like the ODF Toolkit[2] for office documents. The reason for this is the availability of well-documented and reliable frameworks for DOM handling.

The text representation is a sequence of different nodes, representing the tree structure by nested delimiters, called tags. The text representation is parsed from left to right, corresponding to a *preorder traversal* of the tree.

Definition 2.6 *The term document order denotes the order in which nodes are encountered, one after another, as the document that contains them is parsed [DeRose and Clark 1999].*

Definition 2.7 *For two nodes $n_1, n_2 \in A$, $n_1 < n_2$ is true, if and only if n_1 comes before n_2 in document order.*

These definitions ensure an explicit ordering on the XML tree, thus allowing for addressing nodes with respect to their position on the tree.

Within the text representation, element tags are delimited by angle brackets. Subtrees are delimited by an opening and a closing element tag. Elements without children can be represented as a single tag with a special notation (slash at the end).

Figure 2.1 shows an example document, whose text representation is parsed into a tree representation. It contains text and element nodes, the latter one with attributes. During parsing, the tree is constructed from the text representation from top to down, according to the document order. The first line contains the prologue of the text representation, specifying the version of the XML specification used.

Basically, all nodes are written subsequently within the text representation, without separation of the nodes. White-space like blanks and line breaks are only used within text nodes. Obviously, this representation is not very human-readable. To illustrate the tree structure, the text representation is often structured in a way that resembles the tree representation in Figure 2.1, using line breaks and indentations. This process is called pretty-printing, resulting in an undesired side-effect: the white-space used for pretty-printing is parsed as (empty) text node and added to the tree representation. It is basically possible to ignore this white-space, but the decision whether the white-space is significant to the document content is not straightforward [Murata et al. 2005]. Therefore, I expect the text representation to be not pretty-printed.

[2] `http://odftoolkit.openoffice.org/`

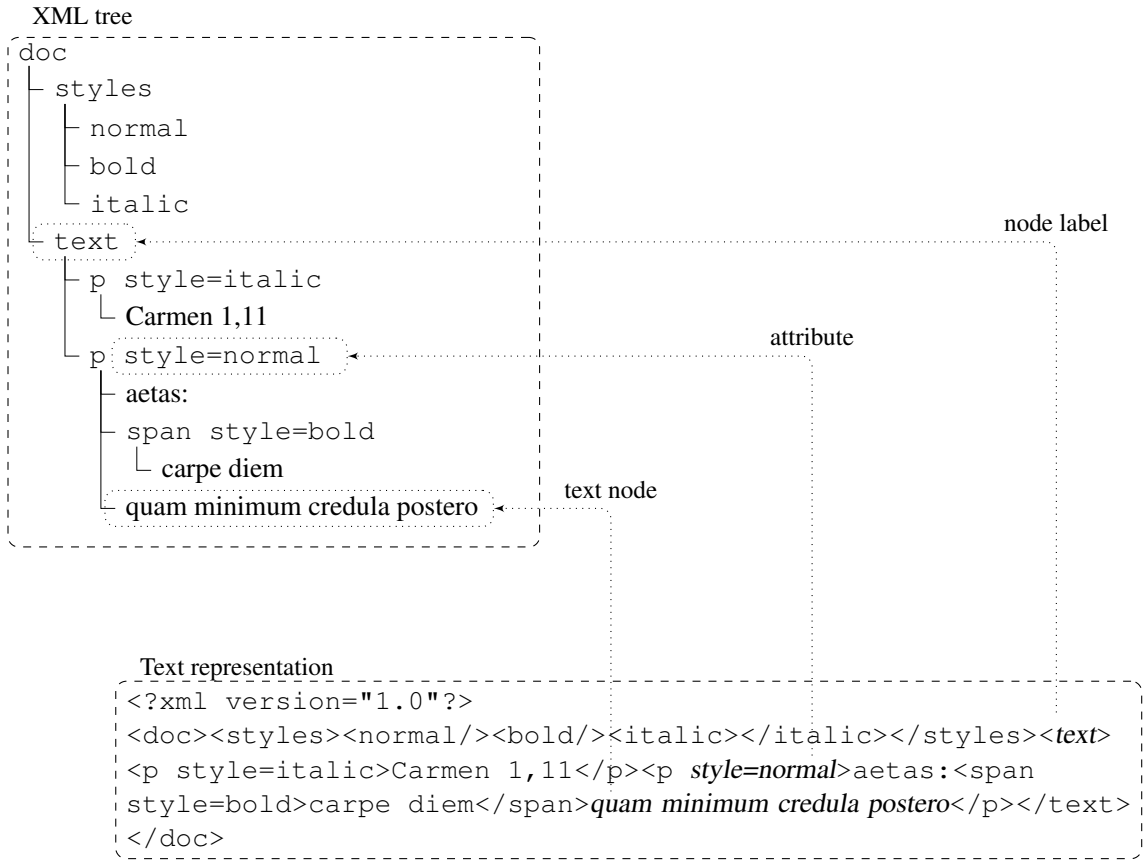


Figure 2.1: The XML tree can be constructed by parsing the corresponding text representation. The line breaks in the text representation have been added for a better readability.

2.2.4 Well-Formedness and Validity

An XML document is required to be *well-formed*. That means that the document fulfills the requirements specified by Bray et al. [2008]. Among others, one important requirement is that the document must conform to the properties of a tree. This includes the rule that overlapping tags must not occur. Although this seems trivial, overlapping tags are commonly used in the domain of Web documents coded in HTML, as many browsers display them despite their non-well-formedness. XML parsers must not process a non-well-formed document.

XML is intended to act as universal meta-language for any kind of data representation. To ensure the interpretability of an XML document, a grammar can be defined. A document conforming to this grammar is called *valid*. An application may only interpret valid documents. The validation is performed using tree automata [Neven 2002]. There exists a large amount of grammar languages for XML. Their major distinction lies in their expressiveness. In this context, expressiveness is mostly influenced by the capability of defining recursions and context-sensitive production rules. The most common grammar languages are (in order

of expressiveness) DTD, XML Schema, and Relax NG. Sadly, an increased expressiveness results in a higher complexity of validation [Murata et al. 2005]. However, Martens et al. [2006] have shown that complex expressions are rarely used in commonly used grammars.

As the word "extensible" in XML may suggest, an XML document is not restricted to one comprehensive grammar. Different types of (sub-)documents may be used within one document. For example, an office document may consist of a text, with drawings included. The drawings can be parsed independently from the text document and vice versa. To enable the parser to distinguish between different sub-documents, name-spaces are used. An XML name-space allows for a non-ambiguous assignment of a node to its respective grammar [Bray et al. 2006].

2.2.5 Addressing Nodes

A major issue when editing XML documents is the ability to address single nodes. XPath has been defined as a unified language for node addressing [DeRose and Clark 1999]. XPath allows for arbitrarily walking through the tree. That includes absolute addresses that start from the root, but also relative addresses, starting from any node within the tree. Using relative addresses, single nodes and node sets can be selected, mostly using the parent-children relationship. To address single nodes using an absolute address, I introduce a simplified notation.

Definition 2.8 *A path to a node denotes the position of a node from the perspective of the root element and is denoted as follows: the root node is denoted with a slash (/). Its children are indexed from left to right recursively across the levels of the tree. The children index* $n \in \mathbb{N}_0$ *starts at* 0*, with an increment of* 1*. Each level is separated by a slash.*

The path `/0/2/2/0` in my notation would refer to `/*[1]/*[3]/*[3]/*[1]` in XPath notation[3]. XPath has been designed to address not only single nodes but complete node sets. Comprehensive queries can be defined, which in turn require a high complexity upon evaluation [Gottlob et al. 2003]. Simple queries require a noticeable evaluation overhead, too. Additionally, the database-like view of nodes as elements of a set is not suitable for the document domain. Therefore, the proposed addressing scheme is more suitable for addressing single nodes efficiently.

2.2.6 Node Ordering

From its original intention, XML is an ordered tree, where the ordering of the children is significant. This *ordered tree model* reflects the properties of documents, where the ordering of the content is significant, too. As I focus on XML documents, I base my research on this model.

[3] XPath starts the node ordering at 1. According to most conventions in computer science, I start the ordering at 0, especially to avoid confusion with other non-XPath addressing schemes.

On the other hand, XML emerged not only as document description language, but also as data exchange format. Especially in the domain of databases, data representation models commonly rely on a set property of the data. In turn, the ordering of elements within the XML tree is neglected. This *unordered tree* model is adequate for data representation, yet unusable for the encoding of documents. One might argue that hypertexts define their reading direction using links. However, the corresponding linked parts have their own reading direction that must be preserved. Furthermore, the use of an unordered tree model leads into a complexity trap when comparing documents, as I will discuss in Section 3.2.3.

The unordered tree model is often used if the XML data is reconciled from a database representation, e.g. by Abiteboul et al. [2006]. However, ordered XML documents may also be stored in (basically unordered) databases, as shown by Tatarinov et al. [2002].

2.3 Document Formats

Different types of documents can be represented by corresponding XML formats. In this section, I describe two major formats, XHTML in the domain of Web documents and ODF in the domain of office documents. I explain their importance in their respective domain as well as their technical properties. Other formats are mentioned at the end of this section.

2.3.1 XHTML

The Extensible Hypertext Markup Language (XHTML) originates from HTML, the Hypertext Markup Language that builds the standard for Web documents, developed by the World Wide Web Consortium (W3C) [Pemberton 2002]. XHTML provides a vocabulary equivalent to HTML, but adopts XML's strict rules on well-formedness. Furthermore, standard tools for XML like parsers, validators, and XPath processors can be used. However, XHTML and HTML are not fully equivalent. XHTML allows for including foreign XML dialects for special purposes like Scalable Vector Graphics (SVG) for graphics representation or Mathematical Markup Language (MathML) for formulas. For this reason, HTML can be easily converted to XHTML without loss of information, but the opposite way may imply a lossy conversion [Carey 2008]. An XHTML document is basically a single XML file. Embedded pictures are not part of the document and are not considered further for this reason.

2.3.2 ODF

Web documents have been developed in an open process by the W3C. A common standard for all participants in the Web is a major goal of the W3C. In the domain of office documents, however, file formats have been developed by the corresponding software manufacturer and tightened to the office application, like Lotus 1-2-3, Microsoft Word, or Word Perfect. These proprietary file formats used a binary encoding, a publicly available documentation was not

available. This helped mostly the larger software manufacturers to defend their market position, as competing applications were not able to edit the widely-deployed office documents flawlessly [Weir 2009].

To overcome this tight bonding of application and file format, an XML-based office document format has been developed that should build the basis for any kind of office application [Eisenberg 2004]. Originally developed by the Open Office Project, this format has been refined in an open process, resulting in the Open Document Format (ODF) standard that is currently maintained by the Organization for the Advancement of Structured Information Standards (OASIS). The ODF version 1.0 has been standardized by the International Organization for Standardization (ISO) as ISO/IEC 26300:2006. Currently, ODF version 1.2 is under active development.

ODF's popularity has increased significantly in recent years. Especially governmental authorities adopt ODF as standard document format. The ODF Alliance [2008] gives an annual overview of the world-wide adoption of ODF. There are several reasons for fostering the use of ODF. On the one hand, agreed-upon standards allow for interoperability and facilitate long-term accessibility of documents [Borghoff et al. 2006]. On the other hand, DeNardis and Tam [2007] have emphasized the importance of open standards for enabling free and democratic markets. In this context, "open" means free access, as well as independence from patent claims.

An ODF document consists of different XML files stored within a JAR archive [Brauer et al. 2007]. A JAR archive is basically a ZIP archive demanding for a defined directory structure which ensures that certain meta-data is contained within the archive. Using the archive has two reasons: First, the archive compresses the contained data to save storage space. Second, resources not native to the document format, like bitmap pictures, can be embedded within the archive and are tightened to the document that way [Eisenberg 2004]. However, in this thesis, only the XML files are respected. The XML files contain information regarding the content and the style-sheets used, as well as meta-data. The most important and largest file by far is the `content.xml`, containing all content and most of the markup. Usually, an ODF document can still be interpreted if the style-sheets and the meta-data are not present, as shown by Rönnau [2004]. Therefore, I consider the `content.xml` to be the core document.

Office suites cover a wide range of applications. They include a word processor, a spreadsheet application, a presentation engine, and many more. One important aspect of office documents is that different document types can be nested within each other. For instance, a text may contain a table that has been edited in the context of the spreadsheet application. A presentation may contain text elements and drawings. In ODF, this nesting of documents is represented by storing the sub-documents in separate sub-directories in the archive that in turn contain their own `content.xml`. Figure 2.3.2 shows the structure of an example ODF document. It contains a sub-document, as well as pictures. For a detailed description of the different files within an ODF document, please refer to Eisenberg [2004] and Brauer et al. [2007].

Office applications can not only create texts and spreadsheets. Slides or drawings may be created. Even simple databases are usually part of office applications. To the best of my

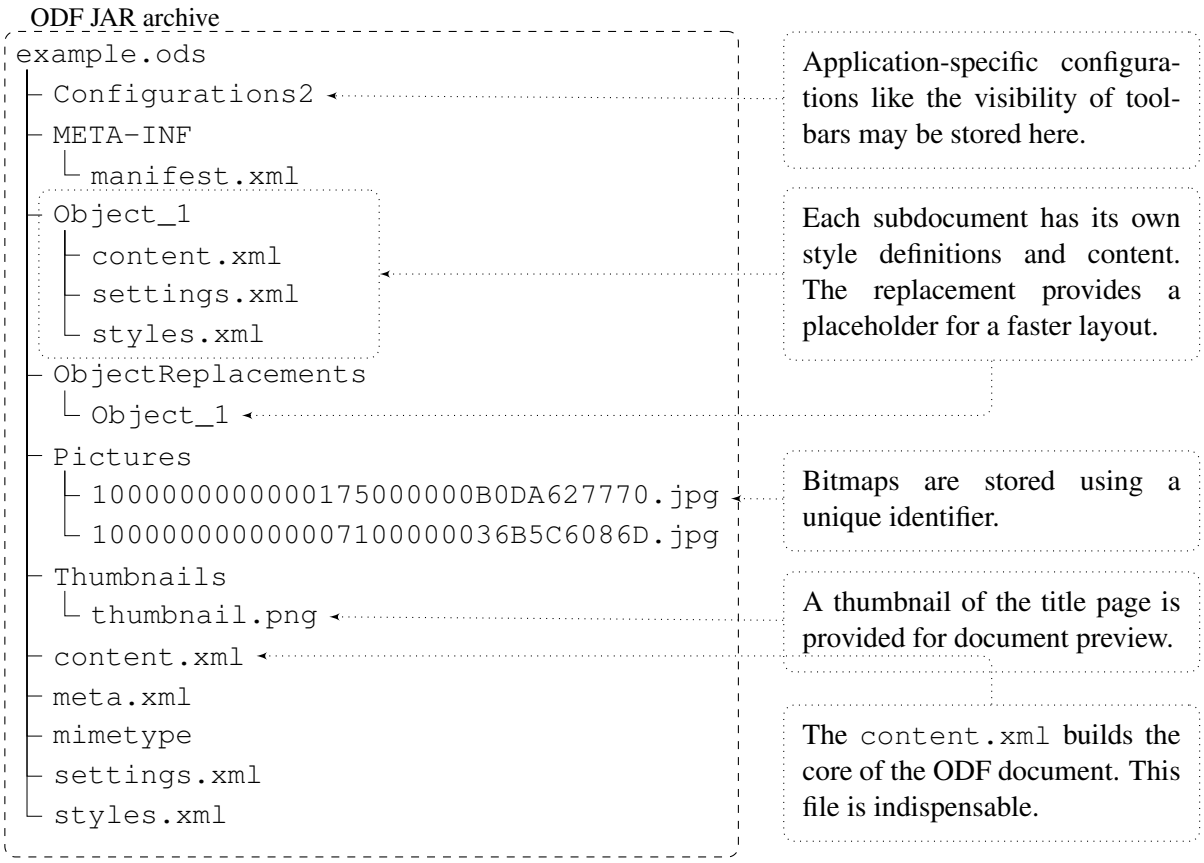


Figure 2.2: An example ODF document structure. It contains a sub-document, as well as pictures.

knowledge, all databases of office applications rely on the relational model by Codd [1970]. They are implemented using the transaction model by Gray [1978]. The key concepts of data sets and transaction systems prevent these databases to be seen as XML documents. For ODF, the database engine and the storage layer are not part of the format specification. Drawings do not provide an inner ordering like texts or spreadsheets. For two non-overlapping lines, the order of their appearance within the XML representation is irrelevant. Therefore, two documents with identical appearance might differ in terms of their representation. For this reason, a drawing does not meet my earlier presented definition of a document. The same holds for presentations. However, the presentation might contain texts and tables as sub-documents, which can in turn be treated as XML documents.

2.3.3 Competing Formats

There is no major opponent to XHTML in the domain of Web documents. In 2009, the W3C announced the discontinuation of the XHTML development[4]. However, it was clarified that this only affects the development of an independent format[5]. The upcoming HTML5 standard contains a specification for encoding Web documents in XML [Hickson and Hyatt 2010]. Therefore, all findings presented in this thesis are applicable to HTML5, too.

In the domain of office documents, Microsoft made great efforts to keep its dominant position. Under the pressure of governmental requirements, the binary office document format has been undisclosed in 2008. In parallel, Microsoft switched to an XML-encoding of their format, too, called Office OpenXML (often abbreviated as OpenXML or OOXML). OpenXML has been standardized by the European Computer Manufacturers Association (ECMA) [Paoli et al. 2006]. An ISO standardization was achieved as ISO/IEC 29500:2008, even if the standardization process came along with some irregularities[6]. The mere size of the specification, consisting of over 7,000 pages and different references to non-disclosed binary data representations are the main reasons for objecting to OpenXML. These are also the reasons why I decided not to consider OpenXML documents separately. However, the basic structure of OpenXML documents strongly resembles the ODF structure, including the usage of a ZIP archive for document storage. Additionally, tools exist that are able to convert these two document formats into each other. Therefore, I assume my findings on ODF to be applicable to OpenXML as well.

2.4 Tree Properties

The specifications do not show how the XML tree of a "typical" document will look like. However, having a deep knowledge of the properties of the tree may help to design an appropriate algorithm for comparing XML documents. To obtain real-life results, I analyze office documents from public repositories and Web pages. The focus of this analysis lies on two aspects. First, I investigate the appearance of the tree, i.e., how the nodes are distributed within the tree. Secondly, I evaluate the similarity of the nodes.

[4] http://www.w3.org/News/2009#entry-6601

[5] http://www.w3.org/2009/06/xhtml-faq.html

[6] For example, in the Norwegian sub-committee, the overall vote on OpenXML was turned into "accept", despite 80% of the committee members voting "reject" (http://blogs.freecode.no/isene/2008/03/31/norwegian-committee-chairman-to-iso-count-the-vote-as-no). Members of the German sub-committee could only decide between "accept" and "abstain" (http://www.heise.de/newsticker/meldung/DIN-sagt-Ja-zur-ISO-Standardisierung-von-OOXML-193470.html).

		Spreadsheet	Text	Web Page
Size in KByte	min	< 1	2	8
	max	16,444	773	233
	average	297	48	89
Nodes	min	10	7	295
	max	238,868	18,826	4,685
	average	4,147	739	1,905

Table 2.1: 177 documents are analyzed, covering a wide range of applications and sizes.

2.4.1 Test Data

For the evaluation, three kinds of document types are taken into account: spreadsheets, text documents, and Web pages. The documents cover a wide variety of application scenarios.

For the spreadsheets, 33 documents are evaluated, from a simple ratings overview up to a complex financial model. Concerning the text documents, 16 different documents, ranging from a one-page Curriculum Vitae up to a 125-page software documentation, are part of the evaluation. Especially the more complex office documents contain several sub-documents that are inspected independently. By this, not only 49, but 160 documents are evaluated.

To evaluate the properties of Web pages, 18 documents are analyzed. They cover major auction sites, news sites, governmental sites, and blogs. In total, 178 documents undergo the analysis, emerging from 67 base documents. Table 2.1 gives an overview of the size of the documents. The resources of the documents including a short description of them are listed in Appendix A.

Apparently, far more office documents than Web documents are analyzed. The decision for this is motivated by following considerations: Web pages are somewhat limited in terms of the amount of content. This limitation does not derive from technical but usability constraints. In the domain of Web page design, it is common knowledge that pages should fit onto the screen of the user [Krug 2000]. Nielsen and Pernice [2009] have enforced this statement using eye-tracking methods. They have shown that the part of Web pages that does not fit onto the screen without scrolling is seldom read by users. Additionally, a comprehensive analysis of Web pages by Fetterly et al. [2003] has revealed that most of the pages have almost the same size.

2.4.2 Node Distribution

The appearance of the tree is mostly influenced by following dimensions: the breadth and the depth of the tree. For document formats, the reading direction of the document is expressed by the breadth of the tree. The depth is mostly used for representing the complexity of the markup, which leads to flat yet wide trees. Admittedly, these are theoretical assumptions

		Spreadsheet	Text	Web Page
Depth	min	6	4	10
	max	13	15	23
	average	9	8	17

Table 2.2: XHTML trees are generally higher than ODF trees, as ODF uses sub-documents to model complex document structures.

		Spreadsheet	Text	Web Page
Node proportions	level 0 (leaves)	55%	56%	56%
	level 1	23%	26%	22%
	total	78%	82%	78%

Table 2.3: XML document trees have far most of their nodes in the lowest levels.

deriving from the specifications of the document formats. The specifications do neither give any hint on the minimum or maximum depth of documents nor the actual appearance.

The analysis reveals that even complex documents are rather flat. Table 2.2 shows the depth of the documents, ordered by their type. Even for a 16MByte spreadsheet document, the maximum tree depth does not exceed 13 levels. In this case, however, I recall the ODF approach to model complex documents using sub-documents. All of the 238.868 nodes are in fact stored within the 13 levels, but some complex structures like figures with captions are independent sub-documents again. In turn, these sub-documents usually have very small trees. For these reasons, XHTML trees are generally higher than ODF trees.

Apparently, the trees are not very high, taking into account the amount of nodes. As already mentioned at the beginning of this chapter, the content of the document is basically stored within the leaves, whereas the non-leaf nodes represent the markup of the document. Following this proposition, one could expect most of the nodes being leaves. The document analysis enforces this proposition partially. Table 2.3 shows the proportion of the nodes from a bottom-up perspective. As average, over 50% of the nodes are leaf nodes throughout all document types. Interestingly, the second-lowest level in the tree contains nearly one quarter of all nodes. All together, the two deepest levels cover around 80% of all nodes.

As the tree depth is varying, comparing more than the deepest levels is not appropriate. To set up comparableness despite different tree depths, I mapped all levels on a relative scale, with 0% meaning the leaves, and 100% the root. Figure 2.3 illustrates the node proportions using this relative scale in a comparative way. Even if displaying different kinds of documents, the ODF dialects for texts and spreadsheets show a similar node distribution. Web documents are even stronger in narrowing towards the root.

The analysis reveals that the trees are flat and wide. However, the decrease of the node

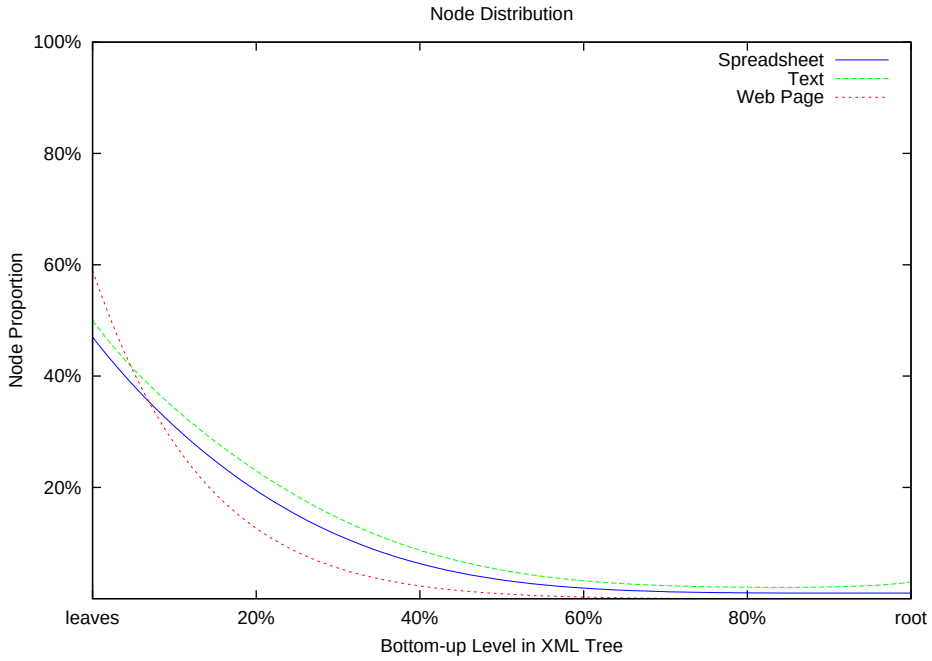


Figure 2.3: XML Document trees are flat and wide. Especially Web documents narrow strongly towards the root.

proportion against the depth is not that high as one could expect due to the fact that the content is mostly stored in the leaf nodes. Nearly one quarter of all nodes lays in the second-lowest level, which means that markup information is extensively used.

2.4.3 Repeating Nodes

Assuming that non-leaf nodes contain the markup, one could expect non-leaf nodes to be mostly identical. This results from the consideration that different markups should be sparingly used in a good document layout [Krug 2000]. To turn the argument on its head, it means that leaves should contain only few identical nodes, as a text should contain only few repetitions [Strunk Jr. and White 1979].

To investigate this assumption, I introduce the *repetition ratio* of nodes, where *values* denotes the amount of different values among all *nodes* on one bottom-up level:

$$\text{repetition\_ratio}(\mathit{nodes}, \mathit{values}) := 1 - \frac{\mathit{values}}{\mathit{nodes}} \tag{2.1}$$

The repetition ratio denotes the likelihood for a given node i that a node j with $i \neq j$ and $\text{height}(i) = \text{height}(j)$ has the same node value. For many different nodes, the repetition ratio tends to 0. If the nodes values are mostly equal, the repetition ratio approximates 1. In the

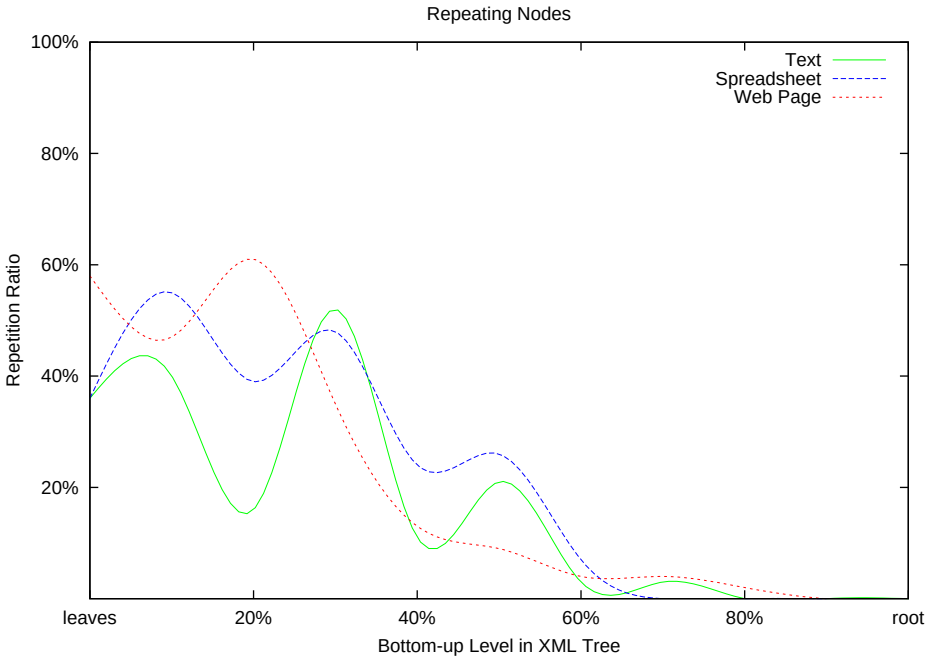


Figure 2.4: Identical markup identifiers lead to a high repetition ratio especially at the lower levels of the tree. The repetition ratio becomes very low towards the root due to the narrowness of the tree.

special case that there is only one node per level, the repetition ratio is 0, which is a likely case near the root node.

It is interesting that the analysis does not enforce the initiatory assumption. Figure 2.4 shows the repetition ratio on a relative scale. Among the leaf nodes, a repetition ratio of 36% for texts and spreadsheets, respectively 58% for Web pages is reached. A closer inspection reveals that many leaf nodes are in fact markup elements. They act as delimiter for formatting options, where the text nodes to format are not child nodes but siblings. Generally, the repetition ratio decreases in direction of the root node, which is mostly influenced by the narrowness of the trees near the root.

The analysis shows a wavelike decrease of the repetition ratio towards the root for all document types. Again, both ODF dialects for texts and spreadsheets show a similar behavior. Repeating nodes are frequent within the tree and at the leaves as well.

2.5 Modification Patterns

XML documents are usually not edited on the tree level. Applications provide user-friendly graphical interfaces to the underlying document tree. For Web sites, content management systems (CMS) assemble a Web site from a given repository. In this section, I analyze the

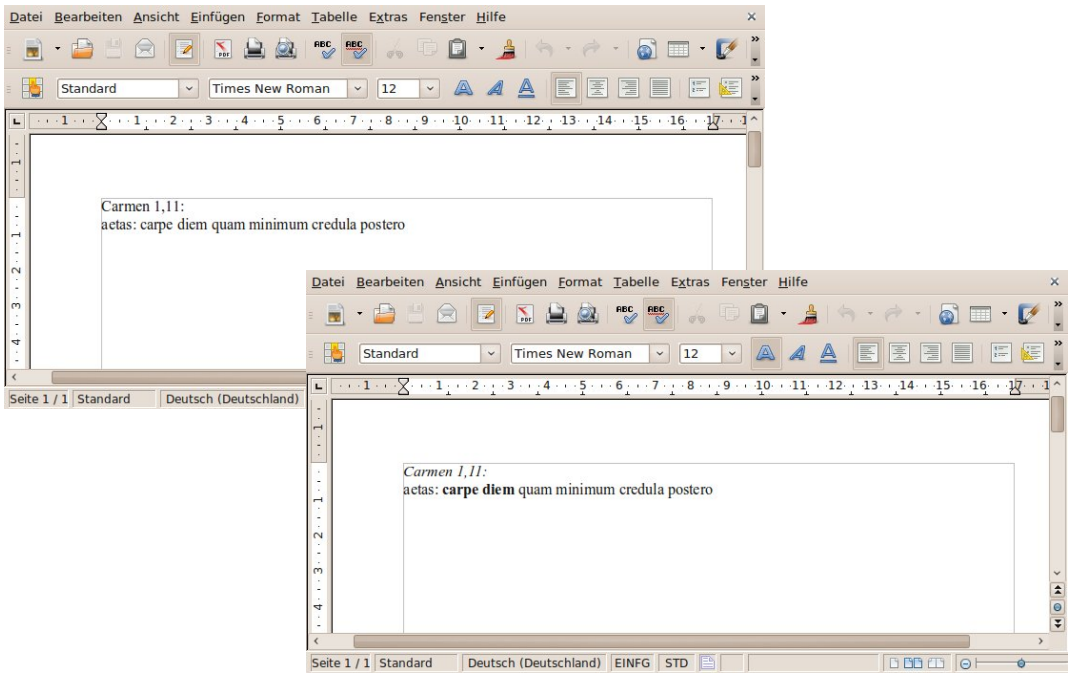


Figure 2.5: Two text snippets are highlighted using a different markup. These changes are not detected by the "compare documents" function of Open Office.

effects of changes on the application-level on the underlying XML tree.

2.5.1 Changes to Office Documents

Office documents are presumably edited using the corresponding office application. The office application tries to ensure a WYSIWYG (*what you see is what you get*) representation of the document. In this context, the document is displayed as it would be printed, not as it is stored in its XML representation. The navigation within the application is also tightened to the optical representation. This WYSIWYG approach appears to be very user-friendly, especially for the less experienced user. It is the de-facto standard for office applications since the 1990's [Myers 1998]. According to Meyrowitz and van Dam [1982], I call the screen representation and the interaction capabilities the *user model* of the document. On the other hand, I call the XML representation the *document model* of the document.

The user model and the document model may diverge. For example, a user can switch to the next line in a text displayed on his screen using the arrow-down button, even if the corresponding paragraph contains no new-lines in the document model. The divergence of the user model and the document model can have a more serious impact, indeed. Figure 2.5 shows two versions of a simple text document with two paragraphs in the user model. Some parts of the text are highlighted using italics and bold font face. If these both versions of the

document are compared by the built-in "compare documents" function of Open Office, the documents are reported to be equivalent, as the text did not change and the structure remained. Figure 2.6 shows the effects of these simple changes on the corresponding XML tree, i.e. the document model. Here, three changes emerge. First, the previously unknown font styles for italics and bold font face have to be defined using two adjacent subtrees. The style change of the whole paragraph is represented by an update of the attribute value. The highlighting of two single words within a paragraph, however, leads to a more complex change. The text node containing the paragraph is split up into three parts – the part before the style change, the highlighted words, and the remaining part of the paragraph.

The previous example shows that slight changes within the user model of a document may have a rather large impact on the XML representation. Boundaries within the two models do not correlate. An implicit page break is a "natural" boundary of a text that is basically not represented in the document model[7]. Vice versa, boundaries set by the tree representation may not be visible within the user model.

For spreadsheets, the user model and the document model are slightly closer. Each cell in the user model is basically represented by a node within the document model. Most edit operations act cell-oriented, which does not provide the same structure-changing impact compared to text documents. Interestingly, all cells within the matrix spanned by the first and the right-most outer-most filled cell are stored in the document model, no matter whether they are filled or not.

Resulting from the example above and similar tests, some assumptions can be made concerning the modification pattern of office documents:

Assumption 2.1 *A leaf node may be replaced by a subtree.*

Assumption 2.2 *Adjacent subtrees may be inserted or deleted.*

Assumption 2.3 *Structure-preserving changes (attribute changes) are a frequent operation.*

The second last assumption affects mostly text documents, but holds for spreadsheets as well. These assumptions are a refinement of my earlier work [Rönnau 2004]. I am not aware of any other investigation of the modification patterns in the XML representation of office documents.

2.5.2 Evolution of Web Pages

The evolution of Web pages has been studied before, e.g. by Fetterly et al. [2003]. However, the main interest of these studies was to obtain statistical data on the update frequency and update quantity for optimizing search engines [Ntoulas et al. 2004]. The quality of the changes has been mostly neglected. Recently, Adar et al. [2009] have performed a qualitative analysis

[7] To be precise, the office application may add a tag to the document, denoting an implicit page break at that point. It is mostly used for a faster layouting. However, the application does not have to respect this implicit page break.

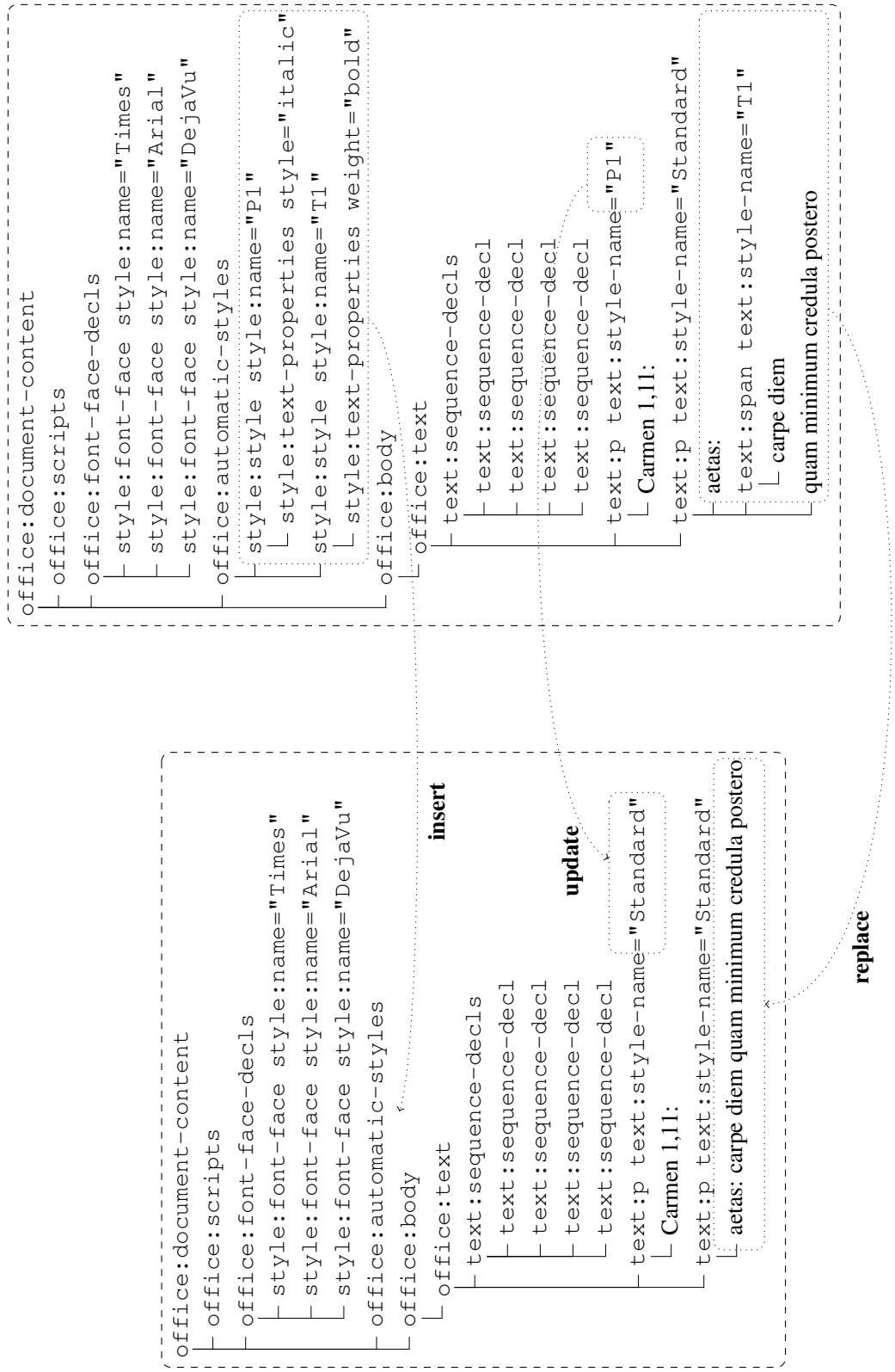


Figure 2.6: The simple markup changes shown in Figure 2.5 lead to three changes in the underlying XML tree. Two of these changes affect the tree structure significantly.

of Web page evolution. As a result, the structure of Web pages appears to be seldom changed. The content is visually organized in blocks that can be swapped or moved. Within these blocks, only minor content changes occur.

An example enforces these findings easily. Figure 2.7 shows the evolution of the Yahoo! portal site within one hour in three steps. On the top left, a list of articles is presented, including a teaser for one of these articles. As one can see, the list of articles is continuously re-arranged to attract as much readers as possible. On the bottom left, a box contains the newsworthy headlines. In this box, minor changes are performed on the headlines. New articles are basically inserted at the top, pushing the lowest headline out. The stock information is updated periodically without affecting the structure. On the right side, advertisements are placed. They are usually taken from a repository of possible advertisements depending on the advertising contract. The advertisement blocks change often, but the advertisements themselves are seldom change.

There is no "universal" way of representing blocks in XHTML. On some pages, the blocks are represented using adjacent subtrees, other pages use a single subtree. The reason for these differences lies in the fact that many different applications are used to create Web documents. A wide variety of Web page editors and content management systems exist which all use different serialization engines to map the user model onto the document model. Additionally, some parts of Web documents (e.g. advertisements) are provided by third parties, leading to a inconsistent representation of blocks. Nevertheless, modification patterns for Web documents can be formulated:

Assumption 2.4 *Structure-preserving changes (text changes) are a frequent operation.*

Assumption 2.5 *Subtrees are often re-arranged.*

Additionally, the insertion and deletion of adjacent subtrees appears to be common, which is already covered by Assumption 2.3.

2.5.3 Modification Amount and Distribution

Another important aspect of document evolution is the question of the modification amount, i.e. how many changes arise between two document versions. This question cannot be answered in general, as the ways of changing a document are numerous. For example, using "search and replace" when editing a document will most likely result in a large amount of changes covering the whole document. Nevertheless, assumptions on the typical modification amount can be made. In an analysis of several major Web sites with frequent changes, Adar et al. [2009] have shown that after two hours, 99.3% of all nodes remained unchanged. After one week, 91.7% of all nodes were still the same. This means that the modification amount is relatively small compared to the document size, even for frequent changes. A data-mining analysis of the change patterns on software repositories by Zimmermann et al. [2006] have shown that updated HTML and text files contained only few changes as well. I assume these

Figure 2.7: The evolution of the Yahoo! portal within one hour shows some typical properties of the evolution of Web documents. The pages are organized in blocks that mostly contain re-arrangements and small updates. The structure is seldom changed.

results to be applicable to the domain of office documents, too. This also confirms previously published assumptions on the modification properties of XML documents, which have been formulated without empiric background by Lee et al. [2004].

Assumption 2.6 *A new version of a document likely has only few changes compared to the size of the document.*

Beside the amount of changes, the distribution of the changes throughout the document is an interesting question. Hill et al. [1992] have developed a notion to display the distribution of changes and reading operations as well, called *edit wear* and *read wear*. They have shown that in general, only few parts of a document are read and changed at once. Large contiguous parts of the document remain unchanged and even unread.

Assumption 2.7 *Changes on a document affect only small parts of the document in general.*

2.6 Conclusions

XML is a powerful meta-language suitable for many tasks. Certainly, the simplicity of its design does not confirm the human perception of documents. Applications map the human-centric user model onto the well-specified document model.

In this chapter, I have analyzed different XML document formats, namely ODF for office documents and XHTML for Web pages. In the course of the analysis, I have studied the properties of "typical" XML documents, gained from public repositories and Web pages. As a main result, the XML document trees appear to be flat and very wide. Additionally, many identical node labels exist. In the following, I have studied common modification patterns. Simple changes on the user model may cause significant changes in the XML document model. The most frequent operations are subtree-oriented insert and delete operations, as well as updates of single nodes.

During the analysis, I have elaborated seven assumption on the modification patterns of documents. These assumptions help to design an appropriate delta model for representing changes between document versions. The knowledge on the tree properties helps to formulate heuristics for efficient differencing algorithms.

3 Differencing Strings and Trees

This chapter gives an overview of the basics in document differencing, with a special focus on their application to the XML domain. One of the first approaches to document differencing was the comparison of strings, with a focus on the longest common subsequence, which I describe first. Later on, algorithms for comparing other data structures have been developed. Since XML documents are a special case of trees, I describe the tree-to-tree editing problem afterwards. It founds the basis of most XML differencing approaches. The most common of them are presented at the end of this chapter.

3.1 String Editing

Strings can be regarded as a sequence of characters. Therefore, theoretical considerations concerning sequence comparison can be applied to the domain of string editing. The most common approach to comparing two strings is the computation of the longest common subsequence, which I introduce first. I briefly describe other notions like the Levenshtein distance afterwards. In the following, I discuss the complexity bounds, before giving a brief overview of the state-of-the-art in text differencing.

3.1.1 The Longest Common Subsequence

The longest common subsequence (LCS) of two strings has been defined by Wagner and Fischer [1974] as the maximum length of a sequence of characters being included in both strings. The common approach to find the LCS is to create a matrix, using both strings to compare as spanning vectors. The cells of the matrix contain the possible prefix combinations of the LCS at the given position.

Using this longest common subsequence, a sequence of *edit operations* can be created that transforms one string into another one. This sequence is also called an *edit script*. The available edit operations are *insert* and *delete*. Most approaches try to find a *minimum edit script* that contains as few edit operations as possible.

To show the computation of the LCS, the strings $S1 := BDAFDFCC$ and $S2 := CADABFC$ shall be compared exemplary. Table 3.1.1 shows the corresponding matrix. Each cell contains the LCS for the prefixes indicated by the indices of the cell. It is computed by taking the prefix LCS of the entries on the adjacent left and upper cell into account. Afterwards, the corresponding characters at the given position are compared. In case of a match, the character is appended to the longest prefix LCS. The cell in the lower right corner indicates one LCS of

	$_0$B	$_1$D	$_2$A	$_3$F	$_4$D	$_5$F	$_6$C	$_7$C
$_0$C	∅	∅	∅	∅	∅	∅	C	C
$_1$A	∅	∅	A	A	A	A	A,C	A,C
$_2$D	∅	**D**	A,D	A,D	AD	AD	AD	AD
$_3$A	∅	D	**DA**	DA	AD,DA	AD,DA	AD,DA	AD,DA
$_4$B	B	B,D	DA	DA	AD,DA	AD,DA	AD,DA	AD,DA
$_5$F	B	B,D	DA	**DAF**	DAF	ADF,DAF	ADF,DAF	ADF,DAF
$_6$C	B	B,D	DA	DAF	DAF	ADF,DAF	ADFC,**DAFC**	ADFC,DAFC

Table 3.1: The LCS can be computed using a matrix and is not necessarily unique. The bold entries highlight one possible path through the LCS matrix.

both strings. As shown in the example, the LCS may not be unique. *ADFC* and *DAFC* are both a valid LCS, which could be created on different paths through the matrix. A possible path is highlighted by a bold font-face. For this path, the resulting edit script transforming $S1$ into $S2$ would be as follows: *delete*($_0$B), *delete*($_4$D), *delete*($_5$F), *delete*($_7$C), *insert*($_0$C), *insert*($_1$A), *insert*($_4$B). Note that the unnecessary characters are removed first, thus transforming $S1$ into the LCS. Afterwards, the LCS is transformed into $S2$ by inserting the new characters. The edit operations must not be swapped, as each operation affects the numbering of the subsequent indices. To transform $S2$ into $S1$, the edit script must be inverted. Each *insert* has to be replaced by an *delete* operation and vice versa. Afterwards, the *delete* operations have to be moved to the beginning of the edit script, without breaking their respective order.

The edit script is intended to reduce the first sequence to the LCS, and to reconstruct the second sequence from the LCS. Deriving from this definition, an *update* operation which means the substitution of a single element is not possible. The same holds for a *move* operation that moves one element to another offset within the sequence. To allow for that kind of operations, the resulting edit script has to be parsed again to find matching pairs of *insert* and *delete* operations [Tichy 1984]. Update operations are easy to find, as the position within the sequence is equal. Move operations are only useful in short sequences or if the sequence does not have many replicating entries. In case of many replications, an unambiguous matching of moved nodes becomes impossible and the resulting edit script becomes less meaningful, as described by Chawathe et al. [1996].

