Formalisation of version control with an emphasis on tree-structured data

Sjoerd Tieleman

August 5, 2006

First supervisor: Vincent van Oostrom

Contents

Preface 3

1 Introduction 4

- 1.1 Relation to Cognitive Artificial Intelligence 5

2 Formal properties 7

- 2.1 Preliminaries 7
- 2.2 Three-way merging and residual systems 9

3 On text 16

- 3.1 Subversion 16
- 3.2 Darcs 28
- 3.3 Miscellaneous notes 33

4 On trees 35

- 4.1 Operations on trees 38
- 4.2 Tree tools 40
  - 4.2.1 3DM introduction 41
  - 4.2.2 Tree matching 47
  - 4.2.3 Chawathe's matcher 49
  - 4.2.4 Edit script disambiguation 53

5 Conclusions & future research 57

- 5.1 Caveats 57
- 5.2 Future research 59

References 60

Preface

I would like to say thank you to the following persons who have in some way contributed to this thesis.

Irene Conradie, for her continued support and love, even at times when I was not too sure about this thesis myself. Thank you for giving it a final read and for your immaculate grammar checking. My parents Herbert and Ineke, for their faith in me even when it all took a bit longer than expected. My supervisors Vincent van Oostrom, Jeroen Scheerder and Albert Visser. Everyone at the "Studielandschap" who was going through the same thing for the past through months (you know who you are!). Xander Schrijen, for being a generally smart person and for pointing me towards Darcs. My (former) colleagues at the IT department, for a mostly great time these past few years. Besides these people, there have been countless others who gave me pieces of advice, went out for beers with me and in general gave me a brilliant time at the University Utrecht. Thank you very much and enjoy this thesis!

This thesis was typeset using the (generally) excellent XeTeX[1] software by Jonathan Kew.

[1] http://scripts.sil.org/xetex/

1 Introduction

As Lindholm put it so aptly in the opening words of his thesis [Lin01]: "Keeping data up-to-date across a variety of devices and environments is becoming more and more important". We are seeing a sharp increase in the use of mobile devices. These are small, portable devices which allow us to communicate and be productive in places (such as public transportation) in which the possibilities were until recently very limited.[2] However, with the coming of these new devices over the last few years, new questions have risen. For example, how do we make sure that the address book on every device is up-to-date? Or that all our scheduled appointments are available on all of our devices? Moreover, how do we propagate changes made on one device to another? How can we allow two different persons to collaborate on a shared document? Most of these issues are of considerable interest and seem to centre on a single topic: the sharing and updating of data. Thus, the central question of this thesis will be:

> How can a formal model of version control on structured data be constructed?

The research question will be answered by taking the followings steps. Firstly, a model will be presented believed to represent version control for arbitrary data structures; later on, the focus will shift towards ordered lists of lines (for version control on texts) and ordered trees (for version control on structured data, such as XML), so a good abstract description will be applicable to both. Secondly, it will be shown that the model holds for version control as it is implemented in several modern VCS[3]s (as operations on ordered lists of lines). Thirdly, the model will be applied to ordered tree structures (moving from ordered lists of lines to ordered trees, which form the basis of the syntax of XML, iCal, LaTeX and other tree-like structures). Finally, the conclusions and the work that still has to be done will be presented.

One of the topics that will be studied is a process known as three-way merging: two data structures that are derived from a single structure can be merged in such a way that they become a single, unified new structure. The three-way merge usually occurs within the context of VCSs. For example, assume that two people are working on a single document. Each one of them can make a copy of this document and make some modifications to it. These two new documents may need to be merged in such a way that the modifications made by both persons are preserved.

[2] One might argue that the ability to be 'productive' anywhere is not A Good Thing™.

[3] Version Control System

As we will see, traditional VCSs operate on ordered lists of lines (texts). They are able to operate on the text representation of other structures (such as trees), but operating directly on the structure instead of the text representation of the structure allows one to extract more information. It is much more meaningful to be able to say that a subtree was moved from position a to position b in a tree, than it is to say that a bunch of text lines was deleted and that some other lines were inserted at a different position. This is more meaningful for three reasons:

- The description that can be given of some modification is more closely related to what actually happens. For example, suppose we are editing some LaTeX document and decide to move an entire chapter with paragraphs, images, tables and all other content to a different position in the document. One would like to say that we have *moved* the chapter (and all of its content) in one single action, instead of saying that we *deleted* the chapter and all of its content, and *inserted* a new chapter somewhere else (which incidentally has the same content as the chapter we deleted).
- It allows us to avoid certain problems that arise when we are performing three-way merging on the text representation (as we will see later on in Section 4).
- The *context*[4] of a node might be used to extract extra information to provide meaningful help messages or better user interfaces.

