XML Diff and Patch Tool

Kyriakos Komvoteas

Supervisor Dr. Joe Wells

September 5, 2003

Master of Science
in
Distributed and Multimedia Information Systems

Abstract

The increasing use of XML the last few years, led to the creation of many differencing and patching tools capable of handling tree-structured documents. However, all of those tools are only able to apply a delta file, which is the collection of the differences between two documents, in one of the original files. This document accompanies an application to create a differencing and patching tool that handles XML documents as ordered labelled trees, and will attempt to overcome the limitation stated above. Moreover, this thesis introduces a new difference output format, named EDUL, and thoroughly documents all the special cases that cause conflicts during the patching.

Acknowledgements

First of all, I would like to thank my supervisor Dr. Joe Wells for his guidance and support throughout the course of this project.

Moreover, I would like to thank my parents and my sister for supporting me all those years.

Special thanks to Vanias for proof-reading the document, and the rest of my friends for putting up with me all those years.

Thank you all,

Kyriakos Komvoteas

Table of Contents

Chapter 1

Introduction

The primary aim of this thesis was to examine all issues that are related to the differencing and patching of arbitrary XML documents. Moreover, the results of this study were used in order to develop an open-source tool, able to fulfil the original requirement. The major contributions of this thesis are summarised in the following paragraph.

First of all, this thesis documents all the requirements for the differencing and patching tool as well as the delta file, which is a file that describes the differences between two documents (a further discussion of delta files is help in section 2.4). Furthermore, a new difference output format, named EDUL, is thoroughly described in this document and used for the implementation of the tool. Moreover, in chapter 4, all the cases where conflicts occur during the patching are thoroughly examined, and the desired behaviour of the tool in all of those cases is clearly described. Finally, a tool that meets the requirements stated and handles properly the special cases was developed.

In order to publish the current work and make it possible for other developers to read and continue this project, the author of this document came into contact with the *SourceForge* [26] Open Source software development environment. After describing the goals of this project as well as the current state of the tool that was developed, they offered some space in their server in order to present the current work to the world. Consequently, this document, as well as the tool and the source code of the application are available on-line at *http://treepatch.sourceforge.net.*

The remainder of this document is structured as follows. This chapter presents some significant notions that will be used throughout this document.

The second chapter introduces the reader to the Extensible Markup Language, known as XML, and explains how XML documents can be represented as tree structures. Moreover, various existing tools for differencing and patching documents are presented. Then, the most interesting difference output formats of those tools are discussed. Finally, a range of differencing algorithms is presented as well as information about patching for line and tree oriented documents.

The following chapter states both the functional and non-functional requirements of the tool that was developed along with this document. The functional requirements are divided into three sections; the requirements of the differencing tool, the requirements of the delta file and the requirements of the patching tool.

Chapter four can be qualified both as the continuation of the requirements or the beginning of the design. It is concerned with the identification of the desired behaviour of the patching tool in special cases where conflicts may occur. A further discussion of this desired behaviour is held in the following chapters.

The design of the tool that was developed during the dissertation is discussed in chapter 5. First of all, a new difference output format, named EDUL, is fully described. This chapter also includes a detailed discussion about context information that will be included in the delta file. After that, the main algorithms of the tool are explained.

In the sixth chapter the implementation of the application is described. All the technologies that were used in order to build the tool are considered. There is also a description of the main classes of the source code as well as the manual pages of the differencing and patching tool.

The following chapter categorises and lists all the tests that we carried out during the design and the implementation of the tools. Moreover, the results of those tests are presented and commented.

Finally, the last chapter evaluates the whole dissertation, discussing both the strength and the weaknesses of the differencing and patching tool. In addition, possible extensions of the tool are suggested.

Keywords

differencing, patching, difference output format, EDUL, XML, arbitrary files, difference output format, ordered labelled trees

1.1 Preliminaries

This section familiarises the reader with some notions that will be used throughout this paper. Firstly, the concepts of differencing, merging and patching are discussed. Afterwards, the differences between line-oriented and tree-oriented comparison of documents are explained.

1.1.1 Differencing

The process of identifying the differences between two documents can be divided in two parts. First, the differences have to be identified in an efficient way and then those differences have to be outputted in a useful format. In the following chapter, a variety of difference output formats will be presented, and their advantages as well as their drawbacks will be pointed out.

1.1.2 Merging and Patching

After locating and outputting the differences between two files, those files are usually combined. In order to do so, the files can be either merged or patched. In the first case the two files are used to create a new version that includes the changes that exist in both of them. In case of patching, the differences between those two files and either one of the original or a third file are used to generate the final version.

1.1.3 Line-oriented and tree-oriented

There are a lot of programs -like UNIX commands- that are able to merge and patch documents in a satisfactory way. However, those programs handle all the documents as a series of lines without taking into account their structure. On the other hand, there are tools that take advantage of the tree structure of some files, and therefore they are said to make a tree oriented comparison.

Chapter 2

Background and Related Work on Differencing and Patching

In this chapter, the Extensible Markup Language and its relation with the tree structure will be explained. Afterwards, some of the existing differencing, merging and patching tools both for line-oriented and tree-oriented files will be explored. Moreover, the most interesting difference output formats of the tools discussed previously will be evaluated. Finally, a wide range of algorithms that have been used in order to find differences between files and patch them successfully will be examined.

2.1 Background on XML

2.1.1 XML – Extensible Markup Language

XML stands for Extensible Markup Language and is leaded by the World Wide Consortium [33]. In fact, XML is a *metalanguage* that allows the programmer to define its own markup languages depending on the application he/she is developing. XML is a simplified subset of the Standardised Generalised Markup Language (SGML).

XML manages to separate the structure and the content of a document from its presentation. Consequently, the same document can be presented in many different ways –in different applications- and it seems that it will become a standardised way

for exchanging data between heterogeneous systems. A simple example of an XML document is shown in Figure 2-1.

```
<?xml version=”1.0”?>
<library>
  <book>
    <title>Java 2 – Black Book</title>
    <author>Steven Holzner</author>
    <year>2001</year>
  </book>
  <book>
    <title>Database Systems</title>
    <author>McGraw – Hill</author>
    <year>2003</year>
  </book>
</library>
```

Figure 2-1 : An Example of XML Document

A Document Type Definition or DTD is a formal description of a particular type of document. It defines the names of all elements, how many times those appear and in what order. DTDs can be part of the XML documents or may exist in a separate file. It is possible that an XML document does not have any associated DTD file, but in this case it is hard to verify that the structure of your document is correct. An alternative to DTD is the XML Schema that is more extensible than the former and easier to handle because is written in XML. However, DTDs and Schemas will not be considered further in this thesis.

In order for an XML file to be valid it has to conform to a specific Document Type Definition (DTD). Moreover, a well-formed XML document follows some general rules of XML. Those rules include that all elements and attributes are case sensitive and that tags must not overlap.

2.1.2 Tree Structure and XML

The most suitable structure for representing XML data is trees, because all the XML elements are strictly nested. All the trees that we will consider in this dissertation are

rooted, labelled and ordered, which mean that they have exactly one root node, all the nodes are associated with a value and the order of the siblings is significant. Thereafter whenever we refer to a tree we imply a rooted, ordered, labelled one.

In order to map XML documents to trees we had to make some premises. First of all, the nodes represent either elements or values of the document. More precisely the root element is the root node of the tree, all the internal nodes represent the elements of the XML document and the leaves are the values of the elements. For example in a XML file that represent a book, the internal nodes will represent the elements "chapter" and the "paragraph" whereas the leaves the actual text of the book. Finally, the attributes of the elements will be the attributes of the nodes. All of those issues will be reconsidered later on, in the design of current tool. The tree mapping of the XML document in Figure 2-1 is shown in Figure 2-2.

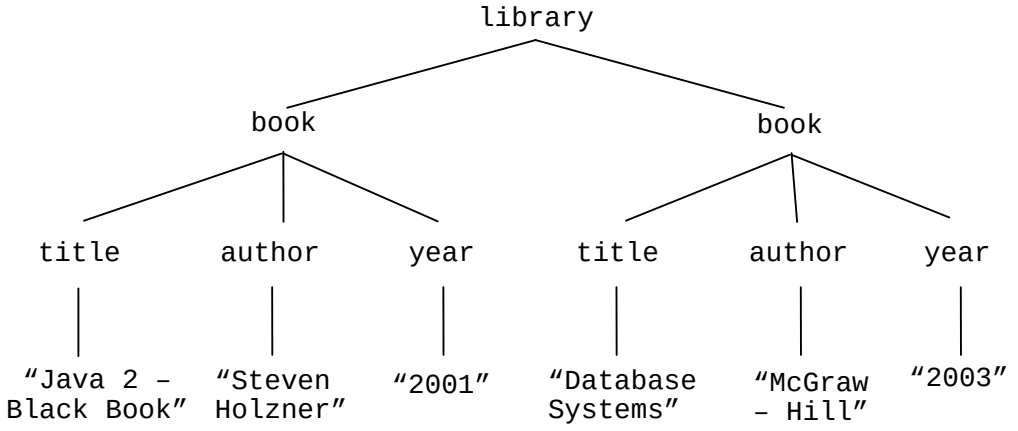

Figure 2-2: The Tree Representation of the XML Document in Figure 1.1

2.2 Differencing, Merging and Patching Tools

The differencing, merging and patching tools can be divided into two main categories: line-oriented and tree-oriented tools. Although at the beginning the tools handled the files as a series of lines, as HTML and later on XML became an important part of the World Wide Web, there was a tendency to create tools for handling tree-structure documents.

A discussion about the line-oriented UNIX commands –they were used and tested in Linux- that are relevant with the differencing, merging or patching of files will open this section. After that, the Revision Control System and the Concurrent Version System as well as the Emacs editor and his features will be considered. Finally, a closer look was taken at the tools that are specifically designed for XML differencing, merging and patching. The discussion will include the following tools; deltaXML, XyDiff, IBM's XML Diff and Merge tool, XML TreeDiff, Microsoft's XML Diff and Patch tool and VM tools.

2.2.1 UNIX Commands

UNIX provides a wide range of line-oriented programs that allow comparing, merging and patching different versions of a document [16, 25]. Some of those programs, like psmerge that merges several postscript files, are not covered in this thesis as they have little relevance with our area of research. A number of different commands are discussed below.

diff

Diff program is able to compare and output the differences of two files or directories. The output of *diff* is called delta file and describes the differences between the two files. Each group of different lines that is included in the delta file is called a hunk. The comparison of the files is line based, so *diff* program does not take advantage of the structure of the files. The user may choose to ignore whitespaces, blank lines and case sensitivity during the comparison.

It is interesting that the output of the comparison can be presented in many formats. The user can choose to output an rcs-format, an ed script, a side-to-side comparison, a context or a unified output. In this thesis a particular interest is given to the context output format, where a few lines of context before and after each change are included in the delta file. Finally, *diff* program is capable of running two different algorithms: the first one is faster whereas the latter is more accurate.

patch

The *patch* program takes as input the delta file that was created by the *diff* command, and one of the original files or a third file to create a patched file. Usually the patched file overwrites the original version and the latter is backed up to a file with the same name and an *.orig* suffix.

With context diffs, p*atch* can identify when the line that is mentioned is incorrect and tries to locate the correct line. This is not possible with other output diff formats because patch does not have enough context information (a further discussion about line oriented patching and its algorithm is held on section 2.5.1). Finally, in case that *patch* is unable to find a line that seems to be correct, it creates a reject file. The reject file is usually named after the original's file name with a .rej extension.

diff3

The main functions of this UNIX command are three, specified by the option the user chooses. First of all, *diff3* is able to find the differences between three files. Moreover, it can generate an *ed script* that describes what editing should be performed to one of the files in order to generate the other. Finally, it is possible to merge two files, informing the user for any possible conflict that may have occurred.

No matter how useful the *diff3* command might be, it only handles files as plain text. Although it is also applicable to tree-structured files like XML files, it fails to produce optimal results.

sdiff

Sdiff is able to merge two files interactively. The two files are printed in the standard output one beside the other, and the user has to choose the left or the right version of the files every time a conflict occurs. Apart from files, *sdiff* can be used to compare two directories, merging the files with the same name in both directories.

sgmldiff

This is a very interesting prgram because it only compares the structure of two SGML files. The comparison is made regardless of the text that lies between the tags. This

can be extremely useful in cases where the user needs to ensure that different translations of the same SGML file have the same structure.

merge

Merge is a simple 3-way merging program. It works fine in case the changes in the two files are not conflicting. However, whenever a conflict occurs, *merge* does not do anything else than including both of the versions that conflict in the merged file. A series of special characters in the final version indicates that a conflict occurred in the specific line.

