Bachelor Thesis

Delta.js — A JavaScript diff and patch engine for DOM trees

Lorenz Schori

January 20, 2012

Advisor: Prof. Dr. Olivier Biberstein
Examiner: Peter Matti

Bern University of Applied Sciences
Department of Computer Science

Abstract

In the software industry any professional team uses version control systems to keep track of their source code. Those systems are all based upon algorithms capable of detecting changes between different versions of a file. This process is commonly referred to as *diffing*. At least as important as detecting changes is the complementary operation namely *patching* or *merging*, i.e. applying those changes to the original document or a slightly changed version thereof. However common version control systems are optimized for plain text files and therefore do not work very well for structured documents. We suppose that this is one of the reasons why generic version control systems are not very useful for people who mainly work with content.

Many software packages already are capable of recording changes to single documents. However sometimes it is desirable that the history of a document or even a set of multiple documents of different types can be tracked without having to rely on the editor software. Because more and more vendors are switching to standardized document formats, it is now feasible to apply a generic approach on versioning structured documents.

In this work we present implementations of a *diff* and a *patch* algorithm suitable for structured documents. We also compare them to related approaches and outline the research which has led to their development.

We hope that this work will serve as a starting point of subsequent projects in the domain of version control and document management.

Contents

1. Introduction

1.1. Motivation and Goals

1.1.1. A Need for Robust and Effective Version Control

When different authors are working on the same document, the ability to track and manage changes is essential. Standard desktop software, e.g., word processors, often provide mechanisms for revision control out of the box. Also many web based collaborative editors like wikis and content management systems provide means to review history and rollback to previous versions of a document. In order to prevent conflicting changes, authors either have to manually coordinate editing sessions among them or a locking system needs to control access to the document. But both solutions typically block access to the entire file during an editing session, preventing effective collaborative work in many cases.

In order to mitigate the locking problem, vendors of office software are now introducing online real-time collaboration features. In some text editors and word processors, authors may share an editing session with other users on the same network and even accross the internet[1]. Also browser based office suites are appearing nowadays which allow real-time collaboration on structured documents hosted on a server in the internet.

However all those version control and collaboration methods are tied to the host application. As a consequence the involved editors are forced to use the same software. While this situation might be acceptable for small teams working in a managed office environment, such a restriction might result in major problems when a workflow requires integration of third party systems.

Also due to the rise of new classes of mobile devices like smartphones and tablets, it will likely become difficult to maintain a uniform software landscape even throughout a single organization. Additionally real-time collaborative editors require a permanent and reliable connection to other participants in an editing session. Obviously this is a problem for mobile users.

Distributed teams in the software industry heavily rely on version control systems which are capable of tracking and merging changes to source code. Often individual developers use different editors and work with a more or less customized set of tools which best fits their working style. In such an environment it is very important that the version control system may be used independently of a document editor.

Source code management systems detect modifications by comparing two versions of a file. Inserted and deleted text is recorded in a so called changeset along with additional

[1]The first implementation of a collaborative real-time editor was presented at legendary "The Mother of All Demos" by Douglas Engelbart in 1968 (refer to Stanfords Mouse Site for more information `http://sloan.stanford.edu/MouseSite/1968Demo.html`). Recent implementations include SubEthaEdit (Mac OS X `http://www.codingmonkeys.de/subethaedit/index.html`) and ACE (Mac/Windows/Linux `http://sourceforge.net/projects/ace/`, the software was developed as part of the Bachelor Thesis of M. Bigler, S. Räss and L. Zbinden at the Bern University of Applied Scinces, Department of Computer Science). Also current versions of Abi Word (`http://www.abisource.com/`) and Microsoft Office (`http://office.microsoft.com/`) provide this feature.

data like edit location and surrounding context lines, revision date and author information. A changeset can be shared with team members, can be posted on a mailinglist for public review or simply kept for later reference. Because in distributed teams it is common that more than one person is working on the same file at a time, it is important that the source code management system is capable of merging changes into a file which has been modified since the changeset originally was recorded. As long as no two editors changed the same text at the same location in a file, the version control system usually is capable of applying the changeset.

The outlined mechanism of version control in source code management systems works very well for plain text files. However current implementations are still not suitable to track and manage changes on structured documents. Therefore a workflow involving an application independant version control system allowing a distributed team of authors to collaboratively edit a set of structured documents without extensive synchronization is currently barely realizable.

1.1.2. The Fruits of Standardization and Open Formats

During the last decade, XML[2] was adopted throughout the software industry as a reliable and standardized way to store structured information into files. Many major vendors are dropping their proprietary file formats in favour of XML based documents, opening up the opportunity for third party software to more easily retreive and process information contained within them. Presumably this shift of paradigms from vendor specific opaque file structures to open file formats has been accelerated by the success of HTML, the lingua franca of the world wide web and also stemming from the same roots as XML.

While XML and HTML define the rules on how structured information gets written to and is read from files, the DOM[3] interface specifies a set of standardized methods allowing us to retrieve and change this information. The DOM interface is implemented in every modern web browser and accessible using JavaScript. Also there are implementations of the DOM in many other programming languages.

1.1.3. JavaScript Beyond the Web

The web browser is probably one of the most widely deployed class of software. People access websites not only using their workstations and laptops but increasingly also using mobile phones, tablets and e-book readers. Although web browser software and underlying operation system vary, the vast majority of those devices is capable of running web applications, a combination of HTML documents generated on a web server and JavaScript code executed on the client. The web browser therefore can be regarded as an application platform just like Windows, Mac OS X or a Linux distribution.

[2]XML: Extensible Markup Language. Overview of Specifications form the World Wide Web Consortium (W3C):
`http://www.w3.org/standards/techs/xml`

[3]DOM: Document Object Model, Specification Overview from W3C:
`http://www.w3.org/standards/techs/dom`

Although the browser is the original and most important platform for applications built using JavaScript, the language is spreading to other domains. Most notably in recent time is the fast pacing development of JavaScript-based server platforms. It has been adopted as embedded scripting language of application software as well[4].

Last but not least, JavaScript has some features which are interesting from a developers point of view.

1.1.4. Putting it Together

The goal of this work is not to present a turnkey solution for versioning XML documents or DOM structure. However our implementation should be capable of detecting changes in structured documents, it should provide suitable file formats for the changesets and it should include methods to merge changes back into the original document. Even if that was modified in the meantime. The solution should run in modern web browsers as well as under *node.js*[5], a JavaScript server environment.

Aside from the implementation goals we want to identify tools and techniques supporting professional software development in JavaScript.

The next section presents an overview of previous work, followed by an indepth analyzis of the LCS algorithm[6] (Section 3). The latter playing an important role in the tree matching algorithm XCC[7] (Section 4) our own implementation is based on. In Section 5 we present a modified version of the original XCC patch format as well as the algorithm utilized to apply such patches on documents. Following that we give an overview of our implementation and important design decision in Section 6 before discussing our results in Section 7.

2. Comparing and Merging Structured Documents

This section is an excerp from our Project Thesis covering difference algorithms and merging strategies for structured documents [23].

2.1. Fundamentals

Before discussing difference detection between trees we will point out some important terms and expressions. The model we present is very simple. As we get deeper into the matter we will introduce more details.

2.1.1. Simple Model for String Sequences

The difference between two documents is commonly represented as a list of operations that will convert the first document into the second [17]. A good diff-algorithm tries

[4]JavaScript is for example available as a scripting language in many products from Adobe Systems. Most interesting in this context probably is the publishing suite around Adobe InDesing

[5]Node.js project website: http://nodejs.org/

[6]LCS: Longest Common Subsequence algorithm [18]

[7]XCC: XML Change Control [21]

to minimize the number of operations required in order to produce a space efficient representation of the changes between two documents. The number of edit operations (see Definition 3) is commonly referred to as the *edit distance*.

Most of the time we are interested in the actual changes instead of just a bare number showing how much a document differs from another. The *edit script* represents this list of operations introduced above (See Definition 1).

Definition 1. Edit Script

Edit Script: A sequence of operations required to convert one document into another.

Anchor: A data unit used by the patching algorithm to identify a location where an operation must be applied.

Sometimes we need even more information in order to allow automatic verification of the applicability of a given edit script to a target document. We refer to those more sophisticated type of change representation as a *Patch* (Definition 2).

Definition 2. Patch

Patch: An edit script with context information.

Context: Content that all compared documents have in common in the neighborhood of where edit script operations will be applied.

Full context patch: A patch containing the intersection of all compared documents in addition to the edit script.

At the very minimum two operations are required to express the conversion from one document into another: *insert* and *delete*.

Definition 3. Operation

insert $(anchor, content[, context])$

delete $(anchor[, oldcontent, context])$

Note that *context* and *oldcontent* typically is not provided in serialized edit scripts but only in patch formats.

2.1.2. Conforming Edit Script

Usually it is not enough to just compute any edit script, instead one wants to find a particular solution conforming to a given criterion. This requirement is easy to fulfill by introducing a cost-function for operations such that we can find a conforming edit script by calculating the total cost of each candidate and selecting the one with minimal cost.

2.1.3. Longest Common Subsequence vs. Shortest Edit Script

Given a constant cost function ($cost : operation \rightarrow 1$) a *Minimal Conforming Edit Script* between two sequences A and B is identical to the *Shortest Edit Script.* Myers [18] has shown that the computation of the shortest edit script is dual to the problem of finding the *Longest Common Subsequence* (LCS) of the two sequences A and B.

2.1.4. Properties of Patch Formats

One way to serialize an *edit script* expressing the changes between two documents is simply recording the *operations* along with their parameters and *anchors* to a text file[8]. The *anchor* corresponds directly to a file location, e.g. a line number for flat text files. No context information is recorded in the edit script file. Therefore an *edit script* in this form can only be applied savely to the unmodified original document.

In addition to file location, the *anchor* in a patch file format[9] also contains *context* information. Context does not only consist of the content affected by the operation but also of neighboring content which both, the original and the changed file have in common. A patching algorithm may leverage this information in order to resolve and verify operation locations even if the original file was slightely modified since the patch was computed.

We require patch file formats to be *invertible*: A patch ($P : A_1 \rightarrow A_2$) containing the instructions to convert a given file A_1 into a later version A_2 should provide all the necessary information such that a reverse patch ($P^{-1} : A_2 \rightarrow A_1$) containing the instructions to convert A_2 into A_1 can be derived from it. Invertibility of patches is especially important in the domain of version control.

Another property we require is *commutativity* of patches: When applying two patches ($P_1 : A_1 \rightarrow A_2$, $P_2 : A_2 \rightarrow A_3$) to the file A_1 resulting in the file revision A_3, P_2 may be applied before P_1 as long as context and operations of the two patches do not overlap.

2.2. Tree Model

An XML document can be regarded as a rooted, ordered, labeled tree.

> **Definition 4.** Document tree: A rooted, ordered, labeled tree T consists of a set of nodes N, a finite alphabet Σ and a labelling function $L : N \rightarrow \Sigma$, which assigns a label to each node. R uniquely identifies the single root node and P denotes a function returning exactly one node representing the parent of a given node. Node order is determined by the ranking function r which returns the position of a node within its siblings. The value function V may return user data for leaf nodes. This definition closely follows the one given by Bille [4].
>
> $$T = (N, R, P : N \rightarrow N, L : N \rightarrow \Sigma, \Sigma, V : N \rightarrow value, r : N \rightarrow int)$$

[8]With GNU diff an edit script representation can be produced using the `-e` switch on the command line

[9]Use the switch `-u` in GNU diff to produce patches in the unified patch format which shows the properties we describe here

Note that especially in graph theory node-labels normally are considered unique. In contrast in the literature on tree comparison, *node labels are not unique* and therefore may not be used for object identification.

Also note that some applications like databases may ignore node order completely while for other applications like word processors the order is an important aspect of the document and must be taken into account when comparing two files. If not stated otherwise node order is important in the following discussion of algorithms.

2.2.1. Requirements for Patch Format

It is time to revisit our definition of edit scripts and patches and extend them in order to match the new requirements of hierarchical structured documents.

Edit Script: *Anchors* must be extended such that we can specify paths relative to a reference node. For practical reasons most of the time the reference will be the root node.

Patch: *Context* must be expressible in terms of nearby nodes, namely ancestors, siblings and descendants.

2.2.2. Tree Operations

When operating on sequences of characters or lines it is sufficient to define two operations in order to construct an edit script, namely insert and delete. In order to express edit scripts in tree structures with the properties we defined above, we introduce one more operation: *relabel*. Definition 5 specifies those basic tree operations. Several algorithms replace this basic set with methods operating on whole subtrees or sequences of subtrees (See Definition 6). Figure 1 depicts basic- and extended operations on trees.

2.3. The Generic Tree to Tree Correction Problem

2.3.1. Tai (1979)

In 1979 Tai expressed the *tree-to-tree correction problem* [24] as a generalisation of the edit distance on sequences. Today his algorithm has no practical relevance anymore but it still serves as the basis of adapted and improved algorithms.

Maximum complexity of Tais algorithm in time as well as in space is $O(|T_1||T_2|D_1^2 D_2^2)$ where $|T_n|$ denotes the number of nodes and D_n the maximal depth of a tree. The worst case where no two nodes from T_1 and T_2 have the same label and both trees have maximal depth therefore results in an upper bound of $O(|T_1|^2|T_2|^2)$ in time and space. If both trees have a similar number of nodes n then we finally get worst case upper bound in time and space complexity of $O(n^4)$.

Definition 5. Basic tree operations

insert $(anchor, node[, context])$
The *anchor* points at a consecutive sequence of siblings S with a common parent P. Remove the node sequence S from P and insert the *new node* N at the place where S was before. Append S as the list of child nodes to N. After the operation P is the new parent of N and N is the new parent of S.

delete $(anchor[, oldnode, context])$
Let P be the parent of the *anchor* A pointing to the node N being deleted. Remove N from P and insert all of N child nodes where N was before.

relabel $(anchor, newlabel[, oldlabel, context])$
Replace the label of the node the *anchor* is pointing at with a new label. Child nodes and subtrees are not affected by this operation.

Definition 6. Extended tree operations

insert tree $(anchor, node[, context])$
Let P be the parent of the *anchor* A, pointing to an existing node or at the end of the children list of P. Insert the new node N including children nodes and subtrees into the children list of P at the position before A.

delete tree $(anchor[, oldnode, context])$
Let P be the parent of the *anchor* A, pointing to the node N being deleted. Remove N including all its child nodes and subtrees from P.

update $(anchor, newvalue[, oldvalue, context])$
Replace the value of the node the *anchor* is pointing at with a new value. Child nodes and subtrees are not affected by this operation.

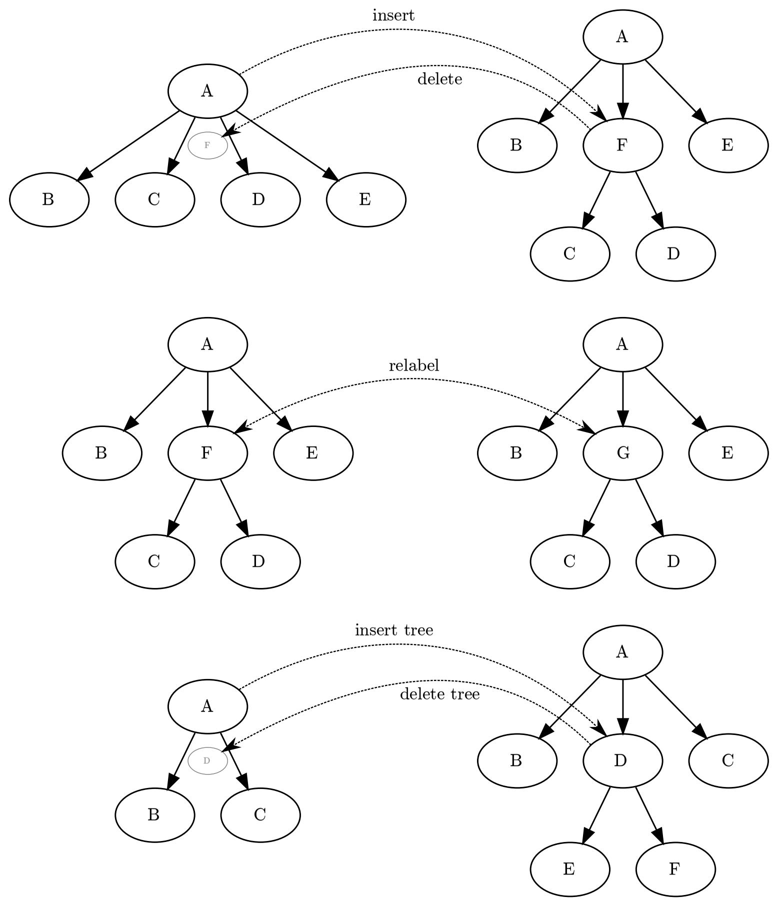


Figure 1: Tree edit operations. "Insert", "delete" and "relabel" operate on node level while "insert tree" and "delete tree" extend to whole subtrees

2.3.2. Zhang and Shasha (1989)

Zhang and Sasha published a new algorithm in 1989 that improves on runtime and space requirements [26]. Key difference to Tais algorithm is the postorder traversal of the trees. In Tais algorithm the comparison of two trees starts on the root node then each of its child nodes is visited recursively until the right most leaf node of both trees is reached. In contrast Zhang and Shasha algorithm starts with the leftmost leaf of both trees working its way through its siblings then going through parents siblings until finally reaching the root node (see Figure 2).

Interestingly enough this algorithm actually operates on ordered forests[10] and therefore solves an even more generic problem than Tais algorithm.

In order to separate tree distance and forest distance calculations, Zhang and Sasha introduced so called keyroots, i.e. nodes having a sibling on their left and the root node. During edit distance calculations, an array of the forest distance between two keyroots is maintained(see Figure 2).