3.1.2 Other Sequence Comparison Approaches

One important property of the LCS formulation is the ability to construct an edit script. Earlier work focused on the question to which degree two sequences are identical. For example, Levenshtein [1966] has defined a metric for the similarity of binary sequences. The Levensthein distance determines the length of the minimum edit script that contains not only the insertion and deletion of bits, but also the substitution of a single bit.

The formulation of the string-to-string correction problem is based on prior work on spelling

correction by Morgan [1970], basing on insert, delete, and update operations, too. Also in the context of bio-nuclear sequences, the problem of finding the longest common subsequence by allowing for inserted or deleted elements has been investigated by Sankoff [1972]. However, these approaches just focused on the computation of the length of the LCS, not on the creation of the edit script.

According to Ishida et al. [2005], recent research focuses on the question of efficiently computing the LCS from the end of the sequences to the beginning. The main application for this is log-file analysis and suffix comparison of biological sequences. Also, the incremental comparison of increasing sequences has been studied by Landau et al. [1998], which is highly useful for the analysis of streams. Other approaches, e.g. by Cormode and Muthukrishnan [2007] increase the computational speed by allowing non-optimal results.

All these approaches do not relate well to the problem statement of comparing document versions, where an exact knowledge about the changes themselves is needed, including the corresponding edit script. Therefore, I will not consider these approaches any further.

3.1.3 Complexity

Basically, the LCS computation can be solved in quadratic time using a dynamic programming approach. The lower bound is $O(n^2)$, with n denoting the length of the sequences. The lower bound relates to two equal sequences to compare. One significant optimization can be done by restraining the input alphabet. A limited input alphabet with size σ allows for a lower bound of $O(\sigma n)$ [Bergroth et al. 2000]. Especially for large documents, this is an important benefit in terms of efficiency, especially for strings. In terms of XML, however, this optimization loses much of its achievements. In this domain, the input alphabet is limited, too, but can be arbitrarily enlarged. Assuming a node-centric granularity, the input alphabet relates to the amount of different nodes within a document to compare. Section 2.4 has shown that the amount of different nodes may be significant. Therefore, this optimization is not suitable for the domain of XML documents.

Beside the time complexity, the space complexity is an important issue when using LCS algorithms. Most approaches compare two strings with sizes n and m using a matrix as shown in the example above. This leads to an $O(nm)$ space complexity. Using clever heuristics, the search space within the matrix can be restrained, thus avoiding the computation of the complete matrix. The main idea is to drop search paths within the matrix that do not promise a meaningful result. As shown in the previous example, most of the entries within the matrix are duplicates that do not offer any new information. Many of the entries would never be reached by a search algorithm for the LCS and may be skipped during computation. Most heuristics still ensure the generation of a minimum edit script.

To illustrate the need for a sub-quadratic space complexity, consider following simple example: An ODF document can easily reach 10.000 nodes. Comparing all nodes using a complete matrix requires 100.000.000 entries. Even if assuming only 4 Bytes per matrix entry, this results in a memory consumption of over 381 MByte just for the LCS matrix. As the example in Table 3.1.1 shows, most entries within the matrix contain several LCS candidates. Therefore,

the actual memory consumption of the matrix would actually be far higher.

Some algorithms are able to compute the LCS with linear space, which is an important achievement when applying the LCS problem to XML documents. One of the best studied algorithms with a good general applicability has been presented by Myers [1986]. It is primarily intended for use-cases where the sequences to compare are similar to a high degree. The algorithms runs in $O(ND)$ time, where $N = n + m$ denotes the summarized size of the compared documents, and D denotes the size of the minimum edit script. Apparently, it is well suitable for documents with $N \gg D$, which relates to a high similarity of the sequences. Its space complexity evolves in a linear way with $O(N)$. The algorithm does not make any assumptions on the size of the input alphabet, which ensures the applicability on the domain of XML documents.

3.1.4 Differencing Line-Based Documents

In the beginnings of the computer age, input and output was performed using punch-cards. Later on, cathode-ray tubes (CRTs) were used for displaying files. These CRTs did not offer a graphics capability that is common nowadays. They were only able to display characters line-by-line [Irons and Djorup 1972]. This line-based orientation is suitable for the domain of document editing, as texts are usually displayed line-by-line, too. The graphical representation of documents was closely tightened to the capabilities of typewriters. For these reasons, first text editing applications stored their files as they have been displayed on the screen. This line-based orientation was also standardized as American standard code for information interchange (ASCII) [Gorn et al. 1963]. This format and its successors are still a popular form for storing text-based documents.

To compare text documents, the sequence of the lines are compared using LCS algorithms. This application is called *diff* and has been introduced by Hunt and McIlroy [1976]. Each differing line leads to a corresponding insertion and deletion of a line. In this approach, lines are considered as atomic entities, which are not inspected further. Additionally, lines are only represented as hash values for the sake of efficiency, as proposed by Miller and Myers [1985]. If each line would be analyzed respectively using the LCS, the runtime of the algorithms would be unusable. Another reason for comparing lines using their hash values is the intended application. Version control systems for source code were the first to use a diff tool to reveal changes during development of software [Tichy 1985]. As lines in source code are rather short, it is quite easy for the human reader to gather the differences between two lines without computer-assistance. For lines containing contiguous text, however, finding the differences manually appears to be an error-prone task [Neuwirth et al. 1992].

The result of a diff run is called a *delta*. Wall [1985] has presented a tool, called *patch*, which is able to apply a delta to the former version of a document. With the usage of diff, patch, and deltas, a repository can be hold up to date efficiently, as only changes are exchanged. Full document versions are seldom transmitted.

3.2 Tree Editing

Early work on tree editing has been highly influenced by a graph-theoretical view towards the tree. With the rise of tree-oriented document languages like XML, other interpretations of the tree property have become popular. The most important distinction between these views is the definition of the edit operations. In general, graph-oriented approaches have a node-centric view towards the change granularity. Document-centric approaches tend to have a subtree-oriented view.

Here, I discuss the graph-oriented aspects of tree editing, as they also build the basis for a subtree-oriented view. First, I introduce the tree-to-tree editing problem and the tree edit distance, which directly relates to it. For the sake of completeness, I shall mention other approaches briefly, before discussing the complexity bounds. Finally, I present first approaches to differencing tree-based documents.

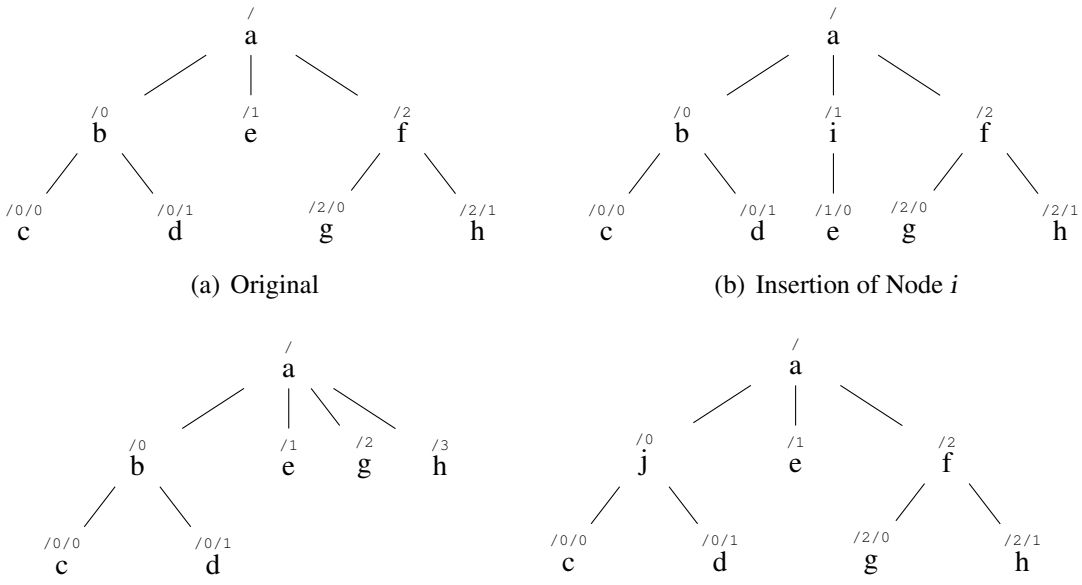
3.2.1 The Tree-to-Tree Editing Problem

Finding and describing the changes between two ordered, labeled trees has been introduced as tree-to-tree editing problem by Selkow [1977]. It is an application of the string editing problem to the domain of trees. From the very first definition, tree editing included not only insert and delete operations, but also an update operation that does not affect the structure of the tree. Only leaf nodes can be inserted or deleted. If the inner structure of the tree has to be changed, the corresponding subtree has to be deleted node by node from the leaves up to the designated position, and re-inserted in reversed order node by node.

Obviously, Selkow's definition of insert and delete operations requires a large amount of operations to represent changes on the tree structure. To overcome this, Tai [1979] has introduced a different notion of tree edit operations that allows for inserting and deleting nodes arbitrarily within the tree. If a node being the root of a subtree is deleted, its subtree is placed in the position of the deleted node, which means that all nodes in the subtree are pulled one tree level towards the root. Vice versa, an inserted node pushes an already existent node forest down one level, becoming the new root of that subtree. According to Bille [2005], this behavior has become the most common node-based edit model of trees. Nevertheless, Selkow's view on leaf-level editing of trees has prevailed in the document-oriented algorithms of Chawathe et al. [1996].

Figure 3.1 shows examples for the edit operations using Tai's edit model. In contrast to this, Selkow's edit model would result in three edit operations for the insertion of node i, that is, *delete*(/1 e), *insert*(/1 i), *insert*(/1/0 e). The deletion of node f would require five operations instead of two, namely *delete*(/2/0 g), *delete*(/2/1 h), *delete*(/2 f), *insert*(/2 g), *insert*(/3 h). Apparently, Tai's model creates significant shorter edit scripts.

Creating a minimum edit script is a challenging task in the domain of tree editing. Whereas the cost of edit operations are usually neglected in the domain of string editing, some tree-editing approaches use a cost-model to take the complexity of operations into account. Each operation is assigned a cost value. The cost of an edit script is the sum of the costs of all edit



(a) Original (b) Insertion of Node *i*

(c) Deletion of Node *f* (d) Update of Node *b* to *j*

Figure 3.1: Inserting a node pushes an existing subtree towards the bottom. Deleting a node lets the descendants raise a level within the tree. Updates are structure-preserving.

a	b	c	d	e	f	g	h

(a) Preorder Traversal

c	d	b	e	g	h	f	a

(b) Postorder Traversal

Figure 3.2: Tree traversal algorithms align the tree onto a sequence. This example shows two possible traversals of the tree displayed in Figure 3.1(a).

operations of this script. The *edit distance* between two trees is the minimum cost of all edit scripts. This view has been introduced by Tai [1979]. The introduction of operation-specific costs is motivated by the consideration that the structure-preserving update operation is less costly in most implementations of trees. Usually, a fixed-cost model is used, assigning each edit operation type a static cost value.

The solution of the tree-to-tree editing problem is not as straightforward as in the sequence comparison domain. Due to their two-dimensional structure, trees do not offer a "natural" linear ordering. Tree traversal algorithms may map the tree structure onto a sequence, hence breaking the parent-children relationship. There are two major traversal algorithms that are commonly used in tree editing algorithms. The *preorder traversal* starts at the root of the tree and visits the subtrees starting at the left-most child, resulting in a *top-down* pass of the tree. The *postorder traversal* is a *bottom-up* pass of the tree, starting at the left-most leaf. Both traversal types are exemplary shown in Figure 3.2. The used traversal algorithm has a significant influence on the resulting edit script. As there usually exist different edit scripts

with equal edit cost, different tree traversal algorithms lead to completely different results, as shown by Barnard et al. [1995].

3.2.2 Other Approaches

Jiang et al. [1995] have introduced tree alignment as an alternative to the tree editing problem described before. The main idea is to first transform the structure of the first tree into the other one using insert and delete operations, regardless of their node labels. Afterwards, the node labels are adjusted using update operations. While the tree edit distance computes the largest common subtree, the tree alignment is used to find the smallest common supertree. A major drawback of this approach is the high increase of edit operations in some scenarios compared to the tree edit distance. However, even the fastest known tree alignment algorithm by Jansson and Lingas [2001] does not provide a gain of efficiency. Therefore, I do not follow this approach.

The tree edit distance is an important factor in the natural language processing domain [Mehdad 2009]. There, the closeness of two trees is most important. Therefore, some research focuses on efficiently computing the tree edit distance, without being able to create an edit script efficiently. Additionally, approximations to the tree edit distance have become popular. Bernard et al. [2008] have introduced a probabilistic cost model for efficient approximation. As the work in this domain neglects the generation of an edit script, I will not follow these approaches either.

All the approaches mentioned up to now considered every node individually. In contrast to this, Valiente [2001] has defined the bottom-up distance of two trees. From his point of view, a node is only considered to be unchanged if none of its descendants has been changed. Using this definition, if one leaf node has been changed, all its ancestors up to the root node are considered to be changed, too. Apparently, this leads to a high increase of edit operations. Although the bottom-up distance is efficient to compute, I do not consider this edit model any further. As each small change results in comprehensive edit scripts, this approach is inappropriate for handling documents for the reason of meaningfulness and space efficiency of the result.

The node-centric definition of insert and delete operations is not suitable for each domain. Beside from document editing and natural language processing, trees are widely used to represent RNA structures in the domain of bioinformatics. The applicability of tree editing for RNA sequence comparison has been shown by Shapiro [1988]. As the RNA has a very limited alphabet, the edit scripts may include operations that do not reflect the nature of the change. Allali and Sagot [2004] have introduced more complex edit operations that allow for more meaningful change representation in the domain of RNA comparison. As this is clearly not the domain of document editing, I do not follow this approach. However, the need for more domain-specific edit operations arises in the document editing domain, too. I will discuss this issue in Section 3.2.4.

3.2.3 Complexity

The tree-to-tree correction problem is usually solved using a dynamic programming approach. For two trees T_1 and T_2, a basic algorithm computes a minimum edit script in $O(|T_1|^2 \times |T_2|^2)$, where $|T_1|$ denotes the number of nodes of T_1 [Bille 2005]. The basic algorithm can be improved by reducing the search space, still ensuring a minimum edit script. Zhang and Shasha [1989] have presented an efficient algorithm that is referred to as best-known general solution. It runs in of $O(|T_1| \times |T_2| \times \min\{\text{depth}(T_1), \text{leaves}(T_1)\} \times \min\{\text{depth}(T_2), \text{leaves}(T_2)\})$. This lowers the complexity bound especially for flat or thin and deep trees. Chen [2001] has presented an algorithm running in $O(|T_1| \times |T_2| + \min\{\text{leaves}(T_1)^2 \times |T_2| + \text{leaves}(T_1)^{2.5} \times \text{leaves}(T_2),\ \text{leaves}(T_2)^2 \times |T_1| + \text{leaves}(T_2)^{2.5} \times \text{leaves}(T_1)\})$ time . This algorithm performs better for thin and deep trees, but is inferior to the approach by Zhang and Shasha in the general case.

Refinements of Zhang and Shasha's algorithm have been presented by Klein [1998] and Touzet [2005]. Klein's algorithm has a better worst-case complexity, but does not perform as well for flat or thin and deep trees. Therefore, it is not suitable for XML documents. Touzet's algorithm requires the maximum tree edit distance to be known in advance. Her algorithm is only more efficient if this maximum edit distance is noticeably lower than the amount of leaves or the depth of the trees to compare. However, it is unlikely to learn about the maximum degree of changes between document versions without prior similarity analysis which cost must be taken into account as well.

All known solutions that compute a minimum edit script still offer a super-quadratic time complexity, which is far less efficient than in the sequence comparison domain. To achieve better complexity bounds, the goal of an edit script conforming to the tree edit distance is dropped. Nevertheless, the edit script should be close to the tree edit distance. Zhang [1996a] has introduced the *constrained edit distance*, where disjoint subtrees are not mapped onto each other, thus reducing the complexity to $O(|T_1| \times |T_2|)$. Chawathe et al. [1996] have introduced the *weighted edit distance*, where the size of subtrees is taken into account when computing the edit distance. Using some assumptions[1] on the characteristics of the input trees, the time complexity bounds can be lowered to $O((\text{leaves}(T_1) + \text{leaves}(T_2)) \times e + e^2)$, with e denoting the weighted edit distance. This is obviously far more efficient than all approaches mentioned previously.

Instead of reducing the tree edit distance property, Zhang [1996b] has proposed to solve the tree-to-tree editing problem more quickly by using parallel algorithms. Since multi-core processors have become standard on today's computers, this is a promising idea. However, the separation of the trees to compare into different parts restricts the computed edit script to leaf-oriented insert and delete operations according to Selkow's edit model. The time complexity of the proposed algorithm yields $O((\log(\text{leaves}(T_1) + \log(\text{leaves}(T_2)) \times \log(|T_1|) \times \log(|T_2|))$. Additionally, the processor complexity of $O((|T_1| \times |T_2|)/\log(\min\{|T_1|, |T_2|\}))$ has to be taken into account. Zhang has related his algorithm to the parallel sequence editing algorithm of

[1] I will discuss the mentioned tree characteristics in more detail in Section 3.3.3.

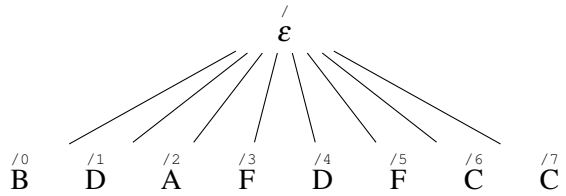


Figure 3.3: Tree-to-tree editing is a generalization of string editing. Strings can be represented by a tree with a height of two.

Apostolico et al. [1990]. According to Apostolico et al., even the best parallel algorithm is inferior to sequential algorithms by means of complexity. As the distribution overhead in parallel algorithm is significant and the change model is inappropriate, I do not follow this approach.

As already mentioned, I only consider ordered trees. Beside domain-specific reasons for that decision, Zhang et al. [1995] have proven the NP-hardness of the tree-to-tree correction problem for unordered trees in general. Later, Akutsu et al. [2008] has shown that the largest common subtree of two unordered trees with bounded height can be computed in polynomial time. Obviously, these complexity bounds do not allow for computing the edit script in acceptable time.

Also in the domain of tree editing, the space complexity is an important yet often forgotten aspect. As with string editing approaches, a matrix is usually used to compute the edit script, thus yielding a quadratic complexity. Wang and Zhang [2005] have presented an optimized algorithm with $O(\log(|T_1|) \times |T_2|)$ space complexity. To the best of my knowledge, this is the only published improvement in this domain. To be able to compute large trees, Chawathe [1999] has proposed an algorithm using external memory, i.e. hard disks.

3.2.4 Differencing Tree-Based Documents

From its intention, the tree-to-tree editing problem has been seen as generalization of the string editing problem [Selkow 1977]. In this view, the characters of the strings are the leaves of a tree with height of two and are therefore not directly connected to each other. Figure 3.3 shows an example string. This view has lead to Selkow's definition of leaf-based insert and delete operations. Later, the string-oriented view towards trees as been relaxed. Most of the work on tree editing in the 1980's and 1990's has been influenced by graph-theoretical aspects, mostly reflected by Tai's interpretation of inner-tree insert and delete operations.

With the rise of highly-structured tree-oriented document formats like SGML, the tree editing problem has been revisited from a document editing perspective. Barnard et al. [1995] have analyzed different tree editing algorithms as well as different modification scenarios of SGML documents. As a result, the insertion and deletion of whole subtrees appeared to be a common task in these scenarios, which are no discrete operations in the traditional tree editing algorithms. To solve this drawback, Barnard et al. [1995] have developed an extension

to the algorithm by Zhang and Shasha [1989], thus allowing for subtree insertion and deletion. Nevertheless, this algorithm still relies on the tree edit distance that does not account for subtree-oriented changes. Therefore, it is unclear how often subtree changes are actually detected. Unfortunately, no evaluation examining the use of subtree changes has been performed.

LaTeX-code can be regarded as tree-oriented document, too. An efficient algorithm for the comparison of trees (with LaTeX-documents in mind) has been presented by Chawathe et al. [1996]. They have adapted the leaf-based edit operations of Selkow [1977]. According to their empirical evaluation, the tree-oriented view appears to be much more suitable for this kind of documents than the line-based view of the conventional diff tool.

3.3 Differencing XML Documents

With the increasing spread of XML data formats, the tree-to-tree correction problem has been revisited. Some tools have been presented that apply existing tree differencing algorithms to XML documents. Other researchers have questioned the existing change models, creating new ones.

In this section, I present four XML differencing tools that implement tree differencing algorithms. After presenting the selection criteria, I briefly present the tools including their underlying algorithm and output format. For the sake of completeness, several non-competitive approaches are mentioned at the end of the section, before discussing the importance of the change representation model.

3.3.1 Selection Criteria and Comparison

A major distinction between XML differencing tools is the algorithm used. All selected tools use different algorithms. As a precondition, the differencing algorithm had to be published to allow for a meaningful investigation. Additionally, an implementation had to be available, at least as research prototype.

For all XML differencing tools, the basic idea is explained. Additionally, the time complexity of the algorithm used is presented. As most of the algorithms have no published statement on their space complexity, it is only mentioned if known.

Another important aspect of a tool is the representation of changes. In the last section, different *change models* have been presented, e.g. the node-oriented editing by Tai [1979] or the leaf-oriented model by Selkow [1977]. The question how these edit operations are represented within a delta has not been discussed yet. In this context, the addressing of an edit operation plays a major role. Each of the presented tools uses a different change model. I do not intend to present the change model of all tools in detail. However, a glance at the generated output quickly shows significant differences. As I relate XML differencing tools to the domain of XML document evolution, the text documents presented in Figure 2.6 (see Section 2.5.1) are used as test case. I repeat the properties of this test case. A simple document, containing two

paragraphs is changed by applying a bold font face and italics. As a result, the underlying XML tree is changed not only at the two parts where the formatting changed. Additionally, new style definitions are added to the document. From a human perspective, 3 changes have been performed on the trees, if "replace" is considered as one operation. Otherwise, 4 operations are needed. As the definition of the tree edit distance does not allow for subtree-oriented operations, the edit distance between both trees numbers to 8. All presented XML tools have to compare these two versions.

3.3.2 Microsoft XML Diff (Zhang-Shasha)

Earlier, the algorithm by Zhang and Shasha [1989] was referred to be the best known general-purpose algorithm for ordered trees. It solves the tree-to-tree editing problem in its formulation by Tai [1979], thus allowing for inserting and deleting nodes arbitrarily throughout the tree. The tree edit distance is computed using a matrix. The basic idea of this algorithm is to restrain the search path throughout the matrix, thus computing only the needed parts. This way, the algorithm computes an optimal solution in terms of the edit distance with time complexity[2] $O(|A_1| \times |A_2| \times \min\{\text{depth}(A_1), \text{leaves}(A_1)\} \times \min\{\text{depth}(A_2), \text{leaves}(A_2)\})$. As XML documents appear to be very flat compared to their size, the time complexity can be converged to a quadratic complexity. The space complexity of the algorithm is quadratic with $O(|A_1| \times |A_2|)$.

Microsoft has developed an XML differencing tool and claims that it implements the Zhang-Shasha algorithm. It is part of the .NET framework and is implemented using C#. The source code is un-disclosed. Additionally, the documentation is rather short, which makes it difficult to retrace the behavior of the implementation. Being strict, this tool does not meet the selection criteria presented before. I decided to discuss this tool despite these deficiencies for different reasons. First, it is the only implementation of the Zhang-Shasha algorithm for XML documents that has left the stadium of a prototype. Second, the delta model is different from the other approaches and should be presented, too. Third, this tool is widely used, which makes it the state-of-the-art on Windows systems. Therefore, it should be considered in the comparative evaluation in Chapter 10.

A curiosity of this implementation is the change model used. Whereas the algorithm by Zhang and Shasha has been designed to support insert and delete operation only on the node level, the Microsoft XML diff also detects subtree insertions and deletions. Additionally, a move operation is supported, which is not known in the original definition of the algorithm either. Following a black-box interpretation, I assume this XML diff tool to be a refinement of the SGML differencing algorithm by Barnard et al. [1995].

The output of the test case is shown in Figure 3.4. As an interesting property, a hash value of the original document version is stored. This is done to prevent the delta to be applied to another document, which would possibly create erroneous results. The idea of the delta is to traverse the target document. Only parts that contain changes are accessed, indicated

[2] Here, I use the notation of A for a document to distinct between the tree differencing algorithms and their applications to the XML differencing domain.

```
xd:xmldiff srcDocHash="375842907363673063"
└ xd:node match="2"
  ├ xd:node match="3"
  │ └ xd:add
  │   ├ style:style style:name="P1"
  │   │ └ style:text-properties style="italic"
  │   └ style:style style:name="T1"
  │     └ style:text-properties weight="bold"
  └ xd:node match="4"
    └ xd:node match="1"
      ├ xd:node match="2"
      │ └ xd:change match="@text:style-name"
      │   └ P1
      └ xd:node match="3"
        ├ xd:change match="1"
        │ └ aetas:
        └ xd:add
          ├ text:span text:style-name="T1"
          │ └ carpe diem
          └ quam minimum credula postero
```

Figure 3.4: Microsoft XML Diff uses a tree-walking relative addressing of nodes.

by instructions of the form `xd:node match="2"`. If the correct position for a change is reached, the change itself is applied, e.g. by `xd:add`. In total, this delta contains 4 edit operations, which corresponds with the human interpretation of the changes to the tree. Note that these edit operations do not correspond to the original definition of the algorithm which ensures a minimum edit script in terms of the minimum edit distance, thus resulting in 8 edit operations.

3.3.3 Diffxml (Chawathe)

Chawathe et al. [1995] have revisited the tree-to-tree editing problem with hierarchically structured data in mind. As a result, they favored Selkow's leaf-oriented edit model over Tai's edit model allowing for arbitrary insert and delete operations. Additionally, they have introduced a *move* operation for subtrees. A corresponding differencing algorithm has been presented later, called FastMatch EditScript (FMES) [Chawathe et al. 1996]. Its initial intention was to find differences in LaTeX document versions. The basic idea is to compute the longest common subsequence of leaves to detect insert and delete operations. FMES works in five phases. In the first phase, structure-preserving update operations are identified. In a second phase, re-arranged subtrees are identified, which are represented using move operations. In the next three phases, the tree is transformed into its target appearance using insert, move, and delete operations. The XML tree is basically parsed bottom-up.

FMES does not guarantee an optimal solution in terms of the tree edit distance. This allows

for reducing the complexity bounds significantly. However, a near-optimal solution shall be ensured in the general case. FMES has a time complexity of $O((\text{leaves}(A_1) + \text{leaves}(A_2)) \times e + e^2)$, with e denoting the *weighted edit distance*, i.e. the length of the computed edit script. The ratio of e/d, where d denotes the tree edit distance, is bounded by $\log n$ for the general case. The general case only holds under the assumption that there exist only few leaves with identical label [Chawathe et al. 1995]. In Section 2.4.3, however, I have shown that typical XML documents contain many repeating leaves. A comparative evaluation in Section 10.4 will show the significant influence on the size of the edit script. The space complexity of FMES has not been investigated by the authors. Performing a sketchy review of the algorithm, I estimate the algorithm to scale linearly in terms of the input size.

Three XML-aware implementations of FMES are available. The first one has been presented by Logilab, a French software manufacturer. The implementation uses the Python programming language, and is part of many Linux distributions under the name xmldiff. The second implementation has been developed by Mouat [2002], called diffxml. It uses Java and has been updated recently. Hottinger and Meyer [2005] have reviewed the xmldiff implementation, stating that the implementation does not fully conform the FMES algorithm. Additionally, this implementation is not competitive in terms of speed, as shown in an empirical scenario by Rönnau et al. [2005]. Hottinger and Meyer [2005] have also presented an own implementation of FMES. However, this implementation is yet in a prototype state and not usable due to several small bugs. Therefore, only diffxml will be considered further on.

In diffxml, attributes are regarded as independent nodes, too. Each edit operation is addressed using an absolute path in XPath notation. Figure 3.5 shows the resulting delta for the test case. It contains 18 edit operations, which is far above the minimum edit script of 8 operations. The reason being that an insertion of a single node is represented by several insert operations, one for the node label, and one for each attribute. The delta model of diffxml is well-defined, yet cumbersome. Additionally, the XPath notation used for the addressing of nodes is hard to read and inefficient in terms of space.

3.3.4 XyDiff (Cobena)

Marian et al. [2001] have tried to track the changes of XML documents on the Web in the context of a data warehouse. They have formulated an edit model that allows for inserting, deleting, and moving subtrees. Single nodes can be addressed by update operations. This edit model is caused by a deductive analysis of a given XML tree. Marian et al. have decided to completely discard the node-orientation for insert and delete operations that is used in the formulation of the tree-to-tree editing problem. Based on this edit model, Cobéna et al. [2002] have presented a new differencing algorithm for XML documents, called XyDiff. This algorithm lowers the complexity bound significantly again. Normally, XyDiff runs in linear time and space with $O(|A_1| + |A_2|)$. The worst case time complexity is $O(|A_1| + |A_2| \times \log(|A_1| + |A_2|))$. This increase of efficiency is gained for the sake of the minimality of the resulting edit script. The basic idea of XyDiff is the so-called *bottom-up, lazy down* (BULD) propagation. XyDiff traverses the tree bottom-up, propagating matching

```
delta
├─ insert childno="1" name="style:style" nodetype="1" \
│      parent="/node()[1]/node()[3]"/
├─ insert name="style:name" nodetype="2" \
│  │   parent="/node()[1]/node()[3]/node()[1]"
│  └─ P1
├─ insert childno="2" name="style:style" nodetype="1" \
│      parent="/node()[1]/node()[3]"
├─ insert name="style:style" nodetype="2" \
│  │   parent="/node()[1]/node()[3]/node()[2]"
│  └─ T1
├─ insert childno="1" name="style:text-properties" nodetype="1" \
│      parent="/node()[1]/node()[3]/node()[1]"/
├─ insert name="style" nodetype="2" \
│  │   parent="/node()[1]/node()[3]/node()[1]/node()[1]"
│  └─ italic
├─ insert childno="1" name="style:text-properties" nodetype="1" \
│      parent="/node()[1]/node()[3]/node()[2]"/
├─ insert name="weight" nodetype="2" \
│  │   parent="/node()[1]/node()[3]/node()[2]/node()[1]"
│  └─ bold
├─ insert childno="2" name="text:p" nodetype="1" \
│      parent="/node()[1]/node()[4]/node()[1]"/
├─ insert name="text:style-name" nodetype="2" \
│  │   parent="/node()[1]/node()[4]/node()[1]/node()[2]"
│  └─ P1
├─ move childno="1" length="12" new_charpos="1" old_charpos="1" \
│      parent="/node()[1]/node()[4]/node()[1]/node()[2]"
│      node="/node()[1]/node()[4]/node()[1]/node()[3]/node()[1]"
├─ insert childno="1" nodetype="3" \
│  │   parent="/node()[1]/node()[4]/node()[1]/node()[3]"
│  └─ aetas
├─ insert charpos="8" childno="2" name="text:span" nodetype="1" \
│      parent="/node()[1]/node()[4]/node()[1]/node()[3]"
├─ insert name="text:style-name" nodetype="2" \
│  │   parent="/node()[1]/node()[4]/node()[1]/node()[3]/node()[2]"
│  └─ T1
├─ insert childno="3" nodetype="3" \
│  │   parent="/node()[1]/node()[4]/node()[1]/node()[3]"
│  └─ quam minimum credula postero
├─ insert childno="1" nodetype="3" \
│  │   parent="/node()[1]/node()[4]/node()[1]/node()[3]/node()[2]"
│  └─ carpe diem
├─ delete charpos="1" length="46" \
│      node="/node()[1]/node()[4]/node()[1]/node()[4]/node()[1]""
└─ delete node="/node()[1]/node()[4]/node()[1]/node()[4]"
```

Figure 3.5: Diffxml relies on a leaf-oriented change model with attribute granularity, resulting in a large amount of edit operations.

```
delta
├ Deleted pos="0:1:3:0:2:0"
│ └ aetas: carpe diem quam minimum credula postero
├ Inserted pos="0:1:2:0"
│ └ style:style style:name="P1"
│   └ style:text-properties style="italic"
├ Inserted pos="0:1:2:1"
│ └ style:style style:name="T1"
│   └ style:text-properties weight="bold"
├ Inserted pos="0:1:3:0:2:0"
│ └ aetas:
├ Inserted pos="0:1:3:0:2:1"
│ └ text:span text:style-name="T1"
│   └ carpe diem
├ Inserted pos="0:1:3:0:2:2"
│ └ quam minimum credula postero
└ AttributeUpdated nv="P1" name="text:style-name" \
   ov="Standard" pos="0:1:3:0:1"
```

Figure 3.6: JXyDiff uses absolute paths. Insertions and deletions target subtrees. Attributes can be updated directly. Old values are stored in the delta to reconstruct former versions of the document.

nodes towards the root. Nodes are weighted using the amount of descendants. In a subsequent top-down pass, the node weight is used to quickly decide whether a modified subtree will be deleted with its original appearance and inserted with its target appearance again, or whether a set of more fine-granular operations is used, which means inserts on lower levels or updates. Generally speaking, the "brute-force" method by deleting and inserting large subtrees is preferred.

XyDiff has originally been implemented using C on Linux. Unfortunately, this implementation is no longer being maintained. As a result, it can only be used on highly modified Linux systems with outdated libraries and compiler versions. A Java implementation, called jXyDiff, has been developed later. However, this implementation lacks a corresponding patch tool to apply a computed delta to an XML document.

The output format of (j)XyDiff is straightforward and intuitive, the delta is shown in Figure 3.6. Each operation uses an absolute path. Delete and update operations store the old value of the changed parts, which allows for reconstructing the former version of a document using the delta. In total, seven edit operations are used to represent the changes.

3.3.5 Faxma (Lindholm)

All previously mentioned approaches traverse an XML document using its tree representation. Another approach, called faxma, has been presented by Lindholm et al. [2006]. They applied the idea of greedy matching of sequences to the domain of XML. Originally, this approach

```
diff:diff op="insert"
└ ref:node id="/0"
  ├ diff:copy run="2" src="./0"
  ├ ref:node id="./2"
  │ ├ style:style style:name="P1"
  │ │ └ style:text-properties style="italic"
  │ └ style:style style:name="T1"
  │   └ style:text-properties weight="bold"
  └ ref:node id="./3"
    └ ref:node id="./0"
      ├ diff:copy run="1" src="./0"
      ├ text:p text:style-name="P1"
      │ └ diff:copy run="1" src="./1/0"
      └ ref:node id="./2"
        ├ aetas:
        ├ text:span text:style-name="T1"
        │ └ carpe diem
        └ quam minimum credula postero
```

Figure 3.7: The output of faxma is a script, containing references to the original document and inserted nodes.

has been used in the domain of binary synchronization algorithms by Tridgell [1999]. A sliding window is used to compare two input sequences using highly efficient rolling-hash algorithms that have been introduced by Rabin [1981]. An XML document is transformed into a sequence using the XAS format presented by Kangasharju and Lindholm [2005]. In this XAS format, nodes are transformed in document order to tokens in a sequence stream. This tokenization prevents the greedy matcher to split the XML document into unparsable fragments. In the general case, faxma runs in linear time and space with $O(|A_1| + |A_2|)$. Faxma has been implemented using Java.

An interesting property of faxma is the change model used. Due to its linear parsing of the document, an edit script in its conventional form is not created. Instead, the output of the document consists of the inserted parts and references to the unchanged parts of the original document. Figure 3.7 shows the delta for the test case. The commands `ref:node` and `diff:copy` reference to a single node or several subtrees, respectively. This change representation is efficient in terms of space, but hard to read by humans. To some extent, it resembles the Microsoft XML Diff format.

3.3.6 Other Approaches

One important distinguishing factor for the selection of XML differencing algorithms is the availability of an implementation to verify the claimed achievements. Xu et al. [2002] have claimed to have developed an XML differencing algorithm supporting the ordered and unordered tree model running in linear time. Unfortunately, I was not able to get the implemented

system. As the algorithm description is rather imprecise, a new implementation cannot be performed using that description. Some approaches do only support selected properties of XML document. For example, the approach by Lee et al. [2004] neglects attributes, which lowers the applicability of this approach significantly. Other approaches, as presented by Al-Ekram et al. [2005] or Iorio et al. [2009], do not provide a better time complexity than previous approaches.

Some approaches make domain-specific assumptions on the XML format. BioDIFF for example has been designed for the annotation of biological data [Song et al. 2007]. In the domain of bioinformatics, Hedeler and Paton [2008] have shown the shortcomings of general-purpose differencing algorithms.

Differencing XML documents has also been investigated from a graph-theoretical perspective. A corresponding tool, SSDDiff, has been presented by Schubert et al. [2005]. However, the graph-theoretical approach is far less competitive in terms of efficiency, thus leading to an unusable runtime already with rather small documents.

An overview of XML differencing approaches for ordered trees has been performed by Peters [2005]. XML differencing algorithms have also been developed for unordered documents. The most prominent solution has been presented by Wang et al. [2003].

3.3.7 Discussion

The resulting deltas for the test case reveal fundamental differences between the four different XML differencing tools. The amount of edit operations ranges from 4 to 18, which is a significant deviation, mostly resulting from the granularity of the change model. Beside the change granularity, the addressing of operations is handled differently. The deltas of Microsoft XMLDiff and faxma basically describe a walk through the source document, creating the target document by in-lining edit operations. This model demands that all edit operations of a delta are subsequently applied. An edit operation cannot be interpreted without taking the rest of the delta into account.

In my opinion, the subtree-oriented representation of changes is far more intuitive for the domain of XML documents, as it does reflect the nature of the changes better that a node-oriented granularity. As the content is stored in the leaves, and non-leaf nodes are used for markup, the subtree-oriented view offers a contiguous representation of the changes relating to an edit operation in terms of the user model.

When taking the readability by humans into account, the absolute addressing scheme by XyDiff and diffxml is far more appropriate than the in-line-representation of Microsoft XML Diff and faxma. Especially the dense representation of XyDiff allows the user for estimating the content of the changes at a glance – which is not possible for the fine-grained diffxml delta format. In my opinion, this is an important aspect concerning the user-acceptance of a differencing tool.

3.4 Conclusions

Finding and representing changes between document versions has been an extensively studied problem since the 1970's. First approaches have only considered the string representation, leading to the definition of the longest common subsequence problem. In this chapter, I have presented the LCS, as it is still widely used. For example, the well-known *diff* tool relies on a common LCS solution. The computation of the LCS is solvable in nearly linear time for similar documents.

Trees have been seen as generalization of strings. The tree-to-tree editing problem maps the LCS problem onto the domain of trees. However, the complexity of the comparison of trees is significantly higher than for strings. Even the best known solutions run in super-quadratic time and space. To deal with this challenge, recent approaches relax the need for optimal solutions, using best-effort approaches to find reasonable small edit scripts, thus yielding a linear complexity in the average case. In this chapter, I have presented different change representation models in the tree editing domain, altogether with prominent solutions for the efficient computation of the edit script for trees.

Differencing XML documents is a kind of natural application of the tree-to-tree editing problem. Several tools have been developed. Some of them implement known tree-correction algorithms. Others take the properties of XML documents into account. In this chapter, I have presented the four best-known algorithms for XML differencing, altogether with their implementation. They all use a different representation of the changes. Several approaches introduce subtree-oriented operations that have not been known in the original formulation of the tree-to-tree editing problem. Using a simple test case, I have shown that the change model and the representation of changes is a major distinguishing feature between XML differencing tools. However, this aspect has mostly been disregarded in the literature up to now. The presented tools will act as competitors in the evaluation of my algorithm in Chapter 10.

4 Document Merging

Parallel editing of documents is an every-day task. Comparing documents versions allows for reconstructing the evolution of a document by extracting the changes. However, the document versions can only be reconstructed linearly. Parallel editing rises the need to merge two document versions that have been edited independently. Merging document versions is a challenging task, as evolving paths and conflicting updates have to be taken into account.

In this chapter, I give an overview of the most common approaches to document merging, including the state-based model on which I base my work. I start by giving a short introduction to parallel editing processes, including the resulting challenges. In the following sections, three major strategies for document merging are presented and discussed: state-based change control, operational transformation, and the annotation of documents. Other approaches are briefly discussed at the end of the chapter.

4.1 Parallel Editing of Documents

Documents are often edited by different persons. In a paper-based office, a physical document is usually tightened to a file. Only one person can perform any action on this document at a given time. Electronic documents, however, may have multiple physical representations (either electronically or printed), and may be distributed arbitrarily. First approaches to support collaborative editing of documents mapped this serial access model on electronic documents, e.g. by Neuwirth et al. [1990]. At this time, parallel editing of a file was already state-of-the-art in version control systems for source code of software [Tichy 1982]. Later, the parallel editing model has been applied to the domain of document editing, too. When editing documents in parallel, merging different document versions including the detection of conflicts becomes the main challenge.

4.1.1 Version Control Systems

Up to now, the source code of software projects is mainly organized in files. These files are usually stored within a repository, which is managed by a version control system [Pilato 2004]. The version control system handles concurrent reading and writing accesses to the repository and ensures that each change is tracked. Any prior version of a file shall be reconstructible.

Bennett and Rajlich [2000] have stressed that software products usually have parallel evolution strings. Even if a new version of the software is available, former versions still receive support, resulting in bug fixes and other changes. Version control systems model this parallel

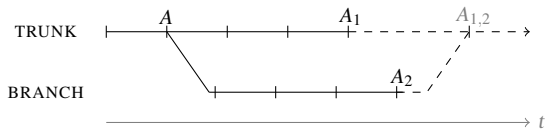


Figure 4.1: In a structured editing process, a document may evolve in parallel in different branches.

evolution by the use of *branches*. One file may have different valid versions in parallel, one in each branch. Figure 4.1 shows a file A, which is branched and independently edited, resulting in the versions A_1 and A_2. In terms of the evolution of the file, A is the *nearest common ancestor* of A_1 and A_2. The main branch is usually referred to as *trunk*. It is generally possible to reunite branches. This requires the *merge* of A_1 and A_2, resulting in $A_{1,2}$. The same holds if a delta should be applied to different branches in parallel, e.g. for bug fixing. I will discuss the issues of merging in Section 4.1.3. For a more detailed description of source code evolution models, please refer to Conradi and Westfechtel [1998].

Branching has to be actively initiated by a user and is used for an intended parallel development. Another important aspect of version control systems is to handle the concurrent (writing) access to the files in the repository. A common approach is to lock a file that is opened for editing, thus hindering other users to perform a write operation onto that file [Borghoff and Schlichter 2000]. Especially for large files, more optimistic collaboration models are used, though. Each user is allowed for changing a local copy of the document. When committing his changes to the repository, his document version is merged with the changes of other users that emerged in the meantime[1]. This optimistic model relies on the assumption that users usually change different parts of the document. Delisle and Schwartz [1986] have enforced this assumption using an empirical analysis.

4.1.2 Collaborative Editing of Documents

Version control systems have a fixed document access model. Each access on a file is logged, changes can be tagged with version numbers. This behavior reflects the precise collaboration models used in the domain of software development, where each minor change may have a great impact. Developers frequently commit their changes to the repository to be able to revert single changes isolated and quickly, which has been shown in an empirical analysis of repository commits by Zimmermann et al. [2006]. Additionally, source code of software is highly structured. Entities like objects, procedures etc. have unique identifiers.

For documents, these restrictions do not hold. Documents have an implicit structure. An explicit structure may be defined, e.g. on the level of sections and subsections [Borghoff and Teege 1993a]. However, tracking the evolution of this structure has to be done explicitly. An

[1] One could consider the local copy to be an implicit branching. Indeed, this terminology has recently become popular in distributed version control systems.

implicit tacking is not possible outside a closed system, as a paragraph does not offer a natural identifier. Additionally, even simple changes on the user level may cause far more changes on the XML representation of a document, as already shown in Section 2.5.

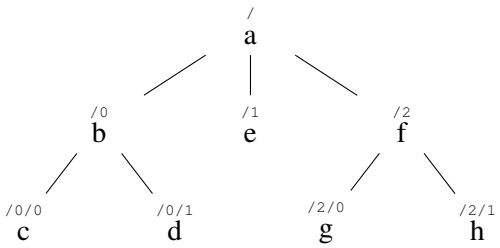
There exist numerous models for the collaborative editing of documents, e.g. by Flower and Hayes [1981]. Koch [1997] has performed a thorough analysis of the different aspects of collaboration and their support by computer-based tools. In my work, I focus on the technical aspects of the merging of document versions. The social aspects of collaborative systems are mostly disregarded. Nevertheless, a social perspective may also help to promote the technical development. Performing an analysis of the social aspects of collaboration, Mitchell et al. [1995] have identified a key requirement for collaborative systems: a user wants to see all changes performed on the document at a glance to be aware of its current state and the prior modification process as well.

4.1.3 Merging

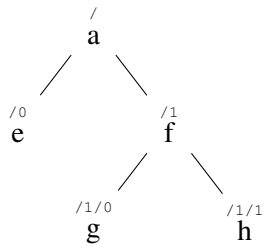
The reconciliation of a new document version out of two (or more) prior versions of the document is called *merge*. Basically, syntactic and semantic merges have to be distinguished [Shen and Sun 2002]. Syntactic merges rely only on the written word, without respect to the semantic meaning of it. A semantic merge takes the semantic meaning of the documents to merge into account. This is a for more complex task than a syntactic merge. For highly structured documents in formal languages like source code, there exist semantic merge approaches. They mostly rely on an analysis of the code tree; a survey of these approaches has been performed by Mens [2002]. For less-structured documents using natural language, no corresponding approach is known to me. Natural language processing is a complex task which is only applicable to a restricted class of problems [Toland 2000], suffering from a low accuracy [Lease 2007]. It appears to be unlikely that usable semantic merge approaches for documents in natural language will be developed within the next decade. Additionally, Shen and Sun [2002] have stressed that a semantically consistent merge can only be achieved by having a proper definition of the authors' roles. According to Koch [1997], a common awareness of the goals of the work is a key prerequisite for meaningful collaborative editing (and merging). As I focus on efficiency and general applicability, I will only consider syntactic merging.

Adams et al. [1986] have presented a first merging application for text files. They distinguish overlapping and non-overlapping changes. Non-overlapping changes do not affect the same lines. These changes can be merged automatically in their approach. Khanna et al. [2007] have shown that even non-overlapping changes may interfere with each other due to identical parts, leading to unwanted merge results. I will discuss that issue in Section 4.2.1.

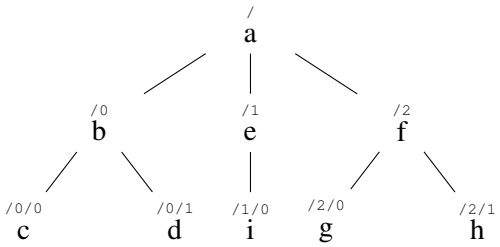
A main challenge in merging documents is the addressing of an operation. In a serial collaboration model, a change addresses the position using a path. A line may be identified by a line number, an XML node by an addressing scheme as presented in Section 2.2.5. Two persons performing actions on a document create two new versions from their nearest common ancestor. In this context, the amount of changes is irrelevant. The changes are addressed basing on the nearest common ancestor. If the second person tries to apply her changes, the



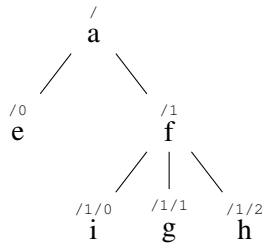
(a) Original version



(b) Version A_1. The subtree rooted at `/0` has been deleted.