Revision Control System (RCS)

Revision Control System [28] is able to manage multiple revisions of a file. It is quite useful for programming language files or documentation of a project where documents change frequently. *Rcs* holds a history of all changes, storing not only the current version of the file but also the author, a message that summarises the changes that took place and the date and time the revision was checked out.

Moreover, it provides commands that allow the user to handle those revisions. *Rcsdiff* finds the differences between revisions of the same file, whereas *rcsmerge* incorporates all the changes of two revisions to a single file. In case that a conflict arises, *rcsmerge* prints a warning message.

The limitation of *rcs* is that the user has to "lock" the file in order to edit it. Consequently only one person has write permission to a particular file at any time.

Concurrent Version System (CVS)

Concurrent Version System [3] is an extension of the revision control system, and overcomes the problem of simultaneously editing of a file. *Cvs* assigns a revision number to every checked-out file and uses the update command to keep all the files up to date. Before any changes are committed to a file, *cvs* checks if the revision number of that file and the current number of the revision system are identical. In case that they differ, the file is updated and any conflicts are reported. Afterwards, the user decides whether his changes must be committed or not.

2.2.2 Emacs Editor

Emacs is a powerful tool rather than a simple text editor [25]. One of its standard components, named Ediff [36] is an interface to allow UNIX *diff*, *merge* and *patch* commands to be executed in a user friendly and interactive way. Subsequently, the comparison of the documents is line oriented.

The files being compared are presented one next to the other, allowing the user to copy text from one document to another. Moreover, it is possible for the user to merge interactively two files into a buffer or apply a delta file to an original file. Finally, Ediff supports version control, allowing the user to compare a document with an olderer version.

2.2.3 DeltaXML

DeltaXML [10, 15] is a commercial tool created by Monsell EDM Ltd, capable of comparing, merging and synchronizing XML documents. It is able to merge both ordered and unordered trees fast, where the size is up to 50MB. However, the state of the trees, either ordered or orderless has to be explicitly stated. The XML documents have to be well-formed and with the same root element, whilst knowledge of the DTD or the Schema that it they follow is not required.

One of its interesting features is that it supports both 2-way and 3-way merging. In the first case, DeltaXML compares and merges two versions of the same documents, whereas in the latter case there is one base file and two versions of that file that need to be merged. 3-way merging is more complex but leads to more accurate results. Furthermore, it uses a tree matching algorithm [14] running in linear time and memory.

Contrary to other tools, it only supports the insert, delete and update edit operations. The move edit operation is not implemented because it is not common [8] and would add complexity to the algorithm. Hence, in case that a node is moved in one of the two files, the differencing algorithm assumes that the node was deleted and then inserted in another location (section 2.4 discuss further the edit operations).

2.2.4 XyDiff

Gregory Coneba and the research department in INRIA created a tool for differencing XML files called XyDiff [18]. It uses a very efficient algorithm that improves the time and memory management by identifying large subtrees that were left unchanged between the two versions of the document. However, the results are not always optimal. The algorithm does not always produce minimal delta files, called XyDelta (XyDelta output format is discussed further in section 2.3.2).

Moreover, the tool supports the insert, delete, update and move edit operations, and comparison of documents greater than 10 MB is possible. Finally, it provides options of ignoring whitespaces and formatting characters in XML files.

2.2.5 IBM's XML Diff and Merge Tool

IBM created a Java tool for differencing and merging XML files [1]. The XML Diff and Merge tool only implements basic functionality. It does not support 3-way merging and it is unable to merge automatically XML files, even if there is no conflict.

The tool prints both of the files on the screen and it is up to the user which version will choose, whenever a difference is identified. It does not offer any kind support, documentation or API for this application.

Moreover, it is incapable of producing always the correct results. IBM’s developers made the assumption that if all children of a node have no conflicts, the node is free of conflicts. However this is not accurate and causes errors in various test cases.

2.2.6 XML TreeDiff

IBM also created TreeDiff [29] that is a set of Java Beans for differencing and updating DOM trees [32]. Although Document Object Model (DOM) is simple to use, it has two main drawbacks; firstly, it is relatively slow and secondly, it cannot handle large documents.

Moreover, TreeDiff uses an optimal tree-differencing algorithm in combination with a fast subtree-matching to produce the results, reducing significantly the time and space complexity. However, the differencing file is only applicable to the original file in order to create the updated version.

2.2.7 Microsoft XML Diff and Patch

Microsoft created a tool [17] that is able to differentiate and patch two XML documents regardless if they are mapped into ordered or unordered tress. Moreover, it offers an option of ignoring whitespaces, XML comments and processing instructions, and offers some interesting options about the namespaces. It represents the differences with a XML Diff Language (XDL) that is called XML diffgram.

However, it is not an open-source tool, it is not accompanied with any kind of documentation and it uses the Document Object Model [32]. Furthermore, it fails to compare files that differ dramatically and the online version of the tool is unable to handle files greater than 100KB.

2.2.8 VM Tools

VM Systems introduced a Java XML-oriented Diff and Patch tool [30], able to output the differences between two versions of a document and apply those differences to one of the original files in order to create a patched file. The difference document is optimised for minimal size and the core code is extensible offering an API.

Although it supports both markup style and data documents, this is not the case with XML processing instructions and comments. Finally, the Diff and Patch tool is not accompanied with any documentation.

2.3 Difference Output Format

The output format of the differencing tool is significant for the successful patching of the documents. Hence, this section presents the most interesting output formats of the tools that we examined earlier. The output format should be reasonably compact but, at the same time, it should include enough information for the differences to be applied to a document.

For the sake of clarity the input documents of all the tools are the same. The two files that were tested are illustrated in Figure 2-3.

```
<?xml version="1.0"
 encoding="utf-16"?>

<catalog>
  <product>
    <status>Available</status>
    <name>Ipod Mp3 Player</name>
    <description>10G HD – Windows
    </description>
    <price>$450</price>
  </product>
  <product>
    <status>Not
available</status>
    <name>Digital Camera</name>
    <description>Canon Ixus II
    </description>
  </product>
</catalog>
```

```
<?xml version="1.0"
 encoding="utf-16"?>

<catalog>
  <product>
    <status>Available</status>
    <name>Ipod Mp3 Player</name>
    <description>10G HD – Windows
    </description>
    <price>$450</price>
  </product>
  <product>
    <status>Available</status>
    <name>Digital Camera</name>
    <description>Canon Ixus II
    </description>
    <price>$399</price>
  </product>
</catalog>
```

Figure 2-3: The Input Files of the Output Format Testing

2.3.1 DeltaXML

Figure 2-4 presents the output format of the DeltaXML [15]. DeltaXML produces either a ‘full delta’ that contains both the changed and the unchanged data or a more compact delta that excludes the unchanged data. This format leads to delta files that are relatively large but easier for the user to read and understand.

```
<catalog deltaxml:delta='modified'>
  <product deltaxml:delta='unchanged'/>
  <product deltaxml:delta='modified'>
    <status deltaxml:delta='modified'>Available</status>
    <name>Digital Camera</name>
    <description>...</description>
    <price deltaxml:delta='inserted'>$399</price>
  </product>
</catalog>
```

Figure 2-4: DeltaXML Difference Output Format

2.3.2 XyDelta

XyDiff uses an output format called XyDelta [19] to represent the differences between two XML documents. XyDelta has useful mathematical properties, like the fact that delta files that follow the XyDelta format can be aggregated and inverted without knowledge of the original file. Moreover, every node is given a unique identifier that is called XID. Figure 2-5 presents the difference output format of the XyDiff tool.

```
<xydelta
  v1_XidMap="(1-30)"
  v2_XidMap="(1-14;18-23;31-33;24-30)">
  <delete xid=(15-17) parent=6 position=1>
    <status>Not Available</status>
  </delete>
  <insert xid=(31-33) parent=6 position=4>
    <price>$399</price>
  </insert>
</xydelta>
```

Figure 2-5: XyDelta Difference Output Format

2.3.3 XML Update Language (XUL)

IBM's XML TreeDiff uses the XML Update Language (XUL) [29] in order to output the differences between two documents. Each XUL document is a set of nested node elements. Although the structure of the original tree has been reduced to what is strictly required, there is still enough structural information to show the possible

interdependence of the operations. In Figure 2-6, it shows that the two operations are independent and subsequently can be processed in parallel.

```
<node id="/*[1]" />
    <node id="/*[1]/*[2]" />
    <node op="add" name="B" type="3" />
        <node id="/*[1]/*[3]"/>
        <node op="add" name="G" type="3" />
    </node>
</node>
```

Figure 2-6 – XUL Difference Output Format

2.3.4 Delta Update Language (DUL)

Adrian Mouat in his thesis [19] described a new output format named Delta Update Language (DUL). DUL supports four edit operations; insert, delete, update and move. The initial aim of DUL was to be able to be used in patching arbitrary XML documents, so the elements of the edit operations include a lot of information about the node, like the type and the name of the node. The output format is demonstrated in Figure 2-7.

```
<?xml version="1.0" encoding="UTF-8"?>
<delta>
  <insert childno="7" name="#text" nodetype="3" parent="/node()[1]/
       node()[4]"></insert>
  <insert childno="8" name="price" nodetype="1" parent="/node()[1]/
       node()[4]"></insert>
  <insert charpos="1" childno="1" name="#text" nodetype="3" parent="/
       node()[1]/node()[4]/node()[2]">Available</insert>
  <insert charpos="1" childno="1" name="#text" nodetype="3" parent="/
       node()[1]/node()[4]/node()[8]">$399</insert>
  <delete charpos="10" length="13" node="/node()[1]/node()[4]/node()[2]
       /node()[1]"></delete>
</delta>
```

Figure 2-7 – DUL Difference Output Format

2.3.5 XML Diff Language (XDL)

XML Diff Language is a XML-based language able to describe differences between XML files that are used by Microsoft in its Diff and Patch tool [17]. An instance of the file is called diffgram. XDL uses path descriptors for identifying nodes that work on DOM. An example delta file is shown below.

```
 <xd:node match="2">
    <xd:node match="2">
      <xd:add match="/2/2/2-3" opid="1" />
      <xd:change match="1" name="price">
        <xd:change match="1">$399</xd:change>
      </xd:change>
      <xd:remove match="2-3" opid="1" />
    </xd:node>
  </xd:node>
  <xd:descriptor opid="1" type="move" />
</xd:xmldiff>
```

Figure 2-8 – Diffgram Difference Output Format

2.4 Differencing Algorithms

This section presents some of the most significant differencing algorithms. One of the primary aims of this discussion was to show the evolution of those algorithms over time, so the algorithms are presented in chronological order.

2.4.1 Initial Approach - Tree-to-tree Correction Problem

Selkow [23] was the first to approach the problem of identifying the differences between two trees, known as tree-to-tree correction problem. For the sake of simplicity, he defined that the edit operations are only allowed to the leaves of the two trees. Although this seems to be restrictive, it makes sense in XML documents where it is not allowed to delete an internal node. For example, the user should not be allowed to delete an element <book> and keep the elements <title> and <ISBN> from

that book. However, applying the delete edit operation to subtrees, instead of single nodes, overcome this problem.

Tai's paper [27] overcame that limitation and described a dynamic programming algorithm that could calculate the distance between two rooted, labelled and ordered trees. The algorithm uses the preorder numbering to number the nodes of the trees. Tai also inserted the notion of *mapping* between the nodes of two trees, which is a graphical representation of what edit operation apply to each node.

2.4.2 Shasha's and Zhang's Fast Algorithms

Zhang and Shasha [39] in 1989 suggested a similar algorithm using a left-to-right postorder numbering. They also introduced the notion of edit distance between two ordered forests, which is the minimum cost of a series of operations in order to transform one forest into the other. The edit operations are still the same but the complexity in time and space is smaller.

Zhang and Shasha suggested in [38] three algorithms trying to solve the problem of editing distance between trees. The first one is based on [27] and manages to improve the results whereas the other two are fast parallel algorithms. Those algorithms will not be considered further in this thesis.

2.4.3 Chawathe's Algorithms

Chawathe introduced various algorithms for differencing tree structured documents. The most significant of them are presented below.

Change Detection in Hierarchical structured information

Although this approach is much faster than Shasha's and Zhang's and always lead to correct results, it does not guarantee to generate minimal deltas.

This algorithm [7] is able to compare two versions of a tree-structured file(the base file and an updated version of it). Object identifiers are not assumed because they are not always present. The edit operations that are supported are insert, delete or

update of a node and move of a subtree. The problem in this approach is divided into the *Good Matching* problem and the *Minimum Conforming Edit Script (MCES)* problem [7]. The first one can be described as the problem of finding a matching between objects in the two versions, and the latter as the problem of computing the minimum cost edit script (in case of ids the process is simplified and speeded-up).