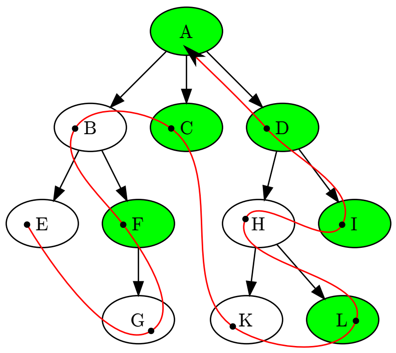


Figure 2: Zhang Sasha: postorder tree traversal (red line) and keyroots (green shaded nodes).

Upper bound in complexity of this algorithm is $O(|T_1||T_2|min(L_1, D_1)min(L_2, D_2))$ in time and $(|T_1||T_2|)$ in space. Again $|T_n|$ denotes number of nodes D_n tree depth and L_n number of leaves. Worst case input for this algorithm is a completely one-sided tree with n nodes where $\frac{n}{2}$ are leaves and with a depth of $\frac{n}{2} + 1$ as shown in Figure 3. Considering two trees having the same size n, worst case time complexity becomes $O(n^4)$ while worst case space complexity remains $O(n^2)$.

2.3.3. Klein (1998)

A refinement on Zhang and Sashas algorithm was proposed by Klein [14] in 1998. The decomposition of trees into "heavy paths" yields another improvement on maximum time complexity. However Zhang and Shashas algorithm still outperforms Kleins on many input sets. On the other hand Kleins algorithm also is suited for edit distance computation between unrooted trees, i.e. trees without a designated root node.

[10]Ordered forest: sequence of ordered trees

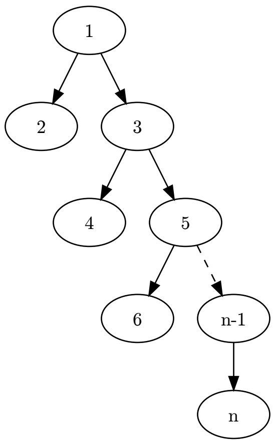


Figure 3: Worst case input tree for Zhang Sasha algorithm

Kleins algorithm runs with $O(|T_1|^2|T_2|log|T_2|)$ in time and with $(|T_1||T_2|)$ in space. For two similar sized trees with n nodes, the complexity therefore becomes $O(n^3\ log\ n)$ in time and $O(n^2)$ in space.

2.3.4. DMRW (2009)

In 2003 Dulucq and Touzet showed that Zhang Shasha as well as Klein algorithm can be described within a more general framework [11]. They introduced the notion of *decomposition strategy* as the discriminator between those algorithms. Based on that work Demaine, Mozes, Rossman and Weimann improved the Klein-algorithm in 2009 and achieved the new worst case time complexity of $O(n^3)$ [10].

2.3.5. Recap

In this section we pictured the milestones in the development of tree edit distance algorithms aimed at solving the generic tree to tree correction problem by finding the minimal conforming tree edit distance. In scientific domains, e.g. when working out the differences between RNA secondary structures, cubic worst case runtime may be acceptable. When comparing structured documents however it might be interesting to trade minimality of the edit script for better runtime performance.

2.4. Unit Cost Algorithms

2.4.1. mmdiff and xmdiff (1999)

Recalling the duality of *shortest edit script* and *longest common subsequence* (see Section 2.1.3) it is possible to design more efficient edit distance algorithms by constraining the cost-function of operations to a constant (unit) value. Chawathe [5] presented two algorithms using the same technique for computing the tree edit distance like the one introduced by Myers [18] for sequences.

While *mmdiff* is designed to run in main memory, *xmdiff* is able to handle arbitrary big documents. For *mmdiff* upper limits in time and space is $O(|T_1||T_2|)$, boiling down to $O(n^2)$ for similar sized trees. While having constant upper limits in memory usage, the *xmdiff* algorithm introduces quadratic IO costs.

2.5. Diff Algorithms for Structured Hierarchical Data

2.5.1. Extended Zhang Sasha (1995)

The changes introduced by people when editing documents are normally more complex than the three basic operations *insert, delete, relabel* defined previously (see Definition 5). Therefore an edit script comprising only of basic operations may mask the actual meaning of the changes found between two document versions. In 1995 Barnard, Clarke and Duncan proposed an extended version of Zhang Sasha algorithm [3] in order to enhance the expressivity of an edit script in the domain of document comparison.

Extended Zhang Shasha introduces three additional operations performed on whole subtrees: *insertTree*, *deleteTree*, *swap*. Swapping is only allowed on adjacent siblings.

While this algorithm improves on edit script semantics and generally produces smaller deltas with less operations compared to the original Zhang Sasha algorithm, it does not help narrowing complexity bounds and memory requirements. However it produces minimal conforming edit scripts, just like its predecessor.

2.5.2. FastMatch EditScript - FMES (1996)

FMES was presented by Chawathe, Rajaraman, Garcia-Molina and Widom in 1996 [7] as a complementary algorithm to Zhang Sasha tailored to generating edit scripts in structured documents. The set of operations is however rather different to the one of Zhang Shasha. In FMES *insert* and *delete* operations are restricted to leaf nodes, *relabel* is substituted by an *update* operation targeting node values instead of node labels. Additionally a *move* operation is introduced capable of changing the parent of a given subtree and also its position within its siblings.

A key property of this algorithm is the separation of change detection into two sub-problems:

1. Find a good matching between two trees
2. Compute the edit script

If object identifiers are present in the data the solution to the first problem is trivial and leads to a speedup of the whole process. The algorithm is capable of assigning object identifiers if necessary based on node labels and values. For interior nodes the matching criterion is based on their child nodes.

The matching algorithm is based on a set of criteria and assumptions appropriate for structured data in the domain of document processing resulting in faster runtime at the expense of potentially non-minimal edit scripts:

Criterion 1: Leaf nodes can be matched only if their labels are equal and their values are similar enough.

Criterion 2: Internal nodes can be matched only if a certain percentage of their leaves match.

Assumption 1: Labels follow a structuring schema where certain labels only are allowed as child nodes of others.

Assumption 2: Every node in one tree only has at most one node in the other tree resembling it closely.

Upper bound in time complexity for this algorithm is $O((L(T_1) + L(T_2))e + e^2)$ where $L(T_n)$ represents the number of leaf nodes of a given tree and e is the weighted edit distance (typically, $e \ll n$) [7] representing the sum of the weights of all operations where an *insert* and *delete* each count 1, an *update* counts 0 and the weight of a *move* is equal to the number of leaf nodes which are descendants of the node in question. Given two similar sized trees of the size n which do not have any nodes in common, worst case time complexity becomes therefore $O(n^2)$.

LaDiff, the authors implementation of FMES, took two versions of a LaTeX document and generated a third one with annotations on additions and deletions as well as indications where parts of text were moved to another location. An example is given in [6].

2.5.3. BULD (2001)

Unlike FMES, the BULD algorithm by Cobéna, Abiteboul and Marian is designed exclusively for XML documents [9]. Operations are similar to FMES however *insert* and *delete* target subtrees and not leaf nodes.

In some XML structures it is common that certain tags occur more frequent and in consecutive sequences — for example think of paragraphs (P-Tag) in an XHTML document. Therefore XML tag names are not the best choice as a mapping criterion. Instead a hash on node values and subtrees is calculated which is used in order to find corresponding nodes and subtrees in the other document. When a DTD is available, BULD will also consider ID attributes.

The BULD algorithm runs in $O(n\ log\ n)$ time and $O(n)$ space where n is the number of nodes of both documents.

2.5.4. faxma (2006)

A rather unique approach on finding differences in XML documents was presented by Lindholm, Kangasharju and Tarkoma in 2006 [15]. Instead of matching the tree structure of documents, the sequence of tokens emitted by the XML parser is compared using a "rolling hash"[11]. A similar method is used in the `rsync` tool. After working out the common parts in the token streams, the results are mapped back to the XML tree structure (the *metadiff*).

Runtime is expected to be linear for two document versions with small and local changes. However for two completely different documents of the same size n, the algorithm runs in $O(n^2)$.

2.5.5. XCC (2010)

Recently Rönnau and Borghoff released a framework consisting of libraries and tools for diffing, patching and merging XML office documents [21]. The diff algorithm shares some aspects of BULD. The operations *insert* and *delete* are extended to address tree-sequences. The *move* operation is realized by simply interlinking equivalent *insert* and *delete* operations. Like in BULD hash-values are calculated for all nodes in a bottom up manner.

Worst case complexity is $O((L(T_1) + L(T_2))D + I(T_1) + I(T_2) + D)$ in time and $O(|T_1| + |T_2|)$ in space where $|T_1| + |T_2|$ is the sum of the number of nodes in both documents, $L(T_n)$ the number of leaves, $I(T_n)$ the number of internal nodes and D the minimal edit distance. Note that the first expression is due to the use of Myers $O(ND)$ difference algorithm for finding the Longest Common Subsequence among the leaves of both trees. Considering two completely flat trees with only one internal node each (the root node), where the value of no two leaves are equal, worst case time complexity becomes $O(n^2)$ for two trees of the size n because of parameter $D > n$. Complexity in space remains linear though.

The authors claim that the final move-detection step does not influence complexity because a hash-map with linear lookup time is used to match equivalent insert and delete operations.

The LCS (Longest Common Subsequence) and the XCC diff algorithm are discussed in depth in Section 3 and Section 4 respectively.

2.5.6. A Note on the move Operation

Recall that no single algorithm for the generic tree to tree correction problem presented in the former section employed a *move* operation. In order to understand the problem imagine an edit script representing the changes between two ordered trees where all move operations have been deleted. The result is exactly an edit script representing the changes between two *unordered* trees. However this problem has been shown to be NP-complete for the general case [4].

So what made it possible to devise algorithms with less than quadratic complexity and support for the *move* operation at the same time? By matching corresponding *insert* and *delete* operations during a post processing phase, minimality of the resulting edit script

[11] A rolling hash is a hash value calculated over the symbols within a window of a fixed size. Symbols outside the window do not contribute to the hash value. However when the window is “slided” along the input sequence, the hash values of overlapping windows can be reused which makes the calculation of subsequent values very efficient. The Rabin Fingerprint is an example of a rolling hash.

is not guaranteed anymore which is unacceptable for the generic case but reasonable in the domain of document comparison [8].

2.5.7. Recap

We can identify several key ideas in the work done by a number of research groups in order to adapt the generic tree edit distance problem to the domain of document comparison:

1. Constrain the cost model of operations (See 2.5.1). Instead of allowing to assign a cost to each and every single operation within an edit script, the cost model is constrained to constant units in order to reduce runtime complexity.
2. Heuristically reduce possible candidates in the matching phase.
3. Accept non-minimal edit scripts in order to further reduce complexity.
4. Adapt scope of operations to enhance readability (i.e. use one *insertTree* instead of many *insert* node operations). Also introduce new operations like *move* to better reflect the meaning of changes.

3. Longest Common Subsequence (LCS)

An algorithm identifying the Longest Common Subsequence (LCS)[12] of two strings plays the most important role, both in traditional line based diff-tools such as GNU diff and in the XCC diff algorithm for DOM trees, which will be described in greater detail afterwards. In this section we discuss Myers widely used LCS algorithm [18] featuring quasi-linear time and space complexity.

3.1. Myers Edit Graph

Myers introduced the concept of the edit graph which shows that finding the Longest Common Subsequence of two strings is a special instance of the single-source shortest path problem. Consider two strings A with the length N and B with the length M. Construct a grid with N columns and M rows, i.e. with $N+1$ vertical and $M+1$ horizontal grid lines. Label the vertical grid lines with the characters from A starting at the second grid line with the first character and ending with the last character at the last grid line. Same procedure is done for string B and the horizontal grid lines. Now each intersection (x,y where $x \in [0, N-1]$ and $y \in [0, M-1]$) between vertical

[12] Note that the Longest Common Subsequence is not equivalent to the Longest Common Substring problem. The Longest Common Substring of two strings is a sequence of *consecutive* characters contained in both strings. In contrast the elements of the Longest Common Subsequence do not have to be contiguous in both strings.

Consider the two strings "zappa" and "appear". The Longest Common Substring of those two strings is obviously "app" (i.e. zAPPa, APPear). The Longest Common Subsequence (LCS) however actually is a,p,p,a (i.e. zAPPA, APPeAer).

and horizontal grid lines represents a mapping from a single character from string A at position x to string B at position y.

Given that $(0,0)$ denotes the top left and (N,M) the bottom right corner of the grid, every vertex is connected to its right hand neighbor alongside the horizontal grid line as well as to the next vertex downwards along the vertical grid line. Each of those orthogonal edges represent an edit operation, i.e. *delete character at position x from string A* for horizontal edges and *insert character at position y from string B* for vertical edges. Additionally a diagonal edge from $(x-1,y-1)$ to (x,y) is introduced for every character in A at position x if it is equal to the character in B at position y.

The Longest Common subsequence is contained in the path from (0,0) to (N,M) with the maximum number of diagonal edges. Analogous the shortest edit script is represented by the path with the least orthogonal edges. By giving orthogonal edges a cost of 1 and diagonal edges a cost of 0, the problem of finding the path with the most diagonal and the least orthogonal edges becomes a shortest path problem in a weighted directed acyclic graph (See Figure 4).

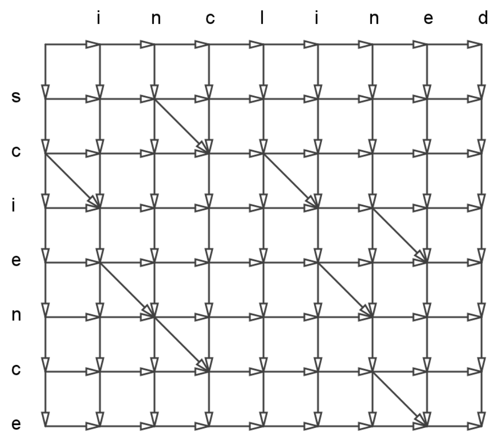


Figure 4: Edit graph: Horizontal edges represent delete- and vertical edges insert-operations. No change is necessary when following diagonal edges.

3.2. X-K Coordinate System

Considering that the most interesting elements of the edit graph are the diagonal edges, we do not use cartesian (x-y) coordinates but rather an alternative system in order to identify and store locations in the grid the edit graph is based on.

With $k \in [-M, N]$ a *k-line* is a straight line through $(k,0)$ with a slope of 1, i.e. a diagonal going through $(k,0)$, $(k+1,1) \ldots (k+n,n)$ in the grid (See Figure 5). Note that we can easily derive the y-component of any point in the grid when x and k are known using: $y = x - k$.

Therefore a *k-point* (x,k) translates to the cartesian point (x, $x - k$). For example the top-left corner of the graph is located at k-point (0,0) and the bottom-right corner is at k-point (N, $N - M$).

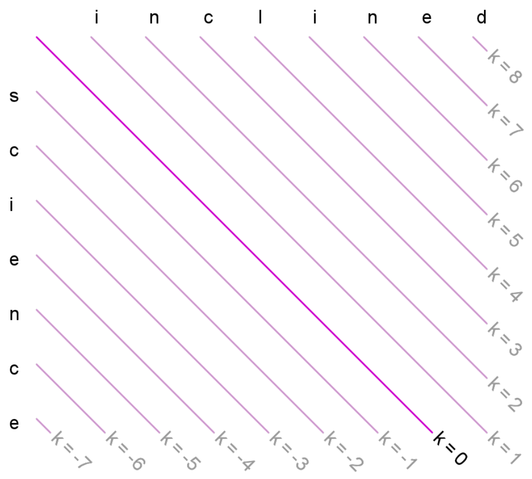


Figure 5: *K-Line*: Diagonal lines going through (k, 0) where $k \in [-N, M]$
.

3.3. Basic Greedy LCS Algorithm

Because the anatomy of the edit graph follows strict rules it is not necessary to build up a complete graph data structure and extract the shortest path using a generic algorithm like e.g. Dijkstra. Myers gives a basic greedy algorithm for finding the LCS in $O((N + M)D)$ where M and N are the number of elements in sequence A and B respectively and D denotes the number of edit operations necessary to turn A into B.

A *D-path* is a path starting at *k-point* (0,0) containing D orthogonal edges and any number of diagonal edges. Therefore a *D-Path* can be composed by a *(D-1)-path* followed by an orthogonal edge (i.e. one edit operation) and by zero or more diagonal edges. Myers used the term *snake* for the diagonal part of a D-path. We refer to the bottom-right end of a snake as the *snake-head*.

Starting from *k-point* (0,0) and $D = 0$ we extend iteratively the *furthest reaching D-path* on every possible *k-line*, incrementing D after each iteration. The resulting x-value on a given k-line is stored into the map V indexed by k. The starting points for new *D-Path segments* are derived by selecting the end points of the furtesh reaching *(D-1)-Path* stored in V. As long as no *D-path* reaches (M,N) this procedure is repeated.

Obviously there is no point in examining each and every k-line from $-N$ to M in every iteration because *D-paths* starting at (0,0) cannot reach *k-points* beyond $k \in [-D, D]$. Also a D-path segment starting at k can only end on $k + 1$ or $k - 1$, therefore we only have to examine half of the k-lines in each iteration.

In order to accomplish this an outer loop is equipped with the loop variable D starting at 0 and incrementing by 1 as long as no *D-path* reaches (M,N). Within an inner loop every second k-line from $-D$ to D is examined in order to find the new *furthest reaching D-path*. In each iteration of the outer loop *D-1-paths* are extended to *D-paths* by generating new *snake-heads* by the inner loop. We can depict the *D-contour* as a line connecting all the endpoints of the newly generated *snake-heads* (See Figure 6).