(c) Version A_2. Node `/1/0` has been inserted



(d) Version $A_{1,2}$. The node is inserted at the wrong place.

Figure 4.2: Interim changes affect the addressing of other operations to merge. If no measures are taken, operations may be performed at the wrong place of the document.

addresses of the operations might be affected by the prior changes of the first person. As an example, Figure 4.2 shows an incorrectly merged tree. Here, the negative offset resulting from a delete operation leads to the insertion of a node into the wrong subtree. Handling these offsets resulting from prior changes is a major challenge in document merging.

4.1.4 Conflict Handling

Up to now, only non-overlapping changes have been considered. If two users try to change the same part of a document, a *conflict* arises [Lippe and van Oosterom 1992]. In general, conflicts cannot be resolved automatically. The most common approaches are to favor one of the versions [Shen and Sun 2002], to reject a change until it does not conflict any more [MacKenzie et al. 2002], or to prompt the user for a decision [Ignat and Norrie 2006].

The previous definition of a conflict relies on a linear representation of a document. However, most documents are structured. Therefore, Borghoff and Teege [1993b] have extended this view. Every change on a higher level of the document marks the subordinate parts as changed, too. This view is also applicable to the domain of trees. A change of a node may be interpreted as change to all of its descendants. I will discuss this issue in Section 8.4 more thoroughly.

Detecting conflicts is a crucial capability needed for the merging of documents. In this con-

text, conflicts are also defined on a syntactic level. Semantic conflicts arising from a contradictory content of the document are not considered. Approaches exist that require the definition of consistency rules for highly formalized documents, e.g. by Scheffczyk [2004]. Using his approach, a merge could be rejected in case of the violation of a consistency rule. However, formal consistency rules are not applicable on documents in general. Proof-reading by the user will most likely prevail as ultimate instance for conflict-detection and resolution on a semantic level.

4.2 State-Based Change Control

In state-based change control systems, changes between document versions are identified by a differencing tool, as described in the last chapter. Documents can either be merged using a three-way differencing tool, which compares the documents with their nearest common ancestor. Alternatively, deltas can be enriched with context information that allows for identifying edit operations using their syntactic context.

4.2.1 Three-Way Differencing

When documents are updated independently, they usually base on a common ancestor version. A three-way differencing tool compares two document versions with respect to their nearest common ancestor. The basic idea is to be able to determine how two operations in both document versions relate to each other. The first three-way differencing tool for line-based documents, called *diff3* has been developed by Smith [1988].

Khanna et al. [2007] have performed a thorough analysis of diff3. The basic approach of diff3 is shown in Figure 4.3. First, the longest common subsequence of the original version with either of the two documents is computed. The two documents are merged using these two sequences afterwards. The nearest common ancestor allows for determining whether an operation does not match in order of a conflict or a matched node. Apparently, a three-way differencing is more complex than a simple comparison of two documents.

Even the availability of the nearest common ancestor does not ensure an unambiguous decision whether a node conflicts. This issue arises already in the context of line-based differencing, where Khanna et al. [2007] have revealed some unwanted behavior of diff3. Conflicts may arise if an element occurs multiple times within a document. At this point, It should be remembered that in common documents, many nodes are identical (see Section 2.4). However, a sufficient distance between two edited parts of the documents lowers the probability for merge errors significantly.

4.2.2 Context-Aware Patching

Three-way differencing has two major drawbacks. First, the high complexity becomes noticeable, especially for large documents. Second, the nearest common ancestor has to be known

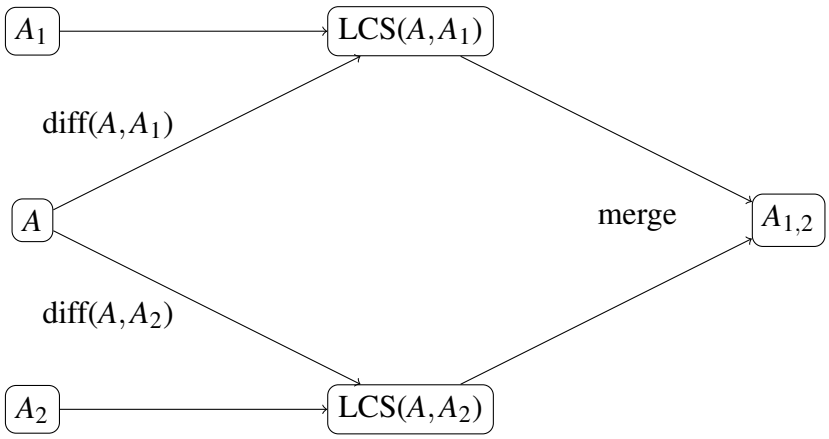


Figure 4.3: Diff3 computes the longest common subsequence between the nearest common ancestor A and the two modified versions first. Afterwards, the merged document is computed.

and to be available. Both may be severe issues, especially in loosely coupled collaboration environments. For example, if documents are exchanged via e-mail, a reliable version-tracking is not available. In distributed version control systems, all versions of a document may not be available locally, too.

To overcome these drawbacks, Davison [1990] has introduced a new delta format for line-based documents, containing context information. Figure 4.4 shows an example. By default, three lines before and after the line affected by an operation are added to the delta, as shown in Figure 4.4(c), where a blank line has been inserted. Afterwards, this delta is applied to a version of the document, where an additional line has been inserted at the beginning (see Figure 4.4(b)). Using the context, the patch tool is able to find the correct insertion point of the blank line, despite the offset of one line caused by the new headline. This context diff is also known as *unified diff format*. Today, a slightly modified version of this format is used in common diff and patch tools [MacKenzie et al. 2002].

4.2.3 XML Support

Merging approaches for XML documents relying on a state-based model are very rare. An XML-aware three-way differencing algorithm has been presented by Lindholm [2004]. The complexity of the merging procedure is reasonable with $O(n\log(n))$. However, the time for comparing the trees twice has to be considered, too, which is super-quadratic (see Section 3.2.3). Additionally, this approach is not able to handle conflicting updates, as the detection of conflicts appears to be far more tricky in the domain of trees [Horwitz et al. 1989].

The definition of syntactic context is straightforward in a line-based document model. However, there is no intuitive mapping onto a tree-based document model. Mouat [2002] has presented an XML delta format, called *Delta Update Language* (DUL) that allows for adding

Carmen 1,11:
Tu ne quaesieris (scire nefas) quem mihi, quem tibi
finem di dederint, Leuconoe, nec Babylonios
temptaris numeros. Ut melius quidquid erit pati!
Seu pluris hiemes seu tribuit Iuppiter ultimam,
quae nunc oppositis debilitat pumicibus mare
Tyrrhenum, sapias, vina liques et spatio brevi
spem longam reseces. Dum loquimur, fugerit invida
aetas: carpe diem, quam minimum credula postero.

(a) Original

Quintus Horatius Flaccus
Carmen 1,11:
Tu ne quaesieris (scire nefas) quem mihi, quem tibi
finem di dederint, Leuconoe, nec Babylonios
temptaris numeros. Ut melius quidquid erit pati!
Seu pluris hiemes seu tribuit Iuppiter ultimam,
quae nunc oppositis debilitat pumicibus mare
Tyrrhenum, sapias, vina liques et spatio brevi
spem longam reseces. Dum loquimur, fugerit invida
aetas: carpe diem, quam minimum credula postero.

(b) Updated

```
-- orig.txt 2010-03-14 21:14:14
+++ upd.txt 2010-03-15 11:10:02
@@ -3,6 +3,7 @@
```
finem di dederint, Leuconoe, nec Babylonios
temptaris numeros. Ut melius quidquid erit pati!
Seu pluris hiemes seu tribuit Iuppiter ultimam,
+
quae nunc oppositis debilitat pumicibus mare
Tyrrhenum, sapias, vina liques et spatio brevi
spem longam reseces. Dum loquimur, fugerit invida

(c) Context delta with meta-data

Quintus Horatius Flaccus
Carmen 1,11:
Tu ne quaesieris (scire nefas) quem mihi, quem tibi
finem di dederint, Leuconoe, nec Babylonios
temptaris numeros. Ut melius quidquid erit pati!
Seu pluris hiemes seu tribuit Iuppiter ultimam,

quae nunc oppositis debilitat pumicibus mare
Tyrrhenum, sapias, vina liques et spatio brevi
spem longam reseces. Dum loquimur, fugerit invida
aetas: carpe diem, quam minimum credula postero.

(d) Merged

Figure 4.4: Adding syntactic context to a delta allows for reliably finding the correct address of an edit operation. In this example, the new headline in the merged document version causes an offset of one line.

syntactic context to edit operations. In DUL, an edit operation may contain the siblings of the node addressed by the edit operation, as well as the parent and children nodes. However, the DUL specification has some weak points, which I will discuss in Section 6.5.4. Additionally, no corresponding patch application is known to me.

In general, most of the work on state-based XML versioning focuses on the aspect of efficient change detection. The subsequent merge of documents has mostly been disregarded in literature. None of the XML differencing tools presented in Section 3.3 is able to merge XML documents.

4.2.4 Discussion

Three-way differencing and context-aware patching are commonly used for merging line-based documents. Established version control systems like CVS or Subversion rely on these state-based approaches [Pilato 2004]. However, they cannot ensure a correct merge result.

Major research efforts have been made to develop merge approaches that try to offer a bet-

ter reliability or accuracy [Mens 2002]. Nevertheless, none of these approaches achieved to supersede the conventional approach. From my personal perspective, the reason for the prevailing success of the classic diff and patch tools is their general applicability, in combination with their ease-of-use. The deltas are human-readable, and the merge of document versions is nearly self-explaining. Additionally, these tools do not bond the user to a special application or edit model. They have been adopted in a large variety of tools on a higher level.

Up to now, no diff and patch toolkit offers this broad applicability and merge-support for XML documents. First approaches exist that apply the well-proven diff3 and context-aware patch tools to the domain of XML. However, these approaches are still in their infancy. A general-purpose XML diff, patch, and merge toolkit would close that gap.

4.3 Operational Transformation

Operational Transformation (OT) has been introduced by Ellis and Gibbs [1989] in the context of a real-time group editor. Each participant in the collaboration environment has its own working context with a local copy of the document, called *site*. Each operation performed on one site is transferred to the other sites, where the operation is applied within the corresponding context.

4.3.1 Basic Idea

In the original definition, a character-based granularity has been defined. Any character change (i.e. a keystroke) results in an insert or delete operation. The basic idea is to transform each operation in a way matching its current context. This transformation mainly targets the adaption of the addressing of a change to handle interim changes caused by the local user or other sites. The transformation step is the real heart of any OT algorithm.

A major goal of OT approaches is to ensure that all copies of the document are in the same state after a certain timespan for transmission and reconciliation. This is hard to achieve, especially in high-latency connections. Due to network constraints, operations may be received not in the order they have been performed, which requires sophisticated transformers [Nichols et al. 1995].

The reconciliation of former versions of a document (i.e. an “undo” operation) is often impossible in OT approaches. Oster et al. [2006a] have introduced an annotation method that marks deleted items as removed. This way, changes can be reverted which is a major improvement. As a drawback, the document size increases linearly with the amount of changes, which can become space-consuming.

4.3.2 XML Support

The operational transformation approach has been mapped from the domain of line-based texts to trees by Davis et al. [2002]. Molli et al. [2002] have presented an XML-aware group editor,

called SAMS. In SAMS, the character-centric granularity of the text-based domain is mapped onto a node-based granularity. Attribute changes are considered as independent operations. A similar approach has been presented by Ignat and Norrie [2003].

Recently, Google has presented a new collaboration environment, called Wave[2]. Google Wave relies on an operational transformation approach to XML documents, described by Wang and Mah [2009], which has been highly influenced by the work of Nichols et al. [1995]. Instead of using a tree-based edit model, Google Wave relies on an XML representation by sequences, similar to the XAS serialization format by Kangasharju and Lindholm [2005], which is used in the faxma XML differencing algorithm (see Section 3.3.5). Node labels are considered to be atomic, whereas text nodes are treated on a character-granularity level.

4.3.3 Merging

The operational transformation approach is basically intended to allow users for the parallel editing of a document. In this context, conflict handling is an important yet difficult issue. In general, only delete and update operations can cause conflicts, as the same item is addressed by two or more operations [Shen and Sun 2002]. Insert operations do not lead to conflicts, as they do not affect items addressed by other nodes. However, insert operations in the same syntactic context may lead to meaningless results, e.g. if a sentence is edited by two persons independently. Here, an awareness for the editing operations of other persons may be appropriate [Schlichter et al. 1998]. Additionally, a paragraph edited by a user can be locked to prevent changes by other users [Borghoff and Teege 1993b].

Basically, the reconciliation of document versions is a transparent process that is not designed for user interaction. Therefore, Shen and Sun [2002] have decided to generally favor the local version in case of conflicts. Sun and Sosič [1999] have proposed a fine-grained locking scheme for real-time editors to prevent the rise of conflicts. Another approach has been followed by Ignat and Norrie [2006]. They display conflicting changes in a separate window, requesting the user to vote for one of the possible versions.

Parallel editing by different users raises the question of scheduling the different operations, especially in low-bandwidth and high-latency environments, or with many parallel users. This way, a semantically correct merge of changes shall be ensured. Preguiça et al. [2003] have presented an approach that respects integrity constraints. However, finding an optimal solution appears to be NP-hard. Therefore, only a heuristic best-effort approach is used.

4.3.4 Complexity

The transformation of the operations may be an complex task. Ignat and Norrie [2008] have shown that the transformation runs in quadratic time $O(n^2)$, where n denotes the number

[2]Due to low success, Google has already discontinued to provide this service. Nevertheless, Google Wave proved the maturity of OT in the XML domain even in large environments. The code was released as Open Source and is actively developed by a vibrant community.

of operations to perform. This is reasonably fast for collaborative editors where changes are exchanged contiguously in short time intervals. However, operational transformation can be used for asynchronous collaboration as well. There, the number of operations to apply increases significantly, leading to a noticeable increase of the runtime.

To achieve a faster reconciliation in asynchronous collaboration system, Ignat and Norrie [2008] have introduced a distributed operation history instead of a centralized operation list. The distribution follows the tree structure of the XML document. By this, the different operation lists can be worked off in their local context, which is more efficient than processing the whole operation list at one time. A second benefit of distributed operation histories has been described by Papadopoulou et al. [2006]. Due to the locally available operation list, a common awareness about the changes can be established efficiently.

In the classic approach, all sites of a collaborative editor are equal and exchange their editing information on a peer-to-peer basis. This approach scales bad in environments with many sites [Oster et al. 2006b]. Many redundant information is exchanged and the number of conflicts may increase dramatically. To overcome this issue, a common server can be defined which hosts the master document. Especially in case of conflicts, this master document represents the valid state of the whole system. The server also integrates different changes into one, which reduces the overhead during the transmission. As an example, Google Wave uses this server-centric operational transformation approach [Wang and Mah 2009].

4.3.5 Discussion

Implementing the operational transformation approach in the domain of group editors has some benefits. Usually, the user input is directly caught and mapped onto the corresponding operations for reconciliation. This way, there is no need to perform complex diff runs, as would be needed in a state-based approach. Even if the overhead due to the transformation of the operations and the high amount of network traffic are significant, editors using operational transformation are quite fast. This results from the fact that the operations are exchanged in short time intervals which distributes the complexity over time to an acceptable degree. However, this only holds for synchronous editors. In asynchronous editors, the drawbacks of the overhead become severe.

Beside the network availability constraints, there are other reasons for favoring an asynchronous approach. Privacy and security become a more and more important aspect of collaborative work. Although group awareness is an important benefit for supporting collaboration and co-operation, private workspaces are essential [Olson et al. 1990]. Especially extensive changes (like the re-organization of chapters) are usually tested in a private workspace before presenting them to other authors. To meet these concerns, Ignat et al. [2008] have introduced so-called *ghost operations* that are not displayed to the other users, but applied to the document. The user acceptance of ghost operations has not been investigated, however. Additionally, this is a kind of internal branching which is hard to reconcile in case of a later merge.

4.4 Annotation of Documents

In a document-centric view of collaboration, all editing information should be part of the document itself. Different approaches have been presented that use annotations of the XML document to represent version information. Most approaches are highly inspired by the domain of temporal queries in databases [Wang et al. 2006]. The basic idea is to represent different states of the document within the document itself.

4.4.1 Basic Idea

In general, single nodes are annotated, thus applying the tree edit model by Tai [1979]. Usually, attributes are used to annotate the node. As a simple example, the node `comment` could be annotated this way: `comment created="March, 2, 2010"`. The annotation contains the operation applied and the date of that operation. This scheme is straightforward for insert and delete operations, yet difficult to apply for update operations. To be able to revert an update operation, the former value of a node, can be stored as an attribute. For the example above, an update could be represented as `newcomment update="March, 3, 2010", ov="comment"`. For subsequent updates on the same node, this model has to be extended. Chawathe et al. [1999] have presented a method that allows for multiple updates of a single node, where all former values are preserved in attribute values. Respecting attribute changes requires an even more complicated representation of the changes [Wang and Zaniolo 2008].

The example above uses a chronological date to order the changes. In this time-based annotation model, each annotation contains temporal information concerning the validity of the performed operation. The granularity of the timestamps should conform the edit pattern used. For a frequently updated document, a more fine-grained resolution than dates may be appropriate. A time-based annotation model for XML documents has been presented, e.g., by Amagasa et al. [2000].

Another approach is to label each operation of a new version with a unique identifier. This way, several edit operations can be part of a new version. The version identifier is usually a number representing the evolution step. This version-based approach to annotation requires the user (or the application) to actively define a new version. For example, this approach is used in the domain of normative documents [Grandi et al. 2005].

An extension of the annotation model is to provide not only the initial date/version of a change, but to define a validity period for a modified node, too. This validity period is defined by the start and the end. For a still valid node, a virtual date (e.g. “December, 31, 9999”) or a code word (e.g. “now”) can be used [Amagasa et al. 2000].

4.4.2 Querying States

The annotations should not be visible to the end-user. In general, the user is given the state of the document at a given time or version. This is usually performed by a middle-ware [Grandi et al. 2005]. Two approaches exist to extract a given state of the document. The first is to

use a specific query language with temporal operators, which is inspired by temporal database queries. A solution has been presented by Nørvåg [2002]. In the second approach, the XML document is transformed into the requested version using a transformation language like XSLT [Grandi and Mandreoli 2000].

4.4.3 Change Tracking in Documents

One of the most prominent applications of the annotation approach is the change tracking feature in office applications. During the editing of a document, all changes are recorded and stored within the document.

Office applications may operate on an internal document model, which is not necessarily equal to the document model given by the XML document format used for serializing the internal representation. The change tracking is performed on this internal representation, which does not have a tree structure like XML. This has some implications. First, the operation types may not conform to the tree structure. For example, ODF supports following operation types: insert, delete and format-change [Brauer et al. 2007]. The last one is not a tree-related operation. Second, changes are tracked with the granularity of characters. Because office document formats usually store a whole paragraph within one text node, measures have to be taken to map this on the XML domain which is less fine-grained. Third, edits are allowed to span over parts of adjacent subtrees, without breaking the original structure, which does not conform to the tree structure of XML, where each node has at most one parent node.

Any tracked change is registered in a change list at the beginning of the document. This list records the change type, the time when it was performed, and by whom. Furthermore, a reference to the change itself is included via ID attribute. In case of a delete operation, the deleted part is stored within the change list, too. The tracked change itself is delimited by two nodes, called `change-start` and `change-end`, identified by their unique ID attribute. By introducing these leaf nodes, changes spanning over adjacent subtrees can be covered. The part in between these delimiting nodes has been inserted or formatted differently, respectively. As the deleted part is stored in the change list, a delete operation is only delimited by one empty node which would act as insert position in case of an inversion of the delete. The reason for storing deletions in the change list is straightforward. This way, a removal of prior versions and the corresponding metadata can be performed easily without parsing the whole document.

As update operations are not supported, each character update results in an insert and a delete operation. This reflects the presentation within the office application, where deleted parts are striked out und inserted parts are underlined. However, this presentation makes it difficult to capture the changes by humans [Neuwirth et al. 1992]. Additionally, the high granularity on the character level complicates a later merging of changes, as a single character only has a low information value.

Basically, the tracked changes should be invertible to recover former versions. However, only inserts and deletes are invertible. In case of format changes, the former format is not stored within the change list, hence the change can not be reverted. There are two reasons for

A: "Hello, world!"

```
document-content
├ styles
└ body
  └ text
    └ p style="standard"
      └ Hello, world!
```

A_1: "*Hello, World!*"

```
document-content
├ styles
│ └ style name="P1"
│   └ font-weight="italic"
└ body
  └ text
    └ p style="P1"
      └ Hello, World!
```

```
document-content
├ styles
│ └ style name="P1"
│   └ font-weight="italic"
└ body
  ├ tracked-changes
  │ ├ changed-region id="ct162024864"
  │ │ └ format-change
  │ │   └ change-info
  │ │     ├ creator
  │ │     │ └ Sebastian Roennau
  │ │     └ date
  │ │       └ 2010-05-19T20:53:00
  │ ├ changed-region id="ct156662296"
  │ │ └ deletion
  │ │   ├ change-info
  │ │   │ ├ creator
  │ │   │ │ └ Sebastian Roennau
  │ │   │ └ date
  │ │   │   └ 2010-05-19T20:53:00
  │ │   └ p style="standard"
  │ │     └ w
  │ ├ changed-region id="ct156989080"
  │ │ └ insertion
  │ │   └ change-info
  │ │     ├ creator
  │ │     │ └ Sebastian Roennau
  │ │     └ date
  │ │       └ 2010-05-19T20:53:00
  │ └ changed-region id="ct161192008"
  │   └ format-change
  │     └ change-info
  │       ├ creator
  │       │ └ Sebastian Roennau
  │       └ date
  │         └ 2010-05-19T20:53:00
  └ text
    └ p style="P1"
      ├ change-start id="ct162024864"
      ├ Hello, world!
      ├ change-end id="ct162024864"
      ├ change id="ct156662296"
      ├ change-start id="ct156989080"
      ├ W
      ├ change-end id="ct156989080"
      ├ change-start id="ct161192008"
      ├ orld!
      └ change-end id="ct161192008"
```

Figure 4.5: Change tracking (right) has a significant overhead compared to single files (left).

this behavior. First, format descriptions are stored in a separate style list at the beginning of the document. Changes on this style list are not recorded by the "track changes" feature. Second, a format change may be applied at a higher hierarchy level, e.g., paragraph level. Still, only the text parts affected by this change would be marked by the format-change delimiters.

Figure 4.5 shows an example. A document *A* has been changed from "Hello, world!" to "*Hello, World!*", which includes a content change by capitalizing one letter and a markup change due to an italic fontface. The format change has been applied using the `style-name` attribute of the corresponding paragraph node `p`. The tracked changes do not capture this, they only mark the parts that have been affected by this change, resulting in two change-regions before and after the character update.

4.4.4 Merging Annotated Documents

Including all editing information within the document itself is basically contradictory to merging. If the document is referred to as an object, it cannot be at two places (i.e. editing processes) at a time. Nevertheless, merging is an important task for annotated documents, too.

Rosado et al. [2007] have proposed to include the graph of the whole document evolution within the document, including branches. This allows for representing parallel editing processes within an annotated document. However, no solution for merging of these parallel versions is presented. To solve the merging issue, Fontaine [2002] has combined the annotation approach with an XML differencing engine. This way, a reliable merge can be performed. On the other hand, this approach is not able to handle parallel branches. A similar approach is performed by office applications. A code review of OpenOffice shows that the content (i.e., the text) is merged on the application level by differencing the two documents to merge.

4.4.5 Discussion

The annotation of documents solves a problem common to all XML differencing approaches. As the changes are in-lined within the document, the addressing does not matter. Changes in other regions of the document do not affect the correctness of the actual change, which is a major benefit compared to other approaches relying on path descriptions for addressing.

On the other hand, all editing information is tightened to the document. This has two significant drawbacks. First, the space overhead might become considerable, especially for often-changed documents. One might argue that other approaches have to store all this information, too. But for annotated documents, all the changes have to be transferred every time when the document is exchanged. A separation of the current version and the history of a document is not possible. Second, the annotation might cause severe security and privacy issues. Especially tracked changes in office applications collect metadata about the author and the time the change was performed, The unwanted side-effects of undisclosured internal workflows may be unpleasant.

Except for office documents, the annotation is not part of the document grammar. An annotated document will most likely not conform to that grammar, as the annotations are not part

of it. To verify the validity of a document, a single version has to be extracted. Otherwise, the corresponding grammar must be adapted to the annotation performed, as proposed by Bertino et al. [2002] However, this is time-consuming and error-prone, and therefore not a viable way in my opinion.

An interesting question arises when inspecting how changes are discovered, before the document can be annotated. Annotation approaches rely either on a state-based diff run [Fontaine 2002] or an operation-based approach, e.g. with the change tracking of office documents. Especially the first one requires additional tools to track the changes and map them onto the annotation model. In conclusion, the annotation approach is not very suitable for document merging. All approaches offering merge support rely on some kind of differencing run to estimate the operations to merge.

4.5 Other Approaches

Research on the structured evolution of documents has been mainly driven by the needs of software engineering. Version control systems like RCS, CVS, or Subversion have been mainly designed to support the software development process [Pilato 2004]. In the evolution of software, branching and merging are every-day tasks, with some special characteristics. For example, if the name of a procedure is changed, all invocations of that procedure have to be updated. Most of the research on document merging focuses on the needs of source code, which is definitely out of scope of my thesis. Mens [2002] has presented a concise overview of the research on software merging, which is a good starting point for further reading.

Other research on document merging is highly inspired by the domain of relational databases. Gropengießer et al. [2009] have presented a novel transaction scheme that applies the transaction model of Gray [1978] to XML. This data-centric approach is applicable to ordered trees, but relies on an unordered XML model by design. Therefore, this approach is not well suitable for the domain of XML documents.

4.6 Conclusions

Merging documents is a complex yet important task. The main challenges in merging are the addressing of changes and the handling of conflicts. Any change within a document affects the paths of all subsequent parts of the document. Other edit operations that address these parts become invalid due to the resulting offset. Conflicts can arise on a syntactic and semantic level. Only syntactic conflicts can be handled by common software tools. The automatic resolution of conflicts is error-prone; usually, the user is prompted for a decision on ambiguous merges. Semantic conflicts have to be resolved manually by proof-reading the document.

Several tools provide a merge capability for documents. Most of them try to merge using a state-based approach, operational transformation, or by annotating the document with version information. In this chapter, I have presented these approaches, including their assets and

drawbacks.

Operational transformation is mostly suitable for on-line collaborative editors. It is designed to frequently merge only few changes. To avoid a complexity trap, a server-side transformation is appropriate for many concurrent users. Operational transformation is also able to support disconnected, i.e. asynchronous operations on the document. However, complexity becomes an issue in this case. Additionally, operational transformation bonds the user to a specific application, which does not match the intended-use case. Koch and Koch [2000] have stressed the importance of a flexible composition of different tools as major prerequisite for the user acceptance of collaborative systems.

In the annotation approach, all changes are in-lined into the document. This is a major benefit, as the problem of addressing the changes is avoided. A conventional differencing tool is used to merge annotated documents. The annotation approach has some drawbacks, as the whole version history is carried with each transmission of the document. First, this is not very efficient in terms of space, especially for frequently updated documents. Secondly, privacy and security questions arise from it. The annotation is performed using an on-line change tracking similar to an collaborative editor or by comparing document states using a diff tool.

Despite several sophisticated merge approaches (especially for software source code), the state-based syntactic merge of documents is still the most common method for merging. In a state-based approach, two document versions are compared with respect to their nearest common ancestor. Another approach is to enrich the edit operations of a delta with the syntactic context of each operation. Both approaches allow for a simple yet reliable merge of document versions. However, XML-aware applications using this model are not available yet.

The presented approaches are not mutually exclusive. Rönnau and Borghoff [2009] have shown how to map the annotations of tracked changes onto a state-based versioning model. Fraser [2009] has developed a collaborative editor using state-based differencing and merging tools. All approaches have their benefits. But in terms of a wide applicability, the state-based approach has a definitive advantage in my opinion. Therefore, I base my work on this model.

Part II

A Context-Sensitive Approach to XML Change Control

5 The XCC Framework

Change control of documents comprises the aspects of document differencing, patching, and merging. The last chapters have shown that the change control support for XML documents is not mature yet. Existing approaches either concentrate only on specific aspects of change control (e.g. differencing) or do not offer the efficiency and reliability that is required for every-day use.

In this thesis, I present a first comprehensive approach to state-based change control of XML documents. In this chapter, I present a framework that enables users and applications to control the evolution of XML documents using a context-oriented approach. I start by describing the basic idea of the context-orientation. Afterwards, I sketch the architecture of the whole framework. Finally, I describe some aspects of the implementation of the framework. The different components of the framework are described in the following chapters.

5.1 Basic Idea

The design of a change control architecture depends on the intended use-case. Here, I describe the collaboration model which I have in mind during the design of my framework. Deriving from this model, I decide to use a context-oriented merge approach, which I reason for afterwards. Finally, I briefly describe the representation of the syntactic context in my approach.

5.1.1 Intended Collaboration Model

Collaboration across organizational borders usually implies the absence of a common collaborative system. Each participant may have his or her own collaborative system, with the possibility to grant the other participants several access rights. However, an overarching collaborative system with a corresponding common concept for roles and privileges will most likely not exist. If the system of one participant is opened towards the user users, all other participants would have to abandon their own systems. Additionally, security issues and inner-organizational policies may prohibit granting foreign participants comprehensive access rights. These issues have lead to the accepted practice of exchanging documents via e-mail or USB pen-drive. Apparently, this is a state-based collaboration model.

The exchange of document versions via e-mail is easy, but the version tracking is cumbersome. In contrast to closed collaborative systems, where the server knows about each access to a document, the highly distributed collaboration via e-mail cannot be tracked automatically. Therefore, organizational solutions are established by the participants. For example, the

changes have to be marked, or written separately as change-wish-list. The user who merges the document versions has to perform the merge manually, which is both time-consuming and error-prone.

5.1.2 Scope

My goal is to support this ad-hoc collaboration by a framework for merging document versions. Additionally, the framework shall be applicable to closed collaboration systems as well. Among all merging approaches presented in Chapter 4, state-based merging and the annotation of document versions are the only ones working outside a closed environment.

The annotation approach needs all changes to be tightened to the document, which is inefficient in the context of the collaboration model as described before. Therefore, this approach is excluded as solution. Among the state-based approaches, three-way differencing is a reliable solution for merging. However, it requires the nearest common ancestor of two document versions to be available. This cannot be ensured in the described scenario. First, it might be physically unavailable. Second, it might be unknown, which version actually is the nearest common ancestor. Among the possible solutions, conventional differencing remains. Two-way differencing and patching have not been applicable in merge scenarios in their initial design. The merge-capability has been achieved by storing the syntactic context of an edit operation, introduced by Davison [1990].

The freedom of the collaboration model is one of the reasons for the prevailing success of the diffutils[1]. The other one is the simplicity of its usage, altogether with the reliability of the merge result. The scope of my thesis is to develop a framework for XML differencing, patching, and merging that provides an ease-of-use and reliability as the diffutils do.

5.1.3 Mapping the Line-Based World onto XML

The basic idea of my approach is to map the context-orientation used in the line-based diffutils onto XML documents. In this model by Davison [1990], for each each operation, the surrounding lines are stored within the delta. This allows the subsequent patch tool for performing a reliable merge taking this syntactic context into account (see Section 4.2.2). The main difficulty of representing context in XML is the two-dimensional structure of the tree. Where the line-based domain offers a "natural" representation of the context, a straightforward solution for XML does not exist. A first approach to mapping the syntactic context onto the domain of XML has been performed by Mouat [2002]. However, this model is partially inconsistent, which I will discuss in Section 6.5.4.

In my approach, the syntactic context of an edit operation is represented using a so-called *context fingerprint*. This fingerprint contains the hash values of the adjacent nodes in document

[1] The GNU implementation of the line-based diff and patch tool is called diffutils, too. I use this term to refer to the framework of differencing and patch tools, including its programming interface. If not indicated otherwise, this implies no restriction to the GNU implementation.

order. Due to the hashing, the fingerprints remain quite small. The document order is used to cover a wide range of XML document types. This is especially motivated by the document analysis which I have performed in Chapter 2. I will discuss this issue in more detail in Section 6.5.4, too.

During a subsequent merge, the fingerprint is used to find the correct path of an edit operation. The merge is performed by the patch tool and can be controlled by several parameters. For example, sub-optimal results may be accepted in case that parts of the context do not match. Additionally, a conflict detection prevents the deletion or update of previously edited parts of the document.

Syntactic context is a commonly used for merging in different domains. As an example, it is a common approach in software source control [Mens 2002] or model merging [Lai 2009].

5.2 Architecture

The architecture of the framework is closely coupled to the provided capabilities and aims to provide a simple yet powerful access to the components of the framework. The framework itself is called XCC framework. It consists of a delta model, a differencing algorithm, and a patch algorithm with merge capabilities. These components are explained in the following pages. Afterwards, I will describe some implementation details, and briefly mention the tools provided by the framework. Finally, I will describe the features of the application interface.

5.2.1 Components

XCC Delta is the delta model that builds the heart of the XCC framework. In a two-way approach, no additional information apart from the delta can be used to decide upon an edit operation to merge. Therefore, a delta must contain all information concerning the changes performed. The expressiveness of the delta model directly relates to the (merge) capabilities of the patch algorithm. Generally speaking, the delta model is the common language of a change control framework. The differencing algorithm (XCC Diff) and the merge-capable patch algorithm (XCC Patch) are built around the delta model, providing the services of the framework.

XCC Diff compares two document versions and represents the changes in a delta. The design of the differencing algorithm is deduced from the document properties and the expected modification patterns that have been elaborated in Section 2.4 and Section 2.5, respectively. The main focus of XCC Diff is to compute the changes efficiently in terms of time and space. Additionally, the computed deltas shall be comparably small and human-understandable. The latter requirement is difficult to evaluate. On the other hand, the comparison of the state-of-the-art in XML document differencing has shown that the amount of edit operations may differ significantly (see Section 3.3). In my approach, I favor a smaller quantity of more comprehensive edit operations.

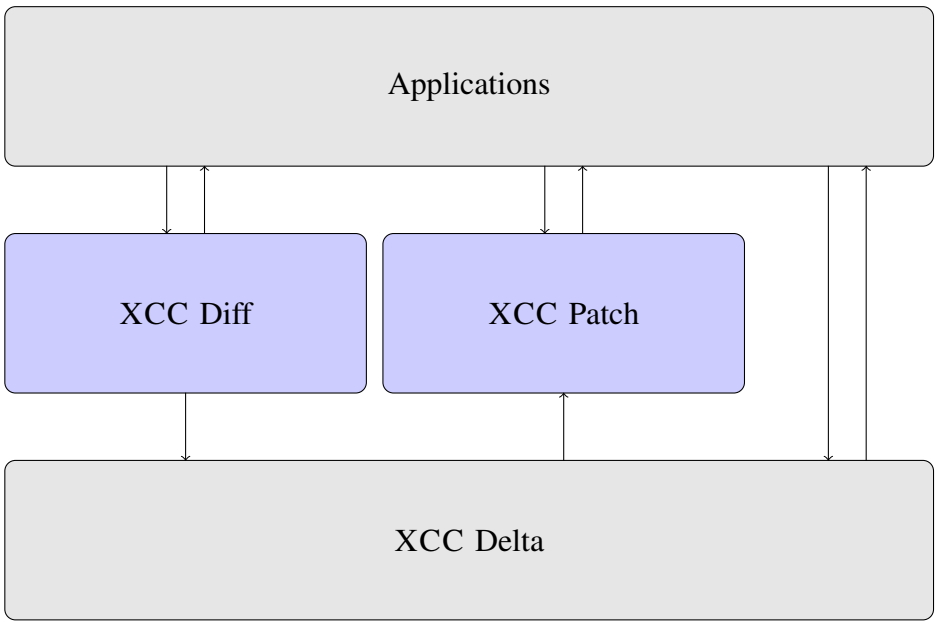


Figure 5.1: The delta model builds the heart of the XCC framework. The diff and the patch algorithm provide the corresponding functionality. Application may also directly access the delta layer.

XCC Patch applies the edit operations contained in a delta to an existing document. In a linear evolution model, the delta is only applied to the file the delta has been computed for, thus creating the expected document version. In a merge scenario, however, the delta is applied to another version of the document. Here, the edit operations may not be applicable. The behavior of XCC Patch can be controlled via different parameters. They control the degree of the acceptable merge quality, as well as the behavior in case of conflicts.

Figure 5.1 shows the architecture of the XCC framework at a glance. The diff and the patch tool exchange information using the delta access layer. In this context, the diff tool creates deltas, which can be processed by the patch tool. Applications can interact with both tools as well, or with the delta access layer itself. The latter one enables the possibility for applications to create conforming XCC deltas themselves, or to merge an existing delta in their respective application context.

5.2.2 Implementation

The XCC framework has been implemented in an evolutionary process by different programmers. The first approach has been performed by Pauli [2008]. Here, only the patching aspects have been implemented using Java and the XOM object model[2]. This implementation has laid the basis of the evaluation of the merge quality [Rönnau et al. 2008]. Philipp [2008] has

[2] `http://www.xom.nu`

performed a re-engineering of the existing implementation to achieve a better runtime. Using a new implementation architecture based on Java and DOM, he was able to achieve an increase of efficiency by an order of magnitude. Due to its advantages, I will briefly describe this implementation in the following.

Nodes are compared using their node value, which is the hash value of the normalized representation of the node (see Section 2.2.2). Nodes may be accessed several times during differencing and patching. To avoid a re-computation of the hash value at each access, the implementation uses a wrapper class around the DOM. The hash value is computed during the first access to the node. Afterwards, it remains in main memory, attached to the node. Therefore, a later access to the node does not increase the runtime. In case that the node is changed, the hash value is re-computed accordingly. One might argue why the nodes are not hashed during parsing while starting up. Two reasons enforce the presented approach. First, the differencing and patching algorithms may be initialized either by giving a path to a file, or handing over a DOM tree. In case of the latter one, the document has already been parsed. A second parse run would need additional time, which directly leads to the second reason. During patching, only the parts of the document that are targeted by an edit operation need to be hashed. A comprehensive hashing does not need to be performed, and would in turn lead to a higher runtime.

The implementation of the differencing algorithm has been performed by Philipp [2009]. The architecture of the implementation is closely related to the architecture of the framework itself and contains the differencing and the patching engine, as well as an XML access layer. The XML access layer relates to the XCC Delta layer in the framework architecture. It provides all features of node addressing, hashing, and fingerprint generation. The whole XCC framework is packed within one JAR archive. This way, an installation of different libraries is not necessary. As the basic libraries for XML handling and the DOM representation are part of the standard installation of the Java Runtime Environment, the XCC JAR is self-containing and works out of the box.

5.2.3 Tools

The XCC framework provides a command-line tool, as well as a graphical user interface for an easy drag-and-drop access to the differencing and patching algorithms.

The command-line tool runs in two modes, which are selected via shell parameter. It can act either as diff or as patch tool. In both modes, the tool offers a variety of different options. I will not describe these options in detail at this point. They are described within the corresponding Chapter later on. The command-line tool provides a user interface familiar to users of the conventional diffutils.

Outside the domain of software development, users are not used to the command-line shell. Therefore, the XCC framework additionally provides a simple graphical interface that allows for differencing and patching documents using a drag-and-drop user interface. This interface offers the same options as the command-line tool. The interface for differencing and patching is the same. It provides three spots – two for XML documents and one for a delta. A user may

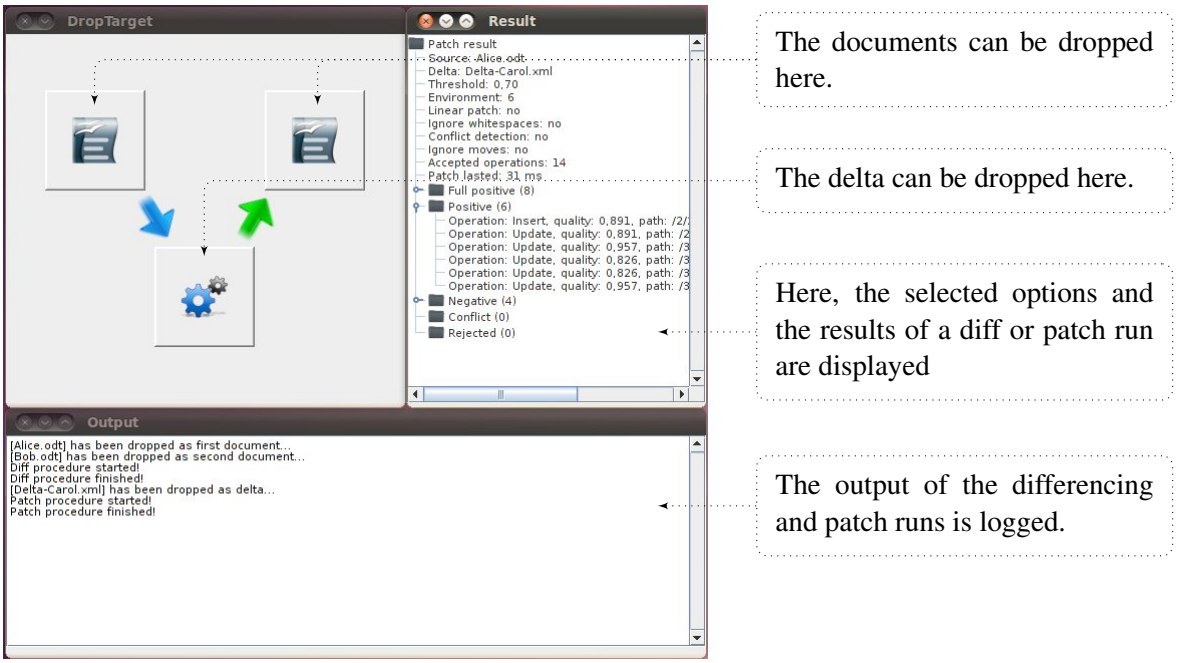


Figure 5.2: XCC provides a graphical user interface that allows for differencing and merging documents using drag-and-drop.

drop files on two of them. If two documents are given, the corresponding delta is computed. If in turn a delta and a document are given, a patch is performed. Figure 5.2 shows a screen-shot of the GUI after a patch run. Here, a merge has been performed. Not all edit operations of the delta have been applied. I will discuss the meaning of such a merge result in Section 8.5 and Section 9.4.

5.2.4 Application Interface

Change control of single documents on a small scale can be ensured by the proposed tools. On a large scale, however, more sophisticated applications are needed to ensure an efficient and user-friendly change control. This may incorporate version control systems, auto-versioning repositories, and many more.

To allow those applications for using the XCC framework, an application programming interface (API) is provided. The framework provides following functionality (in alphabetical order):

- Addressing of nodes using the simplified addressing scheme defined in Section 2.2.5
- Computing of context fingerprints for single nodes and operations
- Differencing document versions

- Hashing of single nodes and complete subtrees
- Merging of documents with the possibility to treat each edit operation individually.
- Patching of documents

The applications may pass an XML document to the framework either as path to a file or as already parsed object model. The latter possibility is important to directly modify a document within the application context, without being forced to serialize the document to its file representation. Different applications already use the XCC framework. I will present them in Section 11.3.

6 A Context-Oriented Delta Model

A key prerequisite to a delta-based change management of documents is a consistent delta model. The most important aspect of a delta model is the definition of the change representation between document versions to be able to reconstruct subsequent document versions. Prior work on XML versioning mainly focused on the question of computing the delta. The question of the delta model itself, mostly in terms of merge-ability and reliability, has mostly been disregarded yet.

In this chapter, I present a context-oriented delta model for XML documents. It is based on the findings gathered from the analysis of XML documents performed in Section 2.4. The delta model shall cover a wide range of applications, not being bound to a specific XML dialect. The delta is defined to ensure enough information allowing for reliably merging document versions, without being gossipy. Human-readability of the output is a second design principle. The user should be able to understand the meaning of an edit operation without tool support. This is mostly important in loosely coupled collaborative environments where the availability of tools cannot be guaranteed at all times.

The remainder of this chapter is structured as follows: First, I give a formal definition of diff, patch, and deltas. Afterwards, I describe the edit operations used in my delta model and motivate them. I introduce different dependencies that can arise between edit operations, followed by a discussion of the implications deriving from my definition of the delta. After that, I introduce my notion of syntactic context. Finally, the aggregation and inversion of deltas are shown.

6.1 Definitions

Up to now, I did use a non-formal description of differencing, merging, and deltas. At this point, I introduce a more rigorous formal model. From this model, dimensions for the correctness of a differencing and merging tools arise.

6.1.1 Equality of Documents

First, the equality of document versions has to be defined. Loosely speaking, two documents are equal if their structure is the same and all corresponding nodes have the same node value.

$$A = A_1, \text{ if and only if } (\forall n \in A : (\exists n_1 \in A_1 : n = n_1 \wedge \mathit{path}(n) = \mathit{path}(n_1))) \wedge (\forall n \in A_1 : (\exists n_1 \in A : n = n_1 \wedge \mathit{path}(n) = \mathit{path}(n_1))) \quad (6.1)$$

As this definition relies on the definition of equality for node values (see Section 2.2.2), the equality of documents is estimated upon the normalized representation of the nodes. This ensures robustness against different encoding formats, name-space representations, and attribute orderings.

6.1.2 Diff and Patch

Two versions of a document are compared by a *diff*-function, which computes the differences and creates a delta δ containing the edit operations to construct one document version from the other one. The basic set of all edit operations is denoted as Δ, with $\delta \subset \Delta$. The signature of the diff function is as follows:

$$\text{diff} : \mathbb{A} \times \mathbb{A} \rightarrow \Delta \tag{6.2}$$

In the following, the term *delta* is used synonymously for a δ describing the differences between two document versions. The counterpart function of diff is called *patch*. It constructs a document version out of the other one by applying the edit operations of a delta:

$$\text{patch} : \mathbb{A} \times \Delta \rightarrow \mathbb{A} \tag{6.3}$$

Thus, given a correct diff and patch algorithm and $A \in \mathbb{A}$, following equations apply:

$$\text{diff}(A,A) = \emptyset \tag{6.4}$$

$$\text{patch}(A,\emptyset) = A \tag{6.5}$$

Obviously, a diff must not detect a change in two equal documents. Additionally, patching a document with an empty delta must not affect the document. To turn this argument on its head, for two differing documents, the delta must contain all information needed to describe the differences. Therefore, for two document versions $A, A_1 \in \mathbb{A}$, with $A \neq A_1$, following equations apply:

$$\text{diff}(A,A_1) = \delta_1, \text{ with } \delta_1 \neq \emptyset \tag{6.6}$$

$$\text{patch}(A,\delta_1) = A_1 \tag{6.7}$$

This ensures the patch function to be the counterpart function of diff.