Comparing hierarchical data in external memory

Chawathe in his paper [5] discusses the problem of comparing hierarchical data in external memory. This is applicable in cases where the files are too big to fit in main memory, so the algorithm tries to minimise the I/O reads from the disk. The allowed operations are insertion, deletion and update a node and they are only applicable in the leaves of the tree. The algorithm can be applied to rooted, labelled, ordered trees.

Handling of unordered trees

S. Chawathe and H. Garcia-Molina presented in [6] the MH-Diff algorithm that allowed even more edit operations than the insertion, deletion and update. It also supports copy, move and glue (inverse of copy) operations. The algorithm does not consider unique identifiers in the nodes of the tress. Although unique identifiers prove to be really helpful and make the execution of the algorithm much faster, they are not always present and sometimes are confusing. Moreover, the algorithm allows the user to decide upon desired operations because the cost of each of them is not fixed.

The suggested solution leads to high complexity and it can be applied to both ordered and unordered trees. The algorithm transforms the problem from finding the edit-script, to finding the edge cover that represents how one set of nodes match to another.

The second algorithm that can be applied to unordered trees is presented in X-Diff [31]. The algorithm assumes that two trees are identical if they are *isomorphic*[1]. However, this thesis will not consider further any algorithms that handle unordered trees.

[1]Two trees are identified as isomorphic if they are identical except for the order among siblings

2.4.4 Other Algorithms

Using Signatures

Khang's paper [13] introduces an algorithm, called top-down that detects changes in XML files using signatures. The latter are simply an abstraction of the information stored in a node. Instead of comparing all the nodes of the two tree-structured documents, it only uses a subset. In fact, if a leaf is modified, the algorithm can detect that change by its ancestors.

The algorithm is applicable for web applications where more interest is expressed upon excluding irrelevant information than on accuracy. Moreover, it only considers changes in element values (elements or attributes cannot be changed). Finally, the document's DTD is required in order to assign the ids.

Structure Comparison of tree oriented files

There are other papers [4, 20] that are only interested in the structural comparison of the XML files. This is particularly useful in order to create a series of possible DTDs files from a collection of XML files. However, those algorithms are out of the scope of this dissertation.

2.5 Patching

Although there are a lot of sources that describe various differencing tools and algorithms, it is almost impossible to find any kind of documentation or scientific paper that is related on patching. The following section organises that information and explains various concepts of line oriented patching. Section 2.5.2 attempts to interpret the notions of line based patching into tree oriented patching.

2.5.1 Line Oriented Patching

As mentioned above, there is no bibliography about patching and the majority of the existing patching tools (like Mocrosoft's XML Diff and Patch Tool [17] or the XML-

Oriented Diff and Patch Tool [30] by VM Tools) are neither open source or offer any kind of documentation. Therefore, the main source of information about patching was the *patch* program [16, 25] that UNIX offers.

Patch takes the output of the *diff* program, which is a collection of differences between two files and is called *patchfile*, and applies those differences to one of the original files or a third file. The *patchfile* is comprised of a series of hunks, which are groups of differencing lines. *Diff* program tries to minimize the total size of the hunks by identifying large sequences of common lines.

It is interesting that in case a hunk cannot be applied to any line, *patch* tries to find the correct location by adding or subtracting the offset used in applying the previous hunk, to the line mentioned. If it is still unable to locate the right place, it attempts to match the context information of that hunk looking backward and forward of the line mentioned. If that also fails, and the *patchfile* was generated with a context diff, *patch* checks a variable named *fuzz factor*. If the value of that variable is greater or equals than one, *patch* ignores the first and the last line of the context and tries to apply the hunk again. In case that this is still not enough, and the *fuzz factor* is greater or equals than two, *patch* ignores the first and the last two lines of the context information. It must be clear that the greater the value of the *fuzz factor*, the greatest the possibility to apply the hunk in a wrong place. If this procedure does not lead to a match, the hunk is placed in a reject file. At the end of the patching the reject file will contain all the hunks that could not be matched. It is quite important to mention that the results are guaranteed only when the patch is applied to one of the original documents.

2.5.2 Tree Oriented Patching

An approach of managing tree oriented patching was introduced by IBM. F. Curbera proposed an algorithm [9] that could take advantage of the tree structure of the files and would operate on DOM. The algorithm uses two passes in order to complete the patching operation. The first approach resolves the node references whereas the second run performs the operations using a depth-first traversal of the XML Update Language (XUL difference output format was discussed in section 2.3.3).

Interestingly, the algorithm may lead to inconsistent intermediate states. This application can be used for version control of documents, collaborative editing of files or synchronization of a central server.

It is important to make clear how the notions of patching are used in tree oriented case. The *patchfile* is a collection of hunks that describe the differences between the two documents. However, the hunks are no longer groups of lines but subtrees that contain the differencing nodes and their context information. The context information of a node, is the neighbouring nodes including its siblings and its parent. The *fuzz factor* represents the same notion as in line oriented files, but in case of tree structure patching its value should be relatively large because it is harder to match the nodes. Finally if a hunk cannot be applied to any of the nodes, it will be placed to the reject file (similarly to the line oriented patching).

2.6 Going Further - Harmony Project

Harmony is a project that is run mainly at the University of Pennsylvania. It is presented at this late point, because it is actually an extension of all the tools that were discussed up to now. Their goal is to develop a generic framework for synchronizing tree-structured data, but the project is still in early stages [12, 21]. More precisely, the research team of Harmony aims to develop a framework that would be able to propagate updates between different copies of tree-structured data that are probably stored in different formats. Examples of applications like that could be the synchronization of bookmarks of different browsers, address books, group calendars, keynote presentations and generic XML & HTML.

In order to synchronize data that is stored in different formats (heterogeneous data), the algorithm transforms the concrete tree-structured data into an abstract form [11]. However, this is not enough because the application should transform the abstract data back into the concrete format. This is a challenging task, since some information is discarded during the transformation into abstract data. To facilitate this demanding task, they created a Bi-Directional language, named Hocus Focus that allowed them to move from one state of the data to the other (from concrete tree-structured data to abstract and vise versa).

Before Harmony, the same research team was developing Unison project, which is a file synchronizer [22]. However, Unison file synchronizer handles all files as a series of lines without taking advantage of the structure of the files.

2.7 Discussion

The conclusions of the background research on differencing and patching are discussed in the following paragraph.

After having studied various algorithms for differencing tree structured files, two of the algorithms that S. Chawathe proposed appear to be the more appropriate to be implemented in this thesis. The first one was described in [6] and it supports a variety of edit operations. The major drawback of this algorithm is that it leads to systems of high complexity. An alternative choice is the algorithm that is presented in [7]. This algorithm manages to run fast but it does not create minimal deltas.

All the difference output formats that were discussed in this chapter seem to be inadequate to satisfy the initial aim of this thesis, which is to create a tool able to patch arbitrary XML documents. However, DUL output format introduced by Adrian Mouat [19] could be a reasonable base to build a more appropriate format. It is also significant to mention that none of those output formats support context information.

Unfortunately there is no bibliography or papers that are related to line or tree oriented patching. So, the patching tool that will be developed in this thesis will be based on the notions of line oriented *patch* program.

Chapter 3

Requirements Capture

This chapter states the general research problem of the thesis, as well as the motivation that guided the author of this document towards that direction. Moreover, both the Functional and Non-Functional requirements of the differencing and patching tool will be described now.

3.1 Thesis Problem Statement

The previous chapter shows that there are many tools and algorithms for differencing tree-structured documents. Moreover, although there are a few tools capable of patching files in a satisfactory way, none of those offer any kind of documentation. The only exception is Adrian Mouat's thesis [19], where can be found a detailed description of the differencing and patching of tree-oriented files. However, in all of those approaches the delta file can be applied only to one of the original files.

This thesis will attempt to address the problem of differencing two XML files, outputting their differences in a delta file and finally applying that delta file to an arbitrary XML document.

Figure 3-1 illustrates the general case. The delta file is the output of the differencing tool and describes the differences between the *base file* and the *file 1*. There is also another version of the base file named *file 2*. The issue is to create a final version that will include both the changes of *file 1* and *file 2*, using only the delta file

and the *file 2*. For the rest of the document, the author will refer to the *base file*, *file1* and *file 2* as they are used in Figure 3-1.

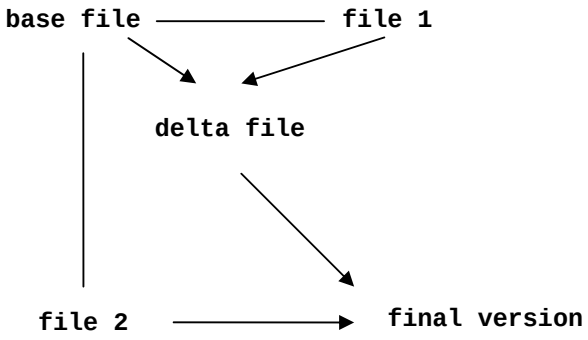

Figure 3-1 – Thesis Statement

3.2 Functional Requirements

The functional requirements of this application are prioritised, using the MoSCoW Rules [2], into the following categories: 'the Must Have', 'the Should Have', 'the Could Have' and 'the Won't Have'. The first category includes the requirements that are vital for the application. The second one consists of the desired features of the project. Those features will be probably implemented but the thesis' success does not rely on them. The next category contains features that will only be implemented in case that there is additional time. The last category makes some suggestions about what might be done in the future in order to develop the tool further. For that reason 'the Won't Have' category will be discussed in the last chapter along with the evaluation of the current work.

3.2.1 Differencing tool

The requirements for the differencing tool can be summarised as follows.

Must Have

- The tool must be able to take as input two well-formed XML documents and identify all the differences between those two files.
- After identifying the differences of the two documents, it must be able to output those differences in a way that is useful for the patching tool. Hence, although it must be relatively compact, at the same time must contain enough context information.
- The algorithm must handle the files as tree structures instead of a series of lines.

Should Have

- The tool should have the look and feel of the UNIX line-oriented commands. Options for the tool must be both letters and names.
- A silent mode should be a possible option. In that case the user only checks if two files are identical without finding the list of differences.
- There should be an option whether or not to ignore white spaces and character case.
- The differencing tool should be able to ignore comments and processing instructions.
- An option of ignoring namespaces should be also available. That means that names with the same local name but different namespace are treated as identical.
- There should be an option of transforming the output format into more user friendly formats.

Could Have

- The tool may implement two different algorithms to give a choice to the user between accuracy and speed.

- The tool may support comparison of directories, mapping the files with the same name in each one of them.
- The algorithm may be able to support arbitrary length files.

3.2.2 Delta File

The requirements for the delta file are described below.

Must Have

- The delta file must be able to describe the changes between two XML documents.
- The delta file must be a well-formed XML document.

Should Have

- The delta file should include enough context information in order to be able to be applied to an arbitrary XML document by the patching tool.
- The delta file should be reasonably compact. The extreme case would be to include the whole file.

Could Have

- The delta file may be transformed in a more user friendly format using XSLT [34] that can transform an XML document to other formats.

3.2.3 Patching Tool

The requirements for the patching tool are listed below.

Must Have

- The tool must be able to apply the patch that was generated to one of the original files in order to produce the other.

- The tool must be able to apply the delta file to an XML file other than one of the original documents. Various accuracy aspects arise in this case, so in the following chapter we will describe in detail the *desired behaviour* of the patching tool in all the cases that it is not trivial what that behaviour should be.

Should Have

- The tool should be able to reverse the patch. That means that all the 'adds' should become 'deletes' etc.
- The tool should offer an option whether to act automatically or not in case that a conflict occurs.
- The tool should offer the option of making a backup of the files and creating rejected files.

Could Have

- The tool might print a report at the end of the patching to describe all the conflicts. In case there was no conflict it may print a message to inform the user.
- The tool might offer a "fuzz factor" to determine the accuracy of the patching.

3.3 Non-Functional Requirements

The non-functional requirements of this thesis are listed below.

- The tool that we will develop along with the research in this thesis should be an open-source application.
- A thorough documentation should accompany the tool, in order to enable other developers to continue and extend this work.
- The tool must run in both UNIX and Windows Operating Systems. The Java Virtual Machine is required to be already installed in those Operating Systems.

Chapter 4

Instances

In case that *file 1* and *file 2* (see Figure 3-1) modify different nodes, patching tends to be straightforward. However, it is expected that both documents will apply an edit operation to the same node. In those cases, the behaviour of *treePatch*, which is the patching tool that will be developed in this thesis, should be thoroughly described.

The chapter explores all those cases and describes the desired behaviour of *treePatch*. Consequently, the chapter is both part of the requirements and the design stage of the application.

4.1 Edit Operations

This section focuses on the definition of the desired behaviour of *treePatch* in cases where conflicts may occur. In this section, only the four edit operations that Chawathe describes in his report [7] are considered; insertion, deletion or update of a node and move of a subtree. Each special case is accompanied with a small diagram illustrating a specific example, as well as a justification on the reasons that the specific behaviour was chosen.