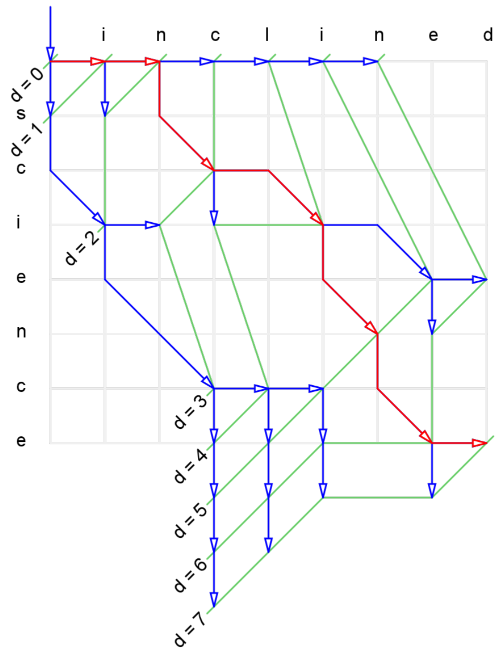


Figure 6: D-Paths, Snake-Heads and D-Contour: A *D-Path* starts at (0,0) and contains D orthogonal edges (edit operations). *Snake-heads* (depicted using arrow-heads) denote the bottom-right end in *D-paths*. Connecting all the *snake-heads* after each iteration of the outer loop results in the *d-contour* (green line). The red *D-path* represents the Longest Common Subsequence (diagonals) as well as the shortest edit script (orthogonal edges).

In order to restrict the runtime of the algorithm the outer loop can be aborted when $D = MAX$ where $MAX < N + M$. Because of this property this algorithm is sometime referred to as a d-band algorithm.

Listing 7a illustrates the complete algorithm using JavaScript Syntax. In this implementation the length of the shortest edit script d is calculated and returned. Also an object appropriately representing a snake is constructed and appended to an array. After $lcs\_forward$ terminates, the Longest Common Subsequence can be determined by backtracking from the last snake which obviously ends in (M,N) through to all its ancestors to (0, 0).

```
function lcs_forward(a, b, dmax, snakes) {
  var N = a.length, M = b.length, V = {}, d, k, x, snake;

  V[1] = 0;
  for (d = 0; d <= dmax; d++) {
    for (k = -d; k <= d; k+=2) {
      // Determine the preceeding snake head. Pick the one whose furthest
      // reaching x value is greatest.
      if (k === -d || (k !== d && V[k-1] < V[k+1])) {
        // Furthest reaching snake is above (k+1), move down.
        x = V[k+1];
      }
      else {
        // Furthest reaching snake is left (k-1), move right.
        x = V[k-1]+1;
      }

      // Follow the diagonal as long as symbol at a[x] equals b[y]
      snake = new Snake();
      snake.setTopLeft(x, x-k);
      while (x < N && x-k < M && a[x] === b[x-k]) {
        x++;
      }
      snake.setBottomRight(x, x-k);
      snakes.push(snake);

      // Memoize furthest reaching x value
      V[k] = x;

      // Termination check
      if (x >= N && x-k >= M) {
        return d;
      }
    }
  }

  // No LCS found for d <= dmax.
  return undefined;
}
```

(a) Forward LCS algorithm

```
function lcs_backward(a, b, dmax, snakes) {
  var N = a.length, M = b.length, delta = N - M, V = {}, d, k, x, snake;

  V[delta-1] = N;
  for (d = 0; d <= dmax; d++) {
    for (k = -d+delta; k <= d+delta; k+=2) {
      // Determine the preceeding snake head. Pick the one whose furthest
      // reaching x value is greatest.
      if (k === d+delta || (k !== -d+delta && V[k-1] < V[k+1])) {
        // Furthest reaching snake is underneath (k-1), move up.
        x = V[k-1];
      }
      else {
        // Furthest reaching snake is right (k+1), move left.
        x = V[x+1]-1;
      }

      // Follow the diagonal as long as symbol at a[x] equals b[y]
      snake = new Snake();
      snake.setBottomRight(x, x-k);
      while (x > 0 && x-k > 0 && a[x] === b[x-k]) {
        x--;
      }
      snake.setTopLeft(x, x-k);
      snakes.push(snake);

      // Memoize furthest reaching x value
      V[k] = x;

      // Termination check
      if (x <= 0 && x-k <= 0) {
        return d;
      }
    }
  }

  // No LCS found for d <= dmax.
  return undefined;
}
```

(b) Backward LCS algorithm

Figure 7: Basic greedy LCS algorithms

3.4. The Middle Snake Algorithm — A Linear Space Refinement

The problem with Myers basic LCS algorithm presented in the previous section is its non-linear space requirements. Given the time and space complexity of $O((N+M)D)$, on input where $N = M = D$ (both strings have equal length but do not contain any common characters), time and space complexity becomes $O(N^2)$.

The non-linear space requirements are due to the fact that each snake has to be kept in memory until the algorithm terminates. In order to reduce space complexity this drawback has to be eliminated. A naive approach would be to just return the last snake when the algorithm terminates and then start over again and return the one before, repeating this process until the starting point is reached. During that process only the snakes required to construct the LCS afterwards have to be kept in memory.

There is however a better approach. Listing 7b shows a variant of the basic LCS algorithm starting at (N,M) and working towards (0,0) effectively discovering snakes by starting in the bottom-right and working towards the top-left corner of the edit graph (see Figure 8).

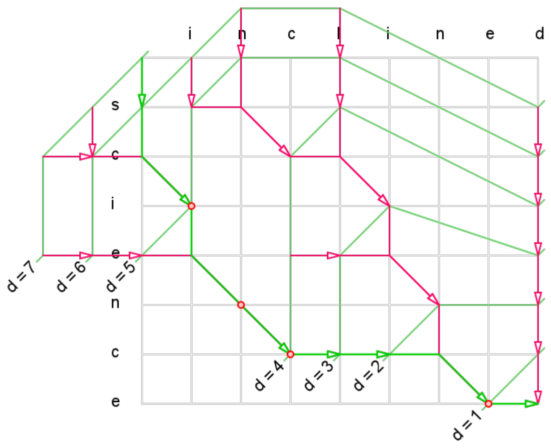


Figure 8: Edit graph produced by backward LCS algorithm version. Note that the choosen path differs from the one discovered by the forward algorithm pictured in Figure 6. However the result ($d = 7$) remains the same.

Considering the striking resemblance of the two versions of the basic algorithm (Figure 7), the idea of running them simultaneously until the furthest reaching *d-paths* from each direction overlap seems pretty obvious. Myers [18] presented an algorithm identifying the *middle snake* in such a way (see Listing 9a). Departing from there the rest of the LCS can be devised by recursively running the algorithm on the remaining parts left- and then right of the middle snake. Listing 9b illustrates the algorithm in JavaScript syntax while Figure 10 shows several steps graphically.

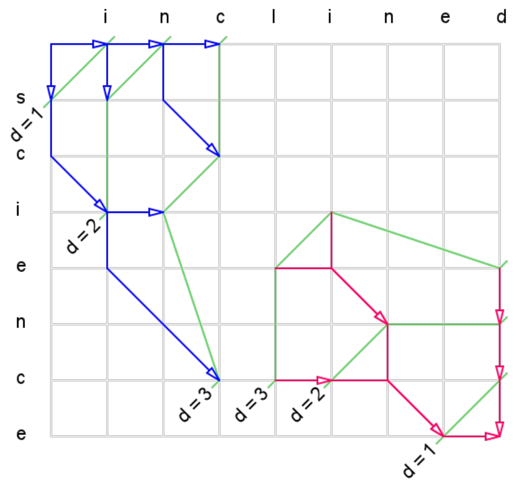
```javascript
function lcs_middlesnake(a, b) {
  var N = a.length, M = b.length, delta = N - M, d, k,
      dmax = Math.ceil((N + M) / 2),
      checkBwSnake = (delta % 2 === 0),
      Vf = {}, Vb = {}, snake;

  // Initialize buffer for furthest reaching x-values indexed by k.
  // Vf for the forward and Vb for the backward pass.
  Vf[1] = 0;
  Vb[delta-1] = N;
  for (d = 0; d <= dmax; d++) {
    // Forward search
    for (k = -d; k <= d; k+=2) {
      snake = next_snake_forward(a, b, k, -d, d, Vf);

      // Check for overlap when delta is odd
      if (!checkBwSnake && k >= -d-1+delta && k <= d-1+delta) {
        if (Vf[k] >= Vb[k]) {
          snake.d = 2 * d - 1;
          return snake;
        }
      }
    }

    // Backward search
    for (k = -d+delta; k <= d+delta; k+=2) {
      snake = next_snake_backward(a, b, k, -d+delta, d+delta, Vb);

      // Check for overlap when delta is even
      if (checkBwSnake && k >= -d && k <= d) {
        if (Vf[k] >= Vb[k]) {
          snake.d = 2 * d;
          return snake;
        }
      }
    }
  }
}
```

(a) Middle snake algorithm. The functions **`next_snake_forward`** and **`next_snake_backward`** are modified versions of **`lcs_forward`** and **`lcs_backward`** depicted in Figure 7, basically lacking the outer `for` loop and returning an object appropriately representing a snake.

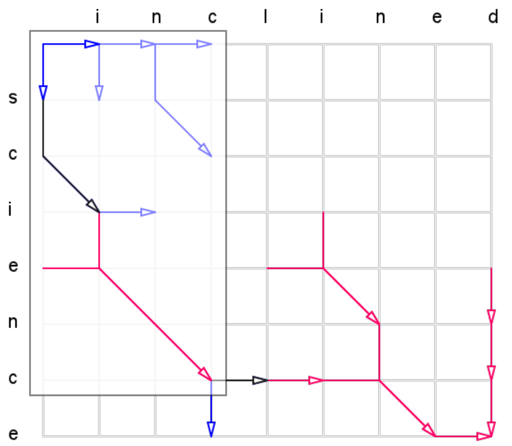
```javascript
function lcs_recursive(a, b, snakes) {
  var N = a.length, M = b.length, ms, d;

  // Return if there is nothing left
  if (N <= 0 || M <= 0) {
    return 0;
  }

  // Find the middle snake
  ms = lcs_middlesnake(a, b);

  if (ms.d <= 1) {
    // Middle snake consists of zero or more diagonal edges. There is also zero
    // or one orthogonal edge (before or after the snake). Add the snake to
    // the result set and return d.
    snakes.push(ms);
  }
  else {
    // Recurse if the middle-snake algorithm encountered more than one
    // operation (orthogonal edge).
    lcs_recursive(a.substr(0, ms.x1), b.substr(0, ms.y1), snakes);

    // Add the middle snake to the result set.
    snakes.push(ms);

    // Recurse on the right half of the edit graph
    lcs_recursive(a.substr(ms.x2), b.substr(ms.y2), snakes);
  }

  return ms.d;
}
```

(b) Recursive LCS algorithm

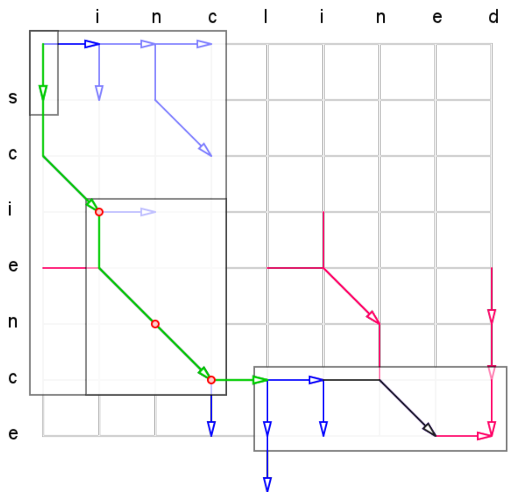
Figure 9: Linear space LCS algorithm sourcecode



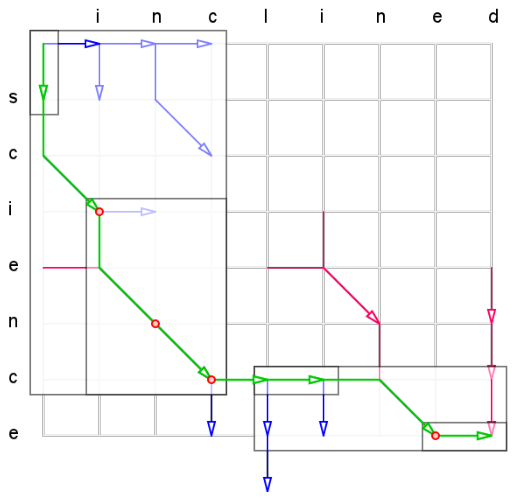
(a) Linear space LCS algorithm starts out from top-left (blue edges) and from bottom-right (magenta edges) simultaneously.



(b) End of first recursion: After identifying the first middle snake (single horizontal black edge), the algorithm starts over within the left part (black frame) and discovers the second middle snake (black edge inside frame).



(c) Left part complete: Algorithm discovered three common symbols (red circles) and continued with right half of the graph.



(d) LCS complete: Note that only the green path segments were kept in memory, red and blue can be discarded as soon as a middle snake is found.

Figure 10: Linear space LCS algorithm illustrated

3.5. LCS Time and Space Complexity

Myers gives $O(ND)$ as the maximal time complexity for his algorithm, where N denotes the number of all symbols from both sequences and D the length of the shortest edit script, i.e. the number of necessary changes. He also showed that the expected time performance is $O(N + D^2)$. Using the middle snake algorithm as outlined above, space complexity is only $O(N)$.

4. XCC Diff in Detail

Virtually any recent diff-algorithm for hierarchical structured documents since Chawathes FMES (discussed in Section 2.5.2) divides the problem of identifying change operations into two subproblems. Generate a *matching* of the two trees and after that construct the *edit script* or *patch*.

4.1. Tree Operations

XCC supports an update operation on a single node, insert and delete operations on tree sequences and a move operation for subtrees. Refer to Definition 6 for an indepth description.

4.2. Tree Matching

During the matching phase, for each node n in T_1 the algorithm tries to find one corresponding node n' in T_2 respecting certain matching rules. A matching between two nodes is a one-to-one relationship. If the nodes n and n' are *partners* in a matching, we also say that they form a *pair*.

4.2.1. XCC Matching Rules

Rönnau and Borghoff distinguish between *structure preserving* and *structure affecting* changes to documents in XCC [21]. In terms of edit script operations, the former expression translates to update operations while the latter subsumes insert- and delete operations. Consequently we can say that structure preserving changes target single nodes taking part in the matching while structure affecting changes target subtrees outside the matching.

The fundamental matching rule in XCC reflects this differentiation and also defines precisely a *structure preserving* change: Let p be the parent of n and p' the parent of n', n and n' may not form a pair if p and p' are no pair either. Note that this rule can be applied recursively in order to decide whether two candidate nodes may form a pair or not. As a consequence no two nodes may form a pair if their tree depth is different. Also update operations may not be applied to any nodes moved from one subtree to another.

Regardless of any other rules, the root nodes of the trees T_1 and T_2 always form a pair.

4.2.2. LCS Among Leaf Nodes

The XCC tree matching algorithm starts out by calculating the Longest Common Subsequence (LCS) between the leaf nodes of T_1 and T_2. In addition to the fundamental matching rule presented above, leaf nodes may only form a pair if their (Hash-)value is equal.

Each pair of nodes (n, n') taking part in the LCS and also complying with the fundamental rule is added to the matching. Additionally all ancestor pairs are added to the matching also, regardless of whether their values are equal or not. The complete algorithm is depicted in Listing 11a.

4.2.3. Extended Update Detection

Because there is a great chance that the values of leaf nodes change when a document is edited, the XCC tree matching algorithm is also capable of detecting modified leaf nodes in a second pass. The matching rules are a little bit different than during the first pass. In addition to the fundamental matching rule defined before, nodes n and n' only may form a pair if their nearest preceeding siblings which take part in the matching also form a pair. Obviously the values of leaves do not have to be equal, however unlike during the first pass, the values of the nodes in their ancestor chain must be the same.

In order to identify potential pairs, the trees are traversed recursively in preorder departing from the root node. When a pair of nodes is found which is not yet recorded in the matching, it is processed according to the rules stated above: if both candidates are leaf nodes, they are added to the matching otherwise they only get added if their values are the same.

Our version of the extended update algorithm departs significantly from the original implementation. This is because the current version of the original implementation does not quite match the stated linear bounds in time complexity under worst case conditions. Figure 12 depicts the worst case input for the original implementation. Listing 11b shows our version of the extended update algorithm. However in some situations our linear time version does not produce the same results like the original implementation.