6.1.3 Edit Operations

A non-empty delta consists of edit operations, conforming to the set property. Thus, the ordering of the operations is irrelevant. An operation must occur only once within a delta. This definition differs from the definition of edit scripts significantly, even if the terms *delta* and *edit script* have been used almost synonymously up to now. I will discuss the impact resulting from the different definitions and my motivation for introducing a novel definition in Section 6.4. Formally, an edit operation is denoted by a tuple.

$$op \in \Delta : (type, path, v, v') \tag{6.8}$$

The first item refers to the *type* of the edit operation, which can turn into *insert*, *delete*, and *update*. The properties of the different operation types will be described in Section 6.2, including the handling of *moves*. The *path* refers to the point where the edit operation shall be applied. I will discuss the addressing scheme in Section 6.4. The entry v contains the new value introduced by the edit operation. The former value is represented by v'. The representation of the value depends on the operation type and will be described in Section 6.2. By storing v and v', the edit operations are *complete* in terms of Marian et al. [2001], thus allowing for reverting changes.

6.1.4 Tree Sequences

A *tree sequence* is a list of adjacent subtrees rooted on the same hierarchy level. All root nodes of a tree sequence share the same parent node. The first root node is called *head*, the remaining nodes *tail*. The head of a single node is the node itself, the tail is empty. A tree sequence can be regarded as ordered forest in terms of graph theory.

6.1.5 Atomicity

Each edit operation is atomic. An edit operation is applied completely or rejected. In this context, a *complete application* means that the new value v is imposed on the document to patch. A partial application of an operation must not occur, which means imposing a subset of v. Note that this definition does not imply a statement on the handling of v'. I will discuss this issue in the context of conflicting edit operations in Section 6.3.3.

6.1.6 Uniqueness

Each edit operation within the same delta will be performed at most once. An edit operation must address clearly one path. An edit operation must not address multiple paths. This definition distinguishes the delta model from document transformation models like XSLT [Clark 1999] or XUpdate [Laux and Martin 2000]. In these models, an operation may be applicable on multiple nodes.

6.1.7 Symmetry

Each delta is invertible. That requires that for each edit operation op, a counterpart operation op^{-1} exists.

$$\forall op \in \delta : (\exists op^{-1} \in \Delta : \text{patch}(A, \delta \cup \{op^{-1}\}) = \text{patch}(A, \delta \setminus \{op\})) \tag{6.9}$$

An inverted delta allows for the reconstruction of a former document version. The inversion of a delta requires symmetrical edit operations.

6.2 Edit Operations

Edit operations reflect the expressiveness of a delta model. In this section, I describe the granularity used in my delta model, before describing the different edit operation types. Finally, I discuss possible other operation types and granularity models.

6.2.1 Granularity of Changes

My delta model supports inserts, deletes, and moves with a granularity of subtrees or tree sequences, respectively. Updates point to single nodes arbitrarily within the tree. The definition of the single operation types emerges directly from the assumptions on the modification patterns of documents which have been elaborated in Section 2.5. I will repeat the five assumptions that affect the granularity of changes:

Assumption 2.1: A leaf node may be replaced by a subtree.

Assumption 2.2: Adjacent subtrees may be inserted or deleted.

Assumption 2.3: Structure-preserving changes (attribute changes) are a frequent operation.

Assumption 2.4: Structure-preserving changes (text changes) are a frequent operation.

Assumption 2.5: Subtrees are often re-arranged.

The delta model meets the first assumption by allowing for the insertion and deletion of a subtree, where a leaf node is a subtree with only one root element. The second assumption is met by the ability to express the insertion and deletion of tree sequences. Update operations represent both attribute and text changes, thus meeting the third and the fourth assumption. The re-arrangement of subtrees mentioned in the last assumption is expressed by the move operation. In general, my delta model resembles the delta model by Cobéna [2003]. However, his delta model is not able to address tree sequences.

6.2.2 Insert

An insert operation addresses a subtree or a tree sequence. The path of an insert operation points to the path where the root of the inserted subtree will be placed after patching. An already existent node at this address and all of its following siblings will be shifted to the right. A subtree designated to be inserted as the last element of a subtree would also refer to the path where it will be placed, even if the path does not exist in the original document already. In an insert operation, v is empty, whereas v' contains the subtree or tree sequence to insert. The counterpart operation of insert is delete.

6.2.3 Delete

The delete operation is the counterpart operation of an insert. It addresses a subtree or a tree sequence. It points to the path where the root of the subtree to delete is placed before patching. In a delete operation, v contains the subtree or tree sequence to delete, v' remains empty. As a delete operation may address single subtrees and tree sequences as well, the question arises how a patch procedure knows which nodes are to be deleted. They can be estimated by taking v into account. As v contains the part of the document to be delete, the nodes to delete directly emerge from that.

6.2.4 Move

Moving subtrees or tree sequences is more difficult than inserting or deleting them. This results from the fact that this operation points to two paths: the first for the source of the move, the second for the target. To avoid an extended notation, I represent a move operation by linking a delete operation with a congruent insert operation.

Definition 6.1 *A delete operation* op_1 *and an insert operation* op_2 *are congruent, if and only if* $v_{op_1} = v'_{op_2}$.

This definition ensures that the targeted subtrees or tree sequences are identical. Both operations receive an attribute containing a common unique identifiers[1]. As a move is atomic, too, the patch application has to ensure that the linked edit operations are either both applied or both rejected. The counterpart operation of a move is a move again, where both operations have been inverted.

6.2.5 Update

In contrast to the other edit operations, an update operation may address a single node arbitrarily within a tree. The path points directly to the addressed node. The former value of the node is stored in v, the new value in v'. The counterpart operation of update is an update again. The main intention of an update is to represent attribute changes in element nodes, or content changes in text nodes. An update is a structure-preserving edit operation.

6.2.6 Discussion

Insert, delete, and update are the basic operations in tree editing. As a re-arrangement of subtrees may be likely, I have introduced the move operation, as, e.g. Chawathe et al. [1995]. However, other operation types would be possible either.

[1] In the current implementation, the uniqueness is ensured by using the system time in nanoseconds. I assume the probability of collisions to be in a negligible magnitude.

Barnard et al. [1995] have defined a *swapping* operation, exchanging two subtrees with each other without changing them. This is a special case of re-arranging subtrees that can be expressed with a move operation, too.

Chawathe and Garcia-Molina [1997] have introduced the *copy* operation that expresses the multiplication of a subtree across the document. As their delta model demands for symmetrical edit operations, too, they have introduced a counterpart operation to copy, called *glue*. A glue represents the collapsing of two identical subtrees into one. The motivation for the copy operation may become apparent, but the glue operation does not reflect a "natural" editing action in a document. Additionally, copy and glue demand for a more complicated notation, which in my opinion hinders the human-readability of that operation. For these reasons, my model does not support copy and glue operations.

Several new edit operations have been introduced by Iorio et al. [2009] that aim to reflect the natural editing process on documents. They have defined a *split* operation that breaks a text node into two nodes. The counterpart operation is *join*. Once again, the readability of these operation is questionable. They have also defined a *upgrade* and a *downgrade* operation. These operations correspond to the node-centric insert and delete operations defined by Tai [1979]. The arguments for using a subtree-oriented granularity of insert, delete, and move are apparent. However, there are also reasons for supporting the node-centric granularity by Tai [1979]. For example, a paragraph may be turned into a subparagraph by adding a parent node within the tree. This would correspond to a node insert [Tai 1979] or a downgrade operation [Iorio et al. 2009]. In my model, this change would be represented by a move or a pair of an insert and a delete operation, which is obviously less meaningful. On the other hand, this example does not justify to switch to a node-centric granularity, which would lead to far more edit operations within a delta, precluding a human-readability.

The Microsoft XML Diff tool presented in Section 3.3.2 uses a hybrid model that allows for subtree-oriented insert and delete operations, as well as for node-centric operations. This way, the benefits of both approaches are combined. However, a node-centric and a subtree-oriented operation cannot be represented within one operation type in my delta model. This results from the merge-capability that is not given in the Microsoft XML Diff tool, as following example shows. Given a single node to delete within the tree with child nodes, the patch algorithm cannot decide whether the children have been there before, or whether the delete conflicts (see Section 6.3.3). To avoid this hazard, my edit model would need additional insert and delete operations that are only defined on single nodes. As this complicates the edit model significantly, I decide to not support node-centric insert and delete operations.

6.3 Dependencies

Edit operations may change the structure of the tree, thus affecting the paths to existing nodes. In this case, the nodes and the corresponding edit operation interfere. A delta may contain an arbitrary number of edit operations. However, they may not address the same nodes to avoid overlapping edit operations. Finally, conflicts can arise during patching. In the following,

these notions are defined. The section ends with an example showing the implications of the definitions.

6.3.1 Interfering Edit Operations

Insert, delete, and move operations change the structure of the document. An edit operation performed on a previous part of the document may affect the paths to all subsequent nodes.

Definition 6.2 *An edit operation $op \in \delta$ interferes with a node $i \in A$, if and only if $path_{op} < i$ and the application of op does affect the path to i.*

An edit operation may only interfere with nodes located in a subtree that shares a common parent node. Update operations are structure-preserving. Therefore, they cannot interfere with other nodes.

Definition 6.3 *For an edit operation $op \in \delta$ that interferes with a node $i \in A$, $offset(op, i)$ denotes the difference that, if added to the path to i, would result in the right path.*

Note that the offset can also become negative in order of deletions.

6.3.2 Non-Overlapping Edit Operations

Edit operations in a delta must not address the same nodes. They must describe non-overlapping parts of the documents.

Definition 6.4 *Two edit operations $op_1, op_2 \in \delta$, with $op_1 \neq op_2$ are non-overlapping, if and only if $v_{op_1} \cap v_{op_2} = \emptyset \wedge v'_{op_1} \cap v_{op_2} = \emptyset \wedge v_{op_1} \cap v'_{op_2} = \emptyset$.*

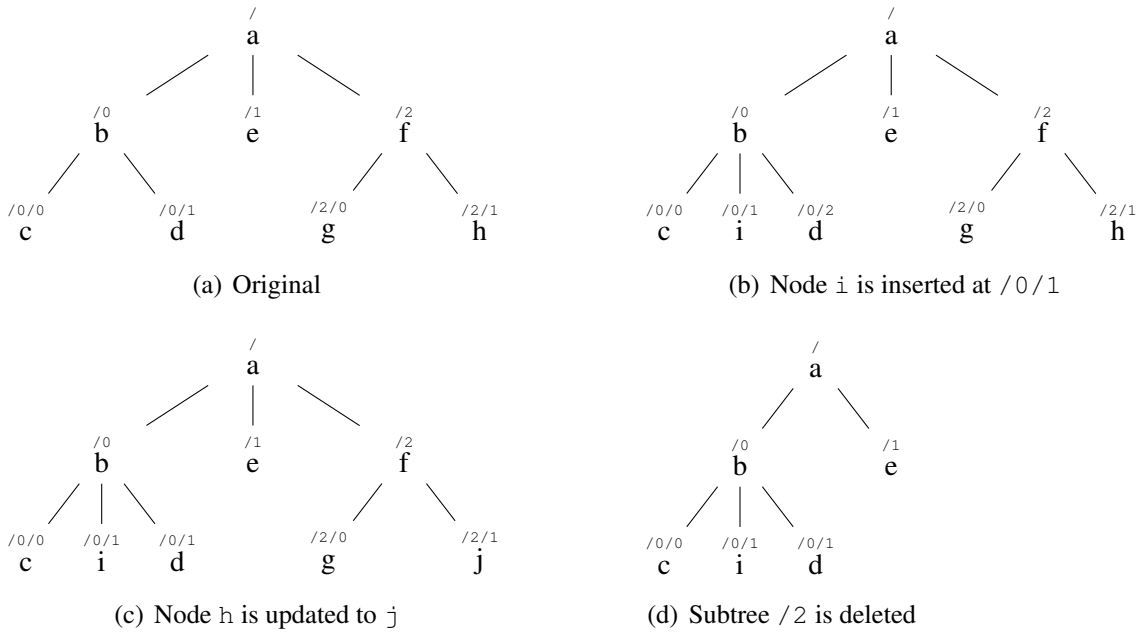
A delta must not contain overlapping changes:

Definition 6.5 $\forall op \in \delta : (\forall op_1 \in \delta \backslash op : op$ *and* op_1 *are non-overlapping).*

These definitions ensure the order of the edit operations within a delta to be irrelevant, which is directly reasoned from the set property of the delta. Non-overlapping edit operations are therefore commutative and can be applied in any order.

6.3.3 Conflicts

A conflict arises in case that for an edit operation $op \in \delta$, v_{op} does not match the corresponding pattern found in the document to patch. As v is empty for insert operations, an insert may not conflict. Conflicts strongly resemble to overlapping changes. However, conflicts arise from changes performed outside the same delta. As conflicts only occur during patching, I will discuss the identification and the handling of conflicts in context of the patch algorithm in Section 8.4.



(a) Original

(b) Node `i` is inserted at `/0/1`

(c) Node `h` is updated to `j`

(d) Subtree `/2` is deleted

Figure 6.1: The insertion of `i` interferes with node `d`, but not with the other subsequent nodes, as they do not share the same parent node. The deletion in 6.1(d) overlaps with the update of 6.1(c), which is not allowed in my delta model.

6.3.4 Examples

Figure 6.1 shows an example for an interfering edit operation and overlapping edit operations as well. A document tree 6.1(a) is first changed by inserting node `i` at path `/0/1` in 6.1(b). This insert interferes with node `d` located at `/0/1`, which is pushed to the path `/0/2`. As no other subsequent node shares the same parent node, this is the only interfering node for this edit operation. In a second step, node `h` at `/2/1` is updated to `j` in 6.1(c). The third edit operation in 6.1(d) deletes the subtree rooted ad `/2`. This edit operation would overlap with the previous update operation. Therefore, the edit operations in 6.1(c) and 6.1(d) cannot be represented within one delta. If both edit operations would be represented by different deltas and applied subsequently, this would result in a conflict.

6.4 Set-Based Delta vs. Edit Script

Usually, the output of a diff algorithm is stored as an edit script. This edit script is used to reconstruct a document version sequentially. Before using a discrete patch tool, edit scripts have been applied by the editing application itself. For example. the classical editor `ed` uses a command language for processing text that has been used for the application of edit scripts [MacKenzie et al. 2002].

Within an edit script, all operations are addressed assuming the previous operations having been applied. This design is appropriate when one can ensure that the edit script is only applied to the document version it was computed for. However, I expect my delta to be applied on a document version different from the version the delta was created upon. This has two major implications. First, the paths of the edit operations may point to a wrong path, resulting from offsets from foreign changes, which corresponds to interfering edit operations. Second, some edit operations of the delta may not be applied, just in case that the correct path cannot be found by the patch algorithm. In both cases, the relative offsets between the edit operations may be broken, requiring a re-computation of the paths of the edit operations.

In my approach, all edit operations are addressed with respect to the original version of the document, on which the delta was created upon. This design is directly derived from the set property of the delta. By this, rejected edit operations from the delta cannot affect the addressing of subsequent edit operations.

The set-based delta has a second big advantage that appears during interactive merging. The user may decide upon ambiguous edit operations in an arbitrary order. This enables the user to navigate through the whole document to estimate the impact of an applied edit operation. I will present a corresponding interactive editor in Section 11.3.1.

6.5 Representing Context in XML

Line-based text documents can be regarded as a one-dimensional entity, i.e., a sequence of lines. The definition of the syntactic context of an edit operation on this kind of documents by Davison [1990] is straightforward: take the surrounding lines and store them within the delta. This definition was made with source code in mind, where code lines are usually limited to 80 characters[2]. Mapping the syntactic context onto XML documents raises two questions. First, how to represent the context within a tree, as it is a two-dimensional entity. Second, how to deal with XML nodes that can have an arbitrary length.

In my approach, I decide to use the document order for the representation of the context, including the node value to normalize the node lengths (and encodings). This results in the definition of the context fingerprint, which shows a standard behavior at the borders of the document, where no context nodes exist. Afterwards, I present two example fingerprints. I discuss my definition related to the context notion of Mouat [2002] at the end of this section.

6.5.1 The Context Fingerprint

The context of an edit operation is stored in a so-called *fingerprint*, which is a sequence of the values of all nodes within a given radius r ($r \geq 0$) around the edit operation in document order.

Before defining the fingerprint, I introduce $A_{fingerprint}$, which denotes the document A where

[2] In fact, this limitation is not a technical one, but most coding guidelines limit the line length.

every node of a delete operation is removed, except its head:

$$A_{fingerprint} \stackrel{\text{def}}{=} A \backslash tail(v) \tag{6.10}$$

This definition ensures that the fingerprint does not refer to the parts to remove within a delete operation. The fingerprint of a node i is a sequence, ordered by the distance relating to i:

$$fingerprint_i[r] \stackrel{\text{def}}{=} \{value(j)\}, \text{where } j \in A_{fingerprint} \wedge 0 < |dist(i,j)| \leq r \tag{6.11}$$

The element of the fingerprint with distance d relating to i is denoted as $fingerprint_i[d]$. The value of $fingerprint_i[0]$ is referred to as *anchor*. The value of the anchor depends on the type of the edit operation:

$$fingerprint_i[0] \stackrel{\text{def}}{=} \begin{cases} value(head(v')) & \text{for insert operations} \\ value(head(v)) & \text{otherwise} \end{cases} \tag{6.12}$$

For a delete operation, the anchor contains the head of the subtree or tree sequence to delete. The anchor of an update operation is the node value before the update. During a patch, the anchor can be used to detect conflicting updates as well as for finding the right offset for a delete operation. For insert operations, however, no former value v is given. Using the head of v' avoids an empty anchor for insert operations and allows for a simpler inversion of these operations (see Section 6.6.1).

The fingerprint has a size of $n = 2r + 1$, with the anchor node in the middle of the sequence. According to the definition of the fingerprint as a sequence, hash values may occur multiple times. This is important to represent repeating node values, which are common in XML documents.

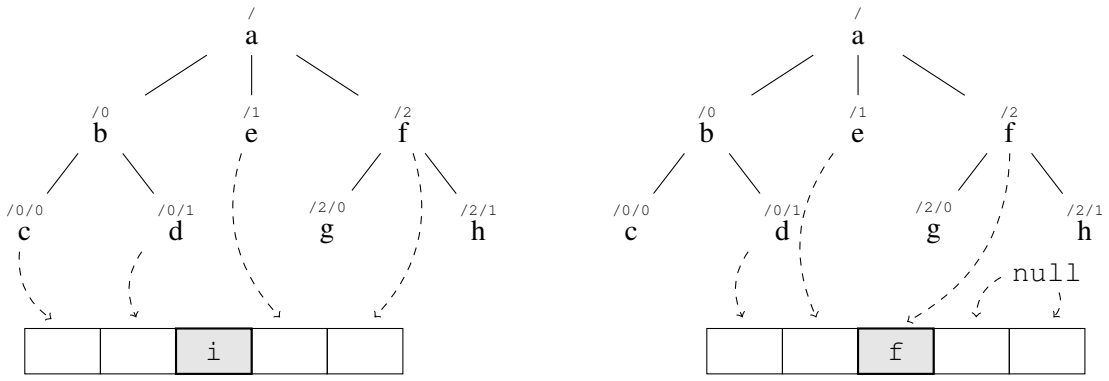
The fingerprint basically does not contain any information about the path of the edit operation within the document tree. The type of the edit operation does not emerge from the fingerprint as well. Additionally, the fingerprint may not only be computed for edit operations. Basically, each node of an XML tree has a fingerprint. In this case, however, $A_{fingerprint}$ does not need to be computed – using A is sufficient.

6.5.2 Fingerprinting Document Borders

In case there are less than the required number of nodes available to fill the fingerprint, a special "null value" is used to represent the document borders. Appendix B decries the computation of the null value.

6.5.3 Examples

Two examples shall illustrate the computation of the context fingerprint. Figure 6.2 shows a simple tree.



(a) Node `i` shall be inserted at `/1` with its tail

(b) The subtree at `/2` shall be deleted

Figure 6.2: The anchor of an insert operation contains the head of the tree sequence to insert. For a delete operation, the subtree is not represented in the fingerprint. At the end of the document, a null value is used to fill the fingerprint.

First, a node sequence shall be inserted at path `/1` in 6.2(a). The anchor is highlighted in gray and contains the head `i` of the tree sequence[3]. The context nodes are taken from the tree in document order, thus leading to the fingerprint `c,d,i,e,f`. This fingerprint has a radius of 2 and a size of 5. Note that the fingerprint itself does not show whether `i` is inserted as single node or whether it has a subtree. This can only be gathered from v of the corresponding edit operation.

The delete operation shown in 6.2(b) exemplifies the use of $A_{fingerprint}$ and the behavior at the document borders. The subtree rooted ad `/2` shall be deleted. Therefore, its descendants have to be removed to compute $A_{fingerprint}$. The root of the subtree is stored as anchor of the fingerprint. As the document reaches its borders, null nodes have to be used to fill the document. On a later application of the delta, they would indicate that the edit operation shall be applied at the end of the document. The resulting fingerprint would be `d,e,f,null,null`.

6.5.4 Document-Order vs. Parent-Child Relationship

A first attempt to define a context-aware delta format for XML has already been performed. Mouat [2002] has presented a delta format, called *Delta Update Language* (DUL). In DUL, an edit operation may contain the siblings of the node addressed by the edit operation, as well as the parent and children nodes. However, DUL raises some questions.

First, the context shall be symmetric, i.e., the same amount of nodes should be stored in either direction. My context fingerprints demand to be symmetric, too. However, the handling of the document borders is not defined in Mouat's delta model. Additionally, a subtree basically contains far less nodes, but only nodes with the same parent node are considered to be

[3]To be precise, the anchor contains the value of `i`. This notation is used for a better readability throughout all examples on fingerprints.

siblings. The DUL specification does not cover the case where not enough siblings are present to fill the context. The same problem arises for leaf nodes. A tree is far larger in direction to the leaves than to the root. The question, how a symmetry should be ensured in a parent-children relationship, especially for edit operations on leaf nodes is not covered by Mouat's delta model.

A second issue affects the inclusion of child nodes for the deletion of subtrees. The DUL specification does not cover this special case that has lead to the definition of $A_{fingerprint}$ in my delta model. Therefore, it seems that a delete operation considers the nodes to delete being part of the operation context, which is obviously not meaningful.

Admittedly, the mentioned weaknesses of the definition could be refined, thus creating a more precise and comprehensive specification of DUL. Nevertheless, I consider my approach to be more appropriate for the domain of XML documents, which is motivated by a simple example. Consider a document where the text of a paragraph is changed. Every paragraph will likely have a similar parent node, as I assume paragraphs to be typeset in a similar way. In that case, the parent-child relationship does not offer any helpful information for merging. Only the content of the surrounding paragraphs (covered by the fingerprint) provides a major possibility of distinction.

6.6 Inversion, Aggregation, and Decomposition

The inversion of deltas is an important aspect of document change control. Shen and Sun [2002] have stressed the importance of inverted edit operations to be able to revert changes. In case that an edit operation cannot be applied, it might be appropriate to revert the other changes, too, even if they have already been applied. First, I show the inversion of single edit operations, before discussing the inversion of deltas as a whole. The section ends with a discussion on delta aggregation and decomposition. Note that I use a non-formal notation as a thorough formalization would require a significant overhead. A more formal notation has been proposed by Vion-Dury [2010].

6.6.1 Inverting Edit Operations

Inverting single edit operations is rather simple. Here, the advantage of defining both the former value as well as the new value of a node becomes apparent. This allows for defining the inversion of edit operations according to Table 6.1. A move operation is inverted by inverting the corresponding insert and delete operation.

6.6.2 Inverting Deltas

The inversion of whole deltas for version reconstruction is slightly more complicated. In this context, interfering edit operations have to be taken into account. The inverted delta is applied to a document version where the non-inverted edit operations have already been applied. The

Original edit operation	Inverted edit operation
$(\text{insert}, \mathit{path}, \emptyset, v', \mathit{fingerprint})$	$(\text{delete}, \mathit{path}, v', \emptyset, \mathit{fingerprint})$
$(\text{delete}, \mathit{path}, v, \emptyset, \mathit{fingerprint})$	$(\text{insert}, \mathit{path}, \emptyset, v, \mathit{fingerprint})$
$(\text{update}, \mathit{path}, v, v', \mathit{fingerprint})$	$(\text{update}, \mathit{path}, v', v, \mathit{fingerprint})$, where $\mathit{fingerprint}[0] := \text{value}(v')$

Table 6.1: Delta inversion is basically done by swapping parameters. Update operations require to modify the fingerprint for inversion. A move is inverted by inverting the corresponding insert and delete operation.

application of these edit operations may have been interfering with nodes targeted by the inversed edit operations. Therefore, the path stored within the inversed edit operations may point to a wrong node.

To avoid this hazard, all paths have to be recomputed, taking the offsets caused by the non-inversed edit operations into account.

6.6.3 Aggregation and Decomposition of Deltas

Deltas that describe subsequent changes may be aggregated into a larger one. Here, two constraints must be considered. First, all edit operations of a delta are addressed relating to the document version it has been computed for. Consider two deltas δ_1, δ_2 that describe the evolution of a document A across the version A_1 into A_2 in the following way $A \xrightarrow{\delta_1} A_1 \xrightarrow{\delta_2} A_2$. The paths of all edit operations in δ_2 have to be adapted with respect to A to build the aggregated delta $\delta_{1,2}$. Once again, this is done by respecting the offsets resulting from the application of δ_1.

The second constraint affects non-overlapping changes. Basically, a node can be targeted by different edit operations from different deltas, thus leading to overlapping changes. To recover a single delta with non-overlapping changes, two overlapping edit operations have to be merged into one edit operation.

Decomposing a delta means splitting a delta up into one or more deltas. A delta $\delta_{1,2}$ with more than one edit operation, describing $A \xrightarrow{\delta_{1,2}} A_2$ can be decomposed into two deltas δ_1 and δ_2 that way that $A \xrightarrow{\delta_1} A_1 \xrightarrow{\delta_2} A_2$ holds. The decomposition can be regarded as a sort of counterpart operation to the aggregation. It is not, however, as merged edit operations are not split up again. During the decomposition, interfering changes are an issue, too.

A delta could be decomposed in a way that divides a delta in one delta with one edit operation and a second delta containing the remaining edit operations. On the one hand, the computation is not that complex. On the other hand, a completely decomposed delta results in single deltas containing one edit operation each. This decomposition is nearly equivalent to the operational transformation approach presented in Section 4.3.

6.7 Conclusions

The ability to precisely describe changes between document versions is a key prerequisite to a reliable change control of documents.

In this chapter, I have presented a concise state-based delta model for XML documents. It allows for describing insert, delete, move, and update operations within a delta. It is completely invertible, which enables the user to revert changes. Additionally, deltas can be aggregated and decomposed. With an aggregated delta, comprehensive changes can be consolidated into a more compact representation. Decomposition allows for a fine-grained depiction of the document evolution, resembling the edit model used in the operational transformation approach of collaborative editors.

An important aspect of my delta model is the enrichment of edit operations with their syntactic context. I have introduced the context fingerprint as an efficient representation of the surrounding nodes around an edit operation. It enables a patch procedure to merge document versions using deltas. I present a corresponding algorithm in Chapter 8. The computation of a context delta using a differencing algorithm is described in the following chapter.

7 An Efficient Differencing Algorithm

A differencing algorithm computes a representation of the changes between two document versions. In this chapter, I present XCC Diff, an XML differencing algorithm, based on the delta model presented in the last chapter. The focus of the algorithm design lies on a competitive time complexity, as well as on human-understandable deltas. In general, a understandable change representation is favored over a small delta. The basic idea of the algorithm is presented in the following section, including the outline of this chapter.

7.1 Basic Idea and Outline

The goal of the diff algorithm is to compute an intuitive representation of the changes between two document versions that should be human-understandable. I do not provide a metric for intuitiveness and human-readability, but a simple rule applies. In general, one larger edit operation is favored over several small edit operations in a cluster. This proposition is based on the findings of Neuwirth et al. [1992] who performed a user study. They have stressed that clustered changes are difficult to understand by humans.

In general, I distinguish between structure-preserving changes, i.e., update operations, and structure-affecting changes, like insert operations. Structure-preserving changes are easier to identify, as they affect only single nodes. Structure-affecting changes address whole subtrees, which makes them harder to delimit by a diff algorithm. Therefore, the algorithm tries to reduce the complexity by working off the structure-preserving changes first. The algorithm works in three steps:

1. Find the common content by calculating the longest common subsequence of leaf nodes.
2. Identify the structure-preserving changes in the common parts.
3. Construct the structure-affecting changes for the differing parts.

During the diff run, matching nodes from both documents are identified and added to a list. Edit operations are created for the non-matched node pairs. At the end of the diff algorithm, each node is either part of the match list or the delta.

The key idea of the algorithm is straightforward: the longest common subsequence of all leaf nodes is computed. The corresponding parent nodes are compared during a bottom-up pass using a dynamic programming approach. The leaf-orientation of the algorithm is chosen to catch the common content of the document versions. At this point, one should recall the assumption on the amount of modifications, which has been elaborated in Section 2.5.3:

Assumption 2.6: A new version of a document will likely have only few changes compared to the size of the document.

Deriving from this assumption, I expect the length of the LCS to be close to the amount of leaves. For these unchanged parts of the document, the corresponding ancestors have to be checked for updates, which are easier to identify, as they are structure-preserving changes.

After the handling of the common parts of the documents, the structure-affecting edit operations have to be caught. Basically, this targets insert, delete, and move operations, which are created upon the leaf nodes that are not part of the common subsequence. In this step, the challenge is to identify non-overlapping edit operations without missing changes. Here, I call to mind the assumption on the distribution of the modifications within the document tree:

Assumption 2.7: Changes on a document affect only small parts of the document in general.

Deriving from this assumption, the algorithm tries to glue adjacent edit operations together into one edit operation containing a tree sequence.

In the following, I describe these steps in more detail. In the end, I analyze the time and space complexity of the proposed algorithm.

7.2 Finding the Common Content

The content of an XML document is stored within the leaves of the tree. The diff algorithm has a content-centric view towards the changes in a document. Therefore, the algorithm starts by computing the longest common subsequence of the leaves of the two trees to compare.

7.2.1 Longest Common Subsequence of Leaf Nodes

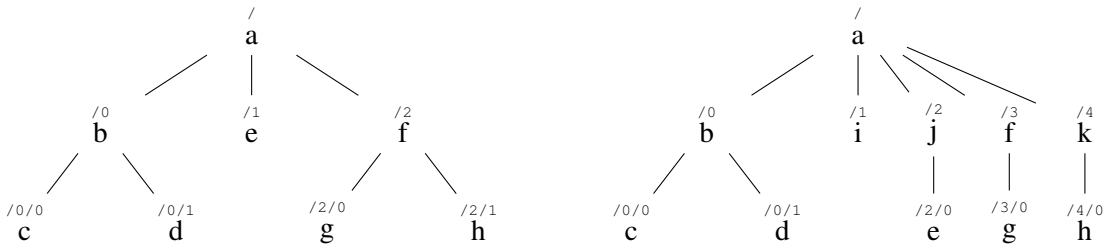
In the first step, the node values of the leaves are transformed into one sequence for each tree, ordered by their respective document order. Basically, this step of the diff algorithm shall identify the unchanged parts of the document.

Certainly, a comparison of leaf nodes does not allow for estimating the changes performed on higher levels of the tree. In this context, I recall the differentiation of structure-preserving and structure-affecting changes. Structure-preserving changes (represented by update operations) cannot be estimated by comparing the leaf nodes. However, leaf nodes allow for catching most structure-affecting changes by taking their depth into account. Therefore, the LCS respects the depth of the leaves to compare.

Definition 7.1 *Two leaf nodes $l_1 \in A_1$, $l_2 \in A_2$ match, if and only if* $value(l_1) = value(l_2) \wedge depth(l_1) = depth(l_2)$.

Two leaf nodes with equal node value but differing depth do not match. This meets the case where a node is inserted or deleted from the ancestors of the leaf, which would require an insert and a delete operation to represent the change using the delta model presented in the previous chapter.

All matched node pairs are added to the match list.



(a) Original (b) Updated

Figure 7.1: Node `k` matches, although it is part of another subtree. Node `e` does not match, as its depth has changed. The resulting leaf LCS is `c,d,g,h`.

7.2.2 Reallocated Leaves

Not all structure-affecting changes are caught by the definition above. A leaf node that has been reallocated to another subtree on the same top-down level will be considered equal, even if this is apparently not meaningful. This step of the diff algorithm strictly focuses on the leaf nodes. A misinterpreted match of leaf nodes will be corrected in a later step of the algorithm, presented in Section 7.3.3.

7.2.3 The Algorithm

Algorithm 1 shows a simplified version of this step of the differencing algorithm. An implementation of this algorithm should rely on an existing algorithm for the computation of the longest common subsequence, e.g., on the algorithm presented by Myers [1986]. A common LCS algorithm would compare two nodes only upon their value, thus neglecting their depth. Therefore, it is important to add the matching criterion to the LCS algorithm.

Algorithm 1 Calculate the longest common subsequence of leaf nodes.

for all $(l_1, l_2), l_1 \in \mathit{leaves}(A_1), l_2 \in \mathit{leaves}(A_2) : l_1$ matches l_2 **do**
  add (l_1, l_2) to *matchList*
end for

7.2.4 Example

Figure 7.1 exemplifies the computation of the LCS of leaf nodes. The tree in 7.1(a) is changed by inserting a new leaf node `i` at path `/1`. Leaf node `e` is pushed a level down by inserting a new parent node `j`. Leaf node `h` is reallocated from its parent `f` to the newly inserted node `k`. Computing the LCS of the leaves between the trees shown in 7.1(a) and 7.1(b) would result in the sequence `c,d,g,h`. The newly inserted node `i` is obviously not part of the LCS. Even

if e keeps its relative position with respect to the other nodes, it is not part of the LCS, as its depth changed due to the insertion of j. Despite having another parent node, h is part of the LCS, as its value and depth did not change.

7.3 Identifying Structure-Preserving Changes

Updates are a structure-preserving edit operation. They may occur on parent elements of matched leaves. This step of the diff algorithm identifies updated parent nodes and detects reallocated leaves that must be removed from the list of leaf LCS. A dynamic programming approach is used to avoid unnecessary computation steps.

7.3.1 Update Operations

This step of the algorithm is performed for all elements of the leaf LCS. Nodes that are not part of the LCS have been inserted or deleted by definition. Therefore, they are part of a structure-affecting change that is not covered by an update operation. Therefore, updates on leaf nodes are handles by a later step of the algorithm (see Section 7.4.2).

To identify update operations, the tree is traversed bottom-up from each matched leaf of the LCS up to the root. This traversal is performed in parallel for both trees compared. The traversal is performed as long as the corresponding nodes match.

Definition 7.2 *Two non-leaf nodes* $n_1 \in A_1$, $n_2 \in A_2$ *match, if and only if* $value(n_1) = value(n_2) \wedge \exists l_1 \in descendants(n_1) : (\exists l_2 \in descendants(n_2) : (l_1 \text{ matches } l_2)) \wedge depth(n_1) = depth(n_2)$.

In case that the nodes do not match, but the descendants, a corresponding update operation is added to the delta: $(update, path(n_1), value(n_1), value(n_2))$.

7.3.2 Optimizing the Tree Traversal

A tree significantly narrows towards the root. The higher the height of a node, the higher is the probability of the node for being ancestor of many different leaves. In a naive approach, all ancestors of all matched leaves would be compared. This approach has two severe drawbacks. First, many unnecessary computation steps would be performed, significantly affecting the runtime. Second, an update would be identified multiple times if the node has more than one leaf. To avoid this, for each identified update operation, the delta must be verified whether the update is already part of it.

To avoid these drawbacks, the algorithm tags a node as soon as it is handled. For the first element of the leaf LCS, the tree is traversed up to the root. For the following elements, the tree is traversed until the first already visited node is reached. This dynamic programming approach ensures that each node is only handled once, thus significantly reducing the time complexity. Additionally, an update cannot be identified twice.

7.3.3 Detecting Reallocated Leaves

As already described in Section 7.2.2, a pair of leaf nodes may be considered matching, even if they are parts of different subtrees. This case occurred seldom during the tests of the diff algorithm, but must be covered in order to guarantee a correct result.

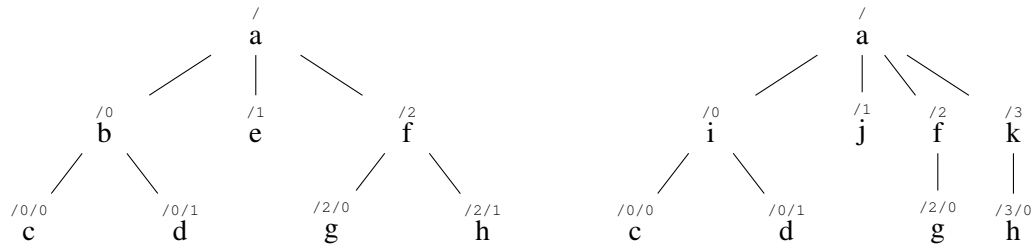
The reallocated leaves are detected during the tree traversal that is performed to identify structure-preserving changes. The detection is based on following consideration: The bottom-up traversal will lead to an already matched node pair. A reallocated node, however, will have other siblings in the same subtree, and a different path towards the root. Therefore, an inner node will already be marked as visited, whereas its counterpart in the other tree has not been visited yet. As a result, the reallocated nodes are removed from the match list. They will be handled during the creation of structure-affecting changes in Section 7.4.

7.3.4 The Algorithm

The code of this step is shown in Algorithm 2. Each node pair is either appended to the match list or added to the delta as update operation. Reallocated leaves are removed from the match list. This way, they can be targeted by insert or delete operations in the next step of the algorithm.

Algorithm 2 Structure-preserving changes on the ancestors of the already matched parts are identified by the second step of the algorithm.

for all $(l_1, l_2) \in matchList$ **do**
  for all $a_1 \in ancestors(l_1), a_2 \in ancestors(l_2)$ **do**
    if (a_1, a_2) has not been visited **then**
      if a_1 does not match a_2 **then**
        add $(update, path(a_1), value(a_1), value(a_2))$ to $\delta_{A_1 \to A_2}$
      else
        add (a_1, a_2) to $matchList$
      end if
      mark (a_1, a_2) as visited
    else if (a_1, a_2) have been visited **then**
      break
    else if (a_1, x_2) **or** (x_1, a_2), with $x_1 \in A_1, x_1 \neq a_1, x_2 \in A_2, x_2 \neq a_2$ have been visited **then**
      remove (l_1, l_2) from $matchList$
      break
    end if
  end for
end for



(a) Original (b) Updated

Figure 7.2: Computing the structure-preserving changes. The leaf LCS is `c,d,g,h`. Node `/0` is detected as update and `h` as reallocated.

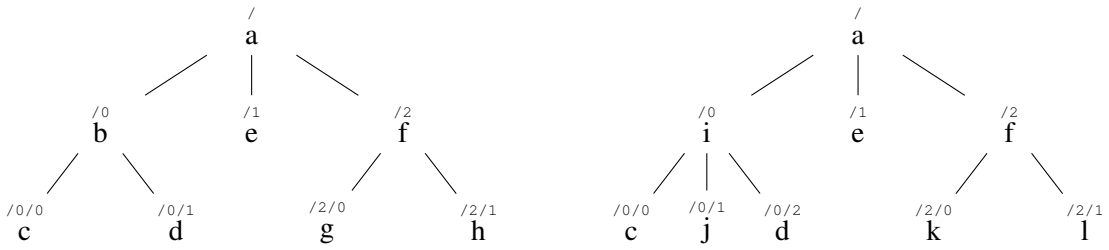
7.3.5 Example

The identification of structure-preserving changes is shown using the documents in Figure 7.2. The non-leaf node `b` is changed to `i`, the leaf `e` to `j`, and the leaf `h` is reallocated to the newly inserted node `k`. The first step of the diff algorithm computes `c,d,g,h` as longest common subsequence of leaf nodes. The subsequent bottom-up pass is started at `/0/0`, because of `c` being part of the match list. As no node has been visited yet, all ancestors up to the root are visited. For the node at `/0`, the node values do not match. Therefore, a corresponding update operation is created and added to the delta. For node `d`, the bottom-up pass stops already at `/0`, as it has already been visited.

The same procedure is performed for node `g`. Node `h`, however, has been reallocated. The bottom-up pass starts at `/2/1` in the original document and at `/3/0` in the modified document. In the original document, `f` is the first ancestor which is accessed. It is marked as visited from the previous bottom-up traversal for `g`. In the modified document, however, `k` is accessed, which is not marked as match for `f`. Therefore, `h` is identified as being reallocated, and removed from the match list created by the LCS run on the leaves.

7.4 Catching Structure-Affecting Changes

After having identified the common parts of the documents including structure-preserving changes, the non-matched parts of the document have to be considered. These are basically the inserted or deleted parts of the document. As they have a subtree-based granularity, the corresponding nodes have to be joined to a subtree. However, an insert or delete on the leaf level may also mean an update operation, which has to be identified, too. Adjacent subtrees targeted by edit operations are glued together into one edit operation containing the whole tree sequence. Congruent insert and delete operations are connected into a move operation.



(a) Original (b) Updated

Figure 7.3: Computing structure-affecting changes. The leaf LCS is `c`, `d`, `e`, the update on `/0` has been identified in a previous step. Node `j` is detected as inserted. The subtree rooted at `/2` is represented as a delete and an insert operation.

7.4.1 Constructing Insert and Delete Operations

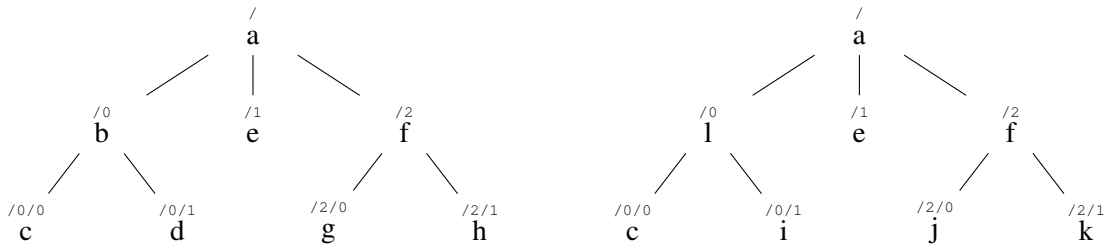
Basically, any non-matched node is part of an insert or delete operation. A node which is only present in the modified document is part of an insert operation, any non-matched node in the original document is part of a delete operation.

The construction of insert and delete operations starts at the leaves, too. Any leaf that is not part of the longest common subsequence of leaves is considered to be either inserted or deleted. This step of the algorithms starts at the first non-matched leaf. A bottom-up traversal is performed, until a node is reached that has already been visited by the previous step of the algorithm. Clearly, this pass can only be performed on one of the trees, as the nodes to insert or delete are not part of the other tree.

An already visited node indicates that at least one of its children has already been matched. Therefore, the structure-affecting edit operation is only created for the subtree rooted below the already matched node. If a subtree is only present in the modified document, an insert operation is added to the delta: $(\mathit{insert}, \mathit{path}, \emptyset, \mathit{subtree}(\mathit{path}))$. Otherwise, a delete operation is created: $(\mathit{delete}, \mathit{path}, \mathit{subtree}(\mathit{path}), \emptyset)$.

After the creation of an edit operation, all nodes from the targeted subtree have to be added to the match list. This ensures that an edit operation is not created multiple times for each leaf node that is part of the affected subtree.

Figure 7.3 shows an example of the construction of insert and delete operations. After computing the LCS of the leaves and the detection of the structure-preserving changes, the algorithm starts at leaf `j`, which has been inserted, as it is only part of the updated document. Its first ancestor is `i`, which has already been visited (and handled as update operation) by the previous step of the algorithm. Therefore, an insert operation is defined and added to the delta. The next non-matched leaf node is at position `/2/0`. The following bottom-up pass stops at the root, as the non-changed node `f` has not been visited before. The subtree spanned by `f` is added to the delta as insert and as delete operation, respectively.



(a) Original (b) Updated

Figure 7.4: The algorithm detects leaf update operations for the nodes at `/0/1`, `/2/0`, and `/2/1`. The update at `/0` has been handled by the previous step.

7.4.2 Updates on Leaf Nodes

In the current state of the algorithm, each change on a leaf will be represented by an insert and a delete operation, as non-matched leaves are not part of the leaf LCS. At this point, I call to mind one assumption on the modification pattern (see Section 2.5.2):

Assumption 2.4: Structure-preserving changes (text changes) are a frequent operation.

Apparently, the representation by insert and delete operations does not reflect the nature of the change. Even if this is human-understandable, a representation by update operations would be more intuitive.

To achieve this, the context of insert and delete operations on the leaf level is taken into account. This step is performed in case that the parent node of the corresponding leaf has already been matched or is part of the update list, respectively. Now, the algorithm checks whether the previous and the next already matched sibling are identical. In that case, an update operation is created on the leaf.

Text nodes are often the only leaf of their parent node. Therefore, an update operation is also created in case that the parent node is identical and no already matched sibling exists which could be analyzed.

Figure 7.4 shows an example of the update identification on leaves. During the bottom-up pass for node `/0/1`, the algorithm checks the parent and the sibling node. The parent node has changed, but is already part of the update list, resulting from the previous step of the algorithm. The sibling is identical. Therefore, an update operation is created. Updates are also created for the nodes at `/2/0` and `/2/1`. Here, the parent node is identical and `j` has no already matched sibling which could be analyzed. When `k` is inspected, its sibling has already been matched.

7.4.3 Gluing Adjacent Operations

The delta model requires changes to be non-overlapping. This property is met by the actual differencing algorithm. However, the resulting delta may contain edit operations that target

adjacent nodes. In my opinion, this is less meaningful than having one operation containing the adjacent changes within a tree sequence. Therefore, adjacent edit operations should be merged into one by gluing the adjacent subtrees together.

As soon as an already matched node is reached during the bottom-up pass, the algorithm checks whether the preceding sibling is affected by an insert or delete operation. In that case, the actual operation is attached to the preceding one. The reason for considering only the previous sibling is simple. As the leaves are traversed in document order, the following sibling cannot be handled by the algorithm, except if it has been already parsed in the step covering structure-preserving changes. In that case, however, the edit operations cannot be merged, as only edit operations of equal type can be glued together.

7.4.4 Move Operations

Move operations consist of a delete operation and a congruent insert operation. Both edit operations are created independently. However, the algorithm keeps a hash map containing the hash values of the subtrees targeted by the already created edit operations. During the creation of a structure-affecting edit operation, a lookup in the hash-map shows whether a congruent edit operation already exists. In that case, both edit operations are linked with an ID attribute. As value of the ID attribute, the system time in nano seconds is chosen. This time stamp is fast to compute and ensures an unique identification of the corresponding operations.