The reader should bear in mind two significant issues while he/she is reading the following subsections. Firstly, the inputs of the patching tool are *file 2* and *treePatchFile*, which is the delta file of the current tool. *TreePatch* does not use *file 1*. As a result, in case that *file 1* and *file 2* are shifted, the results might not be the same.

The remainder of this document assumes that the following statements are not identical.

- Edit operation ed1 is applied to *file 1*, whereas edit operation ed2 is applied to *file 2*.
- Edit operation ed2 is applied to *file 1*, whereas edit operation ed1 is applied to *file2*.

Moreover, *treePatch* does not have only knowledge about the sequence of operations that transformed the two files. Only the final state of the documents is known. Benjamin Pierce discusses that issue in [21], distinguishing between state-based and log-based synchronization. The former is the case that this thesis examines, whilst in the latter the sequence of edit operations that were applied are logged. Hence, in this thesis it is impossible to distinguish between an update and an insertion followed by a deletion.

4.1.1 Delete Edit Operation

- The first case considered is where a node is deleted in *file 1* and updated or moved in *file 2*. Assuming that the author of *file 2* is interested about that node (he applied an edit operation to that node), it will not be deleted. The user will be informed about the situation and the hunk will be inserted in a reject file. The following figure shows that node C is deleted in *file 1* and moved in *file 2*.

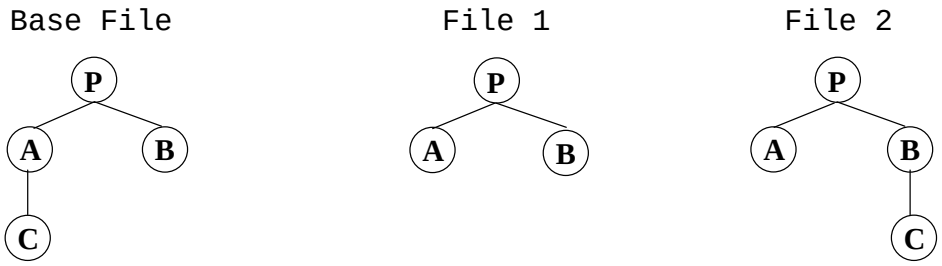

Figure 4-1 – A node that is moved or updated cannot be deleted

- Figure 4-2 illustrates the case where *file 1* deletes node B and *file 2* inserts a node as a child of the same node B. Assuming that node C is significant for the author

of *file 2*, the delete edit operation will not be applied to node B. Once more, the hunk will be inserted in the reject file and the user will be notified.

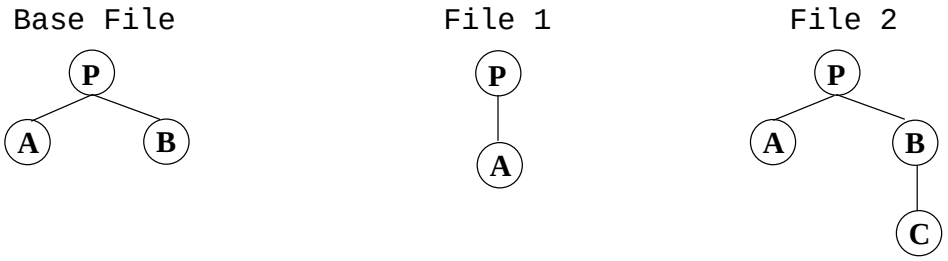

Figure 4-2 – A node that has children cannot be deleted

- The following example considers the case where the same node is deleted in both of the files. It seems that a conflict occurs although it should not. However, this conflict cannot be prevented because *treePatch* is trying to find a node that does not exist anymore. It is interesting that although this conflict occurs in patching, in case of merging the delete operation would be applied without any problems. This is happening because *treePatchFile* contains less information than the whole *file 1*. Figure 4-3 demonstrates an example where both of the files delete node B. The hunk will be added to the reject file and the user will be notified.

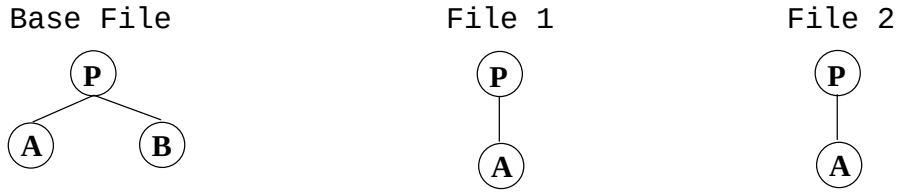

Figure 4-3– The same node is deleted in both of the files

4.1.2 Insert Edit Operation

- Figure 4-4 shows another example where conflict occurs. *File 1* inserts node C as a child of node A, whereas *file 2* deletes node A. In this case it is not feasible for node C to be inserted, because node A does not exist anymore on *file 2* and there is not enough information to undelete it. *TreePatchFile* could include that information but this leads to large delta files (in the extension of this example

maybe more than one nodes were needed to be recovered). Thus, the hunk will be added in the reject file and the user will be informed.

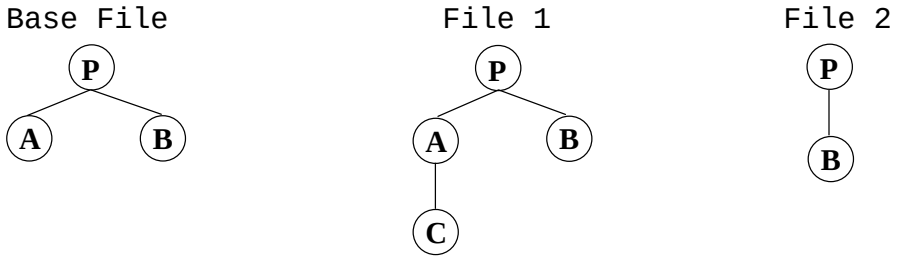

Figure 4-4 – A node is inserted but its parent is already deleted

- The following Figure demonstrates the case where *file 1* inserts node C after node A and before node B. However, in *file 2* the order of nodes A and B has been changed. It is not obvious what the patch program should do because it can either insert the node before node B or after A. It is impossible though to satisfy both conditions. Assuming that the order from left to right is more significant, *treePatch* inserts node C after A. Once more, the user should be notified that a conflict may have occurred.

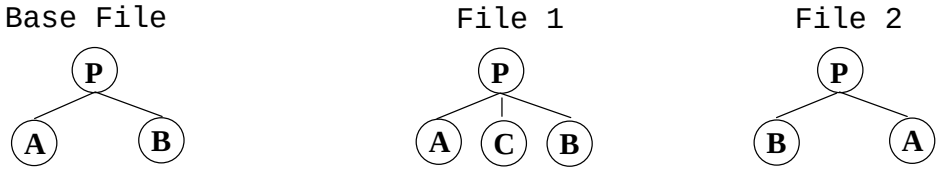

Figure 4-5 – The order of the siblings is modified

- Another similar example is when *file 1* inserts node C after B and before D, and *file 2* deletes both of those nodes. In case that nodes B and D were the only children of node P, there is no conflict. However, in the example illustrated in Figure 4-6, the *treePatch* will not know whether to insert node C before or after A. Because this is not a usual case, there will not be added extra information in the *treePatchFile.* Node C will be inserted as the last child of P, and the user will be informed about the situation.

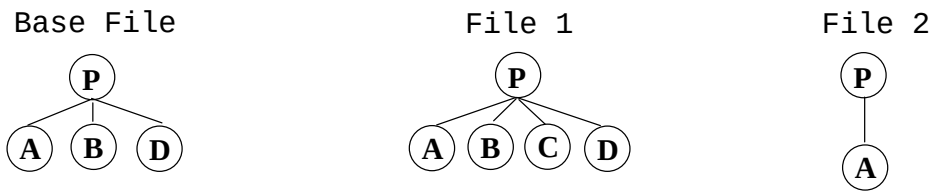

Figure 4-6 – Left and right siblings of the node are deleted

- Another interesting situation is when *file 1* inserts a node after node A and before node B, and *file 2* inserts another node between A and B. It is impossible for *treePatch* to know whether it should insert node C before or after node D. Node C will be inserted after D and the user will be informed that maybe the order of the nodes is incorrect.

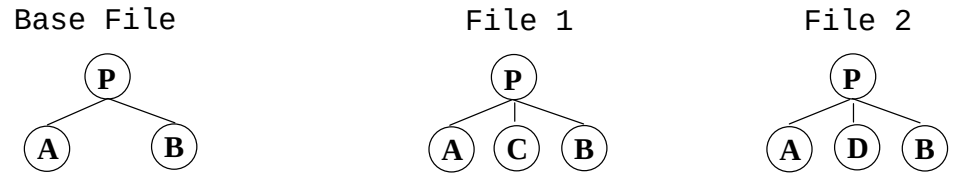

Figure 4-7 – Both of the files insert a node as a child of the same node

- The last example that is associated with the insert edit operation is when *file 1* inserts a node as a child of node A and *file 2* moves the same node A. In this case *treePatch* should insert node C as a child of node A in the new location. The user will not be notified at all.

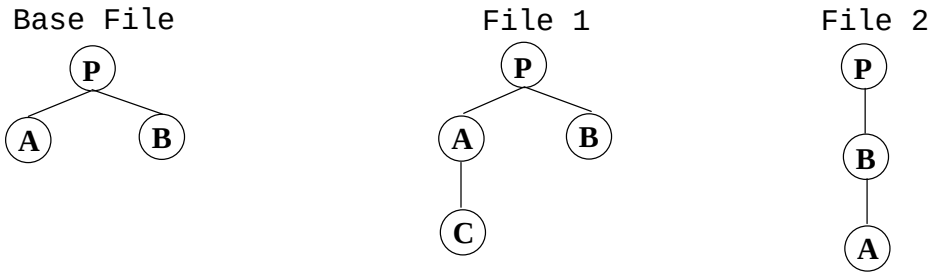

Figure 4-8 – A node is inserted and its parent is moved

4.1.3 Update Edit Operation

- Update edit operation can cause conflicts in case both *file 1* and *file 2* update node A with different values. In that case, the value of the node in *file 2* will be kept and the hunk will be added in the reject file. Moreover, the user should be notified that the value of the node might not be correct. Figure 4-9 illustrates this case.

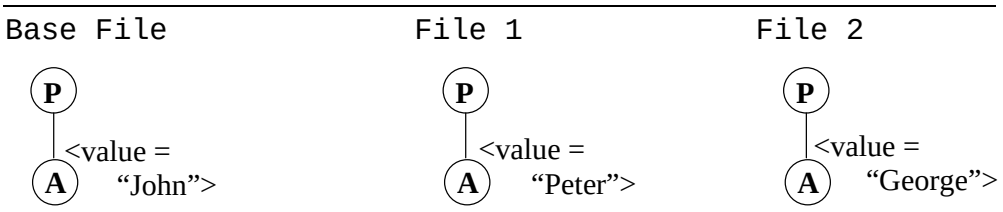

Figure 4-9 – Both of the files update the value of the same node

- Figure 4-10 shows an example where update operation can cause conflicts is illustrated in figure 4-10. The value of node A is updated in *file 1* and the same node is deleted in *file 2*. Although it would be desired to include node A with the updated value in the final version, this is not possible. The delta file would not include the required information to recover the deletion. Consequently, the reject file will includes the hunk and the user will be alerted about the conflict.

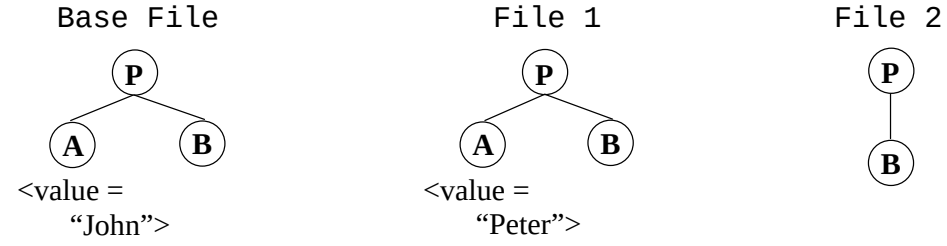

Figure 4-10 – The value of a node is updated and the node is deleted

- It is possible for the update edit operation to be applied to a node in conjunction with other operations without causing any conflicts. So, in cases where one file updates the value of a node and the other moves it or inserts a node as its child or change the order of its siblings should not cause any conflicts. In situations similar to the ones described above both of the operations should be applied without notifying the user.

4.1.4 Move Edit Operation

- Once more, in case a node is moved in *file 1* and deleted in *file 2*, *treePatch* will not have enough information to recover the deleted node. The user will be notified that the node he wanted to move is already deleted and the hunk will be added in the reject file. Figure 4-11 demonstrates an example situation.