We considered dropping extended update detection entirely. However the results of the leaf-LCS pass are sometimes not optimal. If two document versions only have few leaf nodes in common but the overall document structure remained the same, some internal nodes will be included into the delta even if they were not changed at all. This can lead to patches which are difficult to merge. An extreme case is depicted in Figure 13.

```
function xcc_lcs(matching, a_leaves, b_leaves) {
    var lcs = new LCS(a_leaves, b_leaves);

    // Override equals method of LCS instance for tree nodes.
    lcs.equals = function(a, b) {
        return a.depth === b.depth && a.value === b.value;
    }

    // Identify structure-preserving changes. Run lcs over leave nodes of
    // tree a and tree b. Associate the identified leaf nodes and also
    // their ancestors except if this would result in structure-affecting
    // change.
    lcs.forEachCommonSymbol(function(x, y) {
        var a = a_leaves[x], b = b_leaves[y], a_nodes = [], b_nodes = [], i;

        // Bubble up hierarchy until we encounter the first ancestor which
        // already has been matched. Record potential pairs in the a_nodes and
        // b_nodes arrays.
        while(a && b && !matching.get(a) && !matching.get(b)) {
            a_nodes.push(a);
            b_nodes.push(b);
            a = a.par;
            b = b.par;
        }

        // Record nodes a and b and all of their ancestors in the matching if
        // and only if the nearest matched ancestors are partners.
        if (a && b && a === matching.get(b)) {
            for (i=0; i<a_nodes.length; i++) {
                matching.put(a_nodes[i], b_nodes[i]);
            }
        }
    });
};
```

(a) First pass: Match leaf nodes which are part of the LCS

```
function xcc_xud(matching, a_node) {
  var a_nodes = a_node.children,
      b_nodes = matching.get(a_node).children,
      pm = true,  // True if the previous node pair matched
      i = 0,      // Array index into a_nodes
      k = 0,      // Array index into b_nodes
      a,          // Current candidate node in a_nodes
      b;          // Current candidate node in b_nodes

  // Loop through a_nodes and b_nodes simultaneously
  while (a_nodes[i] && b_nodes[k]) {
    a = a_nodes[i];
    b = b_nodes[k];

    if (pm && !matching.get(a) && !matching.get(b) &&
        a.children.length === 0 && b.children.length === 0) {
      // If the previous sibling takes part in the matching and both
      // candidates are leaf-nodes, they should form a pair (leaf-update)
      matching.put(a, b);
      i++;
      k++;
    }
    else if (pm && !matching.get(a) && !matching.get(b) &&
        a.value === b.value) {
      // If the previous sibling takes part in the matching and both
      // candidates have the same value, they should form a pair
      matching.put(a, b);
      // Recurse
      xcc_xud(matching, a);
      i++;
      k++;
    }
    else if (!matching.get(a)) {
      // Skip a if above rules did not apply and a is not in the matching
      pm = false;
      i++;
    }
    else if (!matching.get(b)) {
      // Skip b if above rules did not apply and b is not in the matching
      pm = false;
      k++;
    }
    else if (a === matching.get(b)) {
      // Recurse, both candidates are in the matching
      xcc_xud(matching, a);
      pm = true;
      i++;
      k++;
    }
    else {
      // Both candidates are in the matching but they are no partners.
      // This is impossible, bail out.
      throw new Error('Matching is not consistent');
    }
  }
}
```

(b) Second pass: Detect updates on leaf nodes (linear time variant)

Figure 11: XCC tree matching algorithm

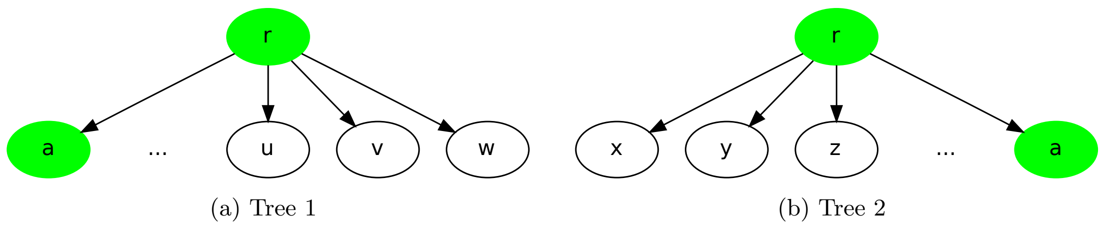


Figure 12: Worst case scenario for original XCC leaf update implementation: The common node *a* takes part in the leaf-LCS while all other nodes do not match and therefore are examined during the leaf update detection pass.

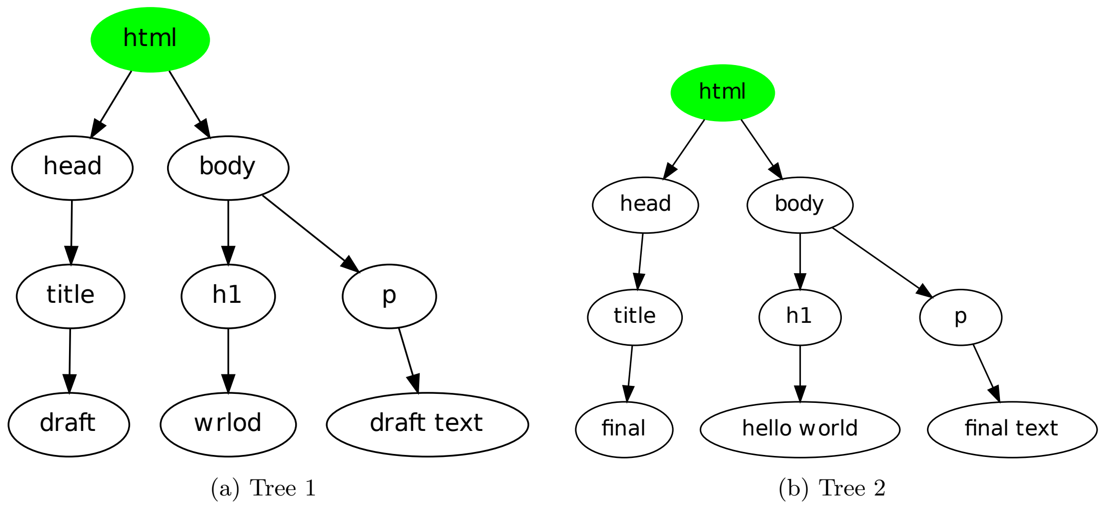


Figure 13: Worst case scenario for XCC when extended update detection is skipped. Because the LCS pass did not detect any common leaf nodes, internal nodes are not matched neither. Even if the document structure is exactly the same. The only node in the matching (green) is the root node.

4.3. Skel-match Algorithm: A Modified Version of XCC

Not satisfied with the way extended update detection works in the current XCC implementation we set out to find a solution which is both, straightforward to implement and also easier to analyze.

We stick to the basic idea of XCC, namely that the actual content is contained near the bottom of the document tree. Consequently we also have to expect that when the content of structured documents is modified, most of the changes will take place on leaf nodes.

Like in XCC we start out with an LCS over document content[13]. Also we apply the exact same matching rules. The result of this pass is a partial matching which contains unchanged content as well as all nodes in the ancestor chain of matched content nodes.

In the second pass we identify unchanched structure. We accomplish this by collecting sequences of unmatched siblings enclosed by matched pairs from both trees. Note that each such sequence from T_1 therefore has a corresponding sequence in T_2. For each sequence pair we collect the bottom-most structure nodes, i.e. all structure nodes which do not have any other structure node as a descendant. We refer to this sequence of bottom-most structure nodes as the structure contour. At this stage we have derived a sequence of pairs of structure contours.

We now compute the LCS for each pair of structure contour sequences. Unlike in the first pass not only the values of the nodes in question have to match but also the values of all their unmatched ancestors pairs. The rule can be implemented easily using recursion. Figure 14 depicts the workings of the common structure detection step graphically. The JavaScript implementation of the second pass is given in Figure 15. Note that the first pass of Skel-match is identicaly to the first pass of XCC diff.

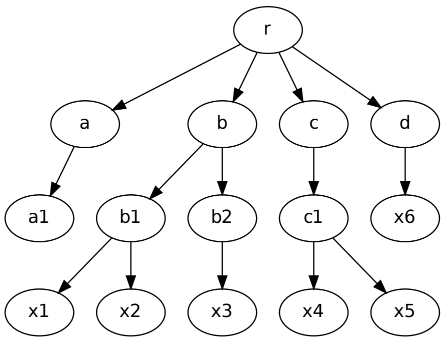
4.4. Skel-match Complexity

As mentioned before (see Section 3.5) the LCS time complexity is given with $O(ND)$. Because our algorithm starts out with the computation of the LCS among the leaf nodes between two trees, the time complexity of this first pass is also $O(ND)$ where N denotes the sum of the number of leaf nodes of both trees and D the number of edit operations.

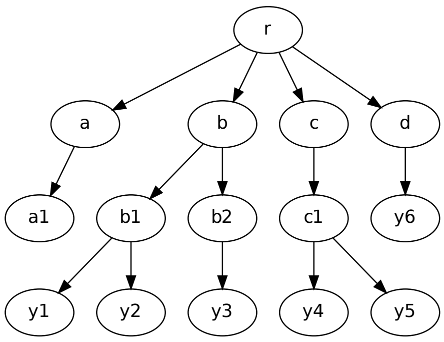
During a subsequent pass, skel-match applies the LCS repeatetly to sequences of parent nodes (together with their unmatched ancestors) of the leaf nodes which were not matched in the first pass. The number of necessary LCS passes depends on the number of changes detected during the leaf-LCS. Therefore at most D additional LCS passes are necessary. Also the maximum length of each parent-node sequence is at most D.

It follows that either few LCS runs with long input sequences or many LCS runs with short input sequences are necessary in the second pass. In order to analyze the different

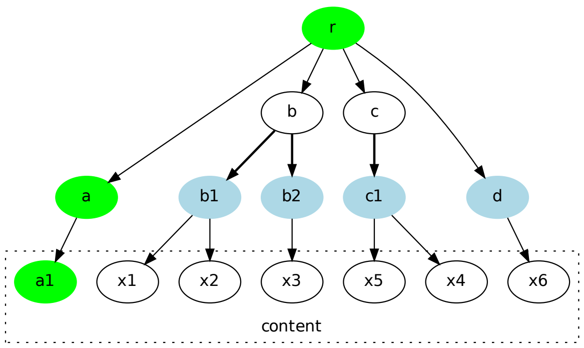
[13]In our implementation it is possible to specify which nodes qualify as document content. In order to allow aggregation of adjacent operations when diffing XML documents, elements who only have one single text node as a child are treated as content while elements with more children are treated as structure.



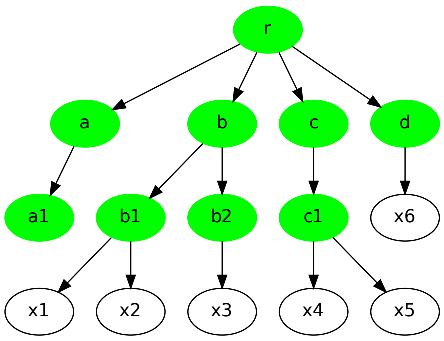
(a) Tree 1 (original)



(b) Tree 2 (modifications in leaf nodes)



(c) Green shaded nodes have been matched by LCS pass over content (aka leaf nodes). Blue shaded nodes *b1, b2, c1* and *d* represent two structure contour sequences. The former derived from the unmatched sequence of siblings *b* and *c*, the latter only consisting of *d*. During the LCS pass over structure contour, also the values of unmatched ancestor nodes will be examined. E.g. when testing if *b1* in tree 1 is the same as *b1* in tree 2, also the values of their parents are examined (*b*).



(d) Matching found using the skel-match common structure detection.

Figure 14: Skel-match: detection of common structure

```
/**
 * Traverse a partial matching and detect equal structure-type nodes between
 * matched content nodes. Specify the starting points using tree nodes a and b.
 */
function skelmatch_match_structure(matching, a, b) {
    var sequence_pair, a_nodes, b_nodes, a_bones, b_bones, lcs, i;
    var sibling_sequences = unmatchedSiblings(matching,
            a.children, b.children);

    // Run lcs over the structure contour of each pair of unmatched siblings.
    for (i = 0; i < sibling_sequences.length; i++) {
        a_bones = [];
        b_bones = [];
        lcs = new LCS(a_bones, b_bones);

        // Override equality test.
        lcs.equals = function(a, b) {
            return equalStructure(matching, a, b);
        };

        // Populate bone array (structure contour)
        sibling_sequence[i].a.forEach(function(n) {
            Array.prototype.push.apply(a_bones, collectBones(n));
        });
        sibling_sequence[i].b.forEach(function(n) {
            Array.prototype.push.apply(b_bones, collectBones(n));
        });

        // Identify structure-preserving changes. Run lcs over lower bone ends
        // in tree a and tree b. Associate the identified nodes and also their
        // ancestors except if this would result in structure-affecting change.
        lcs.forEachCommonSymbol(function(x, y) {
            var a = a_bones[x], b = b_bones[y];

            // Verify that ancestor chain allows that a and b form a pair.
            if (matchingCheckAncestors(matching, a, b)) {
                // Record nodes a and b and all of their ancestors in the
                // matching if and only if the nearest matched ancestors are
                // partners.
                matchingPutAncestors(matching, a, b);
            }
        });
    }
}
```

(a) Skel-match structure matching algorithm. The function `equalStructure` returns true if the two nodes and all their unmatched ancestors have equal values. The `collectBones` function returns an array of the nodes taking part in the structure contour of a sequence of siblings.

```
/**
 * Return an array of pairs of sequences of siblings which do not take part in
 * the matching. Start with the list of siblings a_sibs (from tree a) and
 * b_sibs (from tree b).
 */
function skelmatch_unmatched_siblings(matching, a_sibs, b_sibs) {
    var a_xmatch = [],  // Array of consecutive sequence of unmatched nodes
                        // from a_sibs.
        b_xmatch = [],  // Array of consecutive sequence of unmatched nodes
                        // from b_sibs.
        i = 0,      // Array index into a_sibs
        k = 0,      // Array index into b_sibs
        a,          // Current candidate node in a_sibs
        b,          // Current candidate node in b_sibs
        result = [];

    // Loop through a_sibs and b_sibs simultaneously
    while (a_sibs[i] || b_sibs[k]) {
        a = a_sibs[i];
        b = b_sibs[k];

        if (a && !matching.get(a)) {
            // Skip a if above rules did not apply and a is not in the matching
            a_xmatch.push(a);
            i++;
        }
        else if (b && !matching.get(b)) {
            // Skip b if above rules did not apply and b is not in the matching
            b_xmatch.push(b);
            k++;
        }
        else if (a && b && a === matching.get(b)) {
            result.push({'a': a_xmatch, 'b': b_xmatch});
            a_xmatch = [];
            b_xmatch = [];

            // Recurse, both candidates are in the matching
            Array.prototype.push.apply(result, skelmatch_unmatched_siblings(
                        matching, a.children, b.children));
            i++;
            k++;
        }
        else {
            throw new Error('Matching is not consistent');
        }
    }

    if (a_xmatch.length > 0 || b_xmatch.length > 0) {
        result.push({'a': a_xmatch, 'b': b_xmatch});
        callback.call(T, a_xmatch, b_xmatch, a, b);
    }

    return result;
}
```

(b) Skel-match collection of unmatched siblings

Figure 15: Skel-match algorithm

cases, we introduce $C \in [0, D]$ specifying the number of LCS runs necessary in the second pass. The maximal possible length of an input sequence is then given by $M = D - C$. The symbol M_i denotes the length of the input of the nth LCS run and E_i the length of the corresponding edit script. The complexity of the second step is therefore:

$$O(\sum_{i=1}^{C} M_i E_i)$$

Because of the product $M_i E_i$ and $M_i \geq E_i$, we optain a maximum from the sum when $C = 1$, $M_1 = D - 1$ and $E_1 = M_1$. Therefore the maximal time complexity of the second step is given by:

$$O(DE)$$

Where D corresponds to the number of changes detected by the leaf-LCS from the first pass and E to the sum of changes found in all LCS runs from the second pass. The time complexity of the whole algorithm is therefore given by:

$$O(ND + DE)$$

Our algorithm does not require any additional data structures. Therefore space requirements are the same as for the underlying LCS algorithm. We use the linear space LCS variant and therefore our algorithm runs with $O(N)$ space complexity.

4.5. Skel-match Runtime

We empirically verified the lower and upper bounds of the stated runtime complexity using input trees constructed by a simple tree model with a constant depth and a constant child count. Each internal node has a specified number of child nodes and the leaf nodes all have the same given depth.

For the minimal runtime analysis shown in Figure 16 we used two trees with identical structure and node values such that no changes are detected by the matching algorithm. We used trees of depth 5 with child counts from 1 through 15 ending up with tree sizes up to 800'000. The worst case test graphed in Figure 17 was conducted with a tree depth of 3 and therefore tested a range of tree sizes up to 3500.

4.6. Delta Construction

After optaining a partial matching, a set of operations necessary to turn T_1 into T_2 is collected during a recursive traversal of the tree, departing from the root node.

Recall that the root node always takes part in the matching as well as all the ancestors of a pair of matched leaf nodes. Therefore node-update operations can be detected by comparing the value of every node taking part in the matching with the value of its partner.

The nodes which do not take part in the matching only appear in one of the two trees. Nodes from T_1 which do not have a partner in T_2 should be deleted while nodes from

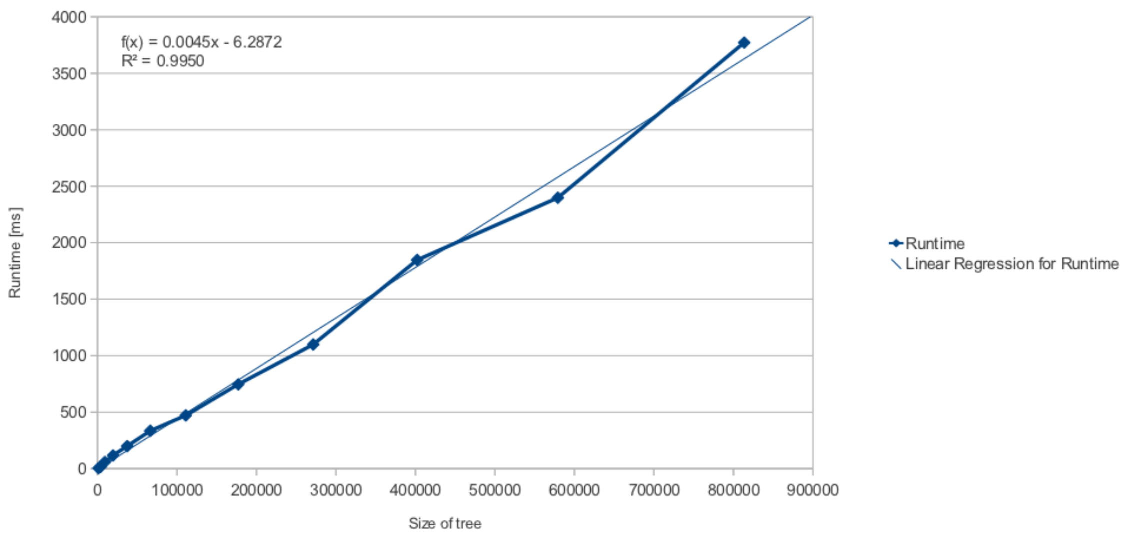


Figure 16: This chart depicts the runtime of the *Skel-match* algorithm for completely identical trees. The number of edits D as well as E is much smaller than the combined size of the trees N. The correlation factor of $R^2 = 0.9950$ found using linear regression indicates a pretty close match to the stated runtime complexity $O(ND + DE)$, i.e. $O(N)$ because D and E can be neglected.