7.4.5 The Algorithm

The third step of the differencing algorithm starts by identifying the insert operations. Each leaf that is part of the modified document but has no matching counterpart in the original document is considered as inserted. Before starting the bottom-up pass, the algorithm checks whether a leaf update has been performed. The corresponding code is shown in Algorithm 3.

After handling the insert and update operations, the delete operations are identified. The code is shown in Algorithm 4. The code is nearly the same as in the bottom-up pass for insert operations. Basically, the identification of leaf updates can be performed during the handling of insert or delete operations as well. However, it must be performed during the run which starts first. Otherwise, the possible counterparts of updated nodes would have been already covered by the edit operations of the first run.

Finally, the algorithm handles the move operations, as shown in Algorithm 5. Here, the hash table lookup is explicitly named. In general, hash tables are widely used in the implementation of the algorithm by Philipp [2009]. They are mostly used to efficiently compare nodes.

7.5 Complexity Analysis

In this section, I analyze the cost of comparing two document versions A_1 and A_2. This covers the time complexity and the space complexity as well, which are presented after elaborating

Algorithm 3 The third step of the algorithm starts by identifying update operations on leaves. Additionally, insert operations are identified.

```
for all l2 ∈ leaves(A2) : l2 ∉ matchList do
  {identification of updates on leaves}
  if (x1, parent(l2)) ∈ matchList or parent(l2) is part of an update operation on x1 then
    for all l1 ∈ children(x1) : (l1, x2) ∉ matchList and l1 is not part of an update operation
    do
      if previousMatchedSibling(l1) = previousMatchedSibling(l2) and
      nextMatchedSibling(l1) = nextMatchedSibling(l2) then
        add (update, path(l1), value(l1), value(l2)) to δ_{A1→A2}
        break
      end if
    end for
    if no update operation has been created then
      add (insert, path(l2), ∅, l2) to δ_{A1→A2}
    end if
  else
    {bottom-up pass including gluing}
    for all a2 ∈ ancestors(l2) do
      if (x1, parent(a2)) has already been visited then
        if previousSibling is root of an insert operation then
          append subtree(a2) to insert operation
        else
          add (insert, path(a2), ∅, subtree(a2)) to δ_{A1→A2}
        end if
      end if
    end for
  end if
end for
```

the complexity of the three steps of the algorithm separately.

7.5.1 Computing the LCS

The computation of the longest common subsequence of leaf nodes is the far most complex step of the algorithm, as I expect over 50% of the nodes being leaves (see Section 2.4.2). The LCS algorithm has to be robust against large input alphabets. In context of the differencing algorithm, the nodes are compared using their value. Therefore,the size of the input alphabet is bounded by the amount of possible hash values, which is a significant parameter.

The current implementation of the differencing algorithm relies on the LCS algorithm presented by Myers [1986]. It is known for its broad applicability and computes a minimum

Algorithm 4 Delete operations are identified during a bottom-up pass.

for all $l_1 \in leaves(A_1) : l_1 \notin matchList$ **do**
  for all $a_1 \in l_1 \cup ancestors(l_1)$ **do**
    if $(parent(a_1), x_2)$ has already been visited **then**
      if *previousSibling* is root of a delete operation **then**
        append $subtree(a_1)$ to delete operation
      else
        add $(delete, path(a_1), subtree(a_1), \emptyset)$ to $\delta_{A_1 \to A_2}$
      end if
    end if
  end for
end for

Algorithm 5 The third step of the algorithm ends by linking moved parts of the document using an ID attribute.

for all $i \in \delta_{A_1 \to A_2} : i$ is insert operation **do**
  if $\exists d \in \delta_{A_1 \to A_2} : hash(i) = hash(j), i \neq j$ **then**
    add *move* flag to i, d, with `ID = systemNanoTime`
  end if
end for

edit script [Bergroth et al. 2000]. One of the main advantages of this algorithm is the lack of assumptions about the sequences to be compared. The LCS algorithm runs in $O(ND)$ time. Here, N denotes the total number of elements to compare, D represents the number of edit operations. Apparently, this algorithm is optimized for a use case where $N \gg D$. At this point the assumption on the modification amount should be recalled, elaborated in Section 2.5.3:

Assumption 2.6: A new version of a document will likely have only few changes compared to the size of the document.

Therefore, the LCS algorithm is suitable for the intended use case, as it runs in almost linear time for similar documents.

In a previous version of my algorithm, common prefixes and suffixes were explicitly removed to restrain the complexity of the LCS run. This idea was inspired by the thoughts of Fraser [2009]. However, the LCS algorithm by Myers [1986] already contains a prefix and suffix detection. Therefore, this step is omitted in the algorithm presented here. Experimental results enforced this decision, showing an increase of runtime when performing an explicit prefix and suffix detection in context of the diff algorithm itself.

Mapping the complexity of the LCS algorithm to the diff algorithm yields a time complexity of $O((|leaves(A_1)| + |leaves(A_2)|) \times D$. In this context, D denotes the size of the minimum edit script between both leaf sequences.

7.5.2 Identification of Structure-Preserving Changes

The second step of the algorithm is performed only for the leaves that have already been matched by the LCS run in the previous step. Here, each ancestor node is visited once due to the dynamic programming approach. Therefore, the bottom-up traversal for the detection of update operations runs in linear time, namely $O(|ancestors(LCS)|)$.

7.5.3 Identification of Structure-Affecting-Changes

Constructing the insert and delete operations creates a bottom-up traversal for each of the document versions. This step of the algorithm is performed only on the leaves that are not part of the leaf LCS computed in the first step. Here, each node is visited once, too. However, the leaves must also be traversed. Basically, this results in a linear time complexity, too.

Each node that is not part of the longest common subsequence is represented in the minimum edit script which can be computed by the LCS algorithm. In the first step of the differencing algorithm, D was used to denote the size of the minimum edit script. Therefore, the amount of non-matching leaves corresponds to D. The resulting time complexity can be denoted as $O(D+|ancestors(\text{non-}LCS)|)$.

One might argue that the search for leaf updates increases the complexity of the second step of the algorithm. The leaf update detection is only started if the parent node has been already visited, which means that a sibling of the leaf has already been matched. The document analysis has shown that on bottom-up level 1, the quantity of nodes reaches half of the quantity of leaves. Deriving from that, I deduce that most leaf nodes do not have many siblings. Therefore, the search for already matched siblings is less complex as the rest of the algorithm by order of magnitude. By this, the leaf update detection can be neglected in the complexity class. I will underline these considerations empirically in Section 9.3.1.

7.5.4 Overall Time Complexity

Adding the complexity bounds of all steps of the differencing algorithm yields $O((|leaves(A_1)|+|leaves(A_2)|)\times D+|ancestors(LCS)|+D+|ancestors(\text{non-}LCS)|)$. Both items covering the bottom-up passes can easily be subsumed as $|ancestors(A_1)|+|ancestors(A_2)|$. Here, I use the notion A, indicating the sum of both trees A_1, A_2 to compare. This leads to the simplified equation $O(leaves(A)\times D+|ancestors(A)|+D)$.

7.5.5 Overall Space Complexity

The space complexity of the differencing algorithm is linear, I claim an upper bound of $O(A)$. I justify this by following considerations: the chosen LCS algorithms runs in linear space, yielding $O(|leaves(A)|)$ [Myers 1986]. During the bottom-up pass, additional space is required to compare all nodes, thus leading to $O(A)$.

Additionally, each node is stored either in the match list or in the delta. Therefore, $O(A)$ has to be added to the complexity consideration again. However, all three steps are in the same complexity class that can be subsumed under $O(A)$.

7.6 Conclusions

In this chapter, I have presented a novel XML differencing algorithm. The algorithm design is directly derived from the analysis of XML documents, which has been performed in Chapter 2. As the content of the documents is stored in the leaves, which represent over 50% of all nodes, the algorithm starts by computing the longest common subsequence of the leaves. Changes on higher levels of the tree are identified using a bottom-up approach.

The changes between the document versions are represented using the delta model presented in Chapter 6. By relying on a well-established LCS algorithm and using dynamic programming techniques, the proposed algorithm achieves a highly competitive time and space complexity. The time complexity is $O(|leaves(A)| \times D + |ancestors(A)| + D)$, with D denoting the amount of changes. Therefore, the algorithm is especially suitable for documents with $A \gg D$, which meets the expected average case. The space complexity is linear with $O(A)$. This is an important aspect, as XML documents can become quite large.

In conclusion, the proposed algorithm promises a highly efficient way of comparing XML files, using a change representation model especially suitable for XML documents.

8 A Merge-Capable Patch Algorithm

In Chapter 6, I have introduced a context-aware delta model. There, some aspects of merging of document versions have already been discussed. This chapter deals with the details of delta application during the patching of a document. The hardest challenge is to reliably merge document versions. This requires the patch tool to be flexible enough to accept non-optimal results, still being strict enough to reject ambiguous results. Obviously, this is a borderline task.

In this chapter, I present XCC Patch, a patching algorithm that is able to merge document versions using a context delta. The basic idea of the algorithm is sketched first. Afterwards, I present the aspects that conform to conventional XML patch tools. In the following, I explain the use of the context fingerprint for a reliable identification of the correct path to an edit operation, before presenting the conflict handling abilities of my algorithm. A weighting method for partially matching fingerprints allows for non-optimal yet reliable merge results. Finally, I present the algorithm at a glance, followed by a complexity analysis.

8.1 Basic Idea

Conventional XML patch tools patch a document as described in the delta. This means that all edit operations are entirely applied at their respective paths in order of their appearance. Of course, this is the standard case in a sequential document evolution model. However, parallel editing of documents cannot be represented by this procedure. Due to a lack of a suitable delta model, other XML patch algorithms are not able to merge document versions using a delta.

To allow for the merging of document versions, I adopt the basic idea from conventional patch tools for line-based documents to the domain of XML documents. In the line-based domain, the patch algorithm starts at the given path of an edit operation and uses the syntactic context stored in the delta to verify whether the context at the document to patch is still the same [Davison 1990]. If not, the patch algorithm tries other paths nearby to search for the correct context. Beside the context, the delta contains also the former value of a line to change. This former value is used to verify whether the line has been edited in the meanwhile, thus leading to a conflict.

My patch algorithm uses the context fingerprint defined in Section 6.5.1 to search for the correct path of an edit operation and to detect conflicts as well. In contrast to one-dimensional line-based documents, XML trees are two-dimensional. Therefore, I will introduce a corresponding tree traversal algorithm to search for the correct path.

During the merging of line-based documents, non-optimal results may be accepted. The

user may decide how many lines of the context have to match in minimum, before an edit operation is applied [MacKenzie et al. 2002]. In this approach, each line of the context is given equal weight. This behavior was inspired by the needs of software source control. Source code is highly structured. The syntactic appearance, e.g., the indentation, is used to represent the structure. Most lines contain only few code, blank lines are widely used.

In XML documents, however, this syntactic structure is not given. The leaves contain the content which in turn should not contain structuring information, as the layout is represented by the tree structure [Goldfarb 1990]. The context fingerprint contains the surrounding nodes in document order, which means that it may contain leaves and inner nodes as well. Especially if changes are clustered, the fingerprints of different edit operations may overlap. If all entries of the fingerprint would be weighted equally, the probability of accepting an edit operation would decrease significantly in that case, even if a human reader would accept it. To handle this issue, I introduce a distance-aware weighting function for the entries of the context fingerprint. An entry near the anchor, i.e., near the edit operation itself is given a higher weight than an entry far away. This allows for a reliable merge without losing a broad applicability.

8.2 Linear Patching

Patch algorithms in XML are seldom described in detail. This derives from the fact that they are only able to apply a delta to the document it has been computed for. If a delta is applied to a modified version of the document, all interfering edit operations will fail (see Section 6.3.1). If the path is still existent, the patch will most probably apply the edit operation at the wrong path. If the path has become invalid, an error will be thrown. To prevent these failures, Microsoft XML Diff (presented in Section 3.3.2) stores a hash signature of the whole document to patch within the delta. It is used to verify whether the delta is applied to the correct document version. If not, the patch aborts with an error message.

XCC Patch is able to act like a conventional XML patch tool, too. This is an important aspect in the sequential reconstruction of document versions, for example in the context of version control systems [Tichy 1985]. Each edit operation is applied as described in the delta.

To be precise, the algorithm actually does not perform the edit operation at the time of the processing. Instead, only a reference is stored to the corresponding path, and the edit operation is added to a list. Two reasons exist for that behavior. First, the delta is a set, edit operations may be in arbitrary order. All paths are described with respect to the original version of the document. If the edit operation would be applied, other interfering edit operations of the delta would point to wrong paths, resulting in errors. This directly leads to the second reason for applying all edit operations at once. Ko and Lee [2006] have stressed the high cost of the re-labeling of XML paths. Each structure-affecting change would require the following nodes to be re-labeled in terms of their path. This re-labeling is switched off before applying all edit operations at the end of the algorithm, as the stored reference is used to access the edit operation.

8.3 Context-Aware Patching

In case that a delta is applied to a modified document version, some of the edit operations in the delta will likely point to wrong paths of the document due to interfering changes resulting from previous edit processes. The patch algorithm uses the context fingerprint to verify whether the given path of an edit operation still points to the correct position within the XML tree. If not, the neighborhood of the given path is searched for the correct position. In the end, I will discuss the suitability of the defined neighborhood for the given use-case.

8.3.1 Matching the Context Fingerprint

Each edit operation within a delta contains a context fingerprint, as introduced in Section 6.5.1. It contains the hash values of the surrounding nodes that have been around the part targeted by the edit operation in the document version for which the delta has been computed. Before applying an edit operation, the patch algorithm verifies whether the context fingerprint matches the syntactic context found at the given path in the document to patch.

Whether the context fingerprint matches is verified by computing the fingerprint of the given path. Each entry of the fingerprint has to match the corresponding nodes of the document to patch. For delete and update operations, the fingerprint has corresponding nodes for each entry of the fingerprint. For insert operations, however, the anchor points to the root of the subtree to insert, not having a counterpart node in the document to patch. Therefore, I define the environment I, containing the distances of the counterpart nodes depending on the edit operation type.

$$I \stackrel{\text{def}}{=} \begin{cases} [-r,..,-1,1,..,r] & \text{for insert operations} \\ [-r,..,r] & \text{otherwise} \end{cases} \tag{8.1}$$

Definition 8.1 *A fingerprint $f \in \delta$ matches a path p in document A, if and only if $\forall n \in I$: $f[n] = \mathit{fingerprint}_p[n]$.*

Before computing the matching of a fingerprint, the type of the edit operation has to be taken into account. At this point, recall the definition of $A_{\mathit{fingerprint}}$ in Equation 6.10, which ensures that the tail of a subtree or tree sequence to delete is not contained in the context fingerprint (see Section 6.5.1). This way, the fingerprint does not refer to itself. During patching, this behavior has to be taken into account.

Figure 8.1 shows an example. The context fingerprint for the deletion of the subtree rooted at `f` would be `d,e,f,null,null`. Computing the fingerprint around node `f` without the knowledge that its children should be deleted would in turn result in the fingerprint `d,e,f,g,h`. Apparently, the patch algorithm would reject the application of that edit operation, as two of the entries do not match. Therefore, for any delete operation, the patch algorithm has to neglect the nodes that correspond to the nodes targeted by the edit operation.

During this step of the algorithm, neither the content of the nodes nor their structure are compared. Non-matching nodes would lead to a conflict, which will be handled in Section 8.4.

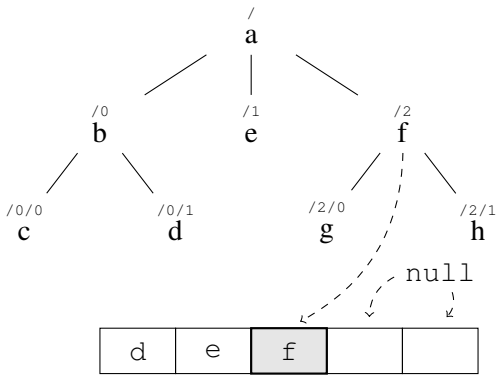


Figure 8.1: The context fingerprint of a delete operation does not refer to the deleted parts. This has to be taken into account during patching by computing the matching fingerprint on the document to patch according to the delete pattern defined in the edit operation.

Instead, the patch algorithm looks up the number of subsequent subtrees contained in the delete operation and hides the same number starting at the targeted path to the fingerprint generation. This is a simple and efficient way of respecting the properties of delete operations. Other operation types do not require such a special handling.

8.3.2 Neighborhood Search

Assuming that the delta is applied to a modified version of the document, prior changes (not known to the patch algorithm) may interfere with the edit operations to apply, resulting in invalid paths. Nevertheless, a reliable merge is still possible. Here, I recall an assumption on the change distribution across a new version of a document, elaborated in Section 2.5.3:

Assumption 2.7: Changes on a document affect only small parts of the document in general.

Additionally, assuming that the authors are aware of their responsibilities and roles, the edit actions will be separated locally from each other [Shen and Sun 2002]. Deriving from these assumptions, the context of an edit operation has only moved with respect to the path stored in the edit operation.

To find the correct application context of an edit operation, I define a neighborhood around the stored path, containing candidate nodes that are considered to be potentially the correct path.

Definition 8.2 *The neighborhood of a path p is a sequence of $2\rho + 1$ nodes on the same top-down level around p in each direction, ordered by their distance to p. The node at position 0 is the node at p. ρ is the radius of the neighborhood.*

In case that the context fingerprint does not match at the stored path, the matching for all nodes of the neighborhood is computed, too. The matching is computed in order of the distance

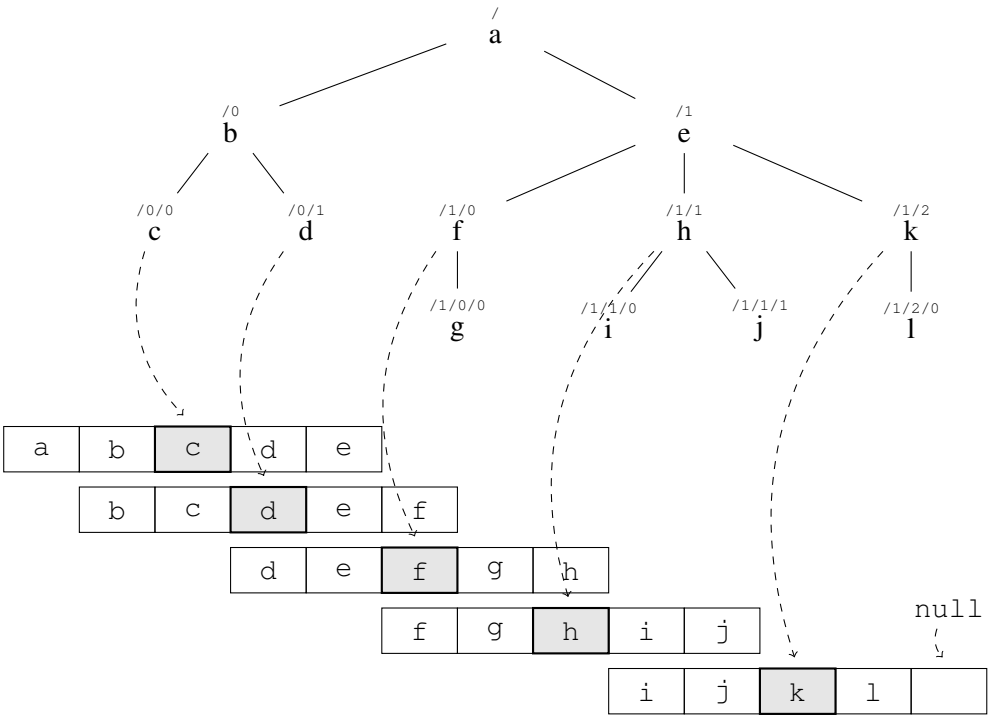


Figure 8.2: The neighborhood with radius 3 around node `/1/0`.

between the candidate node and the path of the edit operation. The reason for this ordering is simple. The patch algorithm does not have any knowledge whether the prior edit operations have been deletions or insertions. Therefore, it has to walk in both directions. Using my ordering, small changes resulting in small offsets are handled before large changes.

Figure 8.2 shows the neighborhood of node `/1/0` with radius of 3. The fingerprints are computed in following order: `f`, `d`, `h`, `c`, `k`.

It is possible that an edit operation has no matching context anymore. This mostly appears if parts of the context (or the anchor node itself) have been modified previously. In that case, the patch algorithm has to reject the edit operation or to perform a best-effort approach (see Section 8.5). The neighborhood has been defined to prevent the patch algorithm from performing an exhaustive search across all nodes of the document, searching for a non-existent context. Additionally, Khanna et al. [2007] have shown that spurious changes may occur if the merge algorithm finds a matching context far away from its original path. As the document analysis in Section 2.4.3 has revealed that the ratio of repeating nodes is relatively high across all levels of the XML tree, I cannot neglect the possibility of a second matching context across the whole document. This is an argument for a restrained search space, too. In case of a small neighborhood, I assume that the probability of identical fingerprints can be neglected.

8.3.3 Discussion

The neighborhood of a node contains only nodes of the same top-down level. This is in fact a noticeable restriction. It is motivated from the following consideration: the top-down level of a node also reflects its position in the XML grammar used for validation. A node on top-down level 7 can be accessed using 7 resolution steps in the corresponding grammar. It cannot be ensured that it can also be accessed by 6 or 8 steps. Examples exist for this possibility. However, I do not assume these examples to be the standard case. One might argue that the context fingerprint would hinder the patch algorithm to apply the edit operation at the wrong path. Although this is correct, I cannot justify the overhead of computing the unnecessary matchings for finding only few possible positive matches. Additionally, a much higher neighborhood radius would be required to get the same width within the XML tree as with the proposed definition.

Apart from that, domain-specific neighborhood generation rules are considerable. For example, within a spreadsheet, only cells on the same column could be taken into account. As those domain-specific rules would prevent my approach from being generally applicable, I do not follow this approach.

In case that the patch algorithm finds the correct path of an edit operation, the resulting offset could be stored. This offset could be added to all edit operations within the delta that interfere with the matched operation. Basically, this is a promising idea. Nevertheless, I do not follow it as the delta is basically a set. Implementing this feature would require to order all edit operations with respect to their interferences before starting a patch run. I estimate the gain of efficiency during the patch run lower than the loss of efficiency due to the prior ordering of edit operations, which must be performed to ensure the correct ordering, even if the edit operations are already ordered.

8.4 Conflicts

An edit operation might target a part of the document that has been edited before. In case that the nodes stored in the edit operation do not match the pattern found in the document to patch, a conflict arises. A conflict is also given if two edit operations from different deltas are overlapping.

8.4.1 Definition

Formally, an edit operation conflicts if one of the nodes of v do not match. Here, one should remember that v contains the previous value of the part affected by an edit operation.

Definition 8.3 *A node $n \in v$ of $op \in \delta$ matches an path $p \in A$, if and only if $value(n) = value(p + path(n))$.*

The matching is defined with respect to the targeted path of the edit operation by adding the offset of the path within v.

Definition 8.4 *An edit operation $op \in \delta$ conflicts for a path $p \in A$, if and only if $\exists n \in v$ of op : n does not match p.*

For insert operations, v is empty, as an insert does not affect the value of an existing node. Therefore, only delete and update operations can cause conflicts.

8.4.2 Conflict Handling

For update operations, a conflict can directly be derived from the matching of the anchor, which is already performed in the previous step. For a delete operation, a matching anchor indicates that the head of the subtree or tree sequence is conflict-free. The remaining parts of the delete operation are compared using the recursively computed hash over the subtrees. In case that the hash values match, the operation is conflict-free, too.

Two possibilities exist to handle a conflict automatically: applying or rejecting the edit operation. In terms of version control systems or operational transformation, the application of the edit operation would correspond to a client-side, the rejection to a server-side preference [Shen and Sun 2002]. In the current implementation, the user may decide which version to keep by setting a flag before the patch run.

8.4.3 Discussion

Another possibility of conflict handling is to reject the whole delta until no conflict arises [MacKenzie et al. 2002]. This forces the user to resolve a conflict manually, preventing an accidental corruption of the document. My algorithm does not support that behavior for two reasons. First, this behavior is mostly intended for software source control, where a delta is assumed to have comprehensive changes on one functionality. There, any rejected edit operation will likely lead to a malfunction of the code in case that the other edit operations of the delta have been applied. In the domain of document editing, however, it is common that only parts of an editing step are applied to the document, e.g., two authors correcting the same misspelling. A second reason is derived from the set property of the delta. Patching algorithms for edit script-based deltas use a subsequent addressing of edit operations. Any rejected edit operation would distract the following edit operations by the missing path offset. Due to this, a complete application or rejection of a delta is preferred in these approaches. As my delta is set-based, all edit operations but the conflicting ones can be applied, whereas the conflicting edit operations could be inspected independently, and applied one after another afterwards.

During the merging of document versions, the user could be prompted whether a conflicting edit operation shall be applied or not [Ignat and Norrie 2006]. This interactive decision is generally favorable over automatic rule-based solutions. However, this requires the user to assist each merge run. I will present an interactive editor for XML documents in Section 11.3.1, which allows for a detailed manual resolution of conflicts.

According to my definition, an insert operation cannot conflict. This directly follows from the fact that an insert does not overwrite any existing node. Nevertheless, an insert operation

may be not meaningful, e.g. because it has been applied before. In these cases, however, the context will most likely be different, thus resulting in a rejection of the insert. I assume the probability of an erroneous insert operation with identical context to be in neglectful magnitude.

8.5 Best-Effort Merging

An edit operation is rejected if one of the entries does not match the given path. However, a user would have probably decided to apply the edit operation, as all other entries did match. To allow for a sub-optimal merge, I introduce partial matchings. In these partial matchings, a node near the anchor is given a higher importance using a distance-aware weighting function that estimates the match quality. Whether an edit operation is applied or not is decided upon a user-defined threshold for the match quality.

8.5.1 Partial Matching

The context fingerprint offers a possibility to identify the correct context of an edit operation. To increase the reliability of the merge result, a user could try to increase the fingerprint radius to store a larger fingerprint. Although this is possible, another problem arises. The larger the fingerprint radius, the higher the probability of erroneously rejected edit operations due to other changes in the surroundings of the context. A partial matching of a fingerprint denotes that there exist matching entries, as well as non-matching ones.

Definition 8.5 *A partial matching of a fingerprint $f \in \delta$ on path p in document A is given, if and only if* $\exists n \in [-r..r] : f[n] = \mathit{fingerprint}_p[n] \wedge \exists n \in [-r..r] : f[n] \neq \mathit{fingerprint}_p[n]$.

8.5.2 Distance-Aware Weighting

The quality of a partial matching could be estimated by dividing the amount of matching nodes by the amount of nodes in total. This would reflect the behavior of the line-based differencing tools for source code, where each line is considered to be equally important [MacKenzie et al. 2002]. In the domain of source code, blank lines are commonly used to structure the code. Assuming that an edit operation contains a new code block, it is likely that an operation is delimited by blank lines. Due to their quantity, blank lines do not offer a high reliability to judge about the correct context of an operation. Therefore, the more remote lines are assigned with the same priority as the lines near the designated path of the operation.

One of the advantages of XML is the separation of content and markup. Therefore, "structuring content" like blank lines is mostly obsolete. One might argue that in text documents, blank lines are used to structure the text, too. First, this is a kind of bad document layouting. Second, in the XML representation of office documents, a blank line is stored using at least two nodes. That is, the paragraph node and a special "empty text" node [Brauer et al. 2007]. Therefore, I do not consider the structuring content as notable.

Especially for large context fingerprints, the probability for a non-matching entry at the borders of the fingerprint becomes significant. In my opinion, a non-matching entry near the anchor of an edit operation will more likely indicate an ambiguous merge than a non-matching entry with higher distance. I will exemplify my considerations. In a text document, each text node contains a paragraph or another rather large portion of text. Each paragraph has a corresponding parent node, denoting the type and the markup of the text. An update on a paragraph with a fingerprint radius of 6 would cover the surrounding three paragraphs including their parent nodes. I assume a text change three paragraphs before less important than on the previous paragraph. Therefore, I introduce a function that weights the matchings of a fingerprint entry depending on its distance to the anchor. Here, I introduce a more formal refinement of the matching of entries of a fingerprint f with respect to a path p (see Definition 8.1 in Section 8.3.1).

$$match(f[n], p) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } f[n] = fingerprint_p[n] \\ 0 & \text{otherwise} \end{cases} \tag{8.2}$$

Once again, the type of an edit operation plays an important role. For insert operations, the anchor of the fingerprint points to the document to insert, which has no counterpart in the document to patch. Therefore, the environment I is used again, which has been defined in Equation 8.1.

Basing on the match function, the quality of a partial matching of f on p can be computed using the weighting function $matchQuality(\delta, A) \rightarrow [0;1]$:

$$matchQuality(f, p) \stackrel{\text{def}}{=} \frac{\sum_{i \in I} \frac{match(f[i], p)}{2^{|i|}}}{\sum_{i \in I} \frac{1}{2^{|i|}}} \tag{8.3}$$

The numerator counts all matches, whereas the denominator normalizes the result onto a value between 0 and 1. A quality of 0 indicates no matchings, 1 means a complete match.

Figure 8.3 shows the appearance of the un-normalized weighting function, which has a bell-shaped figure. The anchor node is given the highest importance, directly followed by the adjacent nodes[1]. Table 8.1 shows the match quality for an update operation, where one of the entries in the fingerprint of radius 3 does not match.

8.5.3 Merging

If a fingerprint does not match completely at its given path, the neighborhood search starts. At this step, the match quality of each candidate is already computed. If a complete matching does not exist for any of the candidate nodes, their match quality is compared. The algorithm applies the edit operation on the candidate with the highest match quality.

[1] For insert operations, the anchor is not considered. Here, the adjacent nodes have the most influence on the weight.

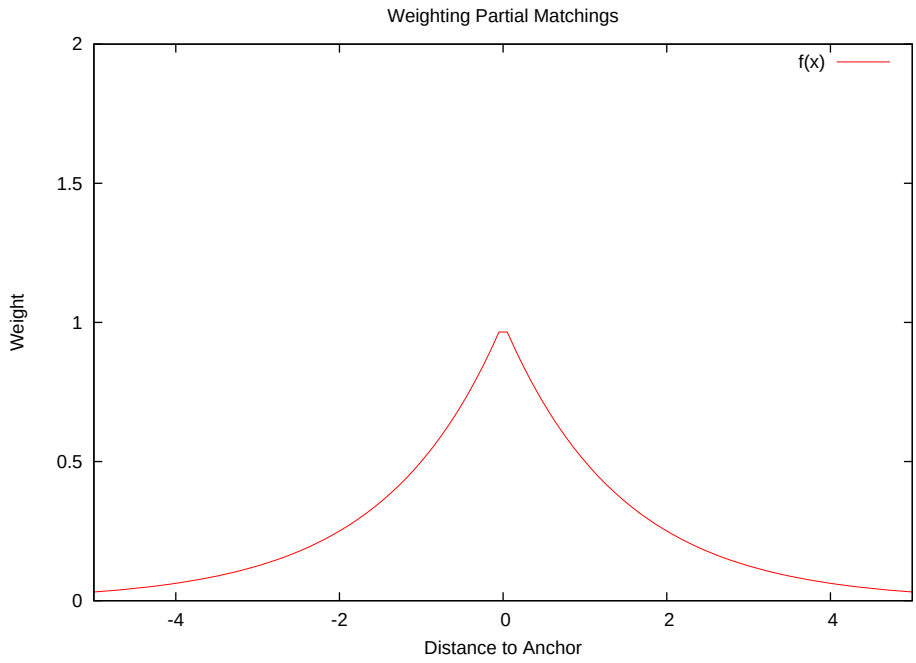


Figure 8.3: The weigthing function shows a bell-shaped figure, highly favoring the nodes next to the anchor.

i	*matchQuality* if $match(f[i], p) = 0$
0	0.636
1	0.818
2	0.909
3	0.955

Table 8.1: The match quality for a fingerprint of an update operation with $r = 3$ for one non-matching node with distance i to the anchor.

To avoid erroneous merges, the user has to define a *threshold value* which will be used as lower bound for accepting a computed match quality. This threshold allows for easily defining to which degree the user accepts ambiguities in the merge result. A threshold of 1 means that only complete matchings are accepted, a threshold of 0 accepts any edit operation, even if it has no matching.

8.5.4 Discussion

The proposed weighting function shows a strong digression depending on the distance to the anchor. A fingerprint radius larger than 4 would not influence the partial matching signifi-

cantly. For example, a Gauss error distribution curve could be used with corresponding parameters for a smoother decrease, or a linear digression. My motivation is to foster the use of small fingerprints. First, a fingerprint means a storage overhead in the delta. Especially for update operations, the fingerprint can easily be larger than the rest of the edit operation. In the line-based domain, 2 to 3 lines are common context radii, too [MacKenzie et al. 2002]. Second, a larger fingerprint increases the probability for partial matchings. A user might intend to use a large fingerprint to get a higher reliability of the merge result. However, the merge run would likely lead to a partial matching resulting from changes far away, making the user feel insecure about the merge result.

The proposed matching function is a binary one. A fingerprint entry matches or it does not. Already small changes in surrounding nodes lead to non-matching entries, probably forcing the patch algorithm to reject an edit operation. This is mostly important for attribute changes or minor changes in text nodes. Here, the matching function could be used to express the similarity between the fingerprint entry and the node. One might argue that the fingerprint contains only a hash value that completely changes even for small changes. This behavior is mostly intended by the needs of cryptography. Recently, Stein [2005] has presented a new class of hashing functions that is able to express the similarity of texts. These functions could be used to allow for an even more meaningful and fine-grained merge. I will discuss this issue in Section 11.4.3.

8.6 The Algorithm at a Glance

The proposed algorithm can be controlled via different parameters. The most important are the *neighborhood radius* ρ and the *threshold* for partial matchings. Additionally, the patch tool can run in different modes. In the first mode, it acts like a conventional patch tool, which I called linear patching earlier. In addition, this mode also supports conflict detection, which can be disabled by the user. The code is shown in Algorithm 6. Here, the context of a node is not respected. Applying a delta to a modified version of the document will most likely lead to rejected edit operations.

Algorithm 6 The algorithm for linear patching with conflict detection.

```
for all op ∈ δ do
  if op does not conflict then
    add op to accepted operations
  end if
end for
```

The second mode respects the context of an edit operation, as well as its neighborhood. Algorithm 7 shows the code. Here, only complete matchings of the fingerprint are accepted. Conflict detection is switched off. This mode mostly corresponds to the behavior of line-based patch tools.

Algorithm 7 The complete merge-capable algorithm with neighborhood search.

```
for all op ∈ δ do
  if matchQuality(fingerprint_op, path_op) = 1.0 then
    add op to accepted operations
  else
    for all n ∈ [−ρ..−1; 1..ρ] do
      if matchQuality(fingerprint_op, path_op) = 1.0 then
        add op to accepted operations
        break
      end if
    end for
  end if
end for
```

Finally, the third mode allows for best-effort merging, including conflict detection. This is the default mode of the patch tool, shown in Algorithm 8. Even if designed for merging, the algorithm checks whether the operation can be applied as it is. This design decision was motivated by the thought that users will most probably use the tool without switching to the linear patching mode, even if they are applying the delta to the correct document version. The proposed design ensures that the performance in this case is not significantly slower than in the first mode of the tool.

Algorithm 8 The complete algorithm in best-effort mode with conflict detection.

```
for all op ∈ δ do
  if matchQuality(fingerprint_op, path_op) = 1.0 ∧ op does not conflict then
    add op to accepted operations
  else
    for all n ∈ [−ρ..−1; 1..ρ] do
      if matchQuality(fingerprint_op, path_op) > threshold ∧ op does not conflict then
        add op to candidates
      end if
    end for
    add candidate with highest quality to accepted operations
  end if
end for
```

In the proposed code, all edit operations are added to a list of accepted operations. As described earlier in Section 8.2, this means that a reference to the corresponding path of the edit operation is stored. The edit operations are applied altogether at the end of the patch run. This is done to prevent erroneous merge results due to previous edit operations with overlapping fingerprints and to avoid the re-computation of paths.

8.7 Complexity Analysis

The proposed patch algorithm can operate in different modes, which have a different complexity. Apparently, the linear patching is the simplest, whereas the best-effort merge is the most complicated. In this section, I present an analysis of the time and the space complexity of the algorithm.

8.7.1 Time Complexity

The algorithm for linear patching is very simple, yielding $O(A \times \delta)$, where δ denotes the amount of edit operations within the delta. Conflict detection would add more complexity. I will discuss that issue later.

Respecting the context fingerprint during patching adds more complexity, too, depending on the fingerprint radius. The resulting time complexity yields $O(A \times \delta \times r)$. In general, the assumption $A \gg r$ holds, except for very small documents. As r remains constant, it can be neglected, thus achieving $O(A \times \delta)$.

Comparing the fingerprint is really simple. One might argue whether the computation of the hash values has to be respected in the complexity analysis, too. The current implementation uses a wrapper class, ensuring that each hash value is computed only once (see Section 5.2.2). Therefore, I do not consider the hash computation separately. The proposed design is especially helpful for the neighborhood search, where each node is accessed multiple times.

The neighborhood search is time consuming. Depending on its radius, the worst-case time complexity increases to $O(A \times \delta \times \rho)$. The complexity for the general case will be better, as the algorithm stops if the correct node is found. As this would not lower the complexity class significantly, I do not distinguish between the different cases. The best-effort merge does not consume more time, as the algorithm structure is the same. Throughout all edit operations, ρ remains constant. Therefore, ρ can be regarded as constant factor, which leads – again – to the complexity class of $O(A \times \delta)$.

For the conflict detection, it is not sufficient to use the context fingerprint. Each node targeted by a delete operation has to be inspected. If the conflict detection is switched on, the previously named complexity classes have to extended by $O(nodes(op_{delete}))$. However, this holds only for very large delete operations, where $A \gg nodes(op_{delete})$ is not valid. Therefore, the overhead for the conflict detection has not to be considered in the average case.

8.7.2 Space Complexity

The algorithm loads the complete document to patch into main memory. Additionally, the delta is loaded. This leads to a space complexity of $O(A + \delta)$. One might argue that the edit operations are interim stored, thus leading to a higher space complexity. However, the list does only contain references to the correct paths of the operations, which makes it comparably small. Additionally, the list is covered by the second parameter of the complexity class, as the amount of edit operations is the upper bound of that list.

8.8 Conclusions

Applying a delta to a different document version is a challenging task. Paths of edit operations may point to wrong addresses due to prior changes. The context of an edit operation might have been altered. Even the nodes addressed by the edit operation might have changed, leading to a conflict. A patch algorithm has to respect all of these issues to offer a reliable merge result.

In this chapter, I have presented a merge-capable patch algorithm for XML documents. It supports three operating modes. First, a delta can be applied as described in the delta, offering no merge support. This mode is fast and intended for sequentially edited documents. It reflects the behavior of most other XML patching approaches. In the second mode, offsets due to prior edit operations are respected. Around the path of an edit operation, the context fingerprint is used to find the correct position for that change. The third mode also accepts sub-optimal matchings. Here, a weighting function is used to estimate the quality of a partially matching context fingerprint. The user decides to which degree a non-optimal merge result is acceptable.

Across all three modes, conflicts can be detected. The user has to decide whether conflicting parts of the document are meant to be overwritten, or if the previous version has to be kept unchanged. This decision is made for all conflicting edit operations within a patch run. I present a graphical user-interface allowing for interactive decisions on conflicts in Section 11.3.1.

9 Evaluation

Theoretical considerations are the basis of any new algorithm. Even if these considerations are based on empiric findings, they should be verified whether they hold in practice. To dos so, I evaluate the proposed XCC framework in this chapter.

First, I describe the experimental setup and the test scenarios used. Afterwards, I analyze the runtime of the diff and the patch algorithm. Finally, I verify the merge-capabilities of the patch algorithm. Here, the focus lies on the reliability of the merge result.

9.1 Experimental Setup

The experiments are carried out on a commercial-of-the-shelf machine, which is briefly described first. Afterwards, I present the test methodology for measuring the runtime. Finally, I show a notation for measuring the merge quality of documents.

9.1.1 Test Environment

All tests are performed on a machine equipped with an Intel Core Duo T2600 processor with 2.16 GHz and 2 GB RAM. The operating system is Linux, kernel 2.6.32 (32bit). The Java environment is OpenJDK 1.6.0_18.

9.1.2 Methodology

The measured runtime is the time elapsed between the start of the differencing or patching algorithm and the finished write of the delta or patched document. In this context, start of the algorithm actually refers to the execution of the first command within the corresponding tool. This methodology has some implications. First, the parsing of the documents is measured, too. This is important, as the parsing has to be performed at every run of the algorithm. Additionally, the XCC frameworks does some initialization during start-up that would not be tracked otherwise. Second, the overhead for loading the Java Runtime Environment (JRE) is not part of the measured runtime. This is reasoned by the fact that this overhead cannot be omitted or affected by XCC. Additionally, the load time highly depends on previously loaded JRE instances that still might be in the cache of the operating system.

Document parsing takes a significant part in the overall runtime. The runtime mainly depends on the document size, as each byte has to be parsed. In a first test, I did order the documents with respect to their node amount. This has lead to misleading test results. The

total size appears to be a better indicator for the complexity of a document than the quantity of nodes.

In each scenario, each step is computed three times. The displayed runtime shows the average of this three runs. This shall lower the influence of the operating system and the hardware. For example, the cache of the CPU or the hard disk may contain parts of the documents to compare or the program code. A cache hit decreases the load time by several milliseconds. On the other hand, the scheduler of the operating system may prioritize other processes. All tests have been performed in the Single User Mode of Linux to prevent other processes to disturb the evaluation. Nevertheless, some processes (e.g., Syslog) are still present and have a higher priority by design, which could affect the runtime.

9.1.3 Measuring the Merge Quality

As already explained in Section 2.2.2, there may exist different ways of syntactically representing a semantically equivalent document. Additionally, the user model of a document may not reflect the actual document model in the file representation (see Section 2.5.1). To estimate whether an edit operation is correctly merged, I rely on the user model. The main motivation for this decision derives from usability considerations. A syntactically correct merge that is misconstrued by the user is unacceptable. Users would not trust that tool. Apparently, a non-trusted merge tool is useless.

For each edit operation to merge, I have defined an expected behavior. For non-conflicting operations, this is the correct path to apply on. If the edit operation conflicts, I expect the patch algorithm to not apply the operation. Each edit operation of an applied delta can be categorized as follows:

- *positives* have been applied to the delta.
- *false positives* are positives which have been applied to a path where they have not been supposed to. This includes conflicting inserts and updates.
- *negatives* have been rejected due to a match quality below the threshold value.
- *false negatives* are negatives which have been rejected, even if a matching path would have been available in terms of the user model.

Apparently, false negatives are the most undesirable result. In this case, an edit operation has been applied, although it has not been supposed to. This is a severe drawback, as no error is reported, and the user would assume a successful merge. To avoid this, stricter merge rules are preferred rather than accepting a false positive. However, this implies that acceptable edit operation may be rejected, leading to a false negative.

9.2 Test Scenarios

The evaluation follows two goals. First, the time complexity shall be verified empirically, which requires a synthetic test suite that ensures a constant modification pattern across all test cases. This test suite is used to verify the runtime of the diff and the patch algorithm. The runtime of the patch algorithm is tested for patching and for merging, which are treated as different scenarios.

The second goal of the evaluation is to estimate the quality and the reliability of the merge results. Here, an every-day scenario is used to gain realistic results.

9.2.1 Differencing

The evaluation of the differencing algorithm follows two goals. First, the runtime of the algorithm shall be evaluated depending on the document size. Second, the impact of some design decisions of the differencing algorithm shall be evaluated.

To evaluate the performance, similar documents with different sizes are taken. In this context, I use the term "similarity" for documents with similar tree structure. The similarity shall lower the influence of the tree structure on the runtime of the differencing tool. Here, ODF spreadsheets are especially suitable, as each cell is stored within the document, thus ensuring a common structure of rows and columns, no matter which value the cells contain. The evaluation is performed on 21 spreadsheets, ranging from 12 to 377 KByte (106 to 3823 nodes). The same documents have been used for the document analysis in Section 2.4. They are listed in Appendix A.

On these documents, changes are performed. Three types of changes emerge, namely an insertion of a subtree, an update of a leaf node, and an update of a non-leaf node. I will discuss these changes later on. Only one change from each type is performed at any time. This way, the quantity of edit operations is isolated from the size of the document, because latter one is an important factor of the complexity class of the algorithm. The influence of the change quantity will be evaluated in Section 10.3. Updates are performed on the leaf level and on the non-leaf level as well, as they are handled by different steps of the algorithm. The insertion consists of a subtree with a height of 2, containing a simple paragraph. This text can be inserted arbitrarily within the cells without breaking the validity of the document.

9.2.2 Patching

The patch scenario relies on the deltas gained from the diff scenario. The deltas describing the leaf updates are applied to their respective original document. This way, no conflict or ambiguity can arise. Here, the linear patching is compared with the default behavior of the algorithm.

9.2.3 Merging

I use two different scenarios for evaluating the merge capability. The first scenario relies on the previous scenario. The leaf updates are applied to the documents that already contain the insert operation. In the document order, all leaf updates occur after the insert operation, causing an interfering edit operation. By this scenario, I intend to evaluate the additional overhead due to the merge case, compared to the normal patching. This includes the influence of the neighborhood radius on the runtime.

The focus of the second scenario lies on the merge quality and reliability. In this scenario, different office text documents are taken and edited independently. The changes are grouped nearby to force a non-optimal merge, and to reflect a natural editing process as described in Section 2.5.3. The merge is performed on 6 documents with a size between 2 and 100 KB. Deltas are applied to a changed document, forcing a merge. The deltas consist of 1 to 10 edit operations, resulting in 60 edit operations in total. The test scenario contains 3 conflicts. Most edit operations interfere with each other to force the tool to perform a neighborhood search. During the tests, the neighborhood radius stays constantly at 6, which is the default radius.

9.3 Runtime

The runtime of the algorithm is one of the most important usability aspects. Especially if a document is handled often by the XCC framework (e.g., in an auto-versioning file-system), runtime becomes a crucial factor.

The runtime of the differencing tool is examined first. Afterwards, time for patching is evaluated. Finally, I evaluate the influence of the neighborhood radius in the merge case, before discussing the results.

9.3.1 Differencing

As the proposed test scenario has a constant quantity of edit operations throughout all document sizes, the runtime of the differencing tool should scale linearly with respect to the document size. This follows directly from the claimed complexity class of $O(|leaves(A)| \times D + |ancestors(A)| + D)$. With $D = 1$, the complexity should yield $O(A)$.

Figure 9.1 shows the runtime for the different types of edit operations. As expected, the runtime increases linearly. The line is fitted with a maximum asymptotic standard error of 5.873%. The gradient approximates 3, the offset is about 200 ms. This is the time needed for initialization, including the parsing of the document.