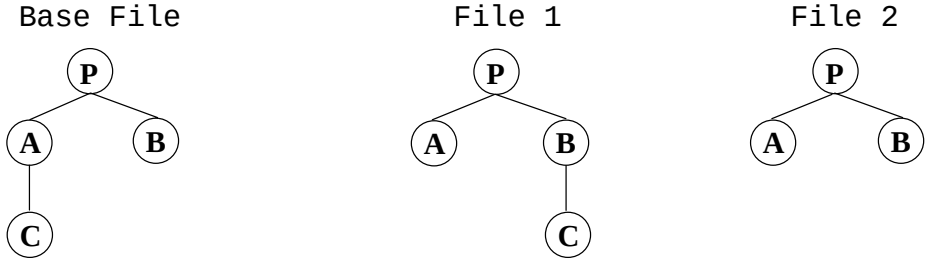

Figure 4-11 – A single node is moved and deleted

- The following figure illustrates an example where *file 1* moves node A to make it a child of B, and *file 2* deletes node B. The move edit operation cannot be applied because the target node is already deleted in *file 2*. The user will be informed and the hunk will be inserted in the reject file.

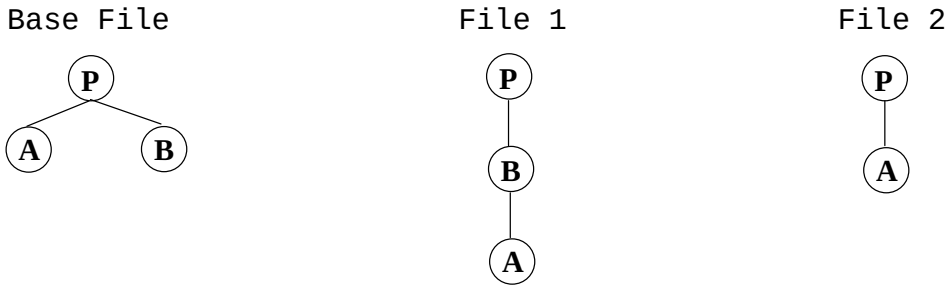

Figure 4-12 – The target node is deleted

- The next example is a very interesting case where *file 1* moves node A and *file 2* inserts a node as a child of A. It is significant to recall that move operation is the only one in this implementation that is applied to subtrees, so *treePatch* will move node A and subsequently node C as a child of A. This behaviour can be characterised reasonable, so the user will not be notified.

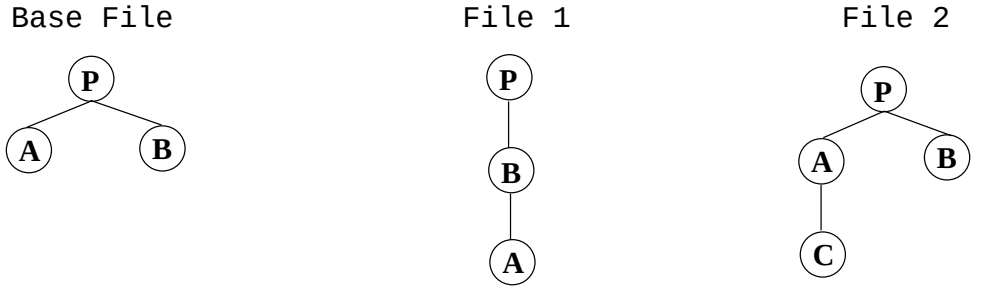

Figure 4-13 – Move operation is applied to subtrees

- Figure 4-14 demonstrates the case when both of the files move the same node in a different position. In this specific example *file 1* makes node B the only child of A whereas *file 2* moves B under node C. *TreePatch* will be unable to decide which one is the correct location for node B, so the move operation will not be applied (node B in the final version will be located under C). As the reader might expect, the hunk will be added in the reject file and the user will be alerted about the possible conflict.

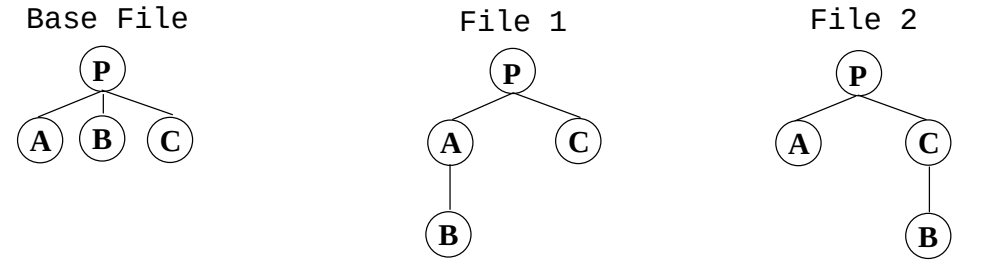

Figure 4-14 – Both of the files move the same node in different locations

- The last figure of this chapter illustrates an example where conflict cannot be avoided. It is shown that in *file 1* node A is the only child of B, whereas in *file 2* node B is the child of A. Keeping in mind that move operation is applied to subtrees, the reader can easily realise that it is impossible for the patch program to apply that operation. The user will be notified for the conflict and the hunk will be added in the reject file.

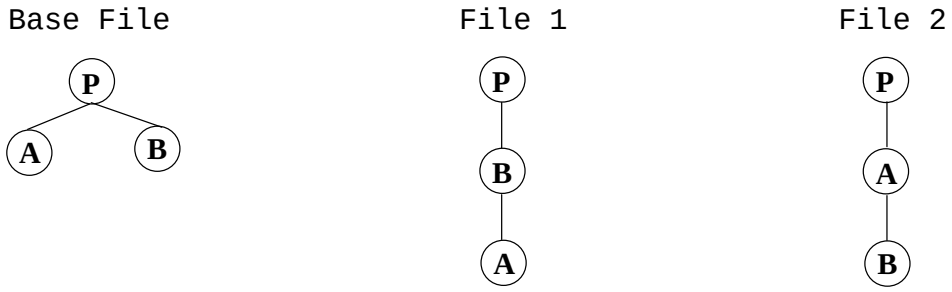

Figure 4-15 – Move operation is applied to subtrees

- Apart from all the above cases that involve the move edit operation and may cause conflicts, there is another category of examples that are related to the order of the siblings. However, all those cases where discussed in insert edit operation (section 4.1.2) and will not be considered again in this section.

Chapter 5

Design

Taken into concern the fact that there are a lot of differencing tools but none for patching arbitrary XML files, inevitably move the interest of this thesis into the latter. With this in mind and taking into account the time limitations of the dissertation, the author of this document decided to use an existing differencing tool. A further discussion about that decision is help in section 5.1. The two following sections describe the difference output format that was used in the current application and the adding of context information in the *treePatchFile*. Finally, the last section considers the design of the *treePatch* tool.

5.1 Differencing Tool

The differencing tool that it will be used in the current application is developed and documented in [19] by Adrian Mouat. There were two main reasons that contributed into choosing that tool. First of all, Adrian Mouat created an open-source tool, so it is possible to use parts of his code and modify them whenever it is necessary. Secondly, although it has some bugs, it satisfies the majority of the requirements for the differencing tool, stated in chapter 3.

Figure 5-1 presents both Adrian Mouat's and the current implementation. The former is presented in the upper part of the figure, where two main parts can be seen; the XML Diff tool and the patching tool. The differencing tool contains the XML Diff

algorithm as well as the DUL difference output format that Adrian Mouat designed and documented in his report.

The lower part of the figure summarises the design of the current implementation. It is shown that the outputs of the XML Diff algorithm are used to create the EDUL difference output format (EDUL is thoroughly documented in section 5.2). Afterwards, the context information is added to create a delta file applicable to arbitrary XML documents. Finally, *treePatch* is used in order to patch the files.

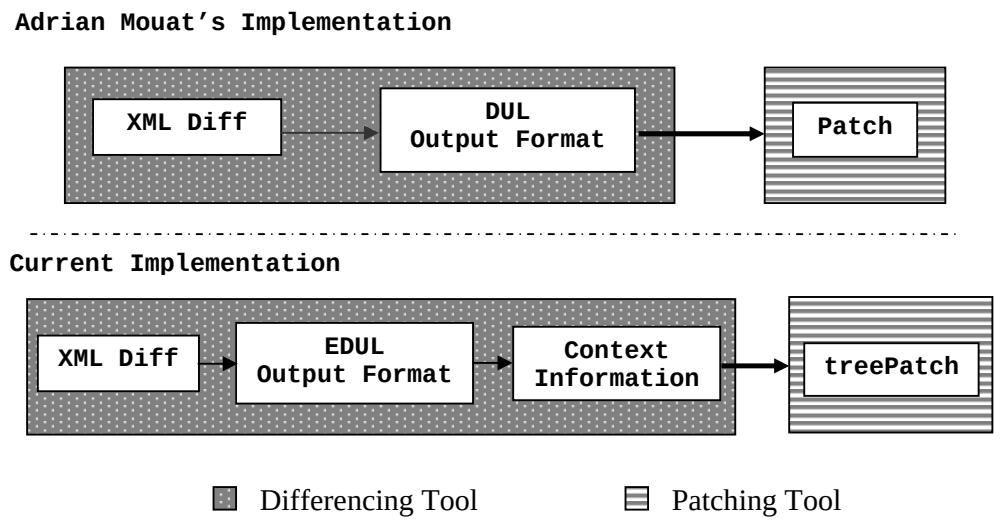

Figure 5-1 – The Relationship of the current and Adrian's Mouat Implementation

The rest of this section examines Adrian Mouat's implementation and reports any possible bugs. Although he fully describes an update operation both for the differencing algorithm and the DUL output format, this was not actually used. Instead, the differencing tool interprets all update operations as an insertion followed by a deletion. This will not change, assuming that it does not affect the operation of the *treePatch*. However, the remainder of the design sections will describe in detail the update operation, assuming that it was used.

Moreover, the delete edit operation is only applied to leaf nodes. This is the only modification of the initial algorithm that Adrian Mouat applied. The result is that in case the user wants to delete an internal node, all its children have to be deleted as well. This problem still remains unsolved.

In addition, the move operation is not working properly. The problem is that the differencing algorithm fails to identify the correct number of children that the node should be moved to. The current implementation overcomes this problem, by adding information about the right and the left sibling of the node that the move edit operation was applied to (a detailed discussion of this issue is carried out in section 5.2.1).

Furthermore, the differencing algorithm is unable to identify the insertion of attributes in an existing node. Although this is a significant feature of a differencing tool, it will not be implemented in the current application because it is not essential for the patching tool.

Finally, not all of options offered by the differencing tool offers are implemented. The current application will implement those that are significant for the output format or the patching tool (this is the case of parent-context option), whereas the rest will not be used (for example the option of ignoring the whitespaces or the case sensitivity).

5.2 Difference Output Format (EDUL)

This section describes the output format of the differencing tool, named EDUL. EDUL stands for Extended Delta Update Language, and it is an extension of Adrian Mouat's difference output format DUL [19]. The first section discusses the differences between those two formats. The four following sections thoroughly describe the insert, delete, update and move edit operations. Finally, the last section gives an example of the EDUL output format.

Throughout this section the XPath standard [35] and DOM Level 2 Core Specification [32] are heavily utilised. For the sake of clarity, the rest of this chapter will refer to the node that an edit operation was applied to as the *changed_node*. Moreover, the elements of the *treePatchFile* that describe the four edit operations will be referred as <insert>, <delete>, <update> and move respectively.

5.2.1 The Differences between EDUL and DUL

Although DUL is well documented, it is not adequate enough to satisfy the requirements of this thesis. As a result, the difference output format is extended in order to be able to be successfully applied to an arbitrary document, as well as handle all the special cases that were discussed in the previous chapter. The following paragraphs describe all the changes from DUL to EDUL. The full description of EDUL output format can be found in sections 5.2.2 to 5.2.5.

Insert

In order to be able to handle all the special cases related to the order of the children after the insertion of the *changed_node*, information regarding the previous and the next sibling of the node was needed. So, the name and the value of those two nodes were added in the <insert> element, which represents the insert edit operation.

Delete

Chapter four defines that if the value of the *changed_node* is updated in *file 2* it should not be deleted. In order to satisfy this condition, an attribute that encloses the value of the *changed_node* in the *base file* is added.

Update

Similarly to the delete edit operation, in case the value of the *changed_node* is updated in *file 2*, it should not be updated again. So, an attribute that keeps that value in the *base file* is added.

Move

Finally, information on the previous and the next sibling of the *changed_node* in the *base file* is added in the move element. Thus, the order of the children can be checked.

5.2.2 Insert Edit Operation

Sections 5.2.2 to 5.2.5 describe in detail the four edit operations that are considered in this thesis. The first XML element that is discussed is <insert>. This element

represents the insertion of a leaf node. All of the attributes of this element are presented below.

nodetype

The *nodetype* attribute determines the type of the *changed_node*. Its value equals to the value of the DOM method getNodeType(). Figure 5-2 shows the values that the DOM method returns.