1.1 Relation to Cognitive Artificial Intelligence

This work is primarily situated in the field of *ubiquitous* computing. The term was first coined by Mark Weiser [Wei91] and covers not only the fact that devices get more and more portable (e.g. taking your computer with you), but also the fact that it is desirable that applications and data can move freely between devices (e.g. having all your stuff available on all your devices). In my view ubiquitous computing is a viable part of Cognitive Artificial Intelligence as it is a great challenge to provide people with a consistent, easy-to-use environment. Not only does it require knowledge of computer science, it also touches upon subjects such as usability, communications and logic. The topics presented in this thesis are meant as low-level foundations for tools that can be used to facilitate work on shared documents. These foundations allow more meaningful information to be extracted and presented to the user, since preserving structure when modifying data allows one to make use of the information

[4] Informally: any relevant information such as its content, its siblings, its children.

in the structure. The design and implementation of these high-level tools will be left out of this thesis; but it is the work that is presented *in* this thesis that allows more "cognitive" tools to be developed.

The model presented in this thesis allows one to construct user interfaces that are more intuitive to understand, like a text processor that can understand when an entire chapter has been moved to a different location in a text and give meaningful feedback about it to the end user. It can also be useful in a situation where multiple persons are working on a single document and conflict situations arise. Tools that can provide meaningful explanations and suggest conflict resolutions can contribute to a more productive work environment.

Before one can concentrate on the higher level tools that enable all this, one has to start at the bottom and work one's way up and that is what this thesis focuses on: working from the bottom up to provide solid ground for the aformentioned tools to be built upon. Therefore, the first step will be to introduce a model and take it from there.

2 Formal properties

In this section a formal description of version control on a single data structure will be introduced. An informal definition of what is meant with "version control" will be given. By version control one can think of two things, roughly split up in a "version" part and a "control" part, respectively:

- Retaining all the changes that were made to some data structure allows us to see the entire history of this data structure from the moment it was created to the moment the last update took place.
- Facilitating the collaborative work on this data structure allows multiple persons (or computer processes) to modify this data structure concurrently without any locking issues[5] or other constraints.

This is a generalised notion of version control, but these are the two subjects that are the focus of interest here. Two very important[6] concepts in version control are:

Differencing The process of obtaining the *difference* between two data structures, also known as obtaining a transition or step to transform one data structure into another.

Merging The process of combining two data structures to create a new one, while retaining as much information from both as possible.

2.1 Preliminaries

Version control will be modelled using an abstract rewriting system (ARS) [Ter03] because an ARS is an elegant, abstract system that can be used to describe changes in objects. By "abstract" it is meant that it is not defined on what kind of data structures the model operates. Defining this model in such a way allows it to be used on a wide variety of data structures, because it is not confined to certain structures. Using it in real-life situations requires an implementation, and it will be this implementation that determines on what kind of data structures the model can operate. For now, an ARS is defined as a quadruple $\langle A, \Phi, \mathsf{src}, \mathsf{tgt} \rangle$, where:

[5] Locking issues occur when some process *locks* the data structure while it is modifying it, such that no other process can modify it at the same time.

[6] They are important in the sense that *differencing* allows us to generate a transition from one object to another and *merging* allows concurrent modifications to an object to be reconsolidated in a new object. *Differencing* has importance in the "version" part, as it allows us to construct transitions between objects, *merging* has importance in the "control" part as it facilitates concurrent work on an object.

- A is a set of objects, denoted $\{a, b, \dots\}$;
- Φ is a set of steps denoted $\{\phi, \psi, \dots\}$;
- src and tgt are the *source* and *target* functions respectively, mapping steps to objects.

For any step $\phi : a \to b$ the following holds: $\mathsf{src}(\phi) = a$ and $\mathsf{tgt}(\phi) = b$. This can be seen as a simplified version control system by assuming that the objects are the different versions of a file. The steps are the *changesets* leading from one file to another; a *changeset* being defined as an ordered structure of actions to be performed on its *source* object to construct the *target* object. For example, let $\mathcal{A}$ be an ARS as introduced above. Suppose there is some original file a that is put under version control. This file is added to a *repository* in its initial state. Now one can make some changes to this file. At a certain point, one may decide that this new version (let's call it b) is finished, and *commit* it to the repository. This new version gets stored in the repository, and so does the step $\phi : a \to b$ which is calculated using objects a and b.[7] This means that in its initial state (after adding a to the repository), the ARS looks like:

$$\langle \{a\}, \emptyset \rangle$$

For brevity the src and tgt functions are omitted; $A = \{a\}, \Phi = \emptyset$. After the commit of b the ARS will look like:

$$\langle \{a, b\}, \{\phi : a \to b\} \rangle$$

This means that there now exists a mechanism to perform version control on an object: starting with an initial object and using steps this results in updated versions of this object. However, this is not sufficient for complete version control. Version control also has the property of being able to move backwards from an updated version as far back as the initial version. Suppose there exist two objects a, b and the step ϕ with $\mathsf{src}(\phi) = a$ and $\mathsf{tgt}(\phi) = b$. From this step, one would also like to be able to calculate the *converse* step ψ with $\mathsf{src}(\psi) = b, \mathsf{tgt}(\psi) = a$, which allows the transition from b to a.[8]

[7] For now it is not specified *how* this step gets calculated, this will be explained later on.