The runtime of the detection of the leaf update is the shortest. This is not surprising, as the leaves are investigated first. The insertion of the subtree takes slightly more time in most cases, as more nodes are affected by this change, and the subtree has to be constructed for the edit operation. It is not possible to give a prediction on the runtime needed by the different edit operation types. Other factors like the path of the edit operation within the tree or the size

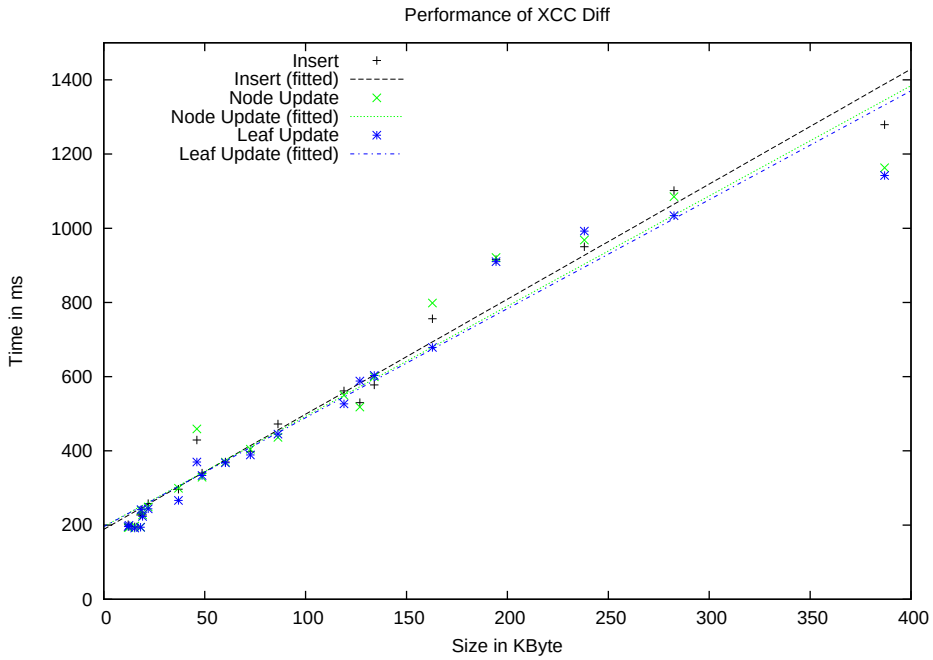


Figure 9.1: The runtime of the differencing tool increases linearly with the document size, conforming to the claimed complexity class.

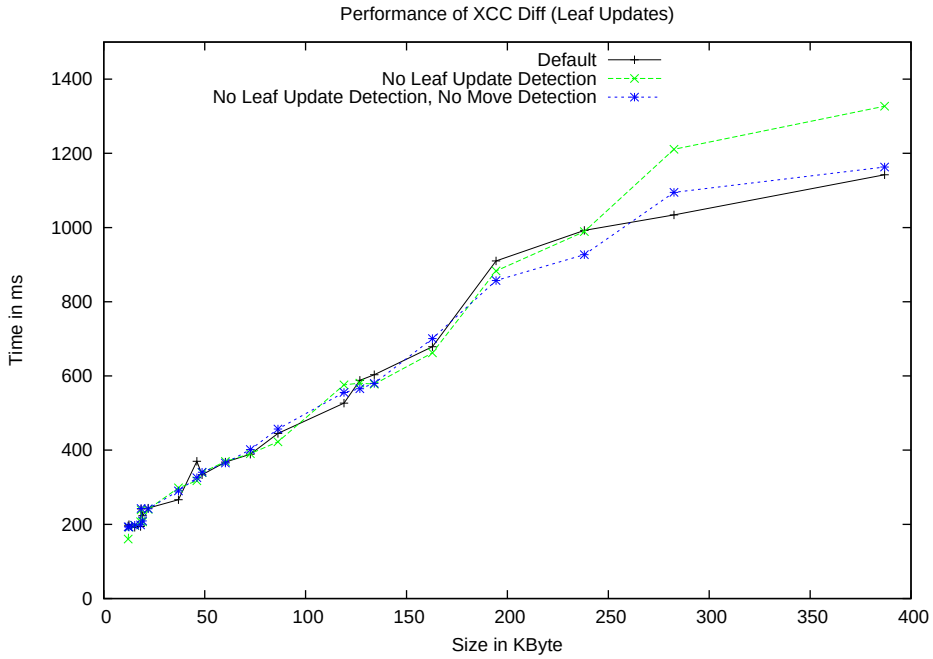


Figure 9.2: The detection of leaf updates does not affect the runtime significantly. In some cases, it even saves runtime.

of a structure-affecting change are also important, which prevents a generally applicable rule. Nevertheless, the evaluation enforces the claimed complexity class.

The ability to detect updates on the leaf level is an important aspect of my algorithm. However, this requires additional computation during the identification of structure-affecting changes (see Section 7.4.2). In the complexity analysis, I argued that this additional time can be neglected (see Section 7.5.3), which ought to be verified by the evaluation. To do this, the test scenario has been solved by a version of the algorithm that omits the detection of leaf updates. Figure 9.2 shows the (non-fitted) results. Interestingly, they are not straightforward. In most cases, the runtime without leaf update detection is slightly shorter than in the default version of the algorithms. For the larger documents, the simpler version of the algorithm runs even slower. This derives from the fact that the leaf updates are represented as a pair of an insert and a delete operation. By this, the algorithms verifies whether these operations build a move operation later on. Figure 9.2 also shows the runtime of the algorithm in case that the leaf update detection and the move detection are both disabled. Here, the tool preforms slightly faster. However, it is still slightly slower than in the default configuration. Obviously, it needs more time to create two structure-affecting changes including their context fingerprints, than creating one update operation. This example shows that the detection of leaf updates does not affect the complexity class of the algorithm.

9.3.2 Patching

Patching a document is even faster than differencing. Figure 9.3 shows the results. The lines are fitted with a maximum asymptotic standard error of 5.519%. The gradient approximates 1.2, whereas the offset counts approximately 130 ms. The initiation is faster than for the differencing tool as only the document to patch and the delta have to be parsed. The runtime increase is significantly lower as the patch algorithm may directly jump to the path where the edit operation has to be applied.

In its default settings, the tool performs a best-effort merge with a neighborhood radius of 6 and a threshold of 0.7. However, the neighborhood is not searched, as the algorithm finds a perfect match at the expected path. Therefore, the linear patching is only slightly faster. Nevertheless, it has an advantage as the neighborhood is not generated and the match quality is never computed.

9.3.3 Merging

Merging is almost as fast as patching. The results are shown in Figure 9.4. The maximum asymptotic standard error yields 5.623%. As with the patch scenario, the gradient approximates 1.2, the offset 130 ms. The conflict detection is not listed explicitly, as it does only affect delete operations (see Section 8.7.1). For update operations, possible conflicts are already detected by the anchor of the context fingerprint. Interestingly, the neighborhood radius does not seem to noticeably affect the runtime of the tool. The runtime for a neighborhood radius of 6 (default) and 30 is almost identical.

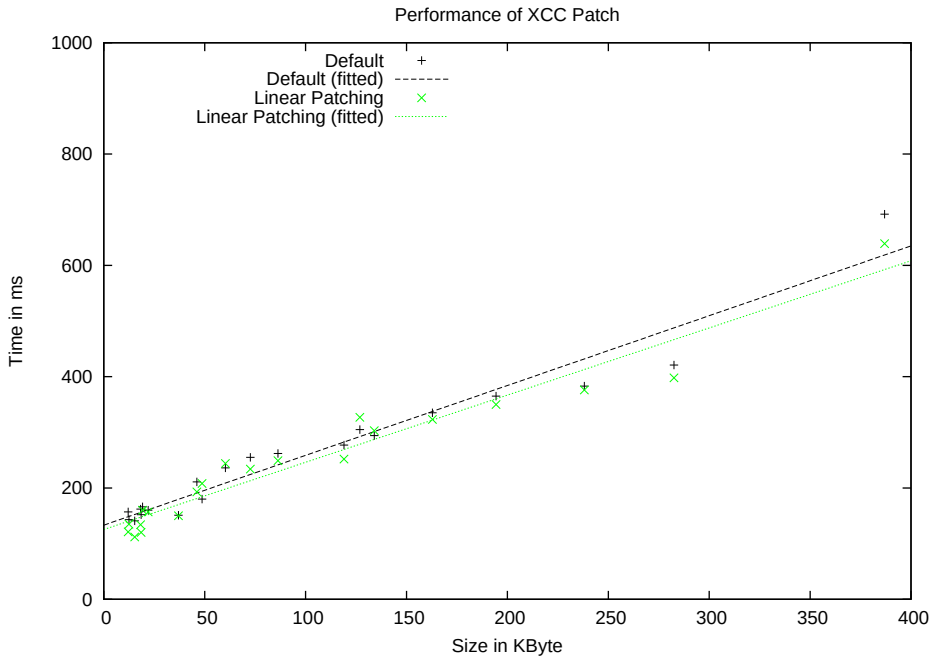


Figure 9.3: The runtime of the patch tool increases linearly with the document size, too. Linear patching is only slightly faster.

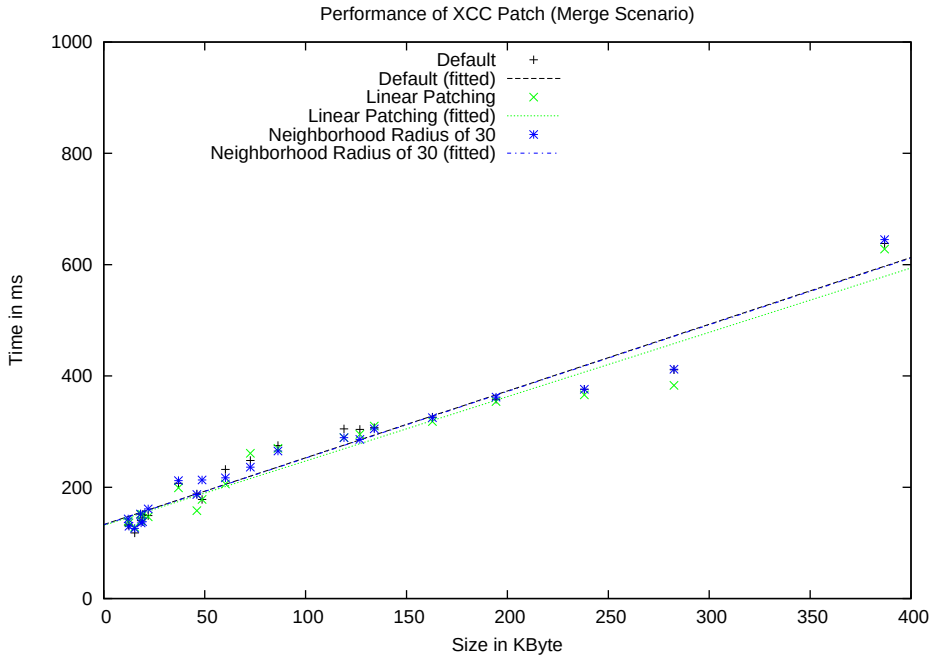


Figure 9.4: Merging is as fast as patching. Even a neighborhood radius of 30 does not affect the runtime noticeably.

9.3.4 Discussion

Each test run has been performed three times. The runtime of the single runs alternates with a deviation of nearly 20% in some cases. This leads to some spurious results, where the less complex linear patching runs slower than merging. This seems to follow from the large caches in hard drives and CPUs, which offer a significant gain of speed in case of a cache hit. One could disable the caches to get more comparable results. However, the results would not reflect the behavior in an usual environment, where the caches are activated. Using the proposed methodology, the maximum asymptotic standard error yields 5.873%, which is an acceptable value. Therefore, I decide to use the caches to represent realistic runtime values.

The runtime of the diff algorithm increases linearly, which conforms the claimed complexity class. In general, an increase of 100 KByte of the document size leads to an increase of 300 ms in runtime. When assuming the size of the documents to be far higher than the amount of changes, the complexity class can be approximated with $O(A)$. According to the analysis of the change patterns in Section 2.5.3, this assumption meets the expected standard case.

Patching and merging are even faster than differencing, as only one document and the delta have to be parsed. An increase of 100 KByte in document size yields a runtime increase of 120 ms. In general, merging is only slightly slower than linear patching. These results enforce the suitability of the proposed approach for every-day use. Interestingly, the runtime is not noticeably affected by the neighborhood radius. Here, I recall the complexity class of the patch algorithm, which is $O(A \times \delta)$. The fingerprint radius r and the neighborhood radius ρ are apparently far smaller than the document size. Therefore, they are neglected in the claimed complexity class. The evaluation enforces this assumption.

In this evaluation, the runtime is only investigated with respect to the document size. Prior evaluations by Rönnau [2004] and Lindholm et al. [2006] have shown that the document size is a major challenge of document differencing. The quantity of changes is the other important dimension that affects the runtime of the algorithm. I will investigate the behavior of XCC diff in a comparative evaluation with other differencing tools in Section 10.3.

9.4 Merge Quality

The behavior of the patch algorithm during merging is deterministic, but can be influenced by different parameters, namely the neighborhood radius, the conflict detection, and the threshold value. Among those parameters, the threshold value is the most important. It defines to which degree a non-optimal matching of an edit operation is acceptable.

The main focus of this part of the evaluation is to determine the impact of the threshold value on the merge quality. The conflict detection is discussed afterwards. Finally, I discuss some issues of false negatives with respect to the user model of a document, as well as the neighborhood radius.

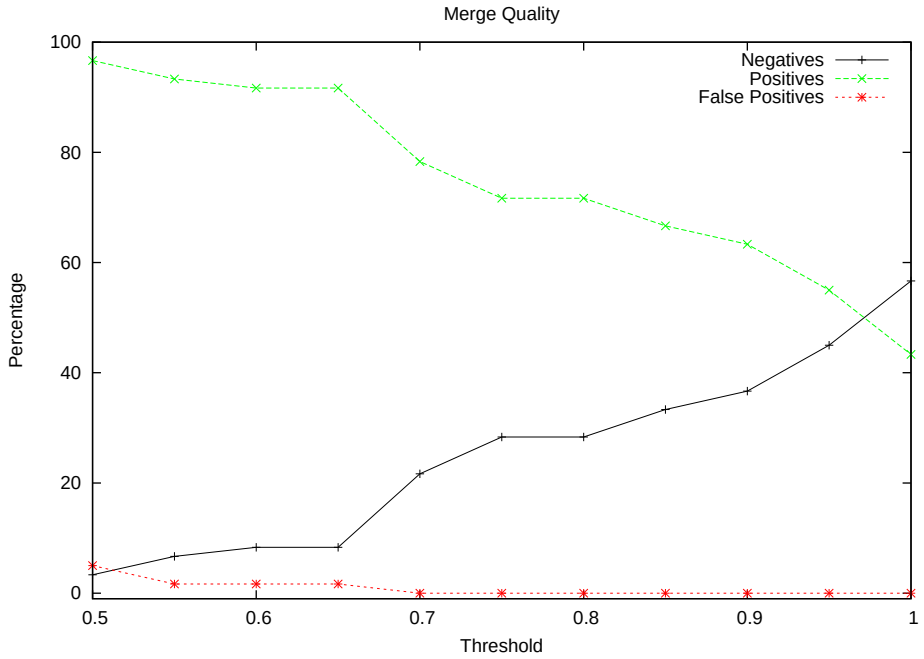


Figure 9.5: The merge quality mainly depends on the threshold value. A low threshold leads to false positives, a high threshold yields a high amount of negatives.

9.4.1 Influence of the Threshold Value

Using a low threshold, an edit operation will be applied, even if its syntactic context does not match completely. A threshold of 0 denotes that the patch algorithm completely disregards the syntactic context. A threshold of 1 requires all nodes within the context to match. Especially in cases where changes are clustered, the probability for a non-matching node in the context of an edit operation becomes significant. Figure 9.5 shows the merge quality depending on the threshold value. Threshold values lower than 0.5 are not displayed as the merge quality is almost constant in this range. A low threshold leads to false positives, which are unacceptable. Increasing the threshold value leads to less matching nodes, but decreases the probability of false positives. Between a threshold of 0.65 and 0.7, the number of negatives increases significantly. In parallel, the number of false negatives drops to 0. This is caused by the properties of the weighing function used in the patch algorithm. A non-matching anchor leads to a match quality of 0.636. The same match quality results if both adjacent nodes of the context fingerprints do not match. Both are major indicators for a false positive.

A higher threshold than 0.7 has no advantage in the test scenario. It leads to a high number of negatives, which are all false negatives. Nevertheless, other scenarios may require a higher threshold to gain a high merge quality.

9.4.2 Conflict Detection

Most of the negatives are false negatives. Only 3 of the edit operations must not be applied (i.e. "true" negatives), as they are conflicting. Therefore, false negatives are not indicated separately. All of them are already rejected using a threshold of 0.5, without using the explicit conflict detection. The conflict detection is only needed for delete operations that target subtrees or tree sequences. As conflicts are well-defined and the conflict detection works straightforward, it is not evaluated separately.

9.4.3 Discussion

Two reasons exist for false negatives. First, the neighborhood radius may be too small to find the correct context of an edit operation. Second, a change may be acceptable in the user model, despite being unacceptable in the document model.

For example, equivalent markup information may be defined using different identifiers in the XML representation [Rönnau and Borghoff 2009]. The displayed document seems to be equal within the office application, but in the underlying XML document, all markup information differs. This kind of mismatch between the user model and the document model cannot be handled by my approach, as it would require application-specific rule-sets. In turn, a higher neighborhood radius can be set using a command-line parameter.

Even if the influence of the neighborhood radius is small on the runtime, the default radius is rather small. This is done to prevent erroneous matches deriving from the high amount of repeating nodes throughout the document. This is a known hazard in the line-based domain [Khanna et al. 2007]. Although this case occurs seldom, I decide to reduce the probability for erroneous matches by restraining the default neighborhood radius (see Section 8.3.2).

9.5 Conclusions

Speed is a crucial prerequisite for user acceptance. This is especially important as merging is known to be rather complex and time-consuming [Mens 2002]. In this chapter, I evaluated the runtime of the tools of the XCC framework. The results are very promising. For document differencing, no run lasts longer than 1.5 s on a standard laptop, which is a fairly good value. Patching and merging is even faster. Here, the tool takes below 0.8 s for merging a document. The runtime of both diff and patch evolve linearly, as the size of the document is far bigger than the amount of changes. Interestingly, merging a document is nearly as fast as patching. This is an important achievement that ensures a high user acceptance, as the merging offers a gain of features without causing a significant trade-off in terms of runtime. All tests have been performed using a synthetic test scenario to ensure a high accuracy.

A second important aspect of merging is the reliability of the merge result. Here, every-day documents are used as test environment to ensure that the proposed solution works outside synthetic conditions. The results are impressive. The context fingerprint appears to be a

reliable source of information to search for the correct path of an edit operation. The main parameter to control the merge procedure is the threshold value. It defines to which degree a partial matching is acceptable. The evaluation shows that a threshold value of 0.7 ensures a high reliability, without rejecting too many edit operations. Using this threshold, no false positive occurs. This is an important aspect, as an erroneously applied edit operation is highly undesirable.

The design decisions of all tools within the XCC framework have been made upon the assumptions on the properties of XML documents and their change patterns, which have been elaborated in Chapter 2. The evaluation has shown that the design holds well in practice, offering a fast and reliable framework for XML change control.

Part III

Conclusions

10 A Comparative Evaluation of XML Differencing Tools

Several differencing tools for XML documents exist. They use different algorithms and delta models which have been presented in Section 3.3. In this chapter, I evaluate XCC Diff compared to these tools which are the best known in terms of their time complexity class. As no other merge-capable patch algorithm is available, a comparative evaluation cannot be performed for XCC Patch.

I start by briefly presenting the test setup and the competitors. The evaluation itself is performed using basic tests and two different scenarios, which are presented afterwards. The evaluation covers the runtime and the memory consumption. Finally, the generated deltas are analyzed.

10.1 Experimental Setup

Five tools are compared in total, shown in an overview first. The test environment and the test methodologies are presented afterwards.

10.1.1 Tools

In Section 3.3, I have presented several XML differencing approaches. One of the selection criteria was the availability of an implementation of the algorithm. These implementations are taken as competitors for XCC Diff. Table 10.1 gives an overview of their algorithmic properties, Table 10.2 shows their on-line resources.

Except for Microsoft XML Diff which is implemented using C#, all tools are written in Java and are platform-independent for that reason. As Microsoft XML Diff only runs on Windows systems, the evaluation has been performed on that operating system. This is also the reason for using the Java-based implementation of XyDiff, called jXyDiff. The original XyDiff implementation has been performed on Linux using C.

Faxma, jXyDiff, and Microsoft XML Diff claim a complexity bound only depending on the document size. For XCC Diff and diffxml, the amount of changes has an influence on the time complexity, too. The space complexity is not listed in the overview, as it has not been published for all approaches. For Microsoft XML Diff it is quadratic, whereas it is linear for XCC Diff.

Tool	Time Complexity	Delta Model	Insert and Delete	Move Supported	Minimum Edit Script	Invertible
XCC Diff	$O(\|leaves(A)\| \times D + \|ancestors(A)\| + D)$	set of edit operations	subtrees	yes	no	yes
diffxml Chawathe et al. [1996]	$O(\|\text{leaves}(A)\| \times e + e^2)$	sequence of edit operations	nodes	no	no	no
faxma Lindholm et al. [2006]	$O(A)$ (average case) $O(A^2)$ (worst case)	insert operations and references to original version	both	yes	no	no
jXyDiff Cobéna et al. [2002]	$O(A \times \log A)$	sequence of edit operations	subtrees	yes	no	yes
Microsoft XML Diff Zhang and Shasha [1989]	$O(A_1 \times A_2 \times \min(\text{depth}(A_1), \text{leaves}(A_1)) \times \min(\text{depth}(A_2), \text{leaves}(A_2)))$	sequence of edit operations	both	yes	yes	no

Table 10.1: XCC Diff and four other XML diff tools are compared in the evaluation. They differ in terms of their complexity class and the delta model used.

Tool	Availability
XCC Diff	`https://launchpad.net/xcc`
diffxml	`http://diffxml.sourceforge.net`
faxma	`http://code.google.com/p/fc-xmldiff`
jXyDiff	`http://potiron.loria.fr/projects/jxydiff`
Microsoft XML Diff	`http://msdn.microsoft.com/en-us/xml/bb190622.aspx`

Table 10.2: All tools are available on-line.

10.1.2 Test Environment

All test runs have been performed on a machine equipped with an Intel Core 2 Quad Q9650 3.00 GHz processor, 4 GB RAM, running Windows Vista Business SP1 (64 bit) and Sun Java JDK 1.6.0_10 (32 bit).

Interestingly, the runtime of the Java-based tools increases when using a 64 bit Java Runtime Environment. Apparently, the XML processor of this Java version is not optimized for handling XML documents efficiently. I did not investigate this issue further, as the 32 bit environment does not constrain the capabilities of the tools.

10.1.3 Methodology

The runtime is measured by a Java-based application that starts the corresponding differencing tool using the shell interface. The system time is stored directly before and after the run of the tool, with a granularity of nanoseconds. The difference is estimated as runtime of the tool. Using this methodology with a shell interface, each tool is loaded including its runtime environment and libraries. Especially as most of the tools are Java-based, this ensures that the whole JRE is loaded again[1]. Otherwise, the measured runtime would be significantly lower, not reflecting the total time a user needs when he invokes the differencing tool the first time.

The memory consumption is measured by a third-party tool that measures the total memory consumption of an application. On the one hand, this includes the overhead that derives from loading the whole runtime environment including all libraries. Especially if a similar application is loaded in parallel, the bare memory footprint of the differencing tool could be estimated to be smaller. On the other hand, however, this methodology reflects the case that a user does only starts the differencing tool if needed, without having further XML editing tools running. I guess this to be more suitable for the average case.

[1] This methodology somewhat contradicts the evaluation setting in Section 9.1.2. In this case, however, the runtime of the whole tool must be measured to guarantee a comparative result, as not all tools rely on Java. In case of Microsoft XML Diff, the .NET framework is loaded upon startup, which affects the overall runtime, too.

10.2 Test Scenarios

All tools have to solve different tests. First, basic tests are performed to verify the correct function of the tools. Afterwards, a test suite based on a synthetic document evolution is presented. The last test suite uses snapshots of a Web page to reflect a more natural document evolution.

10.2.1 Basic Tests

Some reasons exist to normalize XML documents before comparing them. As already described in Section 2.2.2, two semantically equivalent documents may differ syntactically, as there exist different possibilities of representing one and the same node.

First, the attributes of a node may be in arbitrary order. The *attribute ordering test* verifies whether a re-arrangement of the same attributes is detected as edit operation. Of course, this should not happen. Second, different input encodings can be used. Although the XML specification requires all documents to be encoded in Unicode, different Unicode encodings may reflect the same value. In the *Unicode test*, a differencing tool should not detect an edit operation on a different input encoding (UTF-8 and UTF-16) for the same character.

These tests do mostly verify the completeness of the implementation, independent from the algorithm used. Nevertheless, they are important to estimate the usability of these tools for every-day work.

10.2.2 Synthetic Document Evolution

The amount of changes has a significant influence on the runtime of a differencing algorithm. Apparently, a larger delta yields a higher runtime. However, only some of the analyzed algorithms respect the amount of changes within their complexity bounds. To verify the claimed complexity class, a test suite is needed in which the amount of changes increases constantly without significantly affecting the size of the document. A fixed document size (i.e. the quantity of nodes) prevents an influence of the size on the runtime. This is especially important, as the time complexity of all differencing algorithms mainly depends on the document size.

Spreadsheets in ODF have an important property in this context. Each cell within the table spanned by the first and the right-most outer-most filled cell is represented as node in the XML representation, no matter, whether that cell contains any value or not [Brauer et al. 2007]. Pohlemann [2009] has created a test suite on a complex ODF spreadsheet of 375 KByte, consisting of around 5700 nodes. The different versions cover a range from 5 to 100 changes, in steps of 5 changes. In this context, a change is seen as a change on the user model, i.e. by means of the office application. Basically, the user model and the document model may diverge. In this test suite, less complicated changes were performed that do not affect the document model heavily (see Section 2.5.1 for a discussion on the user model).

In this test scenario, a precise knowledge of the changes performed exist. They were performed manually with the aim to have a linear increase of the change amount. This allows for

	Attribute Ordering Test	Unicode Test
XCC Diff	pass	pass
diffxml	pass	pass
faxma	pass	pass
jXyDiff	fail	abort
Microsoft XML Diff	pass	fail

Table 10.3: The basic tests are not solved by all tools. JXyDiff aborts when parsing documents that are not encoded in UTF-8.

a suitable evaluation of the influence of the change amount on the runtime of the differencing algorithm. However, this synthetic test suite does not reflect the "natural" evolution of a document.

10.2.3 Archiving a News Site on the Web

Most Web pages evolve over time. Especially news sites and blogs are updated frequently. However, they do not evolve in a linear way [Adar et al. 2009]. Re-arrangements of headlines and advertisements occur constantly, thus leading to move operations (see Section 2.5.2). In this context, one cannot estimate the scale of the changes between two versions. Nevertheless, tracking the evolution of a Web page is an important task [Masanès 2006].

In this scenario, I archive the changes on a Web page during a day. 17 snapshots of the Wired news site[2] are taken between 7am and 10pm[3]. One snapshot consists of around 2400 nodes and 108 KByte. In this test scenario, I cannot state a correct order of the difficulty of the test runs. Instead, the different snapshots are ordered by the time they have been taken.

10.3 Differencing

All tools have to solve the basic tests first. The runtime of the different approaches is presented afterwards, followed by the analysis of the memory consumption.

10.3.1 Basic Tests

Most of the tools solve the attribute ordering test flawlessly, as shown in Table 10.3. Only jXyDiff recognizes a difference between the documents with modified attribute ordering. This is obviously an implementation error, as the original implementation based on C handles this test correctly. The same holds for the Unicode test, where jXyDiff aborts the run as it is only

[2] `http://www.wired.com`

[3] The site does not provide significant change of content outside these hours.

able to handle XML documents encoded using UTF-8. Microsoft XML Diff interprets two identical documents with different encoding as changed, which is not the expected behavior.

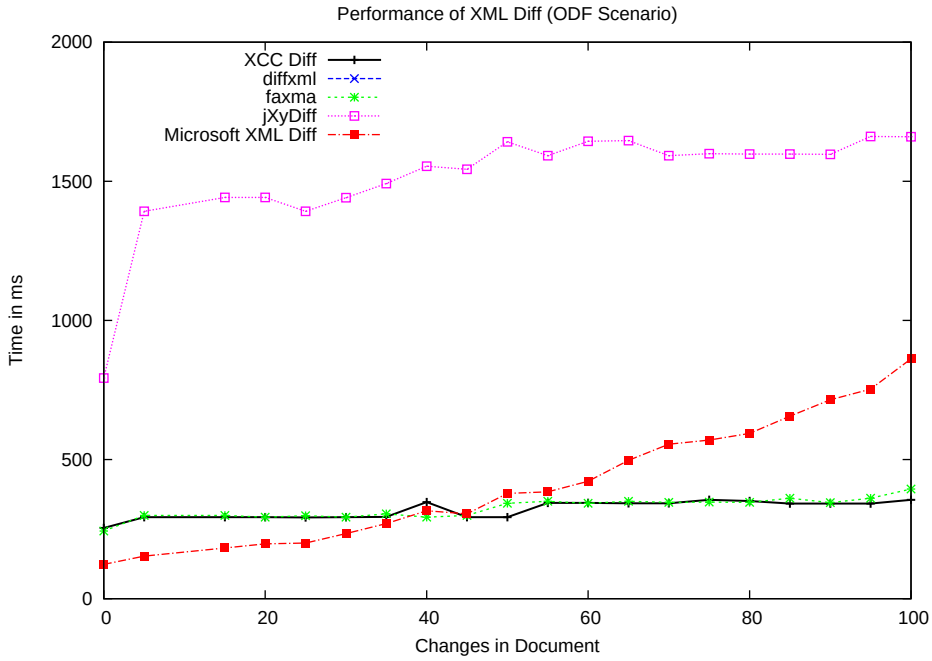
10.3.2 Runtime Comparison

Figure 10.1 shows the runtime of the compared tools. Figure 10.1(a) and Figure 10.1(b) use different scales on the y-axis to ensure the readability of the results. Whereas XCC Diff, faxma, and jXyDiff show a comparable behavior throughout both scenarios, diffxml and Microsoft XML Diff act differently.

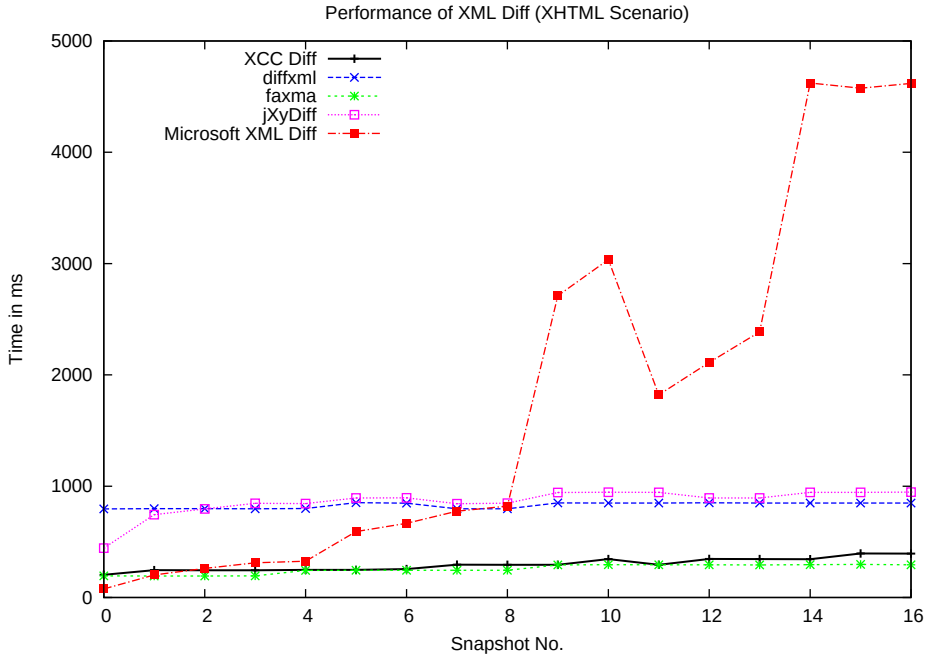
As the ODF scenario is performed on a larger document, the runtime is obviously higher than in the XHTML scenario for all tools. The ODF document has more than twice the amount of nodes compared to the Web page. Nevertheless, the increasing factor is lower for most tools. For diffxml, however, the runtime becomes unacceptably high. Each differencing run takes more than 35,000 ms, which is 40 times the runtime of the XHTML scenario. This is especially interesting as the time complexity class of $O(|\text{leaves}(A)| \times e + e^2)$ suggests that the quantity of changes has a higher influence than the document size. Chawathe et al. [1996] have stated that their algorithm may misbehave in cases where a node value occurs frequently within a tree level. As shown in Section 2.4.3, this pattern is typical for XML documents. In Section 10.4, I will discuss this issue in more detail.

Even if being less complex, Microsoft XML Diff shows a strange behavior in the XHTML scenario. The runtime increases significantly in some test cases. As the documents in this scenario are not ordered by the complexity of the changes but by their change date, one could assume that large changes have been performed in the affected documents. In that case, however, I would expect the competing tools to act likewise, which they do not. Microsoft XML Diff relies on the algorithm by Zhang and Shasha [1989], where special cases are not known. As the implementation is closed-source, a further investigation cannot be performed. Using a black-box approach, I would guess that the search for the minimum edit script is very time consuming in these cases. It is arguable whether the proposed implementation does meet the claimed complexity class of $O(|T_1| \times |T_2| \times \min\{\text{depth}(T_1), \text{leaves}(T_1)\} \times \min\{\text{depth}(T_2), \text{leaves}(T_2)\})$. Here, the amount of changes is not assumed to have a significant influence on the time complexity. However, the evaluation shows that for both scenarios, the runtime increases noticeably depending on the amount of changes.

The other differencing tools show a nearly linear development of the runtime, only slightly depending of the change quantity. Even for XCC Diff, whose complexity class depends on the edit distance, the influence of the change quantity is only minimal for the tested range. This can be derived from the fact that the quantity of nodes is far larger than the amount of changes. Interestingly, jXyDiff is really fast for the first test case, where the original document is compared with itself, thus representing an unchanged document. Here, the greedy bottom-up approach of XyDiff (see Section 3.3.4) offers a fast heuristic to detect unchanged documents.



(a) ODF Scenario



(b) XHTML Scenario

Figure 10.1: The tools differ significantly in terms of their runtime (note the different scales). Most interestingly, some tools show a different behavior for the different scenarios.

10.3.3 Memory Consumption

As far as it has been published, the tested algorithms have a linear space complexity. Only for Microsoft XML Diff, a quadratic space complexity depending on the size of the document is known. Figure 10.2 shows the memory consumption of the tested tools.

The memory consumption of Microsoft XML Diff reflects the runtime shown in the last section. Once again, the reason for this behavior cannot be further investigated, as the tool is closed source. All other tools show a nearly constant memory consumption. A slight increase for a higher amount of changes is acceptable, as the delta to construct has to be represented in memory, too (which is usually not part of the complexity analysis). Especially diffxml seems to have a suitable heuristic to detect equal documents. In this case, the memory consumption is significantly lower than in the other test cases. The memory consumption of all tools is reasonable, allowing for every-day use with even complex documents.

10.4 Delta Analysis

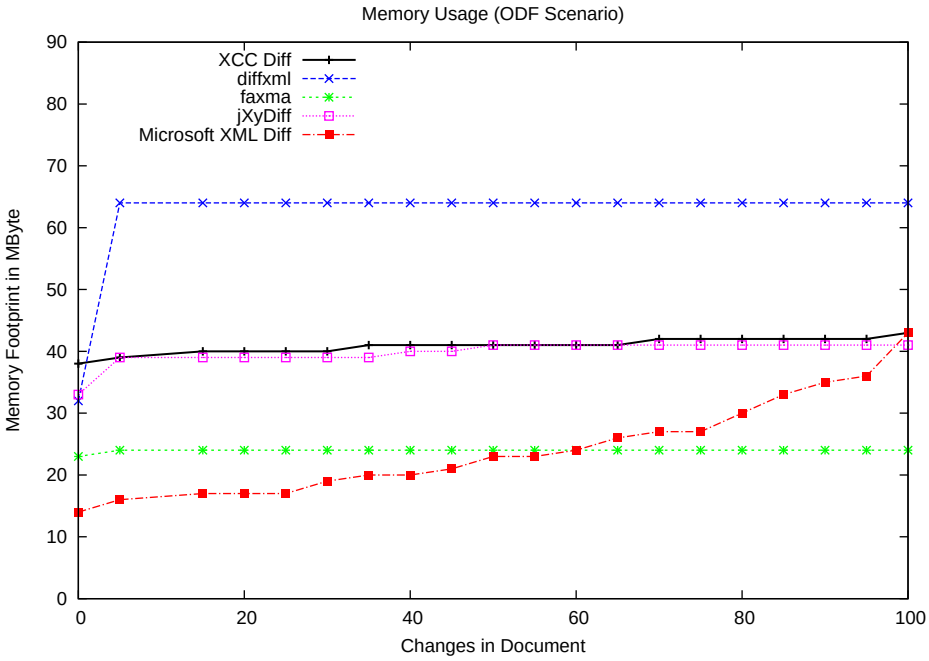
Most evaluations focus on the mere speed of an algorithm. For the usability of an approach, the quality of the resulting deltas is an important aspect as well. Here, I will analyze the quantity of edit operations that are needed to reconstruct a document version. Afterwards, the distribution of the different operation types is investigated. Finally, the total size of the deltas will be evaluated.

10.4.1 Edit Operations

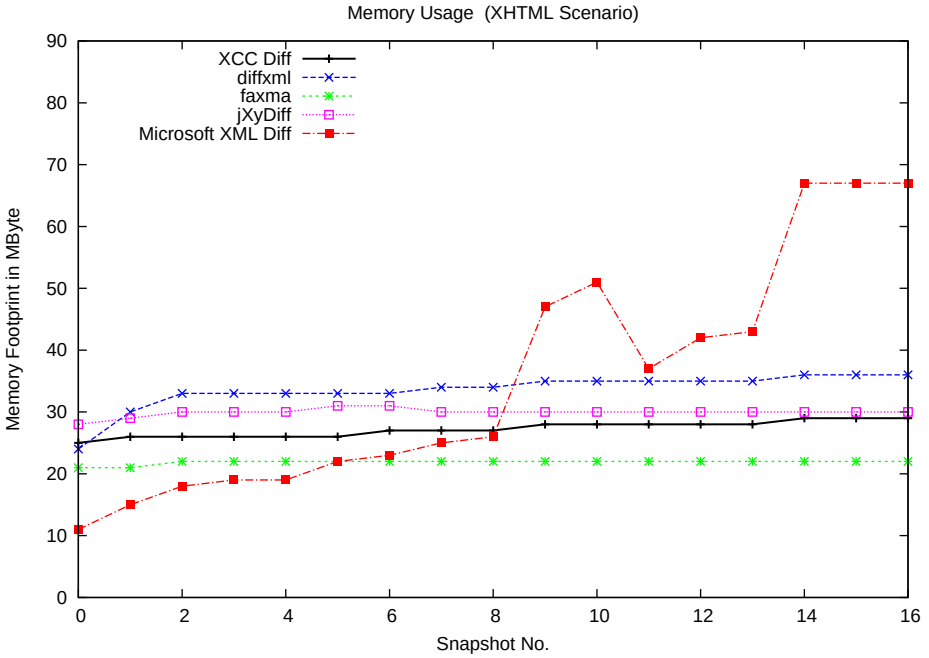
The algorithm by Zhang and Shasha [1989], which lays the basis of Microsoft XML Diff, is the only one which claims to compute a minimum edit script. All other algorithms can only ensure a best-effort approximation. The algorithm of Chawathe et al. [1996], implemented by diffxml, claims to create near-optimal results in the general case.

Figure 10.3 shows the quantity of edit operations that are computed to represent the changes between two document versions. To ensure a better readability, I have used different scales.

Diffxml outreaches all other tools for the ODF scenario and is second worst for the XHTML scenario. At first sight, this contradicts the claim to create near-optimal results. Looking more closely, two reasons exist for this behavior. First, the algorithm relies on the conventional formulation of the tree-to-tree editing problem by Selkow [1977], which has been discussed in Section 3.2.1. Here, each node change is represented by edit operations on the leaf level. Apparently, this leads to far more edit operations than for the subtree-oriented delta model, which is used by the other approaches. Second, the algorithm is known to misbehave if many equivalent nodes exist on the same level of the tree. The algorithm computes the LCS of the nodes on each level. However, the LCS is not unambiguous (see Section 3.1.1). As a result, the matchings on the different tree levels may not correlate. By this, edit operations are created



(a) ODF Scenario



(b) XHTML Scenario

Figure 10.2: The memory consumption is nearly linear for most tools. Only Microsoft XML Diff shows significant increase of memory usage for larger deltas.

to transform one level to match its parent level, although another LCS would have lead to a near-optimal matching.

Basically, Microsoft XML Diff is the only tool to create minimum edit scripts. Nevertheless, XCC Diff needs less edit operations to describe the changes between two document versions throughout all test runs. The reason for this is simple. The delta model of XCC Diff allows for addressing whole subtrees and tree sequences by one edit operation. The algorithm by Zhang and Shasha [1989] relies on the delta model by Tai [1979], where only node-based edit operations are allowed. Basically, this would lead to a far higher number of edit operations within a delta. On the other hand, Microsoft XML Diff uses a modified delta model, which has been presented in Section 3.3.2. In this model, edit operations that span a subtree are glued together into one operation, which makes the delta model similar to mine. This gluing seems to work well in the ODF scenario, but not in the XHTML scenario. Nevertheless, the number of edit operations should not be higher than in the deltas generated by diffxml. This tool also uses a node-based granularity but with non-optimal results. Therefore, Microsoft XML Diff should create deltas with the same number of edit operations, or even less. The evaluation shows that the quantity of edit operations is significantly higher in the XHTML scenario. The amount of edit operations directly correlates to the runtime of the diff tool. As Microsoft XML Diff is closed-sourced, a further investigation is not possible. Apparently, this is an implementation error.

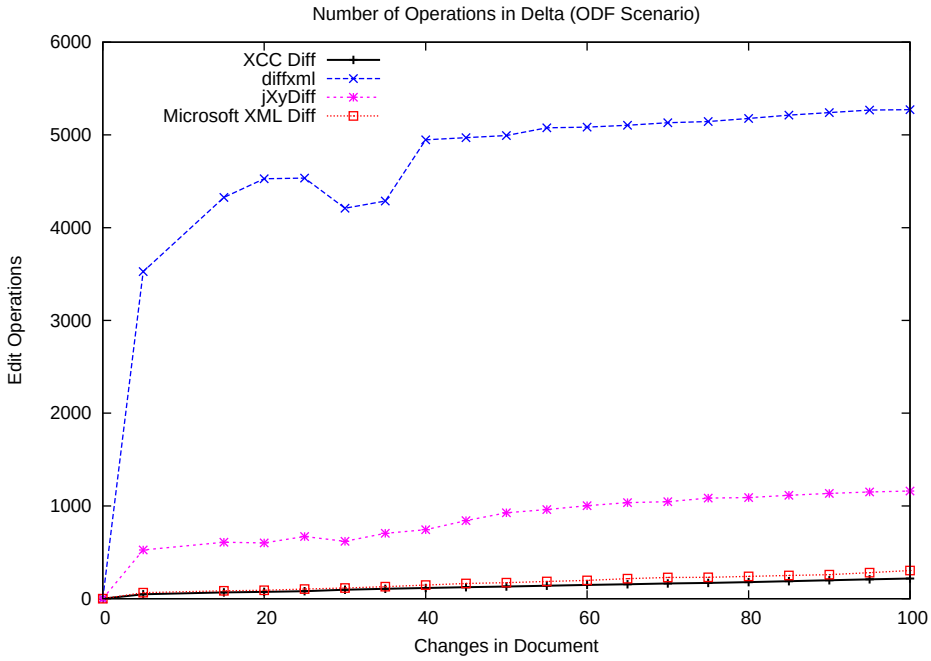
Faxma is not part of this evaluation. This is caused by its delta model. The delta created by faxma consists of a script that contains references to the original document, as well as the inserted parts (see Section 3.3.5). As deletions are not represented and moves (i.e., references to other parts of the document) are only hard to distinguish from regular references, I decided not to investigate this tool in this evaluation, as the result would not be comparable.

10.4.2 Operation Types

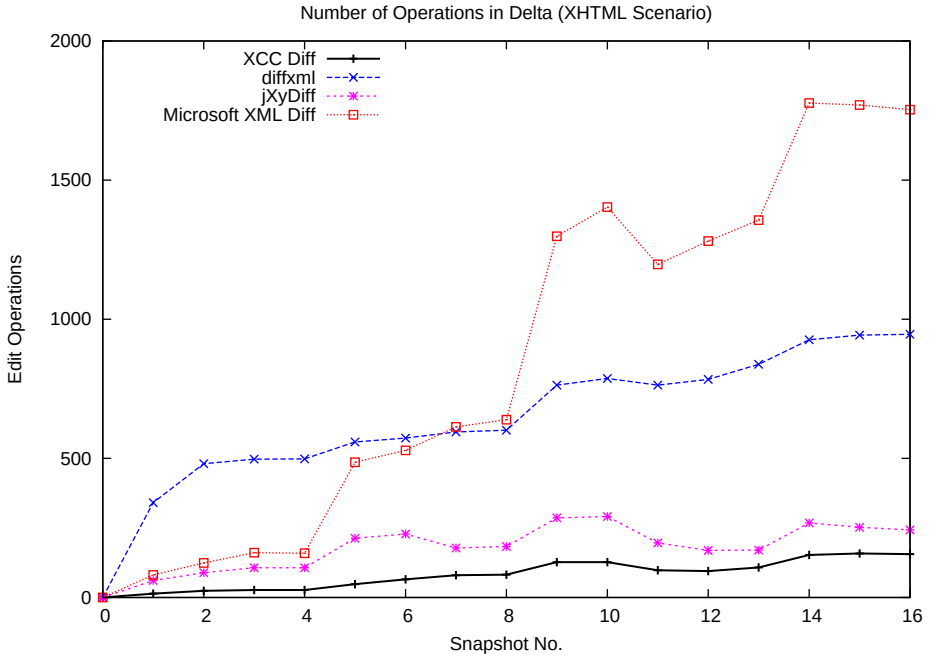
The total number of edit operations already shows a significant difference between the tools. However, the quantity does not reveal how the different algorithm represent the changes between the document versions.

Table 10.4 shows the distribution of the different operation types within a delta. Here, major differences between the tools become apparent. In the ODF scenario, XCC Diff uses insert, delete, and update operations nearly equally. No move is detected. Within the test documents, no move has been performed on the user model as well. Although this does not guarantee that no move occurs on the document model, the strong use of the move operation by diffxml and jXyDiff is conspicuous. I will discuss this issue later. For the XHTML scenario, XCC Diff uses more update operations, including some moves. This behavior reflects the assumptions on the evolution of Web documents stated in Section 2.5.2. In general, the pattern of the used operation types is similar for both scenarios.