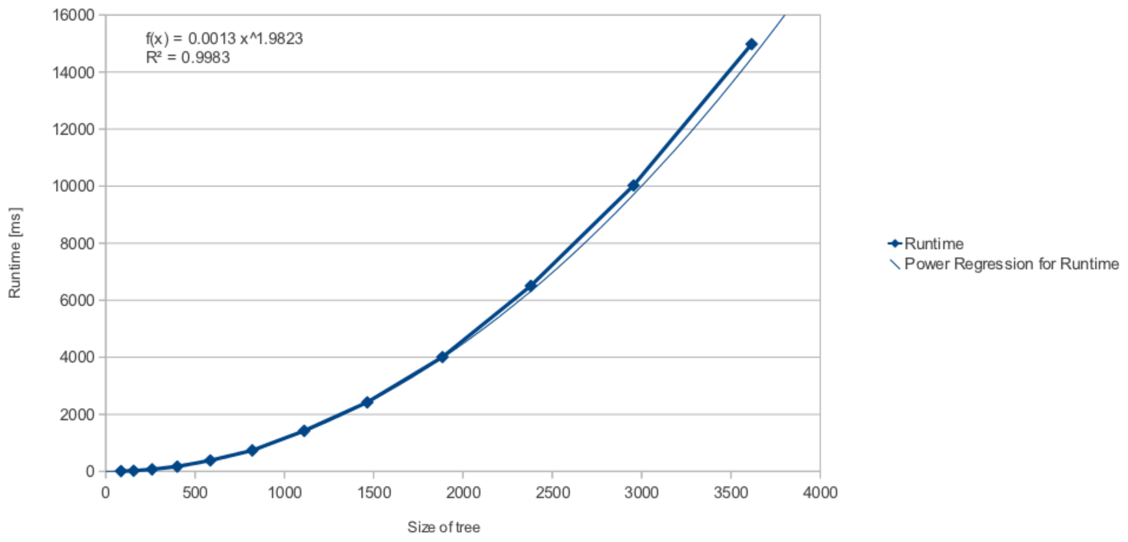


Figure 17: The runtime of the *Skel-match* algorithm with worst-case input is shown in this graph. Input trees are of the same structure but with completely different node values. The number of edit operations D and also E is therefore equal to the number of nodes of both trees N. The measured runtimes fit perfectly with the quadratic polynome calculated using power-regression as indicated by the correlation factor of $R^2 = 0.9983$. For worst case input, the runtime complexity becomes $O(N^2)$

.

T_2 not appearing in T_1 should be inserted. It follows from the XCC matching rule[14] that if a node does not take part in the matching, not one of its descendants may take part in the matching neither. Therefore we can address insert and delete operations by specifying the root nodes of the subtrees.

Our implementation is capable of collecting adjacent sequences of insert and delete operations into a single tree-update operation. This is very similar to the concept of "hunks" used in the unified diff format of GNU diff [2].

5. XCC Patch in Detail

5.1. Patch File Format

A patch file contains a set of operations necessary to turn one structured file into another. Each operation is addressed using a *path* departing from the root node. Each path component is a zero-based index into the children list of the parent node. The root node is addressed by an empty path. A path always points to the position in the original tree (T_1) where the first node is located which is affected by the operation. We refer to this node as the *anchor*. In the case of pure insert operations there are no nodes directly affected in T_1. For such operations, the path points to the first node after the insertion-point. Note that when a subtree should be inserted as the last child of an existing node, the path will point into the blank space after the last child in T_1 (see Figure 18).

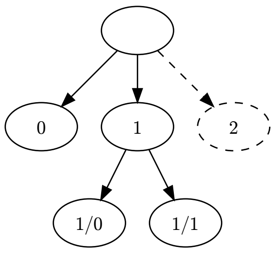


Figure 18: Operations are adressed through XPath-style expressions. In our implementation the root node is specified using an empty path. Insert operations may refer to positions, which do not exist yet (indicated by a dashed node at position 2)

.

In order to support merging changes into documents which have been modified since the delta was recorded, the value of a number of *context nodes* is also stored as part of the operation. The context consists of four values of the nodes immediately preceeding the anchor node and four values of the nodes following the last node affected by the operation in T_1. Obviously a linearization of the tree is required to meaningfully refer to leading and trailing context nodes. We use document order for that matter. Figure 19 shows an example of our patch file format.

[14]Fundamental XCC matching rule: No two nodes may form a pair if their parents do not form a pair either

```
1  <?xml version="1.0"?>
2  <delta>
3    <forest path="0/0">
4      <context>;;df4a3fb;1acdce3a</context>
5      <remove>
6        <title>Information is knowledge</title>
7      </remove>
8      <insert>
9        <title>Frank Zappa: Information is not knowledge</title>
10     </insert>
11     <context>4251892f;662e3aec;1290e7c6;662e3aec</context>
12   </forest>
13   <node path="1">
14     <context>df4a3fb;1acdce3a;6735e4ae;14b60978</context>
15     <remove>
16       <body class="draft"/>
17     </remove>
18     <insert>
19       <body/>
20     </insert>
21     <context>662e3aec;1290e7c6;662e3aec;35380707</context>
22   </node>
23 </delta>
```

Figure 19: Example of *delta.js* patch file format. Node update operations (Line 13) affect a single node, the body-tag in this example. Forest operations target a sequence of subtrees (Line 3). Leading and trailing context is expressed using the hash values of nodes preceeding and following the affected node or tree sequence.

Note that the choosen file format differs in a number of ways from the one proposed by the XCC authors in [22] and [20]. First, we do not include the hash value of any target node into the fingerprint. Considering the example from Figure 19, in the original XCC delta format, the hash values of the body-tag would also be included into the context fingerprint. As a result our delta is even easier to invert. It is sufficient to exchange the contents of the remove- and insert-tag of a given change. Second our delta format supports the aggregation of consecutive insert- and delete-operation into one "forest" operation.

5.2. Weighted Context Fingerprints

The context fingerprints outlined in the previous section are used to ensure that a given change is applied to the correct location during the patch process. Our implementation mostly follows the XCC approach discussed in [22] and [20].

When resolving an operation, a first guess is attempted by following the operations *path*-property in the target tree. The last position which was resolved successfully is used as the departure point for the following phase.

A pattern consisting of a head, a body and a tail sequence is derived from the operation. The head and the tail context is taken directly from the appropriate properties of the operation object. In the case of node-updates, where a single node is the target of the operation, the body corresponds to the node-value of the target node, i.e. the node contained in the remove-part of the operation. When resolving forest operations, the body is computed by flattening all subtrees contained in the remove-section of the operation into one sequence in document order.

Given a fingerprint radius r and an integer value denoting the distance i of a context value to the body, the weighting function $w(i)$ with $i \in [-r, r] \backslash 0$ assigns a weight to each position of the head and tail context. Individual weights are normalized using the factor wn such that the sum of all weights from head and tail is 1 for any r. Figure 20 depicts a pattern and the assigned weights for the head and tail context, Definition 7 gives the normalized weighting function for the context of a fingerprint with radius r.

Definition 7. XCC Weighting Function

$$w(i) = \frac{\left(\frac{1}{2}\right)^{|i|}}{wn} \text{ with } wn = 2\sum_{i=1}^{r}\left(\frac{1}{2}\right)^{i}$$

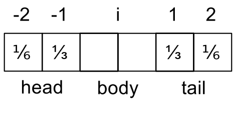


Figure 20: The weights for the head and tail context in a pattern with the fingerprint radius $r = 2$. Note that the body is always required to match, therefore no weights are assigned to body elements.

This pattern is now matched against all positions near the initial guess in a given environment size (default $s = 6$) in document order: $p \in [-s, s]$. For each position a match quality is calculated using the function $q(p)$ given in Figure 21. The position with the maximum match quality is then used as the anchor of the operation. If the match quality does not exceed a specified threshold value, the change is rejected. The authors of XCC found that a threshold value of 0.7 yields good results. A simple matching process with a fingerprint with radius $r = 2$ is outlined in Figure 22.

$$q(p) = \begin{cases} q_h(p) + q_t(p) & \text{if all nodes from body match} \\ 0 & \text{otherwise} \end{cases}$$

$$q_h(p) = \sum_{i=-1}^{-r} w(i,r) \times m(p+i)$$

$$q_t(p) = \sum_{i=1}^{r} w(i,r) \times m(p+i+length(body))$$

$$m(i) = \begin{cases} 1 & \text{if value from pattern matches value of subject at position } i \\ 0 & \text{otherwise} \end{cases}$$

Figure 21: XCC match quality function (q)

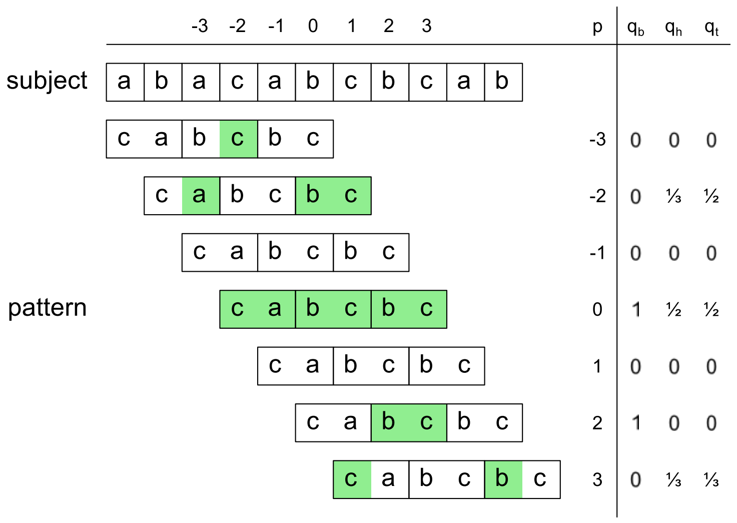


Figure 22: The process of matching a fingerprint with radius $r = 2$ against an environment with a size $s = 3$. In this example the best match is found at $p = 0$. Green shaded squares indicate a match for the represented pattern element.

The resolving algorithm presented here however does not produce optimal results when operating on flat trees where one parent has many child-nodes. This is because in such situations, the initial guess taken by following the path of the operation through the target trees hierarchy can result in a node which is too far away of the intended position. The problem arises if a path points to a node which has many preceeding siblings. When the target tree was modified since the operation was recorded and a node was inserted as a preceeding sibling to the parent of the affected node, the operations path will point to the newly inserted node. This position might then lie outside of the environment size s due to the many siblings preceeding the actual target node.

6. Architecture and Implementation

6.1. Design Goals

We stated in the introduction that we want to provide a framework of algorithms and tools which could serve as a starting point for a version control system tailored to structured documents. Also the architecture should be extendible such that additional file formats like e.g. JSON and new algorithms can be integrated easily. The core algorithms should also be reusable outside the project. Finally the code should run on different browsers and even within other environments like servers or scriptable applications.

Our framework is divided into two layers. The high level interface provides conveniant diff and patch commands and a set of factory classes. The diff and patch algorithms as well as methods for loading and saving data are implemented in a set of loosely coupled classes forming a solid foundation.

6.2. Achieving Reusable Code Units in JavaScript

The inheritance model provided by JavaScript out of the box does not encourage deep class hierarchies. Reusability of code and separation of concerns is easier to achive using *Object Composition* in this language. This approach simplifies the integration of new algorithms and file formats into the framework. It also makes it easy to strip away unneded functionality and minify the code base which is especially important for web applications.

In order to relieve clients of the framework from having to instantiate and associate many collaborating objects, we employ a set of factory classes modelled after the *Abstract Factory* pattern as described in the GoF [12]. The factories are discussed in greater detail in the following section.

Further we will encounter the `Diff` and `Patch` classes. While those classes are not strictly modelled after the *Command* pattern, they nevertheless incorporate the key idea. Upon instantiation the objects are parameterized using the factory classes we discussed above. Execution of the command is triggered by a single public method. However in order to conform to the original pattern, the `diff` and `patch` methods would need to have the same signature, which is not the case in the current implementation. Also not only the factory objects but all of the parameters including input and output documents would have to be properties of the commands and not parameters of the public execute-method.

In order to support different versions of tree matching (diff) and resolver (patch) algorithms, we are using the *Strategy* pattern. The factory classes are responsible for selection of the appropriate strategy, i.e. instantiation of the proper algorithm class.

Another GoF pattern we rely on is the *Adapter* pattern. In order to decouple the diffing and patching algorithms from the document structure (e.g. the DOM document), we introduced a very simple tree-implementation which only provides the functionality we need. We use a set of adapter classes which reproduce our own tree structures from the original document trees, such that the algorithms can interface with the trees without knowledge of the original source.

While not really a design pattern we want to present another important concept commonly used in JavaScript applications. Due to the fact that the language has first-class functions and because objects internally are nothing else than hash tables, it is possible to create and assign new functions to objects at runtime. This technique is commonly referred to as *Object Augmentation*.

We use this technique for example in the LCS implementation. The equality-method is a member of the LCS class. Client code may replace that method on individual instances to provide an equality-method which is adapted to the items the input sequences contain. This allows us to select different equality-methods depending on the input document format. E.g. we may want to compare XML nodes based on their hash values while for other types of trees we stick with the node value. Of course selection of the equality-method is the responsibility of the corresponding factory class.

6.3. High Level Interface

We provide four kinds of factory classes to support the instantiation of the objects required by the `diff` and `patch` commands. Two of them simplify the access to the lower-level diff and patch algorithms (`DiffFactory` and `ResolveFactory`), the other two support the handling of different file formats for document files and patch files (`DocumentFactory` and `DeltaFactory` respectively). Figure 23 depicts the currently implemented factory interfaces and classes whereas Figure 24 show the relations of the command classes to the factory classes.

Data is kept within the two data object classes `Document` and `DeltaDocument` during processing (see Figure 25). Client code should treat them as opaque types. The corresponding factory classes are responsible for their construction as well as for serialization. The string property `type` specifies the document format of the underlying document (xml or json). A reference to the native document structure is kept in the `data` property. When dealing with XML documents, `data` points to an instance of `DOMDocument`, i.e. the in-memory representation of the file. The string representation of the document is kept in the `src` property.

Our algorithms do not operate directly on the underlying document structure. Moreover we maintain our own tree structure which is adapted from the underlying document when a file is loaded. The root of our tree is assigned to the `tree` property of a `Document`. File format specific navigation and retrieval of node values is implemented in a flexible manner by providing the slots `valueindex`, `treevalueindex` and `nodeindex` on the

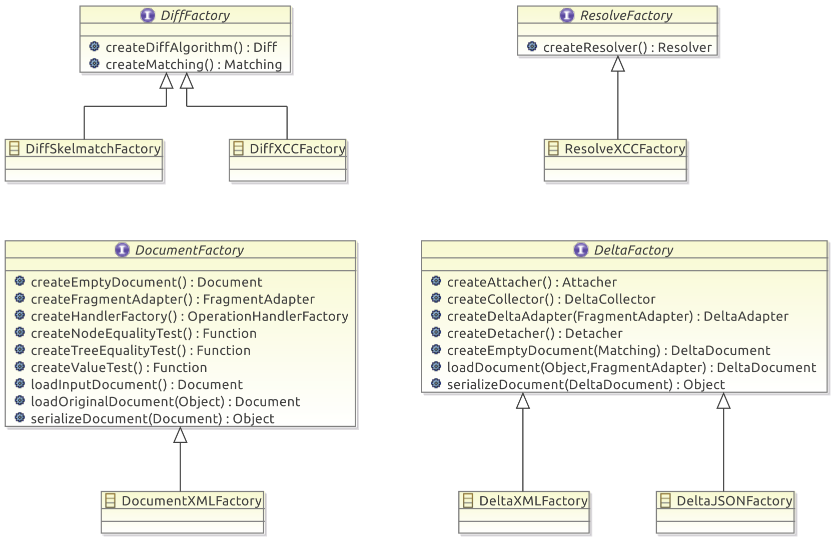

DiffFactory
createDiffAlgorithm() : Diff
createMatching() : Matching
DiffSkelmatchFactory
DiffXCCFactory
ResolveFactory
createResolver() : Resolver
ResolveXCCFactory
DocumentFactory
createEmptyDocument() : Document
createFragmentAdapter() : FragmentAdapter
createHandlerFactory() : OperationHandlerFactory
createNodeEqualityTest() : Function
createTreeEqualityTest() : Function
createValueTest() : Function
loadInputDocument() : Document
loadOriginalDocument(Object) : Document
serializeDocument(Document) : Object
DocumentXMLFactory
DeltaFactory
createAttacher() : Attacher
createCollector() : DeltaCollector
createDeltaAdapter(FragmentAdapter) : DeltaAdapter
createDetacher() : Detacher
createEmptyDocument(Matching) : DeltaDocument
loadDocument(Object,FragmentAdapter) : DeltaDocument
serializeDocument(DeltaDocument) : Object
DeltaXMLFactory
DeltaJSONFactory

Figure 23: The four factory interfaces along with the implementations provided by the current version of the framework. Note that JavaScript does not support any kind of interface-types. Therefore the interfaces `DiffFactory`, `ResolveFactory`, `DocumentFactory` and `DeltaFactory` do not really exist in our implementation. Nevertheless we show them here in order to point out that the classes inheriting from those interfaces are interchangeable.