Node Name	Return value of DOM getNodeType()
Element	1 - ELEMENT_NODE
Attribute	2 - ATTRIBUTE_NODE
Text	3 – TEXT_NODE
Processing Instruction	7 - PROCESSING_INSTRUCTION_NODE
Comment	8 – COMMENT_NODE

Figure 5-2 – The DOM getNodeType() method

name

The second attribute of the <insert> element represents the name of the *changed_node*, and it is equal to the value returned by the DOM method getNodeName(). The *name* attribute will be omitted in cases where the *changed_node* is either a TEXT_NODE or a COMMENT_NODE, because the value that the DOM method getNodeName() returns for all of those nodes is *#text* and *#comment* respectively.

parent

This attribute uniquely identifies the XPath parent of the *changed_node* in *file 1*. The attribute is not used exclusively to locate the parent node of the *changed_node* in *file 2*, assuming that the position of the parent node might not be the same. The algorithm to locate the parent node, it is discussed in section 5.4.

childno

Similarly to the *parent* attribute, this is another attribute that it is not used directly. It only indicates the position where the *changed_node* should be inserted. The attribute

is used in conjunction with four other attributes, named *prev_sib_name*, *prev_sib_value*, *next_sib_name* and *next_sib_value* that are discussed below.

prev_sib_name – prev_sib_value

Although *prev_sib_name* and *prev_sib_value* are two separate attributes, they are considered together, because they both keep information about the previous sibling of the *changed_node* (this node is returned by the DOM method getPreviousSibling()). In case there is no previous sibling, both of the attributes are omitted.

The first attribute stores the name of the previous sibling of the *changed_node*, in the same way that the attribute *name* works. The second attribute represents the value of the previous sibling. The value of this attribute is equals with the value of the DOM method getNodeValue(). So, in case that the previous sibling is an ELEMENT_NODE, the return value of the method will be *null* and the attribute will be omitted.

next_sib_name – next_sib_value

Equally, the *next_sib_name* and *next_sib_value* attributes store information on the next sibling of the *changed_node*. *Next_sib_name* stores the name of the next sibling, whilst *next_sib_value* its value as they are defined by the DOM methods getNodeName() and getNodeValue() respectively. If the value of the next sibling is *null*, *next_sib_value* attribute will be excluded.

The last sour attributes determine where the *changed_node* should be inserted and will handle several conflicts that were discussed in chapter 4.

content

The content of the insert element stores the value of the *changed_node*. In case that the node is an ELEMENT_NODE, its value will be *null*, and the <insert> element will be empty.

example

The following example demonstrates an insertion of an ELEMENT_NODE with name equals to “genre”. As we can observe, the previous sibling is a TEXT_NODE and the next one is a “year” element.

```
<insert nodetype="1" name="genre" parent="/node()[1]/node()[4]" childno="6" prev_sib_name="#text" prev_sib_value="This is the previous sibling" next_sib_name="year"> </insert>
```

5.2.3 Delete Edit Operation

The second XML element discussed is <delete>, and represents the deletion of a leaf node (remember that the differencing algorithm only allows deletions of leaves). The attributes of <delete> element are described below.

node

This attribute uniquely identifies the XPath node in *file 1* to be deleted. However, assuming that the *changed_node* might have been moved in *file 2*, this attribute will not be used exclusively to locate the node. The detailed algorithm that describes how the *changed_node* is being located is described in section 5.4.

charpos - length

The *charpos* and *length* attributes are considered together since they can only be used in conjunction. Those two attributes are only applicable in case that character data is deleted. The value of *charpos* attribute sets the first character of the text to be deleted, while *length* attribute specifies the number of character to be deleted. So, in case that a <delete> element has attributes *charpos=”5”* and *length=”4”*, *treePatch* should delete the 5^{th}, 6^{th}, 7^{th} and 8^{th} character of the node.

In case that the node we want to delete is an ELEMENT_NODE (the type of the node is determined by the return value of the DOM method getNodeType()) the attributes will be omitted.

value

Attribute *value* stores the value of the *changed_node* in the *base file*. The value of the node is determined by the DOM method getNodeValue(). In case the *changed_node* is an ELEMENT_NODE, the attribute will be omitted because the value of all element nodes is *null*.

This attribute was added in order to satisfy the major special case of the delete edit operation that was discussed in the fourth chapter. Before the delete operation is applied to the *changed_node*, *treePatch* needs to check that the value of the node in *file 2* is equals to the value of the same node in the *base file.* In case that those two values are not identical, the operation will not be applied.

example

An example that shows all the attributes of the <delete> operation is illustrated below. The edit operation of this example is applied to a text node, and will delete the whole text of the node, in case that its value in *file 2* is equals to "This is a text node".

```
<delete charpos="1" length="19" node="/node()[1]/node()[2]/node()[4]"
value="This is a text node"></delete>
```

5.2.4 Update Edit Operation

At the beginning of this chapter, all the bugs of Adrian Mouat's implementation were clearly stated. One of those was that the differencing tool does not support the update edit operation (contrary it interprets all the updates as a deletion followed by an insertion). The current implementation uses Adrian Mouat's differencing tool, so it does not support the update operation. However, all attributes of the <update> element will be described in detail, because the difference output format can be used independently of the rest of the tool.

The update edit operation is not applicable to element nodes, because their value is always *null.*

node

The *node* attribute is used in exactly the same way as the <delete> element. It uniquely identifies the XPath node that the update operation is applied to. Similarly to the <delete> element, this attribute will not be used exclusively to locate the *changed_node* in *file 2*, because the position of the node might have been changed.

charpos - length

The first of those two attributes, named *charpos* defines the first character of the text to be updated. The latter stores the number of characters that will be updated.

oldvalue

This attribute was added in order to handle some of the special cases that were discussed in the previous chapter, and stores the value of the *changed_node* in the *base file*. Before an update operation is applied to the *changed_node*, *treePatch* should ensure that *file 2* has not also changed the value of that node. In case that both of the authors of *file 1* and *file 2* have updated the value of the same node, the final version after the patching will contain the node with the value in *file 2*..

content

Correspondingly to the <insert> element, the content of the <update> element stores the new value of the *changed_node*. The value of the node is equals to the value that the DOM method getNodeValue() returns.

example

The following example illustrates the case where the value of a text node is updated. The value of the node is equals to the value of the *oldvalue* attribute, so the update operation will be applied.

<update node="/node()[1]/node()[2]/node()[4]" charpos="1" length="19" oldvalue="This is a text node">This is a text node</update>

5.2.5 Move Edit Operation

The last XML element that will be discussed here represents the move edit operation. Although Adrian Mouat's implementation was unable to successfully apply move operations, the addition of four attributes in the <move> element fix that bug. Those attributes contain information regarding the left and the right sibling of the *changed_node* in *file 1*. The results of that extension are discussed in chapter 7. All of the attributes of the <move> element are described in this section.

node

The *node* attribute uniquely identifies the XPath node to be moved. Once more, the attribute will not be used directly to locate the *changed_node* in *file 2*.

parent

This attribute uniquely identifies the parent of the *changed_node* in *file 1*. The reader should notice that the *parent* attribute contains information about the parent of the *changed_node*, after the move operation is applied. This attribute is the only piece of information that *treePatchFile* contains about the target node.

charpos - length

Those two attributes were previously discussed. They are only applicable in cases where character data is moved. The value of *charpos* attribute sets the first character of the text to be moved, whilst the *length* attribute specifies the number of characters to be moved.

In case that the *changed_node* is an ELEMENT_NODE, both of the attributes will be omitted.

childno

The *childno* attribute is used in equally to the <insert> element. It only implies the position of the child node but in practice the following attributes, named *prev_sib_name*, *prev_sib_value*, *next_sib_name* and *next_sib_value*, are used in order to find the right position.

prev_sib_name - prev_sib_value

Those two attributes store information on the previous sibling of the *changed_node* in *file 1* (the previous sibling is the node that is returned by the DOM method getPreviousSibling() for the *changed_node*). If the *changed_node* is the first child in *file 1*, both of the attributes will be omitted.

The first attribute stores the name of the previous sibling of the *changed_node*. The latter attribute represents the value of the previous sibling. In case that the previous sibling of the *changed_node* is an ELEMENT_NODE, the attribute *prev_sib_value* will be excluded.

next_sib_name - next_sib_value

Likewise, the *next_sib_name* and *next_sib_value* attributes give information regarding the next sibling of the *changed_node* in *file 1*. Attribute *next_sib_name* stores the name of the node, whilst the second attribute stores its value, as they are defined by the DOM methods getNodeName() and getNodeValue() respectively.

If the *changed_node* does have next sibling in *file 1*, both of the attributes will be omitted, whereas in case the value of the next sibling is *null*, the *next_sib_value* attribute will be skipped.

example

The following example shows the case where a node is moved. Its previous sibling in *file 1* is a text node, and its value is equals to “This is the previous node”, whereas the next sibling is also a text node.

```
<move node="/node()[1]/node()[4]/node()[6]" parent="/node()[1]/node()[4]" childno="7" next_sib_name="#text" prev_sib_name= "#text" prev_sib_value="This is the previous node" next_sib_value="This is the next node"></move>
```

5.2.6 EDUL Example

A complete example of the EDUL output format is presented below. In figure 5-3, three edit operations are shown; move, insert and delete.

```
<?xml version="1.0" encoding="UTF-8"?>

<delta>
  <move childno="7" length="0" next_sib_name="genre" next_sib_value=""
   node="/node()[1]/node()[4]/node()[7]" parent="/node()[1]/node()[4]"
   prev_sib_name="year" prev_sib_value=""></move>

  <move childno="7" length="0" next_sib_name="#text" next_sib_value=""
   node="/node()[1]/node()[4]/node()[6]" old_charpos="8"
   parent="/node()[1]/node()[4]" prev_sib_name="#text"
   prev_sib_value=""></move>

  <insert charpos="1" childno="1" name="#text" next_sib_name="#text"
   next_sib_value="Old text" nodetype="3" parent="/node()[1]/node()[4]
   /node()[4]" value="New text">New text</insert>

  <delete charpos="9" length="8" node="/node()[1]/node()[4]/node()[4]
   /node()[1]" value="Old text"></delete>
</delta>
```

Figure 5-3 – EDUL Difference Output Format

5.3 Context Information

Although the difference output format was enhanced, in order to apply the *treePatchFile* in an arbitrary XML document, context information needs to be added (the notion of context information in tree-oriented files was discussed in section 2.5.2). This section considers all the related issues, and describes the algorithm for adding context information that will be used in the current implementation.

The author of this document introduced the notion of *valuable context information*. The idea behind this concept is that some types of nodes can be more useful than other, when the *treePatchFile* is applied to *file 2*. For example, element nodes that contain at least one attribute are more valuable than empty elements without any attributes. Below, a full enumeration of all those cases is listed.

- The node is an ELEMENT_NODE and has at least one attribute (the attribute might or might not be unique).
- The node is an ATTRIBUTE_NODE.

- The node is a TEXT_NODE and it is not an empty node.
- The node is a COMMENT_NODE and it is not an empty node.

The way that the notion of *valuable context information* will be used in the current tool is described in section 5.3.2.

5.3.1 Selection of Context Information

Probably the most important issue related to the context information is the decision that neighbouring nodes should be included in the *treePatchFile.* The tests that were carried out in order to solve that problem are described in chapter 7, while their result is presented in Figure 5-4.

The Figure shows the *changed_node,* the nodes that will be included in the *treePatchFile* as the context information of the *changed_node*, and the rest of the nodes that are not qualified as context information (there is a pattern for each one of those categories). First of all, all the siblings of the *changed_node* should be included. Nodes that exist in the same level but they are not siblings of the *changed_node* are excluded. Moreover, *treePatchFile* should contain the parent of the *changed_node* as well as all the siblings of the parent node. Finally, the parent of the parent node should be included, as well as all its siblings. Nevertheless, this is not absolute; therefore the algorithm that is described in the following section verifies that the *treePatchFile* contains enough *valuable context information.*

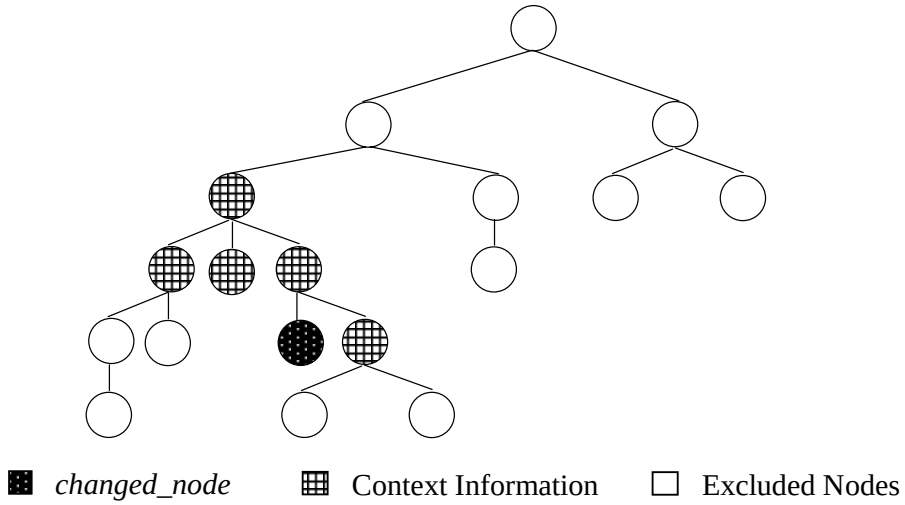

Figure 5-4 – Context information nodes

5.3.2 The algorithm of adding Context Information

The algorithm that allows the context information to be added in the delta file is described in this section. The inputs of this algorithm are listed below.