Microsoft XML Diff shows a completely different usage pattern for both scenarios. In the ODF scenario, the update operation is heavily used. This derives from the fact that the structure of ODF's lay-outing nodes is highly similar. Additionally, there exist far more delete



(a) ODF Scenario



(b) XHTML Scenario

Figure 10.3: Only Microsoft XML Diff ensures a minimum edit script. Nevertheless, XCC Diff creates deltas with less edit operations due to the addressing of subtrees and tree sequences. Note the different scales.

		insert	delete	update	move
XCC Diff	ODF	34.51%	34.43%	31.06%	0.00%
	XHTML	20.45%	31.25%	41.11%	7.19%
diffxml	ODF	2.68%	3.46%	6.49%	87.37%
	XHTML	19.52%	20.04%	5.54%	54.90%
jXyDiff	ODF	5.85%	5.92%	18.77%	69.46%
	XHTML	26.41%	27.72%	20.88%	24.99%
Microsoft XML Diff	ODF	7.88%	20.68%	66.85%	4.59%
	XHTML	38.33%	38.38%	2.36%	20.93%

Table 10.4: The mixture of the operation types in the deltas differs both between the algorithms, as well as between the scenarios.

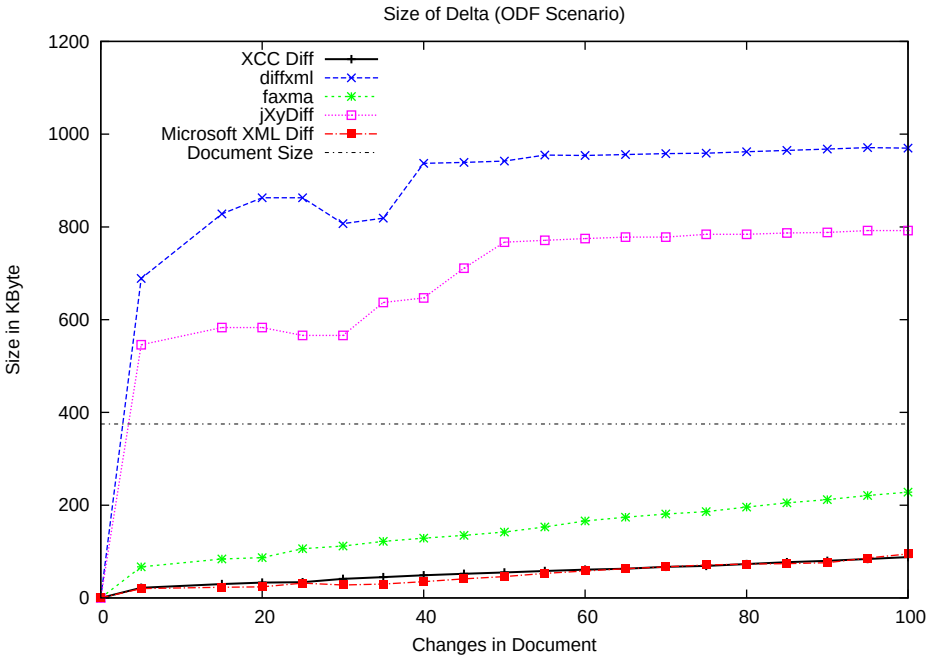
operations than insert operations. This is particularly interesting, as in most cases the quantity of insert and delete operations is nearly identical. This is straightforward, as insert and delete operations are used to represent more complex changes that affect different nodes within a subtree. In this case, however, the gluing heuristic of Microsoft XML Diff works better for delete operations than for insert operations. In the XHTML scenario, however, the quantity of insert and delete operations is nearly identical, with a high usage of the move operation. It is the test run where the implementation error occurs; the deltas deconstruct the old document and reconstruct the new nearly completely.

The test result for diffxml reflects the reason for the misbehavior of the algorithm. The high use of the move operation derives from the fact that diffxml tries to reconstruct the (wrongly matched) document structure. Nevertheless, the move operation is also heavily used in the XHTML scenario. This shows that the worst case of the algorithm design meets the average case in the domain of XML documents.

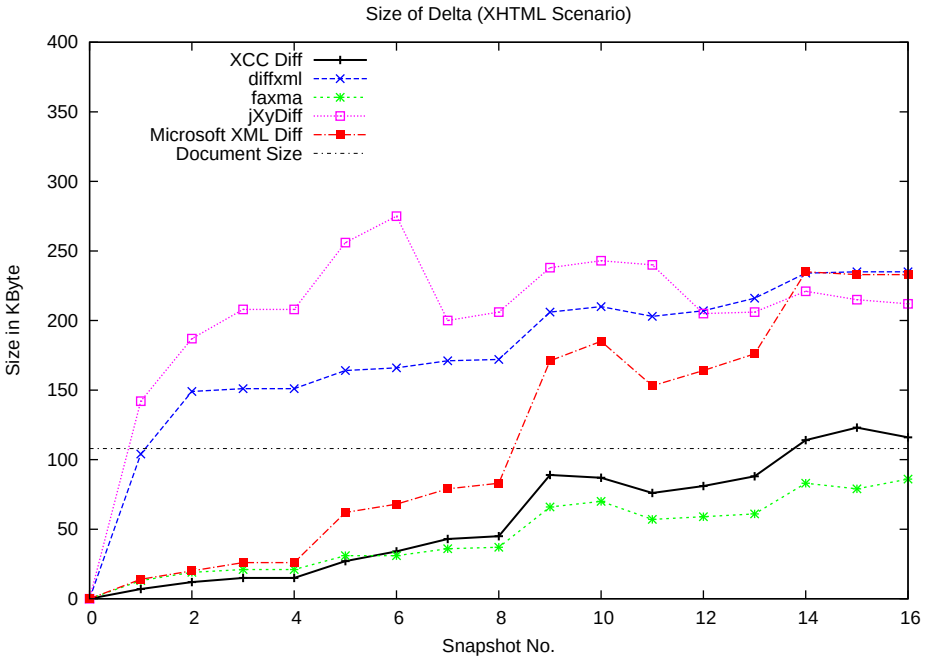
JXyDiff uses many move operations in the ODF scenario, too. This algorithm is also disturbed by the high amount of equal nodes. Its greedy nature leads to matching nodes in different subtrees, although not a real move occurred but an update. Additionally, each move leads to further moves, thus starting a chain reaction. For XHTML, this pattern is not that frequent. Nevertheless, nearly one quarter of the operations are moves.

10.4.3 Delta Size

One reason for storing the deltas instead of whole document versions is to gain a qualitative information on the changes performed. The other reason is to save storage space. However, each delta model adds data to the bare edit operation for addressing and for structuring. Additionally, the last sections have shown that the amount of edit operations used to represent the changes between document versions can become large.



(a) ODF Scenario



(b) XHTML Scenario

Figure 10.4: Some tools create deltas being larger than the compared documents.

Figure 10.4 shows the total size of the generated deltas. The size of the original document is displayed, too. The size of the delta exceeds the size of the document to compare for diffxml and jXyDiff already with minor changes. Apparently, these approaches are less suitable in terms of a storage efficiency. In general, the delta size correlates with the amount of edit operations that have been analyzed earlier.

Although XCC Diff basically has the largest overhead due to the context fingerprints and its invertability, the resulting deltas are comparably small. Especially for the ODF scenario, even large changes lead to deltas that are far smaller than the original document. The XHTML scenario leads to more comprehensive edit operations, thus requiring more storage space with respect to the original document size.

The deltas of faxma have not been analyzed in the last sections, as the delta model is not comparable. The delta size, however, can be compared. The deltas created by faxma are comparably small. Especially in the XHTML scenario, the deltas are the smallest among all tested tools. Here, the ability to address large unchanged subtrees efficiently is beneficial. In the ODF scenario, the subtrees are basically smaller and less deep. Here, faxma loses its advantage, leading to deltas nearly twice the size of XCC Diff and Microsoft XML Diff.

10.5 Conclusions

Two aspects are important when using a differencing tool. The speed of the algorithm should allow its frequent use. Additionally, the resulting deltas should be reasonably small, reflecting the nature of the changes performed.

In this chapter, I have analyzed the performance of XCC Diff in comparison with the best known tools that represent the state-of-the-art in XML differencing. The evaluation has shown that XCC Diff is one of the fastest implementations available. Together with faxma, it shows a nearly constant runtime across a high amount of changes. Even for a 375 KByte office document, the runtime does not exceed 0.5 seconds. The other tools show a significantly lower performance, together with a strange behavior in some cases. Although this might derive from a sub-optimal implementation, it significantly lowers the usability of these tools.

For the resulting deltas, the approaches differ significantly, too. Again, XCC Diff shows a very good and constant performance. The quantity of edit operations needed to describe the changes between two document versions is low, which allows for investigating a delta manually. Here, XCC Diff and jXyDiff are the only tools without freak values. Together with faxma, the size of the deltas by XCC Diff is significantly lower than for the other approaches.

In conclusion, the implementations of XCC Diff and faxma appear to be the most mature among the tested tools. XCC Diff appears to be the best general-purpose algorithm for XML documents. Faxma shows an impressive performance as well. However, its delta model is more complicated to read by human readers and does not allow for merging documents.

11 Conclusions and Future Work

This chapter summarizes the results of this thesis, followed by an overview of the main scientific contribution. Afterwards, I present different applications already using the XCC framework, as well as research directions for future work.

11.1 Summary

In the introduction in Chapter 1, I have presented my motivation for this work. I have described the ad-hoc collaboration model that can be supported by the framework developed in this thesis.

In Chapter 2, I have presented some basic notations regarding documents in general and XML documents in special. As an important aspect, XML documents need to have an inner node ordering. Although this conforms to the XML specification, some database-related XML formats intentionally rely on an unordered tree model. Deriving from that definition, most picture formats are not covered by my approach. In the rest of the chapter, I have given an analysis of XML documents to gain knowledge on their properties, and the change patterns occurring in usual editing processes. The results of this analysis have directed the design of the XCC framework later on.

The comparison of texts and trees is an important and extensively studied domain of computer science. In Chapter 3, I have presented the problem statement as well as the common solutions for text and tree differencing. Text and trees have different properties, thus requiring different solutions. XML documents, however, share the tree property from "natural" trees, as well as the reading direction from texts. Therefore, XML documents require special approaches to compare two document versions. I have given an overview on the state-of-the-art in XML differencing algorithms and tools.

Parallel editing of documents is an every-day task in collaborative environments. Changes performed in parallel have to be merged subsequently to ensure a consistent document. In Chapter 4, I have presented the common approaches to document merging. All of the existing approaches have their assets and drawbacks. I have shown that the existing approaches for XML merging are still not mature, especially for the merging of document states, which is important in ad-hoc collaboration environments without a central repository.

Document change control covers the differencing, patching, and merging of documents. In Chapter 5, I have presented the basic idea of my approach, as well as the architecture of the XCC framework that covers the mentioned aspects of document change control. Important aspects of the implementation have also been highlighted there.

To ensure a reliable merge-capability, I have decided to rely on the syntactic context as indicator for the correctness of the applicability of an edit operation. In Chapter 6, I have presented my delta model that builds the heart of the XCC framework. I have introduced the so-called *context fingerprint* that stores a normalized representation of the syntactic context.

One important challenge in document change control is the efficiency of the differencing algorithm. In Chapter 7, I have presented an efficient differencing algorithm for XML documents, called XCC Diff. The algorithm computes the changes between document versions based on the change model defined in the previous chapter. The complexity class of the algorithm is competitive. For similar documents, a near-linear time complexity is achieved. The space complexity is linear in any case.

Reliable patching is another important challenge. In Chapter 8, I have presented a merge-capable patch algorithm based on the delta model presented before. Using the context fingerprints, it is able to identify the correct position of an edit operation, thus allowing for a reliable merge. In case of a non-optimal match, the algorithm performs a best-effort approach to find the correct position. The user may decide to which degree an ambiguous merge result is acceptable. Additionally, a conflict detection prevents unintended deletions or updates. The complexity of the patch algorithm is even better than that of the differencing algorithm.

The XCC framework consists of the delta model, with the differencing and patching algorithm on top. In Chapter 9, I have evaluated the runtime of these algorithms as well as the quality of the merge. The results have enforced the claimed complexity classes and have proven the reliability of the proposed merge algorithm.

No other framework has the same broad applicability than XCC. Nevertheless, efficient XML differencing tools already exist. In Chapter 10, I have performed an analysis of the runtime and the resulting deltas of the XML differencing algorithms presented in Chapter 3 in comparison to XCC Diff. XCC Diff appears to be one of the best tools available, showing no failure or spurious result (which also holds for only one of the competitors). The deltas are small, describing the actual changes accurately.

11.2 Scientific Contribution

To the best of my knowledge, this thesis is the first comprehensive approach to XML document change control. The scientific contribution of my thesis covers three main aspects.

First, I have analyzed the properties of XML documents and their modification patterns. No other analysis of the properties of XML documents is known to me. Second, I have defined a comprehensive delta model. As a main contribution, I have developed a notion of the syntactic context of an edit operation. Altogether with the corresponding patch algorithm, it is the first delta model known to me that allows for the reliable merge of XML documents. Third, I have developed an XML differencing algorithm that is highly efficient in terms of time and space. Beside its efficiency, it creates deltas that well reflect the actual editing actions performed on the document. To the best of my knowledge, it is the best algorithm for XML differencing that relies on a complete delta model.

11.3 Applications

The XCC framework enables other programs to transparently compare and merge document versions. In this section, I present different applications of the XCC framework. I start by presenting a graphical user interface for interactive document merging. Afterwards, an auto-versioning file-system is presented that is able to automatically store modified document versions. Both tools have already been implemented. Later on, I present three further applications of the XCC framework which have not been implemented yet. Finally, I briefly discuss the applicability of my work on graphic formats.

11.3.1 Graphical Merge Support

XCC Patch allows for controlling the merge process by setting the parameters for the threshold value, the conflict detection, and the neighborhood radius. These parameters are applied to all edit operations within a delta. Defining an all-purpose threshold value is nearly impossible. Setting it high prevents erroneous merges. On the other hand, acceptable edit operations are rejected due to a low match quality. Especially for ambiguous edit operations, an interactive decision by the user is favorable over rule-based solutions.

Basing on the XCC framework, Teupel [2009] has developed a graphical editor. It allows for applying deltas interactively. The basic idea is to show the original document version, as well as the modified document side-by-side. Applied edit operations are shown directly within the XML tree, a list shows all edit operations, including the rejected ones. Figure 11.1 shows a screen-shot of the application. The original document and the updated document are shown side-by-side. Edit operations are highlighted to intuitively show their implications. In the right part of the application, a list of all edit operation of the delta is shown. Selecting one of the operations focuses the document tree on it.

The user can define a threshold for partial matchings, too. All operations with a higher match quality are accepted by default and displayed in green. Operations with a lower match quality are indicated in red. Accepted operations can be hided to allows the user to concentrate on the ambiguous operations. For each operation, the user can decide whether it should be applied or not. Additionally, the path of the operation can be changed by dragging it to the designated path within the tree. Nodes can be edited manually, e.g., for the resolution of semantic conflicts. This is a clear improvement over the conventional patching, as the document can be edited during the merge.

During the edit process, the original document remains unchanged. By this, a user may freely edit the document, without having to fear that the editing actions affect the original document. Any change can be reverted using an “undo” button. document Whenever the user has edited the document, the merge result does not conform to the delta that has been originally applied. Naturally, this is the dedicated usage of the application. Nevertheless, the user may want to store the generated merge result. The editor allows for computing the delta between the original and the modified version after editing it. By this, the resulting delta contains all decisions that have been taken by the user, including all manual changes

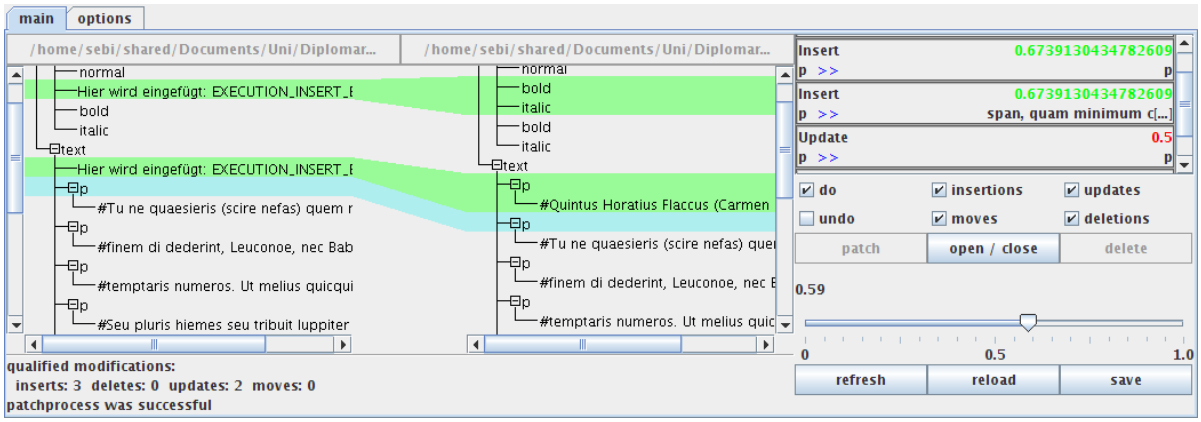


Figure 11.1: The merge GUI displays the original and the updated version side-by-side. Insertions, deletions, moves, and updates are highlighted in different colors. The right list gives detailed information on the operations of the delta.

performed. This feature is especially helpful in the context of version control systems, where only non-conflicting deltas may be committed to the repository. As a second use-case, the user can distribute the merge result to other users that are working on the same document.

11.3.2 Auto-Versioning File-System

In state-based change control systems, the presence of a new state is usually announced actively by invoking the differencing tool or by checking in the new version to the repository. This does not reflect the usual process of editing a document, where the document is loaded, edited, and saved. Users who are not familiar with the usage of change control systems would easily forget to announce the new document version. If check-ins occur only seldom, the deltas become very large and do not reflect the single editing steps, which makes it difficult to revert changes to an interim version. Additionally, large deltas are difficult to merge. In case that the most recent local copy has not been checked in and the local machine has a failure, all changes are lost.

Auto-versioning file-systems try to combine the advantages of file-systems with change control systems. The basic idea is to embed the change control features within the file-system structure in a way that each write access invokes the generation of a new document version. Müller et al. [2010] has presented a general-purpose auto-versioning file-system, which is able to handle XML documents using the XCC framework. It works on top of existing low-level file-systems on Linux systems, and handles different files depending on their MIME type [Freed and Borenstein 1996].

The auto-versioning file-system acts like a conventional one. A file and directory structure is presented. Files and documents can be created, modified, moved, and deleted as one is used to. By default, only the current state of the files is shown. Internally, the current version

is stored, as well as a delta containing the differences to the previous version. In the end, each version can be reconstructed by subsequently applying these deltas. Basically, the user has no direct access to these deltas. Instead, a tool provides an interface for querying former versions. They can be retrieved either by defining a version step (e.g., the previous version), or by defining a date of validity. Especially for older document versions, the definition of a validity period is more appropriate than defining a version step.

Legal constraints are an important issue in document management for processes in enterprises. Auto-versioning file-systems allow for achieving regulatory compliance of document repositories [Peterson and Burns 2005]. Additionally, the auto-versioning feature can be combined with an archiving solution. This especially useful for long-term archiving, where XML documents are qualified for by design [Borghoff et al. 2006]. Additionally, an auto-versioning file-system prevents the user from accidentally overwriting or deleting documents. The implicit versioning ensures a fine granularity that is also helpful for merging.

11.3.3 Other Applications

The XCC framework can be used for numerous other use cases, too. For example, version control systems could be enhanced to handle XML documents adequately. In previous work, I have presented a corresponding API to link XML differencing approaches and version control systems for the change control of office documents [Rönnau 2004]. Here, not only the basic XML document, but all XML documents within the complete office document are compared using an XML diff tool. The resulting delta contains all deltas from the diff runs. Additionally, the embedded bitmaps are compared, too. If they change, they are added to the delta. The Office Versioning API stores a complete delta and by that ensures a comprehensive change control for office documents. A prototype has shown the applicability using XyDiff (see Section 3.3.4) and the Darcs version control system[1] [Rönnau et al. 2005]. This approach is easily extensible towards the XCC framework that also provides a merge capability.

Most office applications offer a "track changes" feature that allows for storing the evolution of a document. This is technically implemented by annotating the document, as described in Section 4.4.3. This feature is popular among end-users, as it offers a simple way of manual document merging. Especially if a document is edited often or by more than two users, the change representation becomes hard to read. Additionally, the tracked changes store lots of meta-data which is probably not intended to be accessible by every participant. Finally, the change tracking is inefficient in terms of storage space. Teupel [2008] has developed a tool to map the tracked changes onto the XCC delta model. The tracked changes are extracted from the office document and converted into an XCC Delta. This way, a better privacy due to reduced meta-data information, and the ability for a more reliable merge through XCC Patch is achieved. Rönnau and Borghoff [2009] have shown the flexibility and efficiency of this approach which combines both the advantages of the change tracking as well as the context-aware merging of XCC.

[1] http://darcs.net, presented by Roundy [2005].

In Section 4.3, I have presented the operational transformation approach as state-of-the-art for collaborative editors. Fraser [2009] has presented a collaborative editor that relies on a state-based differencing and merging approach. The differencing is performed often, which results in the deltas being small and thus lowering the probability for ambiguous merges. It has been shown that the editor is fast enough for on-line collaboration with several users. Using the XCC framework, a collaborative editor for XML documents could be developed.

Recently, libraries became aware that the Web has to be archived, too [Masanès 2006]. This is especially difficult, as Web pages evolve constantly over time. Storing the complete versions of documents is way too inefficient in terms of space. Cobéna [2003] has proposed to use an XML differencing tool to compare the versions of a web page and to only store the delta. This approach is efficient in terms of space. Additionally, the use of an XML diff allows for a qualified description of the Web page evolution. The XCC framework could be easily used to track the evolution of Web pages, too. This way, a long-term preservation of Web content could be achieved efficiently.

11.3.4 Applicability on Graphics

In Section 2.1, I have restricted the term *document* to artifacts with an inner ordering, corresponding to a reading direction. This decision was based on the consideration that equally looking graphics can be represented differently. Following example is based on SVG, an XML-based vector format. Two non-touching lines can be stored in either order, providing the same information. Deriving from that, the use of XCC to track changes appears to be less meaningful. However, Thao and Munson [2010] have shown that common graphics applications store the elements of an SVG in a persistent order. This means that for two unchanged elements, their respective order within the SVG does not change in case that the SVG has been edited by the same application. Basing on this information, XCC can be used to track the evolution of graphics, which broadens the scope of the XCC framework significantly.

11.4 Future Work

The XCC framework is a mostly self-contained system that enables a full change control over XML documents. The last section has shown different applications of this approach. Neverthereless, future research may improve the proposed system. In the following, different research directions are mentioned briefly.

11.4.1 Validating Merge

In the proposed solution, the correct position of an edit operation is solely identified using its syntactic context. A possible violation of the document validity is not verified. During the evaluation, no invalid document occured. However, it is basically possible to create an invalid document in a merge scenario. This is a kind of worst case in terms of user acceptance, as

the merged document cannot be edited by the corresponding application to revise the merge error. To prevent this case, the existing patch tool offers an option to validate a document after a merge. In case of an invalid merge, the patch process is not performed.

This approach prevents invalid documents. However, a more fine-grained solution would be helpful. Here, for each edit operation, the impact on the validity has to be checked. As a re-validation of the complete document is highly cost-intensive, an incremental validation would be appropriate. In that approach, only the modified parts are re-validated. Papakonstantinou and Vianu [2002] have presented a corresponding re-validation algorithm. However, for complex grammars, the space complexity becomes a severe issue, as all possible states of the validating tree automaton have to be held in memory. Barbosa et al. [2006] have presented an improved re-validation algorithm and have shown its efficiency in complex real-world scenarios. Including this approach into XCC Patch would lead to an even better reliability of the merge, without tampering the complexity too much.

11.4.2 Three-Way Merging

Previously, I have argumented that a merge-capable patching offers the simpliest way of merging with a broad applicability. One of the drawbacks of three-way merging algorithms is the need to have the nearest common ancestor version available. In case that this version is given, however, a three-way merge potentially offers a better merge result, as each changed node can be compared with respect to the original version.

The proposed differencing algorithm could be extended to support three-way differencing, too. Here, three documents have to be traversed in parallel instead of two. Khanna et al. [2007] have analyzed the behavior of the line-based diff3 tool. This could act as blueprint for a corresponding XCC Diff3. Additionally, the experience gained by Lindholm [2004] during the design of his three-way differencing algorithm for XML documents should be considered.

11.4.3 Near-Similarity Hashing

For the sake of efficiency, nodes are compared using their value, represented by a hashed representation of their normalized content. Hash functions have mostly been developed in the domain of cryptography, where similar nodes lead to completely different hash values. By this, the hash value does not allow for drawing a deduction on the content of a node. Therefore, even a slightly changed node cannot be matched as syntactic context of an edit operation.

Minor changes of a node, e.g. a corrected typographic error or an attribute change, could be respected, indeed. This would allow the patch algorithm for matching an edit operation even in a slightly modified syntactic context. By this, even more edit operations could be matched at a higher reliabililty, as the threshold could be set to higher values. Nachbar [1988] has shown that accepting a certain inaccuracy leads to significantly smaller and understandable deltas. The use of hash values is a main design decision of my approach. Here, fuzzy hash functions that reflect the similarity of two nodes could be used. These functions have been introduced by Stein [2005].

Bibliography

Abiteboul, S., Segoufin, L., and Vianu, V. (2006). Representing and querying XML with incomplete information. *ACM Transactions on Database Systems*, 31(1):208–254.

Adams, E., Gramlich, W., Muchnick, S. S., and Tirfing, S. (1986). Sunpro: engineering a practical program development environment. In *Proceedings of the 1986 International Workshop on Advanced Programming Environments*, pages 86–96, London, UK. Springer.

Adar, E., Teevan, J., Dumais, S. T., and Elsas, J. L. (2009). The web changes everything: understanding the dynamics of web content. In *WSDM'09: Proceedings of the 2nd ACM International Conference on Web Search and Data Mining*, pages 282–291, New York, NY, USA. ACM.

Akutsu, T., Fukagawa, D., and Takasu, A. (2008). Improved approximation of the largest common subtree of two unordered trees of bounded height. *Information Processing Letters*, 109(2):165–170.

Al-Ekram, R., Adma, A., and Baysal, O. (2005). diffx: an algorithm to detect changes in multi-version XML documents. In *CASCON'05: Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 1–11. IBM Press.

Allali, J. and Sagot, M.-F. (2004). Novel tree edit operations for RNA secondary structure comparison. In Jonassen, I. and Kim, J., editors, *WABI*, volume 3240 of *Lecture Notes in Computer Science*, pages 412–425, Berlin/Heidelberg, Germany. Springer.

Amagasa, T., Yoshikawa, M., and Uemura, S. (2000). A data model for temporal XML documents. In *DEXA'00: Proceedings of the 11th International Conference on Database and Expert Systems Applications*, pages 334–344, London, UK. Springer.

Apostolico, A., Atallah, M. J., Larmore, L. L., and McFaddin, S. (1990). Efficient parallel algorithms for string editing and related problems. *SIAM Journal on Computing*, 19(5):968–988.

Barbosa, D., Leighton, G., and Smith, A. (2006). Efficient incremental validation of XML documents after composite updates. In Amer-Yahia, S., Bellahsene, Z., Hunt, E., Unland, R., and Yu, J. X., editors, *XSym*, volume 4156 of *Lecture Notes in Computer Science*, pages 107–121, Berlin/Heidelberg, Germany. Springer.

Barnard, D. T., Clarke, G., and Duncan, N. (1995). Tree-to-tree correction for document trees. Technical Report 95-372, Queen's University Kingston, Canada.

Bennett, K. H. and Rajlich, V. T. (2000). Software maintenance and evolution: a roadmap. In *ICSE'00: Proceedings of the Conference on The Future of Software Engineering*, pages 73–87, New York, NY, USA. ACM.

Bergroth, L., Hakonen, H., and Raita, T. (2000). A survey of longest common subsequence algorithms. In *SPIRE'00: Proceedings of the 7th International Symposium on String Processing Information Retrieval*, page 39, Washington, DC, USA. IEEE Computer Society.

Bernard, M., Boyer, L., Habrard, A., and Sebban, M. (2008). Learning probabilistic models of tree edit distance. *Pattern Recognition*, 41(8):2611–2629.

Bertino, E., Guerrini, G., Mesiti, M., and Tosetto, L. (2002). Evolving a set of DTDs according to a dynamic set of XML documents. In *EDBT'02: Proceedings of the Worshops XMLDM, MDDE, and YRWS on XML-Based Data Management and Multimedia Engineering-Revised Papers*, pages 45–66, London, UK. Springer.

Bille, P. (2005). A survey on tree edit distance and related problems. *Theoretical Computer Science*, 337(1-3):217–239.

Borghoff, U. M., Rödig, P., Scheffczyk, J., and Schmitz, L. (2006). *Long-Term Preservation of Digital Documents: Principles and Practices*. Springer, Secaucus, NJ, USA.

Borghoff, U. M. and Schlichter, J. H. (2000). *Computer-Supported Cooperative Work: Introduction to Distributed Applications*. Springer, Secaucus, NJ, USA.

Borghoff, U. M. and Teege, G. (1993a). Application of collaborative editing to software-engineering projects. *ACM SIGSOFT*, 18:56–64.

Borghoff, U. M. and Teege, G. (1993b). Structure management in the collaborative multimedia editing system iris. In *MMM '93: Proceedings of the 1st International Conference on Multi-Media Modeling*, pages 159–173, Singapore. Singapore, New Jersey, London, Hong Kong: World Scientific.

Boyer, J. (2001). Canonical XML version 1.0. RFC 3076.

Boyer, J. M., Dunn, E., Kraft, M., Liu, J. S., Shah, M. R., Su, H. F., and Tiwari, S. (2008). An office document mashup for document-centric business processes. In *DocEng'08: Proceeding of the 8th ACM symposium on Document engineering*, pages 100–101, New York, NY, USA. ACM.

Brauer, M., Weir, R., and McRae, M. (2007). OpenDocument v1.1 specification.

Bray, T., Hollander, D., Layman, A., and Tobin, R. (2006). Namespaces in XML 1.0 (second edition). World Wide Web Consortium, Recommendation REC-xml-names-20060816.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. (2008). Extensible markup language (XML) 1.0 (fifth edition). World Wide Web Consortium, Recommendation REC-xml-20081126.

Buckland, M. K. (1997). What is a "document"? *Journal of the American Society for Information Science*, 48(9):804–809.

Carey, P. (2008). *New Perspectives on HTML and XHTML 5th Edition, Comprehensive*. Course Technology Press, Boston, MA, United States.

Chawathe, S., Rajaraman, A., Garcia-Molina, H., and Widom, J. (1995). Change detection in hierarchically structured information. Technical Report 1995-46, Stanford Infolab.

Chawathe, S. S. (1999). Comparing hierarchical data in external memory. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 90–101, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Chawathe, S. S., Abiteboul, S., and Widom, J. (1999). Managing historical semistructured data. *Theory and Practice of Object Systems*, 5(3):143–162.

Chawathe, S. S. and Garcia-Molina, H. (1997). Meaningful change detection in structured data. *SIGMOD Records*, 26(2):26–37.

Chawathe, S. S., Rajaraman, A., Garcia-Molina, H., and Widom, J. (1996). Change detection in hierarchically structured information. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD International Conference on the Management of Data*, pages 493–504, New York, NY, USA. ACM.

Chen, W. (2001). New algorithm for ordered tree-to-tree correction problem. *Journal of Algorithms*, 40(2):135 – 158.

Clark, J. (1999). XSL transformations (XSLT) version 1.0. http://www.w3.org/TR/xslt.

Cobéna, G. (2003). *Change management of semi-structured data on the web*. PhD thesis, École Polytechnique, INRIA Rocquencourt.

Cobéna, G., Abiteboul, S., and Marian, A. (2002). Detecting Changes in XML Documents. In *ICDE'02: Proceedings of the 18th International Conference on Data Engineering*, pages 41–52. IEEE Computer Society.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Commununications of the ACM*, 13(6):377–387.

Conradi, R. and Westfechtel, B. (1998). Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282.

Cormode, G. and Muthukrishnan, S. (2007). The string edit distance matching problem with moves. *ACM Transactions on Algorithms*, 3(1):2.

Davis, A. H., Sun, C., and Lu, J. (2002). Generalizing operational transformation to the standard general markup language. In *CSCW'02: Proceedings of the 2002 ACM Conference on Computer Supported Cooperative Work*, pages 58–67, New York, NY, USA. ACM.

Davis, M. and Collins, L. (1990). Unicode. In *Proceedings of the 1990 International Conference on Systems, Man, and Cybernetics*, pages 499–504.

Davison, W. (1990). Unified context diff tools. Volume 14, Issue 70 of `comp.sources.misc`.

Delisle, N. and Schwartz, M. (1986). Contexts: a partitioning concept for hypertext. In *CSCW'86: Proceedings of the 1986 ACM Conference on Computer Supported Cooperative Work*, pages 147–152, New York, NY, USA. ACM.

DeNardis, L. and Tam, E. (2007). Open documents and democracy. http://odfalliance.org/resources/Yale_Open_Documents_and_Democracy.pdf.

DeRose, S. and Clark, J. (1999). XML path language (XPath) version 1.0. http://www.w3.org/TR/1999/REC-xpath-19991116.

Eastlake, D., Reagle, J., Solo, D., Hirsch, F., and Roessler, T. (2008). XML signature syntax and processing (second edition). http://www.w3.org/TR/xmldsig-core/.

Eisenberg, J. D. (2004). OpenOffice.org XML Essentials - Using OpenOffice.org's XML Data Format. Technical report, SUN.

Ellis, C. A. and Gibbs, S. J. (1989). Concurrency control in groupware systems. *SIGMOD Records*, 18(2):399–407.

Fetterly, D., Manasse, M., Najork, M., and Wiener, J. (2003). A large-scale study of the evolution of web pages. In *WWW'03: Proceedings of the 12th International Conference on the World Wide Web*, pages 669–678, New York, NY, USA. ACM.

Flower, L. and Hayes, J. R. (1981). A cognitive process theory of writing. *College Composition and Communication*, 32(4):365–387.

Fontaine, R. L. (2002). Merging XML files: a new approach providing intelligent merge of XML data sets. In *Proceedings of the XML Europe 2002*.

Fraser, N. (2009). Differential synchronization. In *DocEng'09: Proceedings of the 9th ACM Symposium on Document Engineering*, pages 13–20, New York, NY, USA. ACM.

Freed, N. and Borenstein, N. (1996). Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. RFC 2046.

Goldfarb, C. F. (1990). *The SGML handbook*. Oxford University Press, Inc., New York, NY, USA.

Gorn, S., Bemer, R. W., and Green, J. (1963). American standard code for information interchange. *Communications of the ACM*, 6(8):422–426.

Gottlob, G., Koch, C., and Pichler, R. (2003). The complexity of XPath query evaluation. In *PODS'03: Proceedings of the 22nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 179–190, New York, NY, USA. ACM Press.

Grandi, F. and Mandreoli, F. (2000). The valid web: An XML/XSL infrastructure for temporal management of web documents. In *ADVIS'00: Proceedings of the 1st International Conference on Advances in Information Systems*, pages 294–303, London, UK. Springer.

Grandi, F., Mandreoli, F., and Tiberio, P. (2005). Temporal modelling and management of normative documents in XML format. *Data & Knowledge Engineering*, 54(3):327–354.

Gray, J. (1978). Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK. Springer.

Gropengießer, F., Hose, K., and Sattler, K.-U. (2009). An extended transaction model for cooperative authoring of XML data. *Computer Science - Research and Development*, 24(1-2):85–100.

Hedeler, C. and Paton, N. W. (2008). A comparative evaluation of XML difference algorithms with genomic data. In Ludäscher, B. and Mamoulis, N., editors, *SSDBM*, pages 258–275, Berlin/Heidelberg, Germany. Springer.

Hickson, I. and Hyatt, D. (2010). HTML5. http://www.w3.org/TR/html5/.

Hill, W. C., Hollan, J. D., Wroblewski, D., and McCandless, T. (1992). Edit wear and read wear. In *CHI'92: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3–9, New York, NY, USA. ACM.

Hors, A. L., Hégaret, P. L., Wood, L., Nicol, G., Robie, J., Champion, M., and Byrve, S. (2004). Document object model (DOM) level 3 core specification. W3C Recommendation.

Horwitz, S., Prins, J., and Reps, T. (1989). Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387.

Hottinger, D. and Meyer, F. (2005). XML-diff-algorithmen. Semesterarbeit, ETH Zürich.

Hunt, J. W. and McIlroy, M. D. (1976). An algorithm for differential file comparison. Technical Report CSTR 41, Bell Laboratories, Murray Hill, NJ.

Ignat, C.-L. and Norrie, M. C. (2003). Customizable collaborative editor relying on treeopt algorithm. In *ECSCW'03: Proceedings of the 8th European Conference on Computer Supported Cooperative Work*, pages 315–334, Norwell, MA, USA. Kluwer Academic Publishers.

Ignat, C.-L. and Norrie, M. C. (2006). Flexible definition and resolution of conflicts through multi-level editing. In *CollaborateCom'06: Proceedings of the International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pages 1 –10.

Ignat, C.-L. and Norrie, M. C. (2008). Multi-level editing of hierarchical documents. *Comput. Supported Coop. Work*, 17(5-6):423–468.

Ignat, C.-L., Papadopoulou, S., Oster, G., and Norrie, M. C. (2008). Providing awareness in multi-synchronous collaboration without compromising privacy. In *CSCW'08: Proceedings of the 2008 ACM Conference on Computer Supported Cooperative Work*, pages 659–668, New York, NY, USA. ACM.

Iorio, A. D., Schirinzi, M., Vitali, F., and Marchetti, C. (2009). A natural and multi-layered approach to detect changes in tree-based textual documents. In Filipe, J. and Cordeiro, J., editors, *ICEIS*, volume 24 of *Lecture Notes in Business Information Processing*, pages 90–101, Berlin/Heidelberg, Germany. Springer.

Irons, E. T. and Djorup, F. M. (1972). A CRT editing system. *Communications of the ACM*, 15(1):16–20.

Ishida, Y., Inenaga, S., Shinohara, A., and Takeda, M. (2005). Fully incremental LCS computation. In Liskiewicz, M. and Reischuk, R., editors, *FCT*, volume 3623 of *Lecture Notes in Computer Science*, pages 563–574, Berlin/Heidelberg, Germany. Springer.

Jansson, J. and Lingas, A. (2001). A fast algorithm for optimal alignment between similar ordered trees. In *CPM'01: Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, pages 232–240, London, UK. Springer.

Jiang, T., Wang, L., and Zhang, K. (1995). Alignment of trees – an alternative to tree edit. *Theoretical Computer Science*, 143(1):137 – 148.

Kangasharju, J. and Lindholm, T. (2005). A sequence-based type-aware interface for XML processing. In *EuroIMSA'05: Proceedings of the Conference on Internet and Multimedia Systems and Applications*, pages 83–88.

Khanna, S., Kunal, K., and Pierce, B. C. (2007). A formal investigation of diff3. In Arvind and Prasad, editors, *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*.

Klein, P. N. (1998). Computing the edit-distance between unrooted ordered trees. In *ESA '98: Proceedings of the 6th Annual European Symposium on Algorithms*, pages 91–102, London, UK. Springer.

Ko, H.-K. and Lee, S. (2006). An efficient scheme to completely avoid re-labeling in XML updates. In *WISE'06: Proceedings of the 7th International Conference on Web Information Systems Engineering*, pages 259–264.

Koch, M. (1997). *Unterstützung kooperativer Dokumentenbearbeitung in Weitverkehrsnetzen*. PhD thesis, Technische Universität München.

Koch, M. and Koch, J. (2000). Application of frameworks in groupware – the iris group editor environment. *ACM Computing Surveys*, page 28.

Krug, S. (2000). *Don't Make Me Think!: A Common Sense Approach to Web Usability*. Que Corp., Indianapolis, IN, USA.

Lai, W. (2009). Relationship-Based Change Propagation: A Case Study. Master's thesis, University of Toronto.

Landau, G. M., Myers, E. W., and Schmidt, J. P. (1998). Incremental string comparison. *SIAM Journal on Computing*, 27(2):557–582.

Laux, A. and Martin, L. (2000). XUpdate. http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html.

Lease, M. (2007). Natural language processing for information retrieval: the time is ripe (again). In *PIKM'07: Proceedings of the ACM 1st Ph.D. workshop in CIKM*, pages 1–8, New York, NY, USA. ACM.

Lee, K.-H., Choy, Y.-C., and Cho, S.-B. (2004). An efficient algorithm to compute differences between structured documents. *IEEE Transactions on Knowledge and Data Engineering*, 16(8):965–979.

Levenshtein, V. (1966). Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707–710.

Lindholm, T. (2004). A three-way merge for XML documents. In *DocEng'04: Proceedings of the 4th ACM Symposium on Document Engineering*, pages 1–10, New York, NY, USA. ACM.

Lindholm, T., Kangasharju, J., and Tarkoma, S. (2006). Fast and simple XML tree differencing by sequence alignment. In *DocEng'06: Proceedings of the 6th ACM Symposium on Document Engineering*, pages 75–84, New York, NY, USA. ACM.

Lippe, E. and van Oosterom, N. (1992). Operation-based merging. *SIGSOFT Software Engineering Notes*, 17(5):78–87.

MacKenzie, D., Eggert, P., and Stallmann, R. (2002). Comparing and merging files. http://www.gnu.org/software/diffutils/manual/.

Marian, A., Abiteboul, S., Cobéna, G., and Mignet, L. (2001). Change-centric management of versions in an XML warehouse. In *Journal on Very Large Databases*, pages 581–590.

Martens, W., Neven, F., Schwentick, T., and Bex, G. J. (2006). Expressiveness and complexity of XML schema. *ACM Transactions on Database Systems*, 31(3):770–813.

Maruyama, H., Tamura, K., and Uramoto, N. (2000). Digest values for DOM (DOMHASH). RFC 2803.

Masanès, J. (2006). *Web Archiving*. Springer, Secaucus, NJ, USA.

Mehdad, Y. (2009). Automatic cost estimation for tree edit distance using particle swarm optimization. In *ACL-IJCNLP'09: Proceedings of the ACL-IJCNLP 2009 Conference Short Papers*, pages 289–292, Morristown, NJ, USA. Association for Computational Linguistics.

Mens, T. (2002). A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462.

Meyrowitz, N. and van Dam, A. (1982). Interactive editing systems: Part i. *ACM Computing Surveys*, 14(3):321–352.

Miller, W. and Myers, E. W. (1985). A file comparison program. *Software: Practice and Experience*, 15(11):1025–1040.

Mitchell, A., Posner, I., and Baecker, R. (1995). Learning to write together using groupware. In *CHI'95: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 288–295, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

Molli, P., Skaf-Molli, H., Oster, G., and Jourdain, S. (2002). Sams: synchronous, asynchronous, multi-synchronous environments. In *Proceedings of the 7th International Conference on Computer Supported Cooperative Work in Design*, pages 80 – 84.

Morgan, H. L. (1970). Spelling correction in systems programs. *Communications of the ACM*, 13(2):90–94.

Mouat, A. (2002). XML Diff and Patch Utilities. Master's thesis, Heriot-Watt University, Edinburgh.

Müller, A., Rönnau, S., and Borghoff, U. M. (2010). A file-type sensitive, auto-versioning file system. In *Proceedings of the 10th ACM symposium on Document engineering*, DocEng '10, pages 271–274, New York, NY, USA. ACM.

Murata, M., Lee, D., Mani, M., and Kawaguchi, K. (2005). Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Intermet Technology*, 5(4):660–704.

Myers, B. A. (1998). A brief history of human-computer interaction technology. *ACM Interactions*, 5(2):44–54.

Myers, E. W. (1986). An O(ND) difference algorithm and its variations. *Algorithmica*, 1:251–266.

Nachbar, D. (1988). SPIFF — A program for making controlled approximate comparisons of files. In *USENIX Association*, pages 73–84.

Neuwirth, C. M., Chandhok, R., Kaufer, D. S., Erion, P., Morris, J., and Miller, D. (1992). Flexible diff-ing in a collaborative writing system. In *CSCW'92: Proceedings of the 1992 ACM Conference on Computer Supported Cooperative Work*, pages 147–154, New York, NY, USA. ACM Press.

Neuwirth, C. M., Kaufer, D. S., Chandhok, R., and Morris, J. H. (1990). Issues in the design of computer support for co-authoring and commenting. In *CSCW'90: Proceedings of the 1990 ACM Conference on Computer Supported Cooperative Work*, pages 183–195, New York, NY, USA. ACM.

Neven, F. (2002). Automata theory for XML researchers. *SIGMOD Records*, 31(3):39–46.

Nichols, D. A., Curtis, P., Dixon, M., and Lamping, J. (1995). High-latency, low-bandwidth windowing in the jupiter collaboration system. In *UIST'95: Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, pages 111–120, New York, NY, USA. ACM.

Nielsen, J. and Pernice, K. (2009). *Eyetracking Web Usability*. New Riders Press.

Nørvåg, K. (2002). Temporal query operators in XML databases. In *SAC'02: Proceedings of the 2002 ACM Symposium on Applied Computing*, pages 402–406, New York, NY, USA. ACM.

Ntoulas, A., Cho, J., and Olston, C. (2004). What's new on the web?: the evolution of the web from a search engine perspective. In *WWW'04: Proceedings of the 13th International Conference on the World Wide Web*, pages 1–12, New York, NY, USA. ACM.

Olson, J. S., Olson, G. M., Mack, L. A., and Wellner, P. (1990). Concurrent editing: the group's interface. In *INTERACT'90, Proceedings of the IFIP TC13 3rd Interantional Conference on Human-Computer Interaction*, pages 835–840.

Oster, G., Molli, P., Urso, P., and Imine, A. (2006a). Tombstone transformation functions for ensuring consistency in collaborative editing systems. In *Proceedings of the International Conference on Collaborative Computing: Networking, Applications and Worksharing*, page 38, Los Alamitos, CA, USA. IEEE Computer Society.

Oster, G., Urso, P., Molli, P., and Imine, A. (2006b). Data consistency for p2p collaborative editing. In *CSCW'06: Proceedings of the 20th Anniversary Conference on Computer Supported Cooperative Work*, pages 259–268, New York, NY, USA. ACM.

Paoli, J., Valet-Harper, I., Farquhar, A., and Sebestyen, I. (2006). ECMA-376 Office Open XML File Formats.