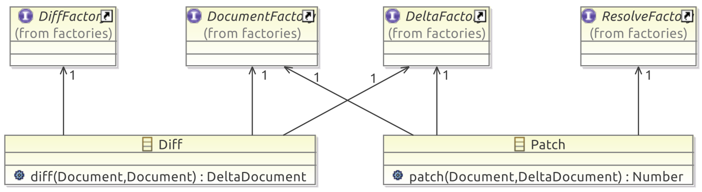


Figure 24: The `Diff` and `Patch` command classes both consume three factory objects upon instantiation. The only factory class that they do not share is `DiffFactory` and `ResolveFactory`, wrapping around the diff and patch algorithms.

`Document` object. For example a document constructed by the `DocumentXMLFactory` installs objects which compute and return hash values for nodes and subtrees into those slots.

The additional properties of the `DeltaDocument` class are a `matching`, which is essentially the result of the diffing process, and two different lists of operations. `Detached` operations represent the contents read from a patch file including head and tail context. The operations contained in the `attached` list represent the changes which were successfully resolved in the target document tree. The `attached` list is also populated after collecting all the changes from the `matching` as a result of a diff command.

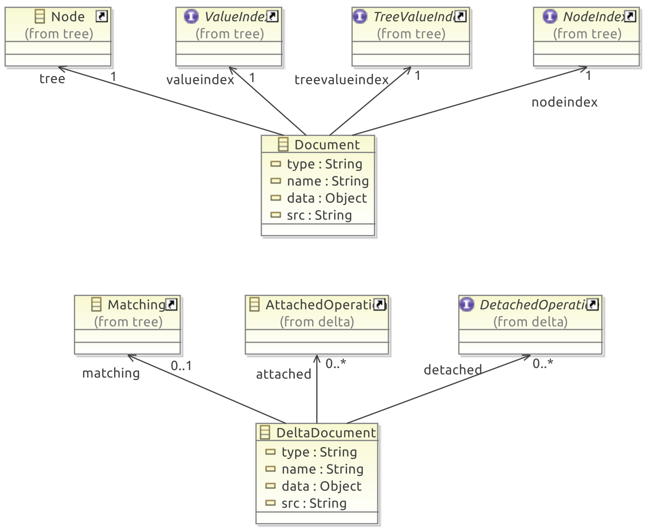


Figure 25: The `Diff` and `Patch` command as well as the four factory classes mainly operate on the `Document` and `DeltaDocument` objects.

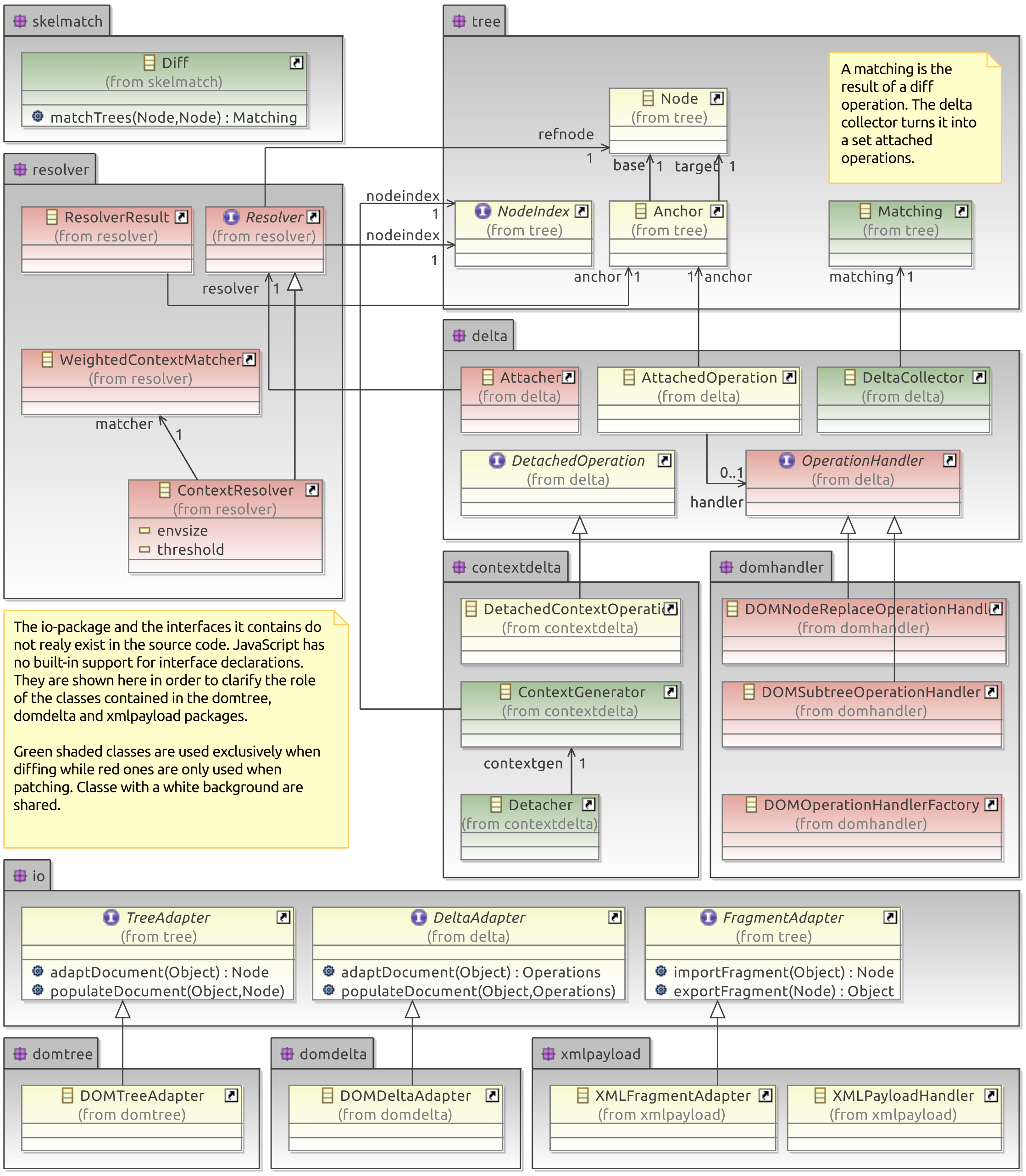
6.4. Foundation

Figure 26 gives an overview of the packages and classes forming the foundation of the framework. In this section we briefly cover the most important ones and point out their interactions.

Remember that JavaScript does not provide any means to declare interfaces for classes. Nevertheless we show some interfaces in the class diagrams in order to emphasize that those classes inheriting from them are interchangeable.

The complete API reference is included on the CD and in the project SVN repository as set of HTML files. They are located under the directory `apiref`. The file `index.html` is a good starting point. The reference documentation was generated from the source code of the framework using Sphinx, the Python Documentation Generator[15] and JSDoc

[15]Sphinx, the Python Documentation Generator
http://sphinx.pocoo.org/



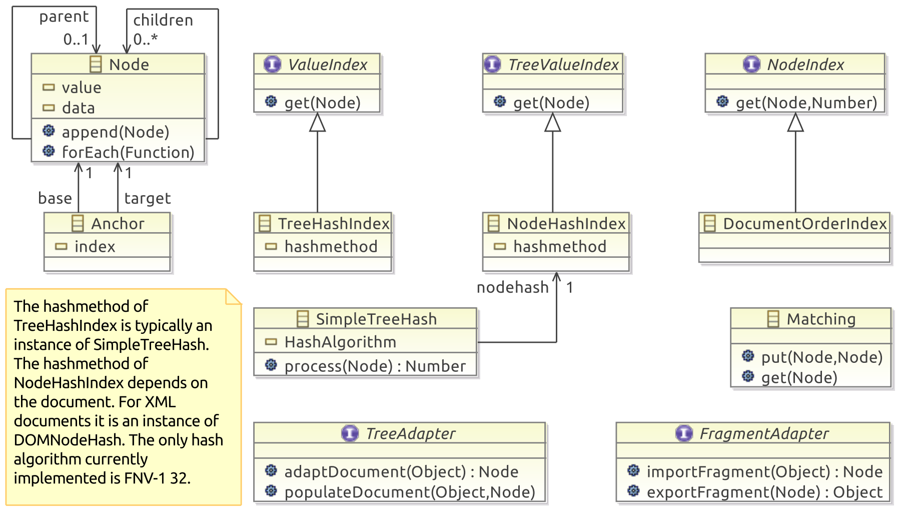
skelmatch
Diff (from skelmatch)
matchTrees(Node,Node) : Matching
resolver
ResolverResult (from resolver)
Resolver (from resolver)
resolver 1
WeightedContextMatcher (from resolver)
matcher 1
ContextResolver (from resolver)
envsize
threshold
The io-package and the interfaces it contains do not realy exist in the source code. JavaScript has no built-in support for interface declarations. They are shown here in order to clarify the role of the classes contained in the domtree, domdelta and xmlpayload packages.
Green shaded classes are used exclusively when diffing while red ones are only used when patching. Classe with a white background are shared.
tree
Node (from tree)
refnode 1
base 1
target 1
nodeindex 1
nodeindex 1
NodeIndex (from tree)
Anchor (from tree)
anchor 1
1 anchor
A matching is the result of a diff operation. The delta collector turns it into a set attached operations.
Matching (from tree)
matching 1
delta
Attacher (from delta)
AttachedOperation (from delta)
DeltaCollector (from delta)
DetachedOperation (from delta)
0..1 handler
OperationHandler (from delta)
contextdelta
DetachedContextOperati (from contextdelta)
ContextGenerator (from contextdelta)
contextgen 1
Detacher (from contextdelta)
domhandler
DOMNodeReplaceOperationHandl (from domhandler)
DOMSubtreeOperationHandler (from domhandler)
DOMOperationHandlerFactory (from domhandler)
io
TreeAdapter (from tree)
adaptDocument(Object) : Node
populateDocument(Object,Node)
DeltaAdapter (from delta)
adaptDocument(Object) : Operations
populateDocument(Object,Operations)
FragmentAdapter (from tree)
importFragment(Object) : Node
exportFragment(Node) : Object
domtree
DOMTreeAdapter (from domtree)
domdelta
DOMDeltaAdapter (from domdelta)
xmlpayload
XMLFragmentAdapter (from xmlpayload)
XMLPayloadHandler (from xmlpayload)

Figure 26: The building blocks of the framework foundation. Note that this package diagram is not complete but it covers the most important part and shows the relations between the packages. In order to reduce complexity, most compartements are collapsed. Many of the classes were already mentioned in Figures 23, 24 and 25.

Toolkit Version 2[16].

6.4.1. The `tree` and `domtree` Packages

The classes `Node` and `Anchor` form the heart of the tree package. The former is used to build up a tree structure which abstracts the underlying document. The class is kept as simple as possible. Multiple nodes are structured into a hierachy by way of the parent-child relationship. The tree structure is not meant to be altered after it was built using a `TreeAdapter`. A reference to the data representing a tree node in the underlying document can be kept in the corresponding property. The equality test of the diff-algorithms default to comparing the value property of two nodes. For XML documents the test method is overriden with a specialized function involving comparison of hash values though.



parent
0..1
children
0..*
Node
value
data
append(Node)
forEach(Function)
1
1
base
target
Anchor
index
ValueIndex
get(Node)
TreeValueIndex
get(Node)
NodeIndex
get(Node,Number)
TreeHashIndex
hashmethod
NodeHashIndex
hashmethod
DocumentOrderIndex
nodehash
1
The hashmethod of TreeHashIndex is typically an instance of SimpleTreeHash. The hashmethod of NodeHashIndex depends on the document. For XML documents it is an instance of DOMNodeHash. The only hash algorithm currently implemented is FNV-1 32.
SimpleTreeHash
HashAlgorithm
process(Node) : Number
Matching
put(Node,Node)
get(Node)
TreeAdapter
adaptDocument(Object) : Node
populateDocument(Object,Node)
FragmentAdapter
importFragment(Object) : Node
exportFragment(Node) : Object

Figure 27: Classes of the tree package.

By means of the `Anchor` class it is possible to specify any position in the tree. The `target` references the node the anchor is pointing at. The `base` represents the targets parent node and the `index` its offset in the children list of the base. The `target` may be undefined. This is the case when the anchor points beyond the end of the children list. Also the `base` may be undefined when the target points at the root node.

The `TreeHashIndex` and `NodeHashIndex` provide access to hash values of single nodes and their subtrees. These classes augment node objects with a property where computed hash values are cached in order to conserver some processor cycles.

Accessing tree nodes relative to a reference node at a given offset in document order can be accomplished with an instance of `DocumentOrderIndex`. This class is mainly used when computing or resolving the values of context nodes relative to an operation.

[16]JSDoc Toolkit:
http://code.google.com/p/jsdoc-toolkit/

The `Matching` class provides a convenient interface for pairing nodes from two trees. The result of running one of the tree-diff algorithm is essentially an instance of this class.

By means of the `DOMTreeAdapter` class from the `domtree` package it is possible to construct a tree structure using the foundations `Node` class from an object implementing the W3C DOM interface, i.e. a `DOMDocument`. Objects of this type can be optained by parsing an XML string using a DOM parser or by accessing the `window.document` property from within a script running in a browser window.

The `domtree` package also provides the `DOMNodeHash` class which implements a method capable of calculating a hash value over an XML element. Namespaces and attributes are normalized in order to prevent different hash values for elements with varied attribute order or changed namespace prefix. The hashing-method and the normalization is implemented in accordance to [16]. The only hashing algorithm available in the current version is FNV1-32 [19].

6.4.2. The `delta`, `contextdelta` and `domdelta` Packages

The packages having the word delta in their name unsurprisingly provide the methods to construct operations and to convert between different kinds of them. Note that we will discuss some classes in pairs. Some of them are complementary but live in different packages. Figure 28 shows the classes of the more generic `delta` package while Figure 29 depicts those of the `contextdelta` package.

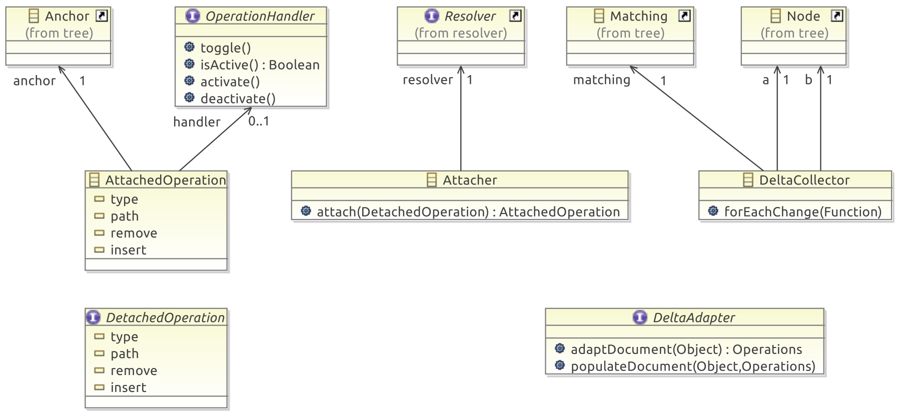


Figure 28: Classes of the delta package.

Most important classes of these packages are the `AttachedOperation` as well as the `DetachedContextOperation`. As already mentioned before, instances of the attached variant are tied to a tree by means of an `Anchor`. In contrast detached operations provide additional information which helps the resolver to locate their entry point in a tree. In the case of the `DetachedContextOperation` values of leading and trailing context nodes are recorded in addition to the standard attributes.

Conversion between attached and detached operations is done by means of the `Attacher` and `Detacher` classes. The former is not much more than a wrapper around the selected

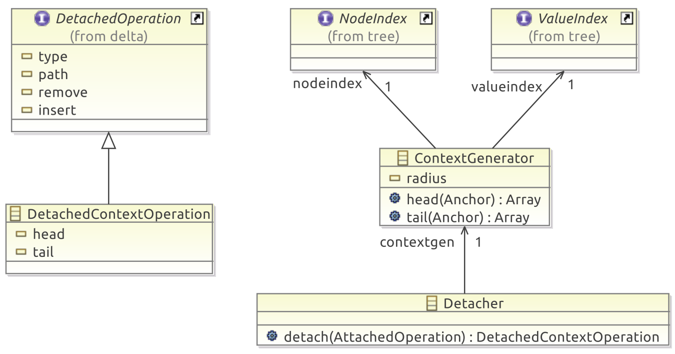


Figure 29: Classes of the contextdelta package.

resolver algorithm. The latter depends on the characteristics of the target delta file. In our case, the values of nodes surrounding the target of an operation need to be collected and stored into the head and tail properties of a new `DetachedContextOperation` instance. Typically `nodeindex` is an instance of `DocumentOrderIndex` and `valueindex` a `NodeHashIndex` object (See previous section for detailed information on those classes).

Finally the `domdelta` package provides the `DOMDeltaAdapter` class capable of converting a list of `DetachedContextOperation` instances into a DOM structure and vice versa. For more details, refer to the API reference or the foundation overview depicted in Figure 26.

6.4.3. Operation Handlers

We already mentioned in Section 6.4.1 that we avoid modifying the tree structure once it was built up using a `TreeAdapter`. That means even during the patching process the adapted structure remains unaltered. Instead the underlying document is changed directly whenever an operation is applied or reverted. For each operation type there must be one handler class per supported document type. Currently only handlers capable of operating on a DOM structure are available, namely `DOMNodeReplaceOperationHandler` for node-update operations and `DOMSubtreeOperationHandler` for inserting and removing sequences of subtrees.

6.4.4. Classes Implementing Diff- and Patch Algorithms

The packages `skelmatch`, `xcc` and `resolver` contain classes implementing the corresponding diff and patch algorithms. They are discussed in depth above in Section 4 and Section 5.

6.5. Frontends

A set of frontends have been produced in order to demonstrate the capabilities of the framework.