- An element that describes the edit operation that was applied to the *changed_node* is already created. That element is mentioned in the pseudo code as <*operation*> to imply that it is an element that represents one of the four edit operations.
- The *par_context* is a variable that defines the number of parent nodes that will be included in the delta file. The default value is two, but the user can change the value of the variable.
- The algorithm adds context information to the *changed_node* to create a hunk as we define it in section 2.5.2. Each hunk is added in the *treePatchFile* as an element named context, which will be refered to in the pseudo code as <*context*>.

Figure 5-5 shows that the ancestors of the changed_node in the treePatchFile are traversed twice. The first time is used to keep a track of all the nodes that will be included in the *treePatchFile*. All of those nodes are added in a vector named *path.* Notice that the valuable context information nodes should be more than 3. In case this condition is not satisfied, more nodes need to be added in the treePatchFile.

The second pass adds the context information to the <*context*> node. Every node of the vector *path*, as well as all of their siblings are added to the *treePatchFile*. In case that the *changed_node* is reached, the <*operation*> element is added in the *treePatchFile* as the first child of the *changed_node*. Finally, the <*context*> element is inserted to the *treePatchFile*.

```
1. Create an element <context> and copy it to a temp node.
2. Copy the changed_node to the current node.

3. For i=0 up to par_context AND valuable context information less
   than 3
      a. Copy the current node to the first position of a vector
         path.
      b. For each sibling of the current node
            i. If the node has valuable context information,
               increase valuable context information.
      c. Make the current node to be its parent node.

4. For each element of the vector path
      a. Add all the siblings of current as children of temp node
         and check if they are valuable context information.
            i. If current is equals to changed_node then make the
               element <operation> the first child of current node.

      b. Copy the next element of the vector path to the current
         node.
      c. Copy the current node to the temp node.

5. Add element <context> at the end of the treePatchFile.
```

Figure 5-5 – Algorithm for Delta File Creation

5.4 Patching Tool

Until now, all the issues that affected the creation of the *treePatchFile* were considered. This section explains how the delta file can be used in order to patch two arbitrary XML documents successfully.

The overall design of *treePatch* is based on the line-oriented algorithm of the UNIX *patch* program. The algorithm can be logically divided into two parts; the first one is responsible for locating the *changed_node* in *file 2*, whilst the latter applies the edit operation.

Function: Find Node

The inputs of this function are listed below:

- The function gets as input the <*context*> element, as it was defined in the previous section.
- The <*context*> element contains the *changed_node*, the context information of that node and the <*operation*> element that was described in the previous section.
- The default value of the *fuzz factor* is set to 2 (its use is described in the following paragraph). However, the user is able to change the value of this variable.

Firstly, the algorithm tries to locate the *changed_node* in *file 2*, using the path of the node in *file 1* (consider the discussion of the attribute named *node* that was discussed in section 5.2). In case that the path points at a node of the same type as the *changed_node* in *file 1*, the algorithm uses the context information to verify that the two nodes are the same. If the context information could not be matched and the *fuzz factor* is greater or equals to 1, the algorithm tries again to match the nodes, ignoring the first and the last child of each level of the context information. If the matching fails once more, and the *fuzz factor* is greater or equals to 2, the algorithm discards the first and the last two nodes of each level of the context information, etc. In case that the nodes are different, the algorithm ignores the path and uses only the context information to locate the *changed_node* in *file 2*.

In order to achieve that, all the nodes of the <*context*> element are traversed and in case that a node is marked as *valuable context information*, that node is compared to all nodes of the same type in *file 2*. If none of the nodes of the *file 2* matches, the <*context*> element is traversed again.

If the algorithm manages to match a *valuable context information* node with a node of the *file 2*, the rest of the context information nodes are used to verify that the algorithm found the *changed_node* in *file 2*. During this procedure, the *fuzz factor* is used in exactly the same way as it was described in the previous paragraph.

If this procedure finishes successfully, the *changed_node* is set as input for the next function. Otherwise, an error message that informs the user about the situation is

printed, and the hunk is added in the reject file. The algorithm is demonstrated in Figure 5-6.

1. For each one of the <*context*> elements

2. Use the path to find the node and if the path points out in a node of the same type as the *changed_node* then
   a. For *i=0* up to *fuzz factor*
      i. If the *context information* matches then return the node.
      ii. Else ignore the first and the last i^{th} children of each level of the context information and try to match the nodes.

3. If the algorithm cannot match the nodes then use the *context information*.

4. For each node in the <*context*> element
   a. If the current node is marked as *valuable context information* then
      i. If any of the nodes of the same type matches then
         1. For *i=0* up to *fuzz factor*
            a. If the *context information* matches then return the node.
            b. Else ignore the first and the last i^{th} children of each level of the context information and try to match the nodes.

5. If the algorithm fails to match any of the nodes in the <*context*>, return a message with the error and put the hunk in the reject file.

Figure 5-6 – The algorithm of the Find Node Function

Function: Apply Edit Operation

The second function of the patching algorithm applies the edit operation. However, in order for the operation to be applied to any node, all the special cases that were discussed in the previous chapter should be handled properly. The behaviour of *treePatch* for each one of those cases is thoroughly described there, so in this section will not be consider further.

Chapter 6

Implementation

This chapter considers all the issues that are related with the implementation of the differencing and patching tool. The discussion begins with a complete description of the technologies that were used during the development as well as a justification on the reasons that those were chosen over other technologies. Moreover, the structure of the source code of the application is demonstrated, in order to aid the continuation of this project be other developers. Finally, the options of the differencing and patching tool will be presented.

6.1 Technology

In order to develop the differencing and patching tool the Java Object Oriented Programming Language was used. The major motivations for this decision are the following; firstly, the differencing tool was already implemented in Java and secondly as it is a widely used programming language, it is easier for other developers to study and continue this work.

In order to parse the XML documents and use their components, a parser was required. The Apache's Xerces parser [37] was chosen, because it is open source and widely used. Apart from the parser, an API to handle the XML documents was also required.

The two most widely used APIs are the Simple API for XML (SAX) [24] and the Document Object Model (DOM) [32]. The former indicates the application every time

a new component appears, whereas the latter which was introduced by the World Wide Consortium, reads the complete XML document and transforms it into a tree structure. The reasons that the DOM Level 2 and Level 3 API was chosen over SAX, are discussed in the following paragraph.

First of all, the DOM API is significantly simpler than SAX. No matter how complex an XML document may be, DOM converts it into a tree, and then by using the interfaces provided by the API is able to modify, create or delete nodes. Everything is a node in the Document Object Tree. Moreover, DOM, contrary to SAX, allows random access to the XML document. Furthermore, even complex searches can be easily implemented. Finally, it provides the ability to read data from a document and write data into it. However, it is slower than SAX and unable to parse large files.

Xerces supports DOM Level 1 and 2 APIs as well as SAX. Finally, XPath [35] is used in the definition of EDUL.

6.2 Description of the Source Code

The four main classes of the current application are illustrated in Figure 6-1. Adrian Mouat uses the algorithm that is described in [7] to find the differences between two well-formed XML documents. This algorithm is divided into two parts, each one of those is represented by the classes named Match and EditScript respectively. The former part identifies the matching between two tree-structured documents, whereas the latter generates a minimum edit script, given the two documents and the matchings between them. This is the part of Adrian Mouat's code that it was used in the current implementation.

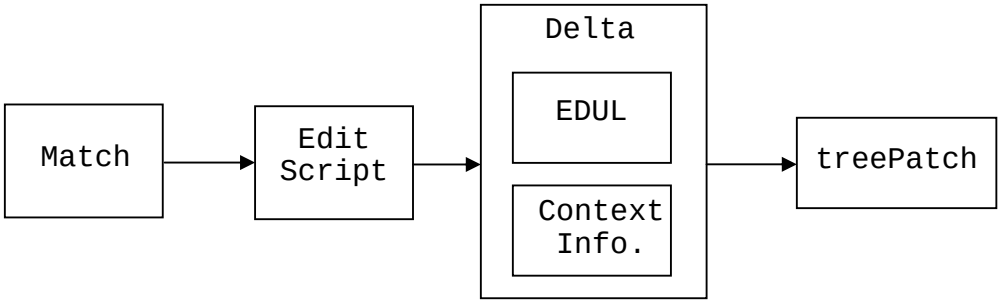

Figure 6-1 – The structure of the source code

The outputs of the EditScript class are the inputs the Delta class. The latter has two main methods; the first creates the treePatchFile using the EDUL Output Format as defined in the previous chapter, whereas the latter adds context information to each one of the nodes that an edit operation was applied to. The inputs of the *treePatch* class are the delta file and *file 2*. This class firstly locates the *changed_node* in *file 2*, and then apply the edit operation taking into consideration all the special cases that were discussed in chapter 4.

6.3 Manual Pages

This section describes both the way that the tool is invoked from the command line and the options that the differencing and the patching tool have. Primary issue during the implementation was to create a tool that would be easy to exercise for users that are familiar with the GNU programs.

6.3.1 Diff Tool

Some of the options of the differencing tool that Adrian Mouat described in his report were not implemented. Those options are excluded and all of the options that are presented here are actually used by the differencing tool. This section will be organised as the GNU tool manual pages.

NAME

diffXML – finds the differences between two XML documents

SYNOPSIS

diffXML [options] from-file to-file

DESCRIPTION

The base file is the *from-file* whereas the latter is *file 1*. The output of this program is the *treePatchFile* that describes the differences between those two files.

Options

Below is a summary of all of the options that diffXML accepts.

-q
--brief Report only if the files differ. It does not output any delta file.

-e
--ignore-empty-nodes
Ignore text nodes that contain only whitespaces.

-r
--ignore-comments
Ignore changes made in comment nodes.

-I
--ignore-processing-instructions
Ignore changes made in processing instruction nodes.

-V
--version Output the version number of the program.

-h
--help Print summary of options and exit.

-t
--tagnames Output tag names of elements rather than “node()” for node tests in XPath expressions.

-C nodes
--context [=nodes]
Specify the number of parent nodes to be added in the context information of the treePatchFile. If *nodes* is not given, it will default to 2.

DIAGNOSTICS

An exit status of 0 means no differences were found, 1 means some differences were found, and 2 means that an error occurred.

6.3.2 Patching Tool

This section is also organised as the GNU manual pages and describes the *treePatch* tool options. Among the options that are presented here, only the *fuzz factor* is not

implemented. Nevertheless, it was added since the algorithm for using it was described in chapter 5. The rest of the options are successfully implemented.

NAME

treePatch – applies a *treePatchFile* to an arbitrary XML document

SYNOPSIS

treePatch [options] [arbitrary file [treePatchFile]]

DESCRIPTION

patch takes a *treePatchFile* containing a difference listing produced by the diffXML program and applies those differences to one or more original files.

Options

Below is a summary of all of the options that *treePatch* accepts.

-F number
--fuzz-factor number
Set the maximum *fuzz factor*. If *number* is not given it will default to 3.

-V
--version Output the version number of the program.

-h
--help Print summary of options and exit.

-d
--dry-run Print results of applying the changes without modifying any files.

-R
--reverse Assume that this patch was created with the old and new files swapped. *treePatch* attempts to swap each hunk around before applying it.

DIAGNOSTICS

An exit status of 0 means no differences were found, 1 means some differences were found, and 2 means that an error occurred.

Chapter 7

Testing

The tests of the implementation were carried out during the whole period of the design and implementation of the tools. Subsequently, the remainder of this chapter is divided into two sections; the first one describes the tests that were carried out during the design of the tools, whereas the latter contains both the working and non-working tests that I did during the development of the tools. All of those tests can be found at the homepage of the dissertation (http://treepatch.sourceforge.net).