Papadopoulou, S., Ignat, C., Oster, G., and Norrie, M. (2006). Increasing awareness in collaborative authoring through edit profiling. In *Proceedings of the International Conference on Collaborative Computing: Networking, Applications and Worksharing*, page 35, Los Alamitos, CA, USA. IEEE Computer Society.

Papakonstantinou, Y. and Vianu, V. (2002). Incremental validation of XML documents. In *ICDT'03: Proceedings of the 9th International Conference on Database Theory*, pages 47–63, London, UK. Springer.

Pauli, C. (2008). Konzeption und Anwendung von XML-Deltas mit Kontextinformationen. Diplomarbeit, Universität der Bundeswehr, München.

Pearson, P. K. (1990). Fast hashing of variable-length text strings. *Commununications of the ACM*, 33(6):677–680.

Pemberton, S. (2002). XHTML™ 1.0 the extensible hypertext markup language (second edition). http://www.w3.org/TR/2002/REC-xhtml1-20020801.

Peters, L. (2005). Change detection in XML trees: a survey. In *Proceedings of the 3rd Twente Student Conference on IT*, Enschede.

Peterson, Z. and Burns, R. (2005). Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage*, 1(2):190–212.

Philipp, G. (2008). Implementierung eines Fingerprint-gestützten performanten XML-Patch-Tools. Studienarbeit, Universität der Bundeswehr, München.

Philipp, G. (2009). Ein effizientes Diff für XML-Dokumente. Diplomarbeit, Universität der Bundeswehr, München.

Pilato, M. (2004). *Version Control With Subversion*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Pohlemann, M. (2009). Eine automatisierte Testumgebung für XML-Versionierungssysteme. Studienarbeit, Universität der Bundeswehr, München.

Preguiça, N. M., Shapiro, M., and Matheson, C. (2003). Semantics-based reconciliation for collaborative and mobile environments. In *DOA/CoopIS/ODBASE'03: Proceedings of the Confederated International Conferences DOA, CoopIS and ODBASE*, pages 38–55.

Rabin, M. O. (1981). Fingerprinting by random polynomials. Technical Report TR-CSE-03-01, Center for Research in Computing Technology, Harvard University.

Rivest, R. (1992). The MD5 Message-Digest Algorithm. RFC 1321.

Rochkind, M. J. (1975). The source code control system. *IEEE Transactions on Software Engineering*, 1(4):364–370.

Rönnau, S. (2004). Versionsverwaltung von XML-Dokumenten am Beispiel von OpenOffice. Diplomarbeit, Universität der Bundeswehr München.

Rönnau, S. and Borghoff, U. M. (2009). Versioning XML-based office documents. *Multimedia Tools and Applications*, 43(3):253–274.

Rönnau, S., Pauli, C., and Borghoff, U. M. (2008). Merging changes in XML documents using reliable context fingerprints. In *DocEng'08: Proceeding of the 8th ACM Symposium on Document Engineering*, pages 52–61, New York, NY, USA. ACM.

Rönnau, S., Scheffczyk, J., and Borghoff, U. M. (2005). Towards XML version control of office documents. In *DocEng'05: Proceedings of the 5th ACM Symposium on Document Engineering*, pages 10–19, New York, NY, USA. ACM.

Rosado, L. A., Márquez, A. P., and Gil, J. M. (2007). Managing branch versioning in versioned/temporal XML documents. In Barbosa, D., Bonifati, A., Bellahsene, Z., Hunt, E., and Unland, R., editors, *XSym*, volume 4704 of *Lecture Notes in Computer Science*, pages 107–121, Berlin/Heidelberg, Germany. Springer.

Roundy, D. (2005). Darcs: distributed version management in haskell. In *Haskell'05: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, pages 1–4, New York, NY, USA. ACM.

Sankoff, D. (1972). Matching sequences under deletion-insertion constraints. *Proceedings of the National Academy of Sciences of the United States of America*, 69(1):4–6.

Scheffczyk, J. (2004). *Consistent Document Engineering*. PhD thesis, Universität der Bundeswehr München.

Schlichter, J., Koch, M., and Bürger, M. (1998). Workspace awareness for distributed teams. In *Coordination Technology for Collaborative Applications*, number 1364 in Lecture Notes in Computer Science (LNCS), pages 199–219, Berlin/Heidelberg, Germany. Springer.

Schubert, E., Schaffert, S., and Bry, F. (2005). Structure-preserving difference search for XML documents. In *Proceedings of the Conference on Extreme Markup Languages*.

Selkow, S. M. (1977). The tree-to-tree editing problem. *Information Processing Letters*, 6(6):184–186.

Shapiro, B. A. (1988). An algorithm for comparing multiple RNA secondary structures. *Computer Applications in the Biosciences*, 4(3):387–393.

Shen, H. and Sun, C. (2002). Flexible merging for asynchronous collaborative systems. In *DOA/CoopIS/ODBASE'02: Proceedings of the Confederated International Conferences DOA, CoopIS and ODBASE*, pages 304–321, London, UK. Springer.

Smith, R. (1988). GNU diff3.

Song, Y., Bhowmick, S. S., and Dewey, C. F. (2007). BioDIFF: An effective fast change detection algorithm for biological annotations. In Ramamohanarao, K., Krishna, P. R., Mohania, M. K., and Nantajeewarawat, E., editors, *DASFAA*, volume 4443 of *Lecture Notes in Computer Science*, pages 275–287, Berlin/Heidelberg, Germany. Springer.

Stein, B. (2005). Fuzzy-fingerprints for text-based information retrieval. In *I-KNOW'05: Proceedings of the 5th International Conference on Knowledge Management*, pages 572–579. Journal of Universal Computer Science.

Strunk Jr., W. and White, E. B. (1979). *The Elements of Style*. Macmillan, New York, NY, 3rd ed. edition.

Sun, C. and Sosič, R. (1999). Optimal locking integrated with operational transformation in distributed real-time group editors. In *PODC '99: Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 43–52, New York, NY, USA. ACM.

Tai, K.-C. (1979). The tree-to-tree correction problem. *Journal of the ACM*, 26(3):422–433.

Tatarinov, I., Viglas, S. D., Beyer, K., Shanmugasundaram, J., Shekita, E., and Zhang, C. (2002). Storing and querying ordered XML using a relational database system. In *SIGMOD'02: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 204–215, New York, NY, USA. ACM.

Teupel, M. (2008). Konzeption und Entwicklung von Konvertern für Fingerprint-basierte XML-Deltas. Studienarbeit, Universität der Bundeswehr, München.

Teupel, M. (2009). Eine interaktive Merge-GUI für XML-Dokumente. Diplomarbeit, Universität der Bundeswehr, München.

Thao, C. and Munson, E. V. (2010). Using versioned tree data structure, change detection and node identity for three-way xml merging. In *Proceedings of the 10th ACM symposium on Document engineering*, DocEng '10, pages 77–86, New York, NY, USA. ACM.

The ODF Alliance (2008). ODF Annual Report 2008.

The Unicode Consortium (1991). *The Unicode Standard: Worldwide Character Encoding. Version 1.0. Volumes 1 and 2.* Addison-Wesley Professional, Reading, MA, USA.

Tichy, W. F. (1982). Design, implementation, and evaluation of a revision control system. In *ICSE '82: Proceedings of the 6th International Conference on Software Engineering*, pages 58–67, Los Alamitos, CA, USA. IEEE Computer Society Press.

Tichy, W. F. (1984). The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, 2(4):309–321.

Tichy, W. F. (1985). RCS—a system for version control. *Software: Practice and Experience*, 15(7):637–654.

Toland, T. S. (2000). An information retrieval system to manage program maintenance reports in a data processing shop. In *ACM-SE'38: Proceedings of the 38th Annual Southeast Regional Conference*, pages 81–87, New York, NY, USA. ACM.

Touzet, H. (2005). A linear tree edit distance algorithm for similar ordered trees. In *CPM'05: Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching*, pages 334–345, Berlin/Heidelberg, Germany. Springer.

Tridgell, A. (1999). *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, The Australian National University.

Valiente, G. (2001). An efficient bottom-up distance between trees. In *SPIRE'01: Proceedings of the 8th International Symposium on String Processing and Information Retrieval*, pages 212–219.

Vion-Dury, J.-Y. (2010). Diffing, patching and merging xml documents: toward a generic calculus of editing deltas. In *Proceedings of the 10th ACM symposium on Document engineering*, DocEng '10, pages 191–194, New York, NY, USA. ACM.

Wagner, R. A. and Fischer, M. J. (1974). The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173.

Wall, L. (1985). patch version 1.3. posted to `mod.sources` on May, 24.

Wang, D. and Mah, A. (2009). Google wave operational transformation. http://www.waveprotocol.org/whitepapers/operational-transform.

Wang, F. and Zaniolo, C. (2008). Temporal queries and version management in XML-based document archives. *Data & Knowledge Engineering*, 65(2):304–324.

Wang, F., Zhou, X., and Zaniolo, C. (2006). Bridging relational database history and the web: the xml approach. In *WIDM'06: Proceedings of the 8th Annual ACM International Workshop on Web Information and Data Management*, pages 3–10, New York, NY, USA. ACM.

Wang, L. and Zhang, K. (2005). Space efficient algorithms for ordered tree comparison. In Deng, X. and Du, D.-Z., editors, *ISAAC*, volume 3827 of *Lecture Notes in Computer Science*, pages 380–391, Berlin/Heidelberg, Germany. Springer.

Wang, Y., DeWitt, D., and Cai, J. (2003). X-Diff: A fast change detection algorithm for XML documents. In *ICDE'03: Proceedings of the International Conference on Data Engineering*.

Weir, R. (2009). OpenDocument Format: The standard for office documents. *IEEE Internet Computing*, 13:83–87.

Xu, H., Wu, Q., Wang, H., Yang, G., and Jia, Y. (2002). KF-Diff+: Highly efficient change detection algorithm for XML documents. In *DOA/CoopIS/ODBASE'02: Proceedings of the Confederated International Conferences DOA, CoopIS and ODBASE*, pages 1273–1286, London, UK. Springer.

Zhang, K. (1996a). A constrained edit distance between unordered labeled trees. *Algorithmica*, 15(3):205–222.

Zhang, K. (1996b). Efficient parallel algorithms for tree editing problems. In *CPM '96: Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, pages 361–372, London, UK. Springer.

Zhang, K. and Shasha, D. (1989). Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262.

Zhang, K., Wang, J. T.-L., and Shasha, D. (1995). On the editing distance between undirected acyclic graphs and related problems. In *CPM'95: Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, pages 395–407.

Zimmermann, T., Kim, S., Zeller, A., and Whitehead, Jr., E. J. (2006). Mining version archives for co-changed lines. In *MSR'06: Proceedings of the 2006 International Workshop on Mining Software Repositories*, pages 72–75, New York, NY, USA. ACM.

A Resources for Document Evaluation

This chapter lists the documents used in the document analysis in Chapter 2, including their on-line sources.

A.1 Selection Criteria

The document analysis shall provide a deeper insight in common Web and office documents to allow for the design of a suitable change control architecture. Therefore, the analysed documents must cover a wide range of real-world scenarios. In my opinion, synthetically created documents are far less suitable than publicly available documents that are commonly used. Therefore, I have selected office documents from public repositories and Web documents from major Web sites.

A.2 Web Documents

The selected Web documents cover four categories:

- Auction sites
- News sites
- Governmental sites
- Blogs

The reason for selecting these categories is motivated as follows. Auction sites are very dense, as they try to display as much information as possible on the screen to attract a broad audience. News sites may be far larger, spanning across multiple screen sizes. Here, I also included stock news sites that are characterized by a high structuring, e.g. by large tables. Governmental sites are usually highly structured and do not contain advertising. Blogs usually contain more text and less items. All of these document types are frequently used on the Web.

Most of these documents have originally been HTML documents. I have converted them to XHTML using the tool *tidy*[1]. Table A.1 lists the analyzed documents.

[1] `http://tidy.sourceforge.net`

Site	URL	Type
eBay	www.ebay.com	auction site
Taobao	www.taobao.com	auction site
My Hammer	www.my-hammer.de	auction site
Bridge2B	www.bridge2b.com	auction site
The New York Times	www.nytimes.com	news site
The Wall Street Journal	online.wsj.com	news site
The Economist	www.economist.com	news site
Wired	www.wired.com	news site
Heise Verlag	www.heise.de	news site
MaxBlue	www.maxblue.de	stock news
OnVista	www.onvista.de	stock news
The CIA	www.cia.gov	governmental site
The Pentagon	pentagon.afis.osd.mil	governmental site
Die Bundesregierung	www.bundesregierung.de	governmental site
Die Bundeswehr	www.bundeswehr.de	governmental site
Lenovo Blogs	www.lenovoblogs.com	corporate blog
The Apple Blog	www.theappleblog.com	corporate blog
Ich Werde Ein Berliner	www.ichwerdeeinberliner.com	private blog

Table A.1: 18 Web pages with different type have been analyzed.

A.3 Office Documents

Office documents may have an arbitrary content and structure. Whereas Web documents are practically limited in content due to the screen size, office document may reach several hundreds of pages. This requires to analyze more documents to get an overview of different applications. Nevertheless, my evaluation cannot give a comprehensive analysis of all possible office documents.

As I focus my research on texts and spreadsheets, I have taken several documents from public repositories. The text documents are taken from the OpenOffice.org documentation repository at documentation.openoffice.org. In this repository, two types of documents are available. Example documents that show the possibilites of OpenOffice (which cover 1-10 pages) and a software documentation. These documents are rather large and contain up to 125 pages. Table A.2 lists the analyzed documents

The analyzed spreadsheets are financial calculations. They range from a simple dept rate calculation with only several cells up to a comprehensive financial models for business companies with 13 different sheets. All spreadsheets are are freely available at www.exinfm.com/free_spreadsheets.html. Table A.3 gives an overview of the analyzed documents.

Office documents may be nested within each other. For example, a table within a text document would be represented as (short) sub-document which could also stand alone. In

Title	Size in KByte	Subdocuments
Code optimization in the Word Processor	52.6	0
Creating Large Documents with OpenOffice.org Writer	377.2	0
Curriculum Vitae	38.1	0
German Letter (DIN-Brief) Template	10.4	0
How to Create and Maintain a Table of Contents	113.1	0
How to Create Columns	57.5	0
How-to dynamically link an image in a Base Form	13.6	0
How-to Get to Grips with OpenOffice.org Draw	731.5	46
How to Modify the Context Menu in OfficeBean	63.5	0
How to Work with Sections	151.7	0
How to Work with Templates	113.0	0
OpenOffice API Training Guide	755.2	0
Personal Letter Cover	9.8	0
Personal References	13.0	0
Personal Resume	28.0	0
Sample Text Article	5.7	0

Table A.2: 16 text documents have been analyed.

the analysis, each sub-document is treated as an independent document, as it has an own root node. By this, not only 67, but 178 documents have been analyzed.

Title	Size in KByte	Subdocuments
Amortization Table	716.6	0
Automation Justification	633.6	8
Capital Budgeting Template	47.2	0
Car Lease Analysis	70.9	0
Cash Flow Analysis	159.0	0
Cash Flow Business Valuation	123.9	0
Cash Flow Matrix	232.5	0
Cash Flow Matrix 2	120.7	0
Cash Gap Days Analysis	130.9	0
CFROI Valuation Model Audit	552.2	0
Cost Estimating Template	567.4	0
Decision Making & Analysis Framework	598.4	2
Debt Calculation	45.0	0
Economic Value Added (EVA)	452.0	0
Entrepreneur Financial Model	16444.2	0
Equity Analysis	125.3	0
EVA Tree Model	189.8	1
FCFE Stable Growth Model	116.2	1
Financial Analysis & Forecasting	1742.3	34
Financial Projections Model	1870.6	0
Implied Risk Premium Calculator	36.2	0
Inflation and the Real Rate of Interest	58.9	0
IT Risk Assesment Template	206.8	0
Merger & LBO Valuation	279.5	0
Option Trading Workbook	730.1	11
Project Management Toolkit	1808.8	7
Risk Register	377.3	2
Synthetic Rating Estimation	61.8	0
Synergy Valuation Worksheet	84.3	0
Valuation Model Assesment	83.9	0
WACC Calculation	59.6	0
What-If Calculation	178.9	1
What-If Model	893.7	0

Table A.3: 33 Spreadsheets covering a large range of sizes have been analyed.

B XML Document Hashing

The XCC framework uses hash values to compare XML nodes and subtrees. This chapter describes the normalization methods used, altogether with the hashing algorithms.

B.1 Normalization

XML provides the ability to encode one and the same semantic meaning in different syntactic ways. Therefore, two nodes that ought to be compared must be normalized first. The normalization covers following aspects: namespace resolution, attribute ordering, character encoding, and white-space normalization. I will briefly describe each of these aspects, including the solution used. My approach mostly conforms to the CanonicalXML specification by Boyer [2001].

B.1.1 Namespace Resolution

XML allows for using elements from different XML grammars within one document. Namespaces are used to enable the XML processor to distinguish between these different grammars, specified by Bray et al. [2006]. Each node has to be assigned to one namespace. References can be used to avoid syntactic overhead, as shown in the following example:

```
<office:document-content
    xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
    xmlns:xforms="http://www.w3.org/2002/xforms">
    <office:body>
        <xforms:form/>
    </office:body>
</office:document-content>
```

Here, the document contains two namespaces, one for office content (based on ODF) and one for forms (based on XForms). For both namespaces, a reference (named `office` and `forms`) is defined at the root node. The root itself is part of the ODF grammar, indicated by the leading `office:`. The inner node, representing an empty form, is part of XForms. However, the use of a reference is not mandatory. The following document would be fully equivalent:

```
<office:document-content
    xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0">
```

```
    <office:body>
        <http://www.w3.org/2002/xforms:form/>
    </office:body>
</office:document-content>
```

In this example, the inner node has been described using its *qualified name*, which means that the reference at the beginning has been resolved. In my approach, each node is hashed upon its qualified name to avoid ambiguousities.

B.1.2 Attributes

Attributes of an element node may be listed in arbitrary order. For example, the nodes `<elem fst="1" snd="2"/>` and `<elem snd="2" fst="1"/>` are fully equivalent. In my approach, all attributes are alphabetically ordered before hashing. Additionally, elements may contain attributes from different namespaces. According to the last section, these attributes are hashed upon their qualified name, too.

B.1.3 Encoding

XML documents are required to be encoded in Unicode, which is an encoding standard that aims to support all existing characters on the World [The Unicode Consortium 1991]. However, different Unicode standards exist. In the Western hemisphere, UTF-8 is the most common standard which is designed for the efficient encoding of Roman characters. In the Eastern hemisphere, UTF-16 is more common which also supports Chinese and Japanese characters efficiently (but is in turn less efficient for the Roman alphabet as its size is far smaller than that of the Eastern alphabets).

Most characters have a different representation in UTF-8 and UTF-16. If two strings are compared from different encodings, one of them has to be translated into the other one. In my approach, all strings are represented in UTF-16, as this encoding is more efficient for a World-wide use.

B.1.4 White-Space Normalization

In XML, white-space like blanks and newlines are treated as essential information. However, white space is often used to structure the XML tree for better readability, also called *pretty printing*. According to the XML standard, a pretty printed document is not equivalent to its original representation. Nevertheless, my approach allows for removing any leading and trailing white-space in text nodes, thus normalizing pretty printed documents. This feature is not mandatory and disabled by default.

Node Type	Byte Signature
Element	0001
Attribute	0002
Text Node	0003
Processing Instruction	0007

Table B.1: Different node types are identified by adding a leading byte signature.

B.2 XML Hashing

After transferring the nodes into a normalized representation they can be hashed. My approach supports the hashing of single nodes as well as whole subtrees and tree sequences. First, I describe how nodes are assembled before hashing, followed by the definition of the recursive hashing. After that, I present the hashing algorithms used.

B.2.1 Node Assembly

I already defined how single nodes and attributes are normalized. If a node is hashed, the different parts of it have to be assembled into one byte array to allow the hash algorithm for computing the hash value.

XML knows different node types. Two nodes with identical name but different node type differ semantically. This has to be respected by the hash function. Therefore, for each node, a leading type signature is added. Table B.1 shows the signatures used in my approach. They conform to the signatures used by Maruyama et al. [2000]. After marking the node types, the node is assembled into one byte array, starting with the qualified node name and the attributes in normalized order.

B.2.2 Recursive Hashing

My approach allows for computing hash values over entire subtrees and tree sequences. The hashing is performed recursively. First, each leaf is hashed. After that, these hash values are added in their respective order to the parent node, separated by the byte array 00. The complete byte array is hashed again. This way, the whole subtree can be hashed recursively. For a tree sequence, the hash value is computed over all root elements of the respective subtrees (which are already recursively hashed).

B.2.3 Hash Algorithms

Basically, any hash algorithm can be used to compute a hash value. The current implementation of the XCC framework allows for two different algorithms: MD5 and FNV.

MD5 [Rivest 1992] is widely established as fast and reliable hash algorithm with efficient implementations on most platforms. It is a cryptographic hash function that computes hash values of 128bit.

Cryptographic hash functions are especially designed for a high orthogonality of similar values to prevent attacks on the encryption. In my use case, it suffices that changes are detected by the hash algorithm. A high orthogonality is not necessary. Therefore, the current implementation uses the FNV hash algorithm by default[1]. This algorithm does not meet cryptographic requirements, but is highly efficient to compute. It resembles the hash algorithm presented by Pearson [1990] and is used in many scenarios where large data sets have to be hashed efficiently[2]. The current implementation relies on 32bit hash values.

B.3 Related Work

Normalization has already been discussed earlier, leading to the definition of CanonicalXML by [Boyer 2001]. My approach is an extended version of it, as my approach allows for omitting trailing and leading white-space.

Hashing of XML has been previously discussed, leading to the definitions of DOMHash by Maruyama et al. [2000] and XML-Signature by Eastlake et al. [2008]. DOMHash assigns a hash value to each XML element, where the hashes are computed recursively over the subsequent tree of that element. Hence, a change in one node affects all nodes on the path up to the root element. A single node within the tree cannot be hashed, which does not meet my requirements. Furthermore, DOMHash does not allow for the addressing of text nodes directly. If a text node shall be hashed, its parent node must be addressed, too.

XML-Signature was designed to create a signature for a whole document or single parts of it. This approach has some major drawbacks for my scenario, too: First, each signature holds a substantial header. Second, the header must contain a path expression, which defines the signed part of the document or must contain the part itself. The latter is obviously useless when using hash values for space saving reasons. The former is not very useful either, because in my use case, a fingerprint should be matched with its counterpart which has probably moved in terms of an absolute path.

[1] Fowler / Noll / Vo (FNV) Hash, `http://www.isthe.com/chongo/tech/comp/fnv/index.html`
[2] http://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function

C XCC Delta Specification

This chapter defines the file format of the context-oriented deltas used by the XCC framework. The rules are defined in natural language.

1. The root node is called `delta`.
2. The delta may contain an arbitrary number of operations that are named `insert`, `delete`, or `update`.
3. Each operation contains following attributes:
   a) `path` contains the path to the edit operation
   b) `radius` denotes the radius of the fingerprint as natural number.
   c) `digester` refers to the hashing algorithm used. It can turn into `fnv` or `md5`.
   d) Insert and delete operations may have the additional attribute `move` that contains the ID of the move operation.
4. Each operation consists of following elements:
   a) The `fingerprint` contains a text node that consists of the hash values of the fingerprint in hexadecimal notation, separated by a semicolon.
   b) The subtree / tree sequence to delete or the node to update is stored as child of `oldvalue`. In case of an insert operation, this element may be omitted.
   c) The subtree / tree sequence to insert or the root of the new value of the updated node is stored as child of `newvalue`. In case of a delete operation, this element may be omitted.

Figure C.1 shows an example delta that is computed in the example in Appendix D.

This definition is comprehensive, yet not very formal. One might expect a formal definition, e.g. in EBNF or XML Schema. However, this is not very useful for the following reason: the content of the edit operations is not known in advance. This content must be validated against the grammar of the compared document. Even if this grammar would be known in advance. these parts could not been validated as they consist only of a fraction of the document. For insert and delete operations, a corresponding production rule could be found. As update operations do only store the updated node, its descendants are not part of the delta, which would in turn lead to a broken validation. For these reasons, even if I would provide a more formal definition, it could not be used for validation. Therefore, I omit this step.

```
delta
├─ insert digester="fnv" path="/2" radius="4"
│  ├─ fingerprint
│  │  └─ c387e295;7a514983;662e3aec;bd3e2f21;48ebf6de;d7ac76f0;0e284785;9a31afad;09ec381d
│  └─ newvalue
│     └─ p style="bold"
│        └─ We got two papers accepted
├─ delete digester="fnv" path="/6" radius="4"
│  ├─ fingerprint
│  │  └─ 662e3aec;1334ff2d;662e3aec;e1388821;c6948277;e1d08bf3;e1d08bf3;e1d08bf3;e1d08bf3
│  └─ oldvalue
│     └─ p
│        └─ DocEng papers are submitted
├─ update digester="fnv" path="/2/0/0" radius="4"
│  ├─ fingerprint
│  │  └─ 662e3aec;bd3e2f21;d7ac76f0;0e284785;9a31afad;09ec381d;03ee618f;ea603659;c015f5c8
│  ├─ oldvalue
│  │  └─ column orientation="left" title="Conference"
│  └─ newvalue
│     └─ column orientation="center" title="Conference"
├─ update digester="fnv" path="/2/1/3/0" radius="4"
│  ├─ fingerprint
│  │  └─ a9b8034c;746ebc4a;a600cea3;746ebc4a;40894c87;c015f5c8;746ebc4a;47b30b11;746ebc4a
│  ├─ oldvalue
│  │  └─ pending
│  └─ newvalue
│     └─ accepted
├─ update digester="fnv" path="/2/2/3/0" radius="4" shortened="true"
│  ├─ fingerprint
│  │  └─ 834f6e47;746ebc4a;226afc6a;746ebc4a;40894c87;c015f5c8;746ebc4a;eb17031c;746ebc4a
│  ├─ oldvalue
│  │  └─ pending
│  └─ newvalue
│     └─ accepted
└─ update digester="fnv" path="/4/0" radius="4"
   ├─ fingerprint
   │  └─ d8f56d3b;e98a5cfe;e864cfac;662e3aec;1334ff2d;662e3aec;e1388821;662e3aec;ea2cb0ac
   ├─ oldvalue
   │  └─ DocEng notification is late.
   └─ newvalue
      └─ DocEng reviews are there. Revise until July, 8.
```

Figure C.1: An example delta containing insert, delete, and update operations.

D A Running Example

In this chapter, I show the procedure of differencing, patching, and merging on an example documents. The main goal is to give a deeper understanding of the algorithms used in the XCC framework.

D.1 Setting

The example document contains a list of research activities and is shown in Figure D.1. This document A is changed independently into two versions, namely A_1 and A_2. Both versions are displayed in Figure D.2. From the user model, all changes are non-interfering. Figure D.3 shows the corresponding XML tree, including all changes performed by A_1 and A_2. The presented tree has been highly simplified for exemplifying the essentials of the XCC algorithms; it does not conform to a common XML document format.

In the following, I will exemplify the computation of the deltas $\delta_{A\to A_1}$ and $\delta_{A\to A_2}$, as well as the merging by applying $\delta_{A\to A_2}$ to A_1.

Research in 2010

This document describes our research in 2010

Conference	Paper Type	Title	Status
DocEng 2010	Demo	Document Archival via Auto-Versioning File-Systems	pending
DocEng 2010	Short Paper	A File-Type Sensitive, Auto-Versioning File System	pending
Springer CSRD	Journal	XCC: Change Control of XML Documents	under revision

Comments

DocEng notification is late
CSRD reviews are there
DocEng papers submitted in time

Figure D.1: The user representation of the example document.

Research in 2010

This document describes our research in 2010
We got two papers accepted

Conference	Paper Type	Title	Status
DocEng 2010	Demo	Document Archival via Auto-Versioning File-Systems	accepted
DocEng 2010	Short Paper	A File-Type Sensitive, Auto-Versioning File System	accepted
Springer CSRD	Journal	XCC: Change Control of XML Documents	under revision

Comments

DocEng reviews are there. Revise until July, 8.
CSRD reviews are there

(a) A_1 adds a new line at the top, changes the orientation of the first column, changes the status of two papers, and updates the comments.

Research in 2010

This document describes our research in 2010

Venue	Paper Type	Title	Status
DocEng 2010	Demo	Document Archival via Auto-Versioning File-Systems	pending
DocEng 2010	Short Paper	A File-Type Sensitive, Auto-Versioning File System	pending
Springer CSRD	*Journal*	XCC: Change Control of XML Documents	submitted

Comments

Submitted revised CSRD paper
DocEng notification is late
CSRD reviews are there
DocEng papers submitted in time

(b) A_2 changes the title of the first column, changes style and status for one paper, and adds a line to the comments.

Figure D.2: The document is edited independently, leading to two versions, A_1 and A_2.

D.2 Differencing

The changes between A and A_1 comprise the insertion of a subtree, leaf updates, and the deletion of a subtree.

In the first step, the LCS of all leaves is computed, resulting in the matchlist shown in Table D.1. The leaves that are only part of A indicate a deletion (see Table D.2). Table D.3 shows the leaves that are only part of A_1, indicating an insertion.

After computing the LCS, the parent nodes of all matched node pairs are compared for

Figure D.3: The XML representation of the original document, showing the operations performed in both versions.

Matched Leaves
Research in 2010
This document describes our research in 2010
`column orientation="left" title="Paper Type"`
`column orientation="left" title="Title"`
`column orientation="center" title="Status"`
DocEng 2010
Demo
Document Archival via Auto-Versioning File-Systems
DocEng 2010
Short Paper
A File-Type Sensitive, Auto-Versioning File System
Springer CSRD
Journal
XCC: Change Control of XML Documents
under revision
Comments
CSRD reviews are there

Table D.1: Leaves that exist in both versions.

Leaves only in A
`column orientation="left" title="Conference"`
submitted
submitted
DocEng notification is late
DocEng papers submitted in time

Table D.2: Leaves that are possibly deleted.

Leaves only in A_1
We got two papers accepted
`column orientation="center" title="Conference"`
accepted
accepted
DocEng reviews are there. Revise until July, 8.

Table D.3: Leaves that are possibly inserted.

updates or changes in parent-children relationship. Here, no change occurred. All parent nodes are marked visited, as shown in Figure D.4. This finishes the second step of the diff algorithm.

In the third step, structure-affecting changes are identified. One of the key issues of this step is the identification of leaf updates. Basically, leaf updates are detected by comparing the siblings of the non-matched nodes. In this example, however, only the leaf `column orientation="center" title="Conference"` has siblings that could be compared. The other non-matching leaves are text nodes, which often have no siblings. In that case, the algorithm compares the parent nodes. If they match, a leaf update is estimated. In the given example, all parent nodes match, leading to the creation of leaf updates for three leaves.

The first unmatched leaf of A_1 ("We got two papers accepted "), however, has no counterpart in A. This also holds for its parent node. Therefore, an insert operation rooted at the parent node is created. The same procedure is performed for the unmatched leaf of A ("DocEng papers submitted in time"), which is estimated as deletion.

The complete delta $\delta_{A \to A_1}$ using FNV hashes and a fingerprint radius of 4 is shown in Figure D.5. Here, the hash values of the delete show some important aspects of the delta model. First, the deletion occurs at the border of the document. Therefore, the fingerprint is filled with a special *null node*, which has the hash value `e1d08bf3`. Second, all fields right of the anchor (which is the middle field) refer to the null node, although the anchor refers to path `/6`, which means that there is one node left in document order (its child node). Here, all the elements that are addressed by the delete operation have been removed, so that the root of the subtree to delete appears to be the last node of the document. This has been done to prevent the fingerprint to refer to the nodes to delete, which would match anyway in a non-conflicting merge scenario, thus lowering the information value of the fingerprint, especially for larger subtrees to delete.

The computation of $\delta_{A \to A_2}$ is nearly the same. Although, there is one difference. In this scenario, a non-leaf update occurs at path `/2/3/1`, where the attribute *style* with value *italic* is inserted. Figure D.6 shows the complete delta.

D.3 Merging

After computing both deltas, $\delta_{A \to A_2}$ shall be applied to A_1 to merge both document versions. When comparing the paths of the edit operations in both deltas, one could easily determine a conflict on node `/2/0/0`. In the following, I will exemplify the behavior of my patch procedure during merging this update operation.

The patch algorithms identifies path `/2/0/0` in $\delta_{A \to A_2}$ and tries to compute the fingerprint around this node in the document to patch. Due to the insertion of a subtree at the beginning of A_1, the path `/2/0/0` is not existing any more. Therefore, the patch algorithm has to compute the neighborhood around that path. Here, I recall that in the current setting, only nodes on the same top-down level are respected by the neighborhood.

Basically, the neighborhood is computed in both directions of the document order. In that

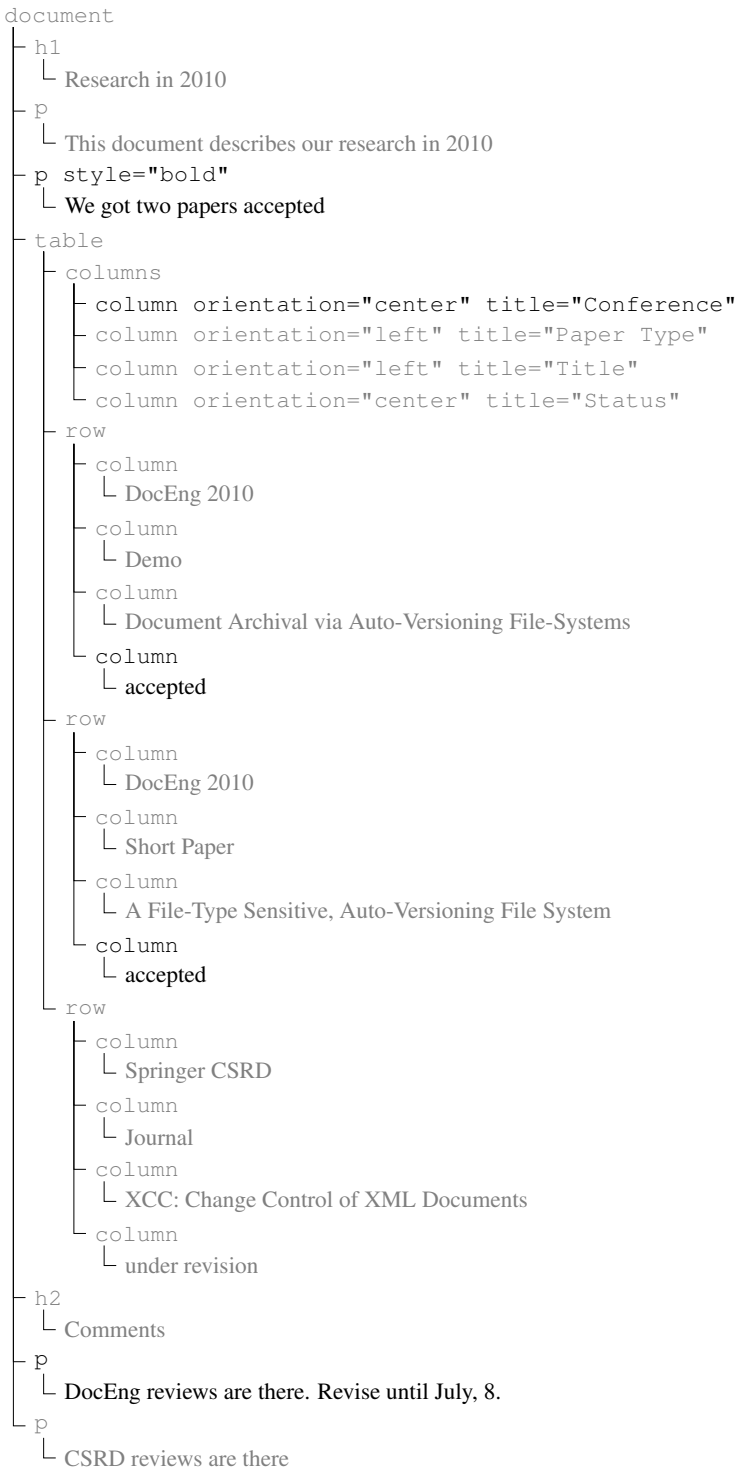


Figure D.4: After the first bottom-up traversal, most nodes are already visited (showing the tree of A_1, visited nodes in gray).

```
delta
├ insert digester="fnv" path="/2" radius="4"
│ ├ fingerprint
│ │ └ c387e295;7a514983;662e3aec;bd3e2f21;48ebf6de;d7ac76f0;0e284785;9a31afad;09ec381d
│ └ newvalue
│   └ p style="bold"
│     └ We got two papers accepted
├ delete digester="fnv" path="/6" radius="4"
│ ├ fingerprint
│ │ └ 662e3aec;1334ff2d;662e3aec;e1388821;c6948277;e1d08bf3;e1d08bf3;e1d08bf3;e1d08bf3
│ └ oldvalue
│   └ p
│     └ DocEng papers are submitted
├ update digester="fnv" path="/2/0/0" radius="4"
│ ├ fingerprint
│ │ └ 662e3aec;bd3e2f21;d7ac76f0;0e284785;9a31afad;09ec381d;03ee618f;ea603659;c015f5c8
│ ├ oldvalue
│ │ └ column orientation="left" title="Conference"
│ └ newvalue
│   └ column orientation="center" title="Conference"
├ update digester="fnv" path="/2/1/3/0" radius="4"
│ ├ fingerprint
│ │ └ a9b8034c;746ebc4a;a600cea3;746ebc4a;40894c87;c015f5c8;746ebc4a;47b30b11;746ebc4a
│ ├ oldvalue
│ │ └ pending
│ └ newvalue
│   └ accepted
├ update digester="fnv" path="/2/2/3/0" radius="4" shortened="true"
│ ├ fingerprint
│ │ └ 834f6e47;746ebc4a;226afc6a;746ebc4a;40894c87;c015f5c8;746ebc4a;eb17031c;746ebc4a
│ ├ oldvalue
│ │ └ pending
│ └ newvalue
│   └ accepted
└ update digester="fnv" path="/4/0" radius="4"
  ├ fingerprint
  │ └ d8f56d3b;e98a5cfe;e864cfac;662e3aec;1334ff2d;662e3aec;e1388821;662e3aec;ea2cb0ac
  ├ oldvalue
  │ └ DocEng notification is late.
  └ newvalue
    └ DocEng reviews are there. Revise until July, 8.
```

Figure D.5: The complete delta $\delta_{A \to A_1}$.

```
delta
├ insert digester="fnv" path="/4" radius="4"
│ ├ fingerprint
│ │ └ 746ebc4a;d8f56d3b;e98a5cfe;e864cfac;1f7f4b76;662e3aec;1334ff2d;662e3aec;e1388821
│ └ newvalue
│   └ p
│     └ Submitted revised CSRD paper
├ update digester="fnv" path="/2/3/1" radius="4"
│ ├ fingerprint
│ │ └ 40894c87;c015f5c8;746ebc4a;eb17031c;746ebc4a;b0c694e7;746ebc4a;387a9b01;746ebc4a
│ ├ oldvalue
│ │ └ column
│ └ newvalue
│   └ column style="italic"
├ update digester="fnv" path="/2/0/0" radius="4"
│ ├ fingerprint
│ │ └ 662e3aec;bd3e2f21;d7ac76f0;0e284785;9a31afad;09ec381d;03ee618f;ea603659;c015f5c8
│ ├ oldvalue
│ │ └ column orientation="left" title="Conference"
│ └ newvalue
│   └ column orientation="left" title="Venue"
└ update digester="fnv" path="/2/3/3/0" radius="4"
  ├ fingerprint
  │ └ b0c694e7;746ebc4a;387a9b01;746ebc4a;d8f56d3b;e98a5cfe;e864cfac;662e3aec;1334ff2d
  ├ oldvalue
  │ └ under revision
  └ newvalue
    └ submitted
```

Figure D.6: The complete delta $\delta_{A \to A_2}$.

special case, however, no node with the same depth exists before the node. Therefore, the neighborhood can only consist of nodes coming after the stored path. In this example, only one of the nodes of the neighborhood provide a partial matching for the given fingerprint, as shown in Figure D.7. The patch algorithm expects this path to be the correct one. However, the anchor node does not match due to the conflict. Therefore, the edit operation is not applied to that document.

Insert operations cannot rely on their anchor for determining the correct insert position. Here, only the surrounding nodes stored in the delta can be used. Figure D.8 shows how the patch procedure deals with the insert operation of $\delta_{A \to A_2}$. Once again, the path points to the wrong address, caused by the insertion at the top of the document. Therefore, the patch procedure creates the neighborhood around path `4`. As you can see, the neighborhood covers nearly the whole document, as only nodes of the same depth are considered as possible insert path. The best matching is given at `/5`, which is in fact the correct path. However, it does not match completely, as the subsequent paragraph has been updated.

After the complete application of $\delta_{A \to A_2}$ to A_1, the merged document is shown in Figure D.9.

Figure D.7: Merging the conflicting change.

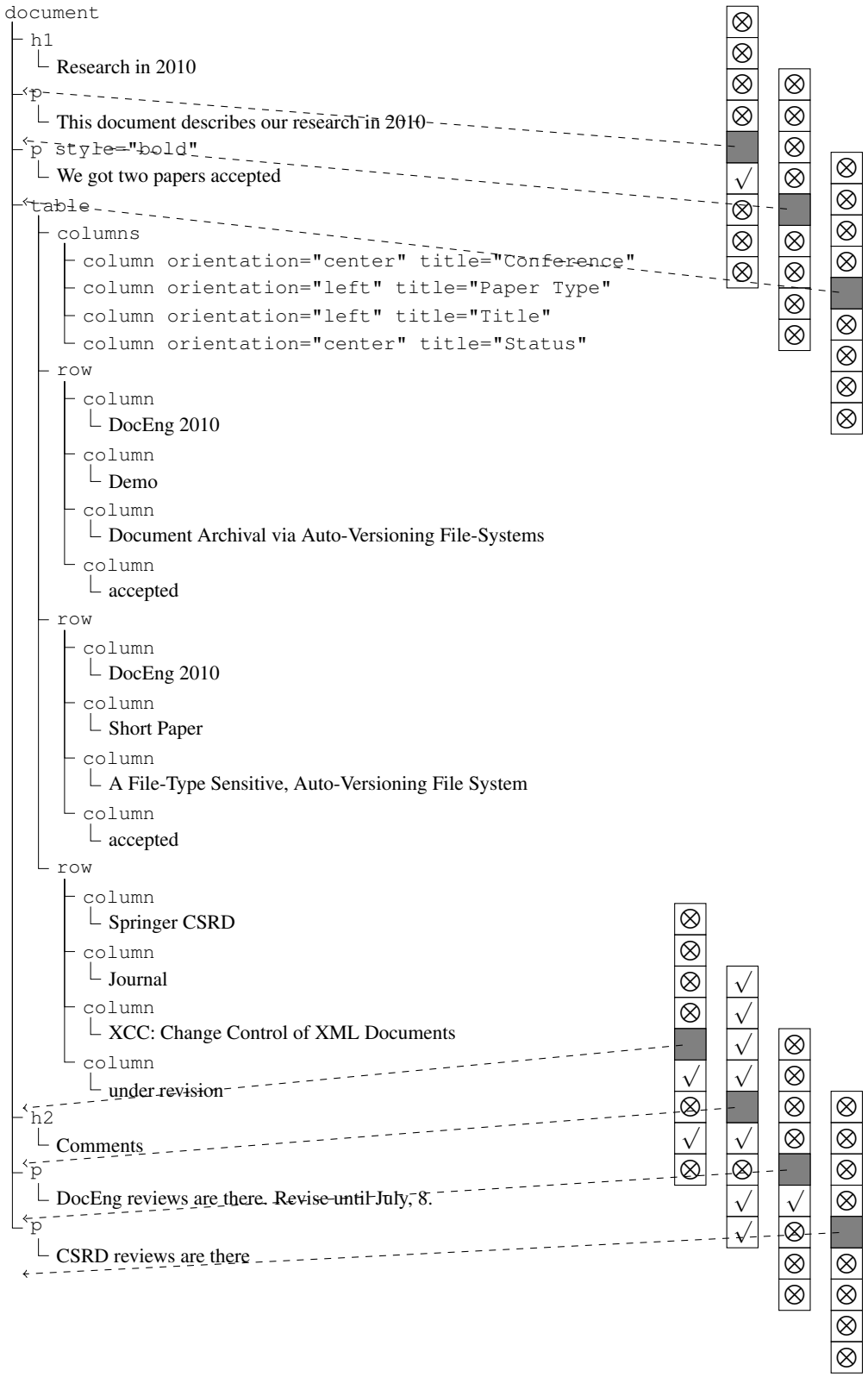


Figure D.8: Merging the insert operation.

Figure D.10 shows the corresponding user representation of it. As you can see, all changes except the conflicting one have been applied.

D.4 Conclusions

In this chapter, I have shown a running example of the XCC framework. The computation of the deltas has been exemplified, as well as the merging afterwards. In this example, one edit operations conflicts, which is detected by XCC Patch. This conflict could be used manually, or by overwriting the conflicting nodes. In case of conflicts, however, merging is not symmetric. In the proposed example, merging A_2 with $\delta_{A \to A_1}$ would lead to another merge result w.r.t. the conflicting node.

```
document
├ h1
│ └ Research in 2010
├ p
│ └ This document describes our research in 2010
├ p style="bold"
│ └ We got two papers accepted
├ table
│ ├ columns
│ │ ├ column orientation="left" title="Venue"
│ │ ├ column orientation="left" title="Paper Type"
│ │ ├ column orientation="left" title="Title"
│ │ └ column orientation="center" title="Status"
│ ├ row
│ │ ├ column
│ │ │ └ DocEng 2010
│ │ ├ column
│ │ │ └ Demo
│ │ ├ column
│ │ │ └ Document Archival via Auto-Versioning File-Systems
│ │ └ column
│ │   └ accepted
│ ├ row
│ │ ├ column
│ │ │ └ DocEng 2010
│ │ ├ column
│ │ │ └ Short Paper
│ │ ├ column
│ │ │ └ A File-Type Sensitive, Auto-Versioning File System
│ │ └ column
│ │   └ accepted
│ └ row
│   ├ column
│   │ └ Springer CSRD
│   ├ column style="italic"
│   │ └ Journal
│   ├ column
│   │ └ XCC: Change Control of XML Documents
│   └ column
│     └ submitted
├ h2
│ └ Comments
├ p
│ └ Submitted revised CSRD paper
├ p
│ └ DocEng reviews are there. Revise until July, 8.
└ p
  └ CSRD reviews are there
```

Figure D.9: The merged document in XML representation.

Research in 2010

This document describes our research in 2010
We got two papers accepted

Conference	Paper Type	Title	Status
DocEng 2010	Demo	Document Archival via Auto-Versioning File-Systems	accepted
DocEng 2010	Short Paper	A File-Type Sensitive, Auto-Versioning File System	accepted
Springer CSRD	*Journal*	XCC: Change Control of XML Documents	submitted

Comments

Submitted revised CSRD paper
DocEng reviews are there. Revise until July, 8.
CSRD reviews are there

Figure D.10: The merged document in its user model.