6.5.1. Command Line Interface

The command line interface consists of the tools `djdiff` and `djpatch`. The former can be used to detect the changes between two versions of a XML document and save them into a patch file. The latter has the capability of applying a patch file to a document. By means of a set of command line options, the behaviour of the tools can be adapted to different needs. Figure 30 gives an overview.

Built in help for djdiff:

```
Usage: djdiff [options] FILE1 FILE2

Available options:
  -h, --help            Show this help
  -p, --payload STRING  Specify payload type (xml or json, default: detect)
  -g, --algo STRING     Specify algorithm (skelmatch or xcc, default: skelmatch)
  -x, --xml             Use XML patch format (default)
  -j, --json            Use JSON patch format
  -d, --debug           Log actions to console
```

Built in help for djpatch:

```
Usage: djdiff [options] FILE PATCH

Available options:
  -h, --help              Show this help
  -p, --payload STRING    Specify payload type (xml or json, default: detect)
  -r, --radius NUMBER     Search radius for fuzzy matching (default: 6)
  -t, --threshold NUMBER  Threshold value for fuzzy matching (default: 0.7)
  -d, --debug             Log actions to console
```

Usage example:

```
cd test/fixtures/
node ../../bin/djdiff.js -p xml zappa-quote-1.html zappa-quote-2.html > zappa-patch.xml
node ../../bin/djpatch.js -p xml zappa-quote-1.html zappa-patch.xml > zappa-quote-patched.html
```

Now `zappa-quote-2.html` and `zappa-quote-patched.html` will have the same content.

Figure 30: Command line options and usage example for djdiff and djpatch. Note that support for JSON is very limited at the moment.

There is also the tool `djhash` which prints the hash values of each XML node encountered in a document to the console. This tool is helpful when one needs to manually verify the head and tail context contained in a patch file. Figure 31 explains the output of the djhash tool in greater detail.

```
$ ./bin/djhash.js test/fixtures/logo-1.svg
 svg                                   35642048 953b08a9
     defs                              60179796 5b38b0cd
     sodipodi:namedview                33b20bf6 f206371d
     metadata                          9bcae695 b14ced8f
         rdf:RDF                       c299b76e 6724a49b
             cc:Work                   89b99f02 3c8c20d7
                 dc:format             57b2f766 aca5381d
                     #text             b11f2c1c e48f1925
                 dc:type               4116347a de39a86a
                 dc:title              54b5f69b 9caa22b5
     g                                 6ed152c 9127d360
         g                             4770a978 a882df54
             text                      391e63db cbabd22b
                 tspan                 78a84384 88be8a30
                     #text             1310d1e6 b19f6351
             text                      459f02a7 7a292e10
                 tspan                 30b5c1ee c0ee219d
                     #text             eb7a706b ee4102ad
```

Figure 31: Output of the djhash command when run against the `logo-1.svg` fixture. In the left half the xml tree structure is displayed by means of the tag-names. The two columns on the right side represent the node hash and the tree hash values respectively.

6.5.2. LCS Demo

In order to understand the Longest Common Subsequence problem we implemented a small JavaScript application capable of drawing the edit graph produced by different input strings. Because of the visual feedback, the LCS demo also proved to be very useful when we had to analyze bugs in our implementation. Additionally we could use the application to generate figures for the documentation, e.g. the ones shown in Figure 10.

We used the visualization framework Processing.js[17] to produce the graphics. The idea on how to visualize the algorithm was heavily inspired by Nicholas Butlers tutorial "Investigating Myers' diff algorithm"[18].

6.5.3. Tree Matching Demo

Also for illustration purposes as well as for runtime examination of the algorithm we provide a JavaScript application which visualizes the XCC tree matching algorithm and the different variants we implemented. Like the LCS Demo, this application is based on Processing.js for the graphics.

[17]Processing.js Visualization Framework
http://www.processing.js

[18]Nicholas Butler: Investigating Myers' diff algorithm
http://www.nickbutler.net/Article/DiffTutorial1

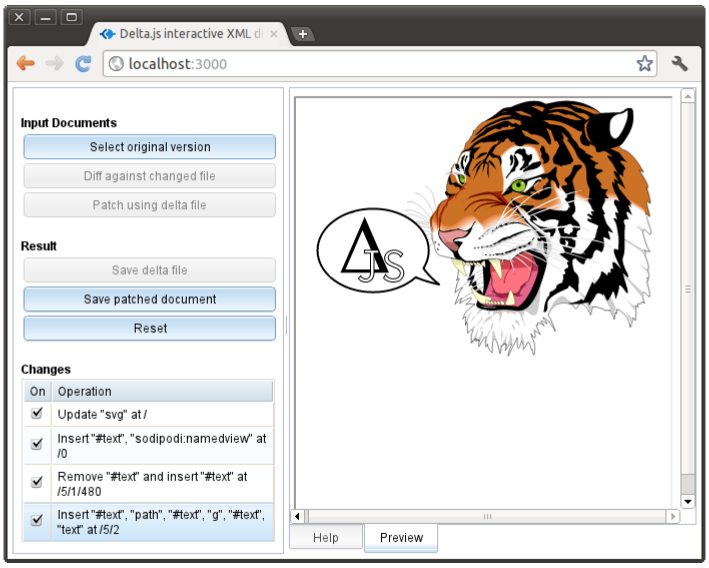
6.5.4. XML Source Diff Demo

The XML Source Diff Demo lets you choose two XML, HTML or SVG files, then it shows the differencies between them by highlighting the relevant parts in the XML source code. The syntax highlighting is done using Google Code Prettify[19]. We also reused the `style_html` function from JS Beautifier[20].

6.5.5. XML Visual Merge Demo

As an optional objective we intended to explore possibilities to visualize changes to documents by highlighting affected parts in the layout. E.g. by framing changed parts of a website with red borders or by shading unaffected parts such that the changes will stand out.

As a first step we implemented a JavaScript application capable of generating patch files as well as selectively applying them to a document. During this interactive process, the influence of certain changes can be monitored by keeping an eye on the built-in live preview. The preview can handle all XML file formats which a browser can render, thus also SVG images are fully supported. A screenshot of the application is shown in Figure 32.



Changes

On	Operation
✓	Update "svg" at /
✓	Insert "#text", "sodipodi:namedview" at /0
✓	Remove "#text" and insert "#text" at /5/1/480
✓	Insert "#text", "path", "#text", "g", "#text", "text" at /5/2

Figure 32: A screenshot of the XML Visual Merge demo. Individual changes can be selected using the checkboxes displayed in the left column. The preview is refreshed in real-time.

[19] Google Code Prettify Syntax Highlighter: http://code.google.com/p/google-code-prettify/

[20] JS Beautifier: https://github.com/einars/js-beautify

The user interface was built using the Dojo Toolkit[21]. This demo application also shows how our framework can be used in conjunction with an AMD module loader[22].

6.6. Tools

As a secondary objective we also wanted to identify tools which support development of JavaScript based applications. In this section we present some we relied on heavily during the development.

6.6.1. Testing and Test-Coverage

Unit-testing is even more important for projects written in a dynamic language like JavaScript than it is for statically typed programs. There are numerous unit-testing frameworks available for JavaScript, some tailored to the browser, other more useful on the server side. We settled with nodeunit[23] because it is capable of executing the tests on the command line as well as in a browser.

The Makefile included with the source code provides targets for running the test-suite server-side as well as in the browser. The command `make test` will execute all unit tests from the command line. By invoking `make browser`, the framework as well as the test-suite is aggregated into one file and stored within the directory `dist/browser-test`. The test-suite now can be executed by opening the file `dist/browser-test/test.html` from within a browser.

Note that some browsers enforce very restrictive security measures and may not allow the test suite to load additional files from the file system[24]. For this case a little helper script spawning a basic http server delivering static files from a directory is included in the source tree. When invoked as specified below, all the files within the directory `dist/browser-test` will be served using HTTP on TCP port 3000:

```
node scripts/http.js dist/browser-test
```

The test suite can then be run by entering the url `http://localhost:3000/test.html` into the location bar of any browser.

Measuring test coverage is currently a bit tricky in JavaScript. Most tools instrument the code, i.e. inject additional statements before each line of code in order to keep track which lines were evaluated how often. When done sloppily, the instrumentation may introduce bugs into the code.

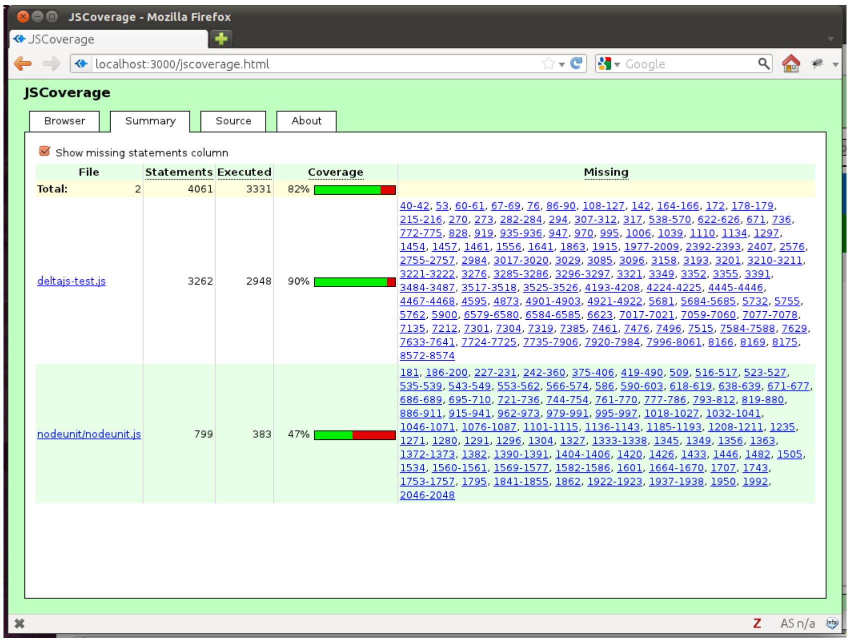
[21]Dojo Toolkit
`http://dojotoolkit.org/`

[22]AMD: Asynchronous Module Definition API
`https://github.com/amdjs/amdjs-api/wiki/AMD`

[23]Nodeunit Test Framework:
`https://github.com/caolan/nodeunit`

[24]E.g. Google Chrome denies any file system access when a page was loaded from a url of the form file:///path/to/some/file.html.

The tool we use to analyze test coverage is JSCoverage[25]. At the moment our tests reach around 90% of the framework code. A screenshot of the JSCoverage user interface is shown in Figure 33.



JSCoverage - Mozilla Firefox
JSCoverage
localhost:3000/jscoverage.html
Google
JSCoverage
Browser
Summary
Source
About
Show missing statements column
AS n/a

File	Statements	Executed	Coverage	Missing
Total: 2	4061	3331	82%	
deltajs-test.js	3262	2948	90%	40-42, 53, 60-61, 67-69, 76, 86-90, 108-127, 142, 164-166, 172, 178-179, 215-216, 270, 273, 282-284, 294, 307-312, 317, 538-570, 622-626, 671, 736, 772-775, 828, 919, 935-936, 947, 970, 995, 1006, 1039, 1110, 1134, 1297, 1454, 1457, 1461, 1556, 1641, 1863, 1915, 1977-2009, 2392-2393, 2407, 2576, 2755-2757, 2984, 3017-3020, 3029, 3085, 3096, 3158, 3193, 3201, 3210-3211, 3221-3222, 3276, 3285-3286, 3296-3297, 3321, 3349, 3352, 3355, 3391, 3484-3487, 3517-3518, 3525-3526, 4193-4208, 4224-4225, 4445-4446, 4467-4468, 4595, 4873, 4901-4903, 4921-4922, 5681, 5684-5685, 5732, 5755, 5762, 5900, 6579-6580, 6584-6585, 6623, 7017-7021, 7059-7060, 7077-7078, 7135, 7212, 7301, 7304, 7319, 7385, 7461, 7476, 7496, 7515, 7584-7588, 7629, 7633-7641, 7724-7725, 7735-7906, 7920-7984, 7996-8061, 8166, 8169, 8175, 8572-8574
nodeunit/nodeunit.js	799	383	47%	181, 186-200, 227-231, 242-360, 375-406, 419-490, 509, 516-517, 523-527, 535-539, 543-549, 553-562, 566-574, 586, 590-603, 618-619, 638-639, 671-677, 686-689, 695-710, 721-736, 744-754, 761-770, 777-786, 793-812, 819-880, 886-911, 915-941, 962-973, 979-991, 995-997, 1018-1027, 1032-1041, 1046-1071, 1076-1087, 1101-1115, 1136-1143, 1185-1193, 1208-1211, 1235, 1271, 1280, 1291, 1296, 1304, 1327, 1333-1338, 1345, 1349, 1356, 1363, 1372-1373, 1382, 1390-1391, 1404-1406, 1420, 1426, 1433, 1446, 1482, 1505, 1534, 1560-1561, 1569-1577, 1582-1586, 1601, 1664-1670, 1707, 1743, 1753-1757, 1795, 1841-1855, 1862, 1922-1923, 1937-1938, 1950, 1992, 2046-2048

Figure 33: The test coverage results for *delta.js* as reported by JSCoverage

6.6.2. Debugging and Profiling

Most modern browsers now provide quite comfortable developper tools out of the box. Many of those tools exist for the server side as well (*node.js*), but they are far from being easy to use let alone well integrated. When we needed to examine running code, we used node-inspector[26] is capable of connecting to a running *node.js* instance and serving the debugging interface to a WebKit based browser[27].

In order to analyze the contribution of individual methods to the overall runtime, the v8 profiler comes in handy[28]. Recall the experiment we outlined Section 4.5 which shows the contribution of the LCS-complexity to the overall skel-match runtime. When run with the profiler enabled, we optain the output shown in Figure 34. The top 7 entries accounting for 85.1% of the runtime clearly are part of the LCS module.

[25] JSCoverage: http://siliconforks.com/jscoverage/
[26] https://github.com/dannycoates/node-inspector
[27] Browsers based on WebKit include Chromium, Google Chrome and Apples Safari
[28] http://code.google.com/p/v8/wiki/V8Profiler

6.6.3. Static Analyzis

Because JavaScript is an interpreted language and therefore source code is not compiled before it is executed, programming errors are only detected at runtime. Additionally due to the fact that JavaScript interpreters are required to autonomously insert missing semicolons at the end of lines, the syntax can sometimes become ambigous. In order to mitigate the risk of introducing hard to find bugs, a program capable of analyzing the source code and flagging potential problems is very helpful.

We regularely check our code with JSLint[29]. The tool is also available as a command line version[30]. This is especially attractive for integration into a build script or a `Makefile`.

```
Statistical profiling result from v8.log, (40821 ticks, 36729 unaccounted, 0 excluded).

[...]

 [JavaScript]:
   ticks  total  nonlib   name
  21696   53.1%   53.2%  Function: LCS.middleSnake lib/delta/lcs.js:293
   9308   22.8%   22.8%  Function: LCS.compute lib/delta/lcs.js:53
   2281    5.6%    5.6%  Function: LCS.nextSnakeHeadForward lib/delta/lcs.js:191
    772    1.9%    1.9%  Function: LCS.forEachCommonSymbol lib/delta/lcs.js:169
    666    1.6%    1.6%  Function: Diff.matchContent.lcsinst.forEachCommonSymbol.a
                                    lib/delta/skelmatch.js:224
    464    1.1%    1.1%  Function: LCS.nextSnakeHeadBackward lib/delta/lcs.js:242
    263    0.6%    0.6%  Function: KPoint.translate lib/delta/lcs.js:411
    186    0.5%    0.5%  Function: Diff.equalContent lib/delta/skelmatch.js:115
[...]
      3    0.0%    0.0%  Function: <anonymous> lib/delta/tree.js:55
[...]
      2    0.0%    0.0%  Function: Diff.collectBones lib/delta/skelmatch.js:264
[...]
      1    0.0%    0.0%  Function: Diff.matchStructure.forEachUnmatchedSequenceOfSiblings
                                    .lcsinst.forEachCommonSymbol.a lib/delta/skelmatch.js:376
      1    0.0%    0.0%  Function: Diff.matchStructure.forEachUnmatchedSequenceOfSiblings
                                    .lcsinst.forEachCommonSymbol.a lib/delta/skelmatch.js:373
      1    0.0%    0.0%  Function: Diff.matchStructure lib/delta/skelmatch.js:360
[...]
```

Figure 34: Profiling information as reported by v8 profiler. The top 7 entries accounting for 85.1% of the runtime clearly are part of the LCS module. Some lines which were deleted from the output are marked using [. . .].

[29] JSLint, The JavaScript Code Quality Tool: http://www.jslint.com/
[30] JSLint command line version: https://github.com/reid/node-jslint

6.7. JavaScript Pecularities

In this section we will point out some specialities of the JavaScript language and we point out the measures we took to cope with them in our code base.

6.7.1. Modularization: CommonJS vs AMD Module Format

JavaScript currently does not provide a standard way to load additional code from external ressources. Also there is no support for isolation of components from each other. Numerous approaches exist which simplify code reuse and try to work around this limitation with more or less success.

In order to prevent polution of the global namespace it is generally recommended that JavaScript code is enclosed into a immediately executed anonymous function. Using this method it is possible to only selectively export symbols which should be available to client code. All advanced modularization methods and loaders are built around this basic pattern. An example of the module pattern is shown in Figure 35.

```
1  var GLOBALOBJECT={};
2
3  (function(exports) {
4      var answer = 42;
5
6      function getanswer() {
7          return answer;
8      }
9
10     function tellme() {
11         console.log(getanswer());
12     }
13
14     exports.tellme = tellme;
15 }(GLOBALOBJECT));
16
17 GLOBALOBJECT.tellme();
```

Figure 35: The JavaScript module pattern. The symbols `answer` and `getanswer` are kept within the anonymous closure starting at line 3. The function is immediately executed after its definition on line number 15 with the `GLOBALOBJECT` as its only parameter.