7.1 Tests during Design Phase

First of all, some tests were carried out to verify that Adrian Mouat's differencing tool satisfied the majority of the requirements that were described in the third chapter. Those tests showed the competences and the limitations of Mouat's differencing tool. They also gave solutions in problems that the previous implementation suffered from. So, they can be identified as part of the design of this dissertation.

Even more tests were performed during the design of the output format and the patching tool. The most significant of those were carried out in order to decide the kind and the amount of context information that should be included in the *treePatchFile*. The first tests only included the previous and the next sibling of the *changed_node*, as well as its parent node. However, that information was not enough, and the algorithm could easily be tricked. So, in the next tests more and more context information around the *changed_node* was built. The fact that the delta file is intended to be applied to an arbitrary XML document makes it necessary to include a lot of

context information. The results of those tests lead to include the context information that is shown in Figure 5-4. Of course this is not absolute, but seems to work well for most of the cases where the XML documents are not too large.

A list of all the tests that were carried out during the design of the current application is shown below.

- Some tests were made to explore the strengths and weaknesses of Adrian Mouat's differencing tool.
- A significant amount of time was spent on testing how much information should be included in the *treePatchFile.*
- Also, the notion of *valuable context information* was tested. The tests shown that in order handle the majority of XML documents, at least three of the node of the context information should be qualified as *valuable context information.*

7.2 Tests During and After the Implementation

Neither the working nor the non-working test files will be added here (the home page of this project includes one test for each one of those cases). Instead, this section will include a description of those tests.

7.2.1 Working Tests

This section describes the most significant working tests of the current application. It is also the proof that the differencing and patching tool fulfils the requirements stated in the third chapter. The categories of the working tests are listed below.

- The insert edit operation is working properly for all the types of nodes.
- The delete edit operation is working properly for all the types of nodes.
- The move edit operation is working properly for most of the cases (section 7.2.2 describes the non-working examples and explains the problem).
- All the special cases of the delete edit operation are working.
- All the special cases of the insert edit operation are working.

- All the special cases of the move edit operation are working.
- Edit operations were tested in conjunction and most of the cases are handled satisfactory.
- All the options that the differencing and the patching tool offer were successfully tested, apart from the *fuzz factor* that is not implemented.

7.2.2 Non-Working Tests

As it was mentioned in chapter 5, the differencing tool is unable to successfully identify the new position of the *changed_node* after the move operation is applied to it. In order to overcome this problem, information about the previous and the next sibling of the *changed_node* in *file 1,* was added in the *treePatchFile*. So, in the majority of the cases, the move operation is working fine. However, if both of the previous and next siblings are deleted in *file 2,* the patching will not be successful.

Moreover, a series of cases where the patching tool is tricked was tested. The major problem is that although *treePatch* does not take for granted that the nodes have unique attributes, it assumes that a group of those attributes is probably unique. Consequently, in case that the same group of attributes exist in more than one places in *file 2*, *treePatch* can be fooled.

Chapter 8

Evaluation and Conclusion

The final chapter of this document discusses the contributions of this thesis. Moreover, it compares the initial requirements of this project with the tool that was finally developed. Section 8.3 considers the limitations of the current implementation and makes suggestions for possible improvements. Finally, an overview of the thesis is presented.

8.1 Fulfilment of Requirements

In chapter 3, the requirements of the differencing and the patching tool were stated. Those were categorised into “Must Have”, “Should Have” and “Could Have”. The requirements of the first category are briefly listed below:

- Create a differencing tool that would take as input two well-formed XML documents, and would identify the differences between those two files. The algorithm should handle the documents as tree structures.
- Create a delta file that would describe the differences between two XML documents. The delta file should contain context information in order to be able to be used by the patching tool.
- Create a patching tool that would be able to apply the delta file that we previously created, with an arbitrary XML document.

All those requirements were met by the current implementation. Moreover, the majority of the “Should Have” and “Could Have” requirements were implemented.

The most important requirement that it was not implemented in the current tool is probably the use of the *fuzz factor*. The patching tool, after locating the *changed_node*, tries to match the node using as much as context information is possible, without using the *fuzz factor*. However, the algorithm is thoroughly described in chapter 5.

8.2 Achievements

It is true that the time available to design and implement this project was insufficient. Consequently, it was almost impossible to develop a bug-free tool that would be able to handle all XML documents. However, the author of this document believes that the contribution of this document and the tool is valuable.

First of all, this thesis documents all the requirements for the differencing and the patching tool as well as the delta file. Furthermore, a new output format, named EDUL, was thoroughly described in this document and it was used for the implementation of the tool. Moreover, in chapter 4 there is a careful study of all the cases where conflicts during the patching may occur, as well as a description of the desired behaviour of the tool in all those cases. Finally, a tool that meets the majority of the requirements stated and handles properly all the special cases was developed. The proof of those achievements can be found in the tests of the tool that are described in the previous chapter and exist in the home page of the project.

8.3 Limitations and Further Work

Although most of the requirements that were described in the third chapter were met, there are various ways in which the current tool could be improved. The rest of this section discusses those suggestions.

First of all, most of the bugs of the differencing tool that Adrian Mouat developed still exist. The reason is that the interest of this thesis moved towards the creation of a useful difference output format and the development of a patching tool for arbitrary XML documents. So, the bugs of the differencing tool were fixed only in cases that they were thought to be an obstacle for the development of the patching tool.

The limitations of the differencing and patching tool are listed below.

- The differencing tool could support more edit operations. For example, it could support the deletion or copy of a subtree.
- The differencing algorithm does not always produce the minimum cost edit script. This results to slower executions of the program and delta files that grow increasingly.
- Options of ignoring whitespaces or character case are not supported by the differencing tool.
- The patching tool is not interactive. It only creates a reject file in case a conflict occurs and notifies the user.
- The tool is not bug-free. There cases where they fail to find the differences and patch correctly a pair of XML files.
- The tool is implemented using the DOM. The limitations of DOM were discussed in section 6.1.
- Namespaces and DTDs are ignored by the tool. This is a significant limitation that should be overcome in future versions of the tools.
- The tool is invoked by the command line. Alternatively, a user graphical interface could accompany the tool.
- The tool could be invoked through the World Wide Web. This is an interesting extension for the current implementation. Chapter 2 shown that some of the existing tools already support this feature.

8.4 Public Release

A home page that contains this document, the source code and all the tests that were carried out throughout this period, was created to encourage other developers to read and continue this work. In order to do that the author of this document contacted the Source Forge Open Source Software Environment [26], and explained them the reasons that make this project worthwhile. All the material can be found at http://treepatch.sourceforge.net.

8.5 Conclusion

All the existing tree-structured patching tools are limited to apply the delta file to one of the original files. This document accompanies a project that managed to overcome this limitation and create an open-source tree-structured patching tool for arbitrary XML documents. Although the time constrains of this dissertation did not allow the development of a robust and bug-free tool, the outcoming results are significant.

The notion of synchronising XML files can be extremely useful because XML documents can encapsulate data. However, this research area is still unexplored. A really worthwhile effort is done by the research team of the University of Pennsylvania, called Harmony Project. Although the project is still in early stages, their aim is to create a tool able try to synchronise heterogeneous tree structured data.

To conclude, the author of this document anticipates that the end of this document will not be the ending of this project. Hopefully, the open source community will show an interest in this work, and this document will be a starting point for further development.

References

[1] Alphaworks Emerging Technologies. http://www.alphaworks.ibm.com/. accessed May 2003.

[2] D. Avison, G. Fitzgerald In *Information Systems Development: Methodologies, Techniques and Tools;* McGraw-Hill Publishing Company; Third Edition, published 2003, ISBN 0077096266.

[3] B. Berliner. "CVS II: Parallelizing Software Development" Proceedings of the USENIX Winter 1990 Technical Conference.

[4] E. Bertino, G. Guerrini, M. Mesiti, I. Rivara, C. Tavela. "Measuring the Structural Similarity among XML Documents and DTDs", 2001.

[5] S. Chawathe. "Comparing hierarchical data in external memory", Proceeding of the 25th International Conference on VLDB, Philadelphia, 1999.

[6] S. Chawathe, H. Garcia-Molina. "Meaningful change detection in semistructured data", SIGMOD, Tuscon, Arizona, May 1997, p. 26-37.

[7] S. Chawathe, A. Rajaraman, H. Garcia-Molina, J. Widom. "Change detection in hierarchical structured information", SIGMOD, vol. 25, num. 2, 1996, p. 493-504.

[8] G. Coneba, T. Adbessalem, Y. Hinnach. "A comparative study for XML change detection", 2002.

[9] F. Curbera, T. Poon, D. A. Epstein. IBM Research Division, T.J. Watson Research Center. www.ibiblio.org/bosak/conf/xmldev99/curbera/curbera.pdf, August 1999.

[10] DeltaXML by Mosell EDM Ltd. http://www.deltaxml.com. accessed May 2003.

[11] M. B. Greenwald, J. T. Moore, B. C. Pierce, A. Schmitt. "A Language for Bi-Directional Tree Transformations", Technical Report, Department of Computer and Information Science, University of Pennsylvania, 2003.

[12] The Harmony Project, http://www.cis.upenn.edu/~bcpierce/harmony/. accessed August 2003.

[13] L. Khan, L. Wang, Y. Rao. “Change Detection of XML Documents Using Signatures”, Workshop on Real World RDF and Semantic Web Applications, 2002.

[14] R. La Fontaine. “A delta format for XML: Identifying changes in XML and representing the changes in XML”. XML Europe 2001, Berlin.

[15] R. La Fontaine. “Merging XML files: A new approach providing intelligent merge of xml data sets”. XML Europe 2002, Barcelona.

[16] Linux Online Inc. http://www.linux.org. accessed May 2003.

[17] Microsoft Corporation. http://apps.gotdotnet.com/xmltools/xmldiff/. accessed June 2003.

[18] A. Marian, S. Abiteboul, G. Cobena, L. Mognet. “Change-centric management of versions in an XML warehouse”, Proceedings of the 27th VLDB Conference, Roma, Italy, 2001.

[19] A. Mouat. “XML Diff and Patch Utilities”. Master’s thesis, Heriot Watt University, School of Mathematical and Computer Sciences, 2002.

[20] A. Nierman, H. V. Jagadish. “Evaluating Structural Similarity In XML Documents”. Proceedings of the Fifth International Workshop on the Web and Databases, Madison,WI, June 2002.

[21] B. C. Pierce. “Synchronize globally, compute locally”, Research day 2002 on Global computing, EFPL, Lausanne. 8 July 2002.

[22] B. C. Pierce, J. Trevor, J. Vouillon. “Unison, A file synchronizer and its specification”, University of Pennsylvania, TACS 2001, Sendai.

[23] S. M. Selkow, “The tree-to-tree editing problem”, Information Processing Letters, 6, (6), 184--186 (1977).

[24] Simple API for XML (SAX). http://www.saxproject.org

[25] E. Siever, P. J. Hekman, S. Spainhour, S. Figgins In *Linux in a Nutshell;* O’Reilly UK; Third Edition, published August 2000, ISBN 0596000251.

[26] Sourceforge. Open Source Development Environment. http://sourceforge.net.

[27] K. Tai. “The tree-to-tree correction problem”, Journal of the ACM, 26(3), July 1979, p. 422-433.

[28] W. F. Tichy, F. Walter. “Design, Implementation, and Evaluation of a Revision Control System” Proceedings of the 6th International Conference on Software Engineering, IEEE, Tokyo, September 1982.

[29] XML TreeDiff by IBM. http://alphaworks.ibm.com/tech/xmltreediff/. accessed May 2003.

[30] VM Tools by VM Systems. http://www.vmguys.com/vmtools/. accessed May 2003.

[31] Y. Wang, D. J. DeWitt, J. Cai. “X-diff: An effective change detection algorithm for xml documents”. Technical report, University of Wisconsin, 2001.

[32] World Wide Web Consortium. The Document Object Model Level 2 Core. http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-0001113/. accessed July 2003.

[33] World Wide Consortium. Extensible Markup Language (XML). http://www.w3c.org/XML. accessed June 2003.

[34] World Wide Consortium. Extensible Stylesheet Language Transformation (XSLT). http://www.w3.org/TR/xslt. accessed July 2003.

[35] World Wide Consortium. XML Path Language (XPath). http://www.w3.org/TR/xpath. accessed June 2003.

[36] XEmacs Webmaster. http://www.xemacs.org/Documentation/ packages/html/ediff.html. accessed May 2003.

[37] Xerces by Apache Group. XML Parsers in Java and C++. http://xml.apache.org.

[38] K. Zhang, D. Shasha. “Fast algorithm for the unit cost editing distance between trees”, Journal of algorithms, vol. 11, p. 1245-1262, December 1990.

[39] K. Zhang, D. Shasha. “Simple fast algorithms for the editing distance between trees and related problems”. SIAM Journal of Computing, vol. 18, p.1245-1262, December 1989.