In addition to isolation, the two popular module formats CommonJS[31] and AMD[32] provide methods which allow the client to retrieve and execute additional code files. The CommonJS specification is reasonable simple to use and is popular on the server-side. Node.js provides functions capable of loading CommonJS modules.

[31]CommonJS module specification:
http://wiki.commonjs.org/wiki/Modules/1.0

[32]AMD: Asynchronous Module Definition
https://github.com/amdjs/amdjs-api/wiki/AMD

However CommonJS has the drawback that the loader is executed synchronously. This does not blend in well with the asynchronous nature of JavaScript, especially on the browser platform. The JavaScript code of an application either is concatenated into one single file before it is served to the browser, or an AMD loader is delivered to the client which takes care of retrieving additional code modules. Lukily modules written in the CommonJS module format are easily convertible into AMD style by means of the `r.js` script available as a part of the require.js package[33]

6.7.2. Lack of Integer Data Type

JavaScript does not provide a data type for integer values. The specification[34] requires that the implementation of the `Number` primitive must be based on 64bit floating point nubers. Although operations within the range of 32bit signed values is guaranteed to produce results without rounding errors, bitwise operations may still produce unexpected results[35].

We had to work around this issue in our implementation of the FNV1-32 hashing algorithm because JavaScript lacks proper integer support.

7. Discussion and Conclusion

7.1. Research Results and Prior Work

Before being able to choose a suitable algorithm as the base for our implementation, we had to acquire quite a bit of knowledge in that domain. Especially important was to work out the features which set the specialized diff algorithms for structured documents appart from the generic tree to tree correction problem and subsequent refinements. A part of this preliminary work has been completed within the context of the Project Thesis 2 [23] during the preceeding semester.

Although our curriculum comprehends lectures on *Algorithms and Datastructures* as well as *Complexity and Computability*, virtually all of the presented algorithms were new to us. We mostly relied on scientific articles and also on free and open source code in order to analyze the methods discussed in this work.

7.2. Fulfillment of Project Goals

The amount of work that was necessary to reach the goals we declared was roughly what we expected. However we did not plan to depart from the original XCC implementation in such an extent. While the development of the *Skel-match* variant did not endanger the fulfillment of the mandatory targets, there was only little time left for the optional

[33]require.js Module Loader:
http://requirejs.org/

[34]The JavaScript language is defined by the ECMA-262 standard [1]. Currently the 5th edition is supported by many platforms.

[35]Refer to the Complete JavaScript Number Reference for some interesting details:
http://www.hunlock.com/blogs/The_Complete_Javascript_Number_Reference

ones. The web application presented in Section 6.5.5 partially covers the requirements defined by the first two optional goals (See **??**). The third optional goal was dropped completely.

On the other hand we are very happy with our implementation. We delivered a modular and extendible framework with an architecture which takes into account the peculiarities of JavaScript as well as common practice on coping with them. Our code base is compatible with competing module formats. The vast majority can be shared between different platforms, i.e. server software and the browser. There are automated unit tests in place for every algorithm and most of the supporting code and the project is organized such that the test coverage can be verified easily. The command line interface and the demo web applications demonstrate a broad range of options on how to interface with our framework.

We took a bit of a risk implementing this project in JavaScript, a language that we barely knew before. Because many web ressources and books on JavaScript are targeted at programming novices, we first had to identify trustworthy sources in order to quickly get up to speed with the language. The book "Pro JavaScript Techniques" by John Resig[36] and the screen cast series "Crockford on JavaScript" by Douglas Crockford[37] were extremely helpful for that matter.

7.3. Future Work

The design of our framework encourages the addition of new file formats, patch formats and diff-/patch algorithms. From an academic point of view it would be interesting to implement more of the algorithms we discussed (e.g. BULD, Section 2.5.3) and compare them performance-wise. However one might consider reimplementing the framework in a programming language which behaves less uncertain than JavaScript in order to produce meaningful results.

Recalling that the runtime of the presented diff algorithm highly depends on the complexity of the LCS, further investigations in that direction could lead to improved performance. It has been shown that Myers LCS algorithm can be refined such that the runtime is cut by a factor of 2 [25]. While swapping the LCS algorithm will not have any inpact on worst case complexity, we can expect a better overall runtime from faster LCS implementations.

In some situations the approach taken to resolve operations in the target tree fails due to a bad initial guess (See: Section 5.2). Because the context fingerprints are arranged in document order, an algorithm based on the document order instead of the document hierarchy even for the first guess should produce better results.

[36] John Resig, Pro JavaScript Techniques, ISBN 1590597273, Apress 2006, `http://jspro.org`
[37] Douglas Crockford, Crockford on JavaScript, `http://yuiblog.com/crockford/`

A. Project Setup and Build Instructions

A.1. Build Instructions for node.js

Delta.js runs within modern web browsers as well as under *node.js*. In order to use it as a module for *node.js* based projects, just drop this directory into an appropriate module path. For example the `node_modules` folder of your project.

The *delta.js* source directory is organized as an npm package. Invoking the command `npm install` will download any dependencies and install them into the directory `node_modules`.

It is recommended to verify the installation by running the automated test suite once by invoking `make test`.

A.2. Building AMD modules

In order to build a version suitable for AMD module loaders like dojo or require.js, invoke `make amd`. The built modules are put into the directory `dist/amd`.

A.3. Building Single File Browser Version

A single-file browser version of the framework can be built using the command `make browser`. The result is placed into `dist/browser/delta.js`. Note that this version is not compatible with AMD modules.

A.4. Running the Command Line Tools

The command line utilities are located in the `bin` directory. They may be invoked directly without a prior build. There are some sample XML files in `test/fixtures` which are useful to quickly test the command line interfaces.

Follow this example in order to produce a patch by diffing two versions of an XML file as well as apply it afterwards back to the original version.

```
./bin/djdiff.js -p xml ./test/fixtures/logo-1.svg \
    ./test/fixtures/logo-2.svg > /tmp/logo-diff.xml
./bin/djpatch.js -p xml ./test/fixtures/logo-1.svg \
    /tmp/logo-diff.xml > /tmp/logo-1-patched.svg
```

The file `/tmp/logo-diff.xml` will contain the changes between `logo-1.svg` and `logo-2.svg` while the file `/tmp/logo-1-patched.svg` will contain the same contents as `logo-2.svg`.

A.5. Running the Browser Based Examples

In order to build the examples, invoke `make examples`. Run `node scripts/http.js examples/lcs` in order to start a local webserver for the LCS example. Then point your browser at http://localhost:3000 in order to access the LCS web application.

The following examples are available:

`example/lcs`: A visualization of Myers Longest Common Subsequence algorithm.

`example/xcc`: A web application allowing to step through the XCC tree matching algorithm.

`example/srcdiff`: Given two versions of an XML file, this web application will highlight the differencies on the XML source code.

`example/vizmerge/src`: A web application allowing diffing and selectively merging of changes in XML documents. This demo application also features a live preview where the effects of a change are shown in realtime.

A.6. Bulding the API Reference

In order to build the API reference, the Python Documentation Generatior Sphinx and the jsdoc toolkit version 2 is required. After invoking `make doc`, the directory `doc/_build/html` contains the built documentation in HTML format.

B. Project Management

B.1. Methodology

While not sticking strictly to an agile methodology/framework like Scrum or XP the core principles of agile project management are taken into account while developing this project. Risk and exploration factor are high and therefore change of coverage and even objectives should not be precluded until late in the project.

Meetings with the supervisor are held frequently on a mostly weekly basis where current and future work is discussed.

B.2. Project Livecycle

Agile methodologies are frequently divided into four to five phases ("Envision", "Speculate", "Explore", "Adapt", "Close") [13]. Applied to the livecycle of this project thesis the first phase comprises the project proposal (before the start of the semester). The second phase comprises the first couple of weeks where identifying and resolving the difficult problems is important, the third phase mostly consists of writing activity while the last one is used to close up the project.

B.3. Risk Management and Planning

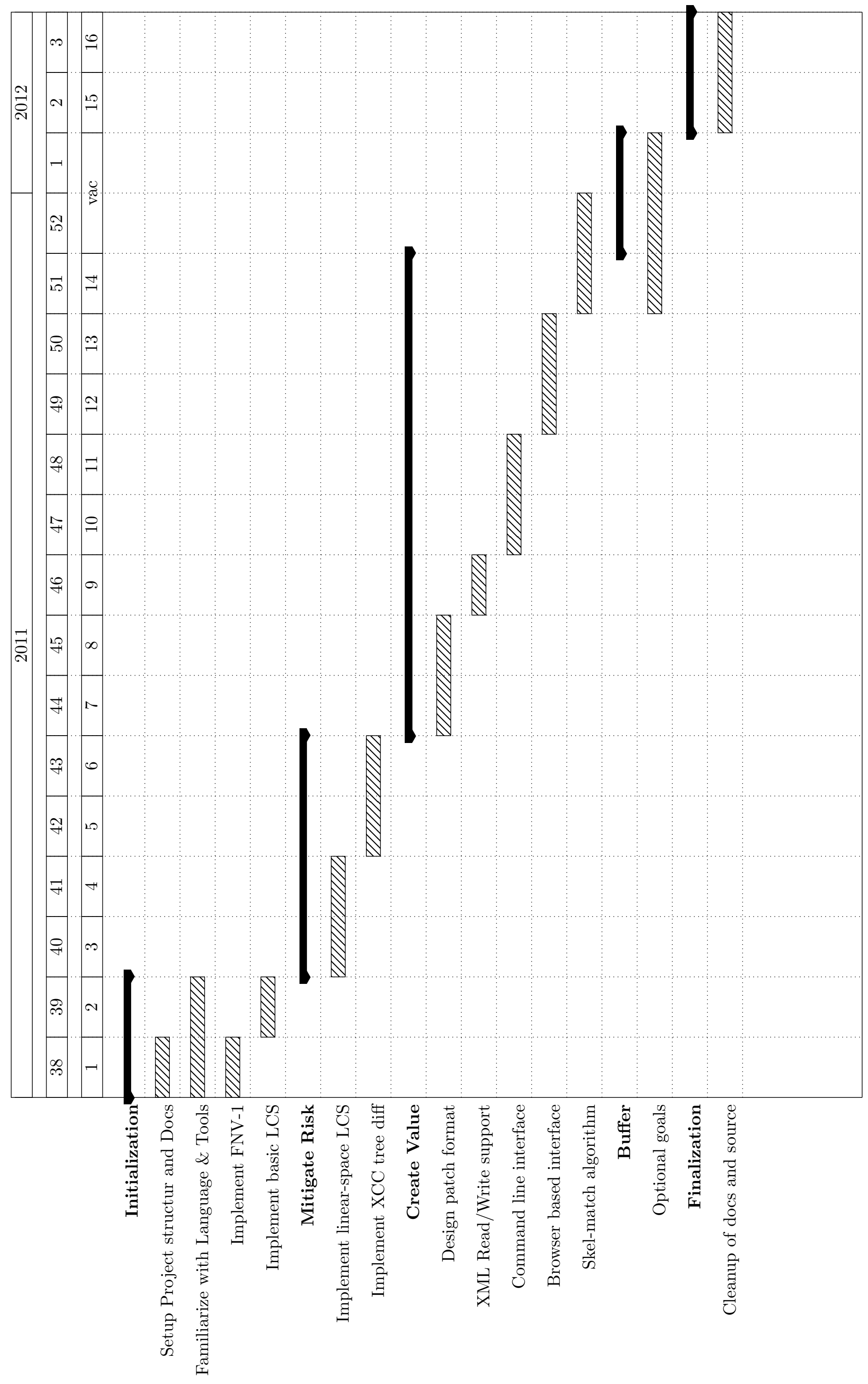
One of the most important aspect of agile software development methodologies is that they encourage continuous reduction of risks. Instead of trying to work out a detailed model of the system up front, important features are added to the system as the project advances.

Features which supposedly are afflicted with greater risk, either because their implementation is expected to be complicated or because additional knowledge has to be acquired in order to tackle them, should be done as early as possible. We always adhere to this principle when planning the next steps in the project. Refer to the gantt chart shown in Section B.5 for an overview of the project plan.

B.4. Tools

No specialized tools are used in order to schedule and prioritize tasks. Reading notes, progress reports, meeting minutes and also tasks which either are marked with [open] and [done] are recorded in a simple textfile (journal.txt). Using a text editor which is able to highlight lines matching a given pattern (like vim), it is very easy to keep the overview of open tasks.

B.5. Chart


2011
2012
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
1
2
3
1
2
3
4
5
6
7
8
9
10
11
12
13
14
vac
15
16
Initialization
Setup Project structur and Docs
Familiarize with Language & Tools
Implement FNV-1
Implement basic LCS
Mitigate Risk
Implement linear-space LCS
Implement XCC tree diff
Create Value
Design patch format
XML Read/Write support
Command line interface
Browser based interface
Skel-match algorithm
Buffer
Optional goals
Finalization
Cleanup of docs and source

C. List of Figures

D. Listings

E. References

[1] *ECMA-262 ECMAScript Language Specification.* 2009. Available from: http://www.ecma-international.org/publications/standards/Ecma-262.htm. 49

[2] Several Authors. GNU diffutils manual, April 2010. Available from: http://www.gnu.org/software/diffutils/manual/. 30

[3] David T. Barnard, Gwen Clarke, and Nicholas Duncan. Tree-to-tree correction for document trees technical report 95-372. Technical report, January 1995. Available from: http://ftp.qucis.queensu.ca/TechReports/Reports/1995-372.pdf. 10

[4] Philip Bille. A survey on tree edit distance and related problems. *Theoretical computer science*, 337(1-3):217–239, 2005. doi:10.1016/j.tcs.2004.12.030. 5, 12

[5] Sudarshan S. Chawathe. Comparing hierarchical data in external memory. In *Proceedings of 25th International Conference on Very Large Data Bases*, page 90–101, September 1999. Available from: http://www.cs.umaine.edu/~chaw/pubs/xdiff.pdf. 9

[6] Sudarshan S. Chawathe. *Managing change in heterogeneous autonomous databases.* PhD thesis, Stanford University, 1999. Available from: http://www.cs.umaine.edu/~chaw/pubs/cm.pdf. 11

[7] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data - SIGMOD '96*, pages 493–504, Montreal, Quebec, Canada, 1996. doi:10.1145/233269.233366. 10, 11

[8] Grégory Cobéna, Talel Abdessalem, and Yassine Hinnach. A comparative study for XML change detection. *Research Report, INRIA Rocquencourt, France*, 2002. Available from: ftp://tfalati.inria.fr/INRIA/Projects/gemo/gemo/GemoReport-221.pdf. 13

[9] Grégory Cobéna, Serge Abiteboul, and Amélie Marian. Detecting changes in XML documents. In *Proceedings of the International Conference on Data Engineering - ICDE 2002*, pages 41–52, February 2002. doi:10.1109/ICDE.2002.994696. 11

[10] Erik D. Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann. An optimal decomposition algorithm for tree edit distance. *ACM Transactions on Algorithms*, 6(1):1–19, December 2009. doi:10.1145/1644015.1644017. 9

[11] Serge Dulucq and Hélène Touzet. Analysis of tree edit distance algorithms. In *Combinatorial Pattern Matching*, volume 2676, page 83–95, 2003. doi:10.1007/3-540-44888-8_7. 9

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Professional, 1995. 34

[13] James Highsmith. *Agile project management : creating innovative products.* Addison-Wesley, Upper Saddle River NJ, 2nd ed. edition, 2010. 52

[14] Philip N. Klein. Computing the edit-distance between unrooted ordered trees. *Algorithms - ESA'98*, page 1–1, 1998. doi:10.1007/3-540-68530-8_8. 8

[15] Tancred Lindholm, Jaakko Kangasharju, and Sasu Tarkoma. Fast and simple XML tree differencing by sequence alignment. In *Proceedings of the 2006 ACM symposium on Document engineering*, page 75–84, 2006. doi:10.1145/1166160.1166183. 11

[16] H. Maruyama, K. Tamura, and N. Uramato. Digest values for DOM (DOMHASH), April 2000. Available from: `http://tools.ietf.org/html/rfc2803`. 40

[17] Webb Miller and Eugene W. Myers. A file comparison program. *Software: Practice and Experience*, 15(11):1025–1040, November 1985. `doi:10.1002/spe.4380151102`. 3

[18] Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, November 1986. `doi:10.1007/BF01840446`. 3, 5, 9, 13, 18

[19] Landon Curt Noll. FNV hash. Available from: `http://isthe.com/chongo/tech/comp/fnv/`. 40

[20] Sebastian Rönnau and Uwe M. Borghoff. Versioning XML-based office documents. *Multimedia Tools and Applications*, 43(3):253–274, March 2009. `doi:10.1007/s11042-009-0271-2`. 32

[21] Sebastian Rönnau and Uwe M. Borghoff. XCC: change control of XML documents. *Computer Science - Research and Development*, November 2010. `doi:10.1007/s00450-010-0140-2`. 3, 12, 21

[22] Sebastian Rönnau, Christian Pauli, and Uwe M. Borghoff. Merging changes in XML documents using reliable context fingerprints. In *Proceeding of the eighth ACM symposium on Document engineering*, page 52–61, 2008. `doi:10.1145/1410140.1410151`. 32

[23] Lorenz Schori. Difference algorithms and merging strategies for structured documents, June 2011. 3, 49

[24] Kuo-Chung Tai. The Tree-to-Tree correction problem. *Journal of the ACM (JACM)*, 26:422–433, July 1979. ACM ID: 322143. `doi:10.1145/322139.322143`. 6

[25] Sun Wu, Udi Manber, Eugene W. Myers, and Webb Miller. An o (NP) sequence comparison algorithm. *Information Processing Letters*, 35(6):317–323, 1990. `doi:10.1016/0020-0190(90)90035-V`. 50

[26] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262, December 1989. `doi:10.1137/0218082`. 8