[8] This may seem redundant because a and b are both in the repository at this time. However, we will see that VCSs almost never store all the objects in the repository, but use a sequence of steps to obtain objects.

2.2 Three-way merging and residual systems

Important in version control is the three-way merge, a combination of differencing and merging. Three-way merging in version control occurs when there are two modified versions of a single file; for example, when two persons are working on the same file. Both may have started from the same file and may have made modifications to it. At some point these files need to be reconsolidated and preferably this should be done in an automated fashion.[9]

Three-way merging in version control (on a single data structure, such as a single file) will be defined in the context of a residual system with composition [Ter03]. A residual system, which is an extension of an ARS, requires that for any pair of co-initial steps from a single state, the residual of the first after the second and the residual of the second after the first have the same target. This can be illustrated by using a picture (see Figure 1).

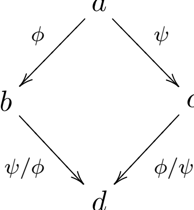


Figure 1: Simple residual system

This picture looks attractive to our notion of three-way merging, because there exist three objects a, b, c and by combining these three objects (and their steps) we can obtain a new object d which is a result of both ϕ and ψ.

An abstract residual system with composition is defined as $\langle \mathcal{A}, 1, /, \cdot \rangle$, where:

- $\mathcal{A}$ is an ARS;
- 1 is a function from objects to steps;
- $/$ is a function from pairs of co-initial steps to steps;
- $\cdot$ is a function from steps to steps.

For every object a, its *trivial* step $1_a : a \to a$ exists, and for every pair of composable steps $\phi : a \to b, \psi : b \to c$, their *composition* $\phi \cdot \psi : a \to c$ exists. Two steps ϕ, ψ are

[9] Recall the "control" part in version control.

composable iff $\mathsf{tgt}(\phi) = \mathsf{src}(\psi)$. The composition of two steps $\phi \cdot \psi$ is a new step in Φ. In some cases it can be reduced to a simpler new step δ where:

$$\begin{aligned} \mathsf{src}(\delta) &= \mathsf{src}(\phi) \\ \mathsf{tgt}(\delta) &= \mathsf{tgt}(\psi) \end{aligned}$$

It should be noted that by performing this reduction, it becomes impossible to track which part of δ came from ϕ and which from ψ. To retain this information one simply should not perform the reduction.[10] Let us assume that this reduction can be performed in some cases, because it is dependent on the implementation whether this is actually the case.[11] However, it should be borne in mind that this may be used later on. For example, suppose there is some step ϕ that modifies a set of lines in some text document and suppose there is have some other step ψ that modifies a different set of lines in the resulting document from step ϕ, then these two steps can be combined into one step that modifies both sets of lines. It should be noted that it cannot be determined anymore from which step which change came, as both steps are now combined into one.

ϕ/ψ can be read intuitively as ϕ *after* ψ, it is a step consisting of that part of ϕ which remains to be done after performing ψ.

Residual systems are required to satisfy certain identities as listed in Table 1, the first four identities are relevant for all residual systems, the last three only apply to residual systems with composition.

The first identity is the most important property concerning residual systems, it is known as the *cube identity*, or *cube law* and states that different paths can be equivalent. See, for example, Figure 2. Step δ can be written in two ways, depending on which path has preceded it; first taking the step $\psi : a \to b$, or first taking the step $\chi : a \to d$. For good measure, ψ and χ are said to be *co-initial*. Likewise, the steps ψ/χ and χ/ψ are *cofinal*. δ can be written as $(\phi/\psi)/(\chi/\psi)$ or $(\phi/\chi)/(\psi/\chi)$. It should be noted that this does not only hold for the paths we have drawn in Figure 2 (using the double arrows), but for any of the final steps with *target* $\bullet$. From this we may conclude that for any three co-initial steps, the order in which they are traversed is not relevant. This means for version control that the order in which any three versions are committed is not relevant for the end result of a merge between these versions.

[10] This reduction can be of use in situations where storage space is limited and one is striving for a space-efficient solution.

[11] Nowhere in residual systems is it said that this reduction can actually be applied, so we leave it to our implementation to take care of that.

$$
\begin{array}{rcl}
(\phi/\psi)/(\chi/\psi) & = & (\phi/\chi)/(\psi/\chi) \\
\phi/\phi & = & 1 \\
\phi/1 & = & \phi \\
1/\phi & = & 1 \\
\hline
1 \cdot 1 & = & 1 \\
\chi/(\phi \cdot \psi) & = & (\chi/\phi)/\psi \\
(\phi \cdot \psi)/\chi & = & ((\phi/\chi) \cdot (\psi/(\chi/\phi)))
\end{array}
$$

Table 1: Residual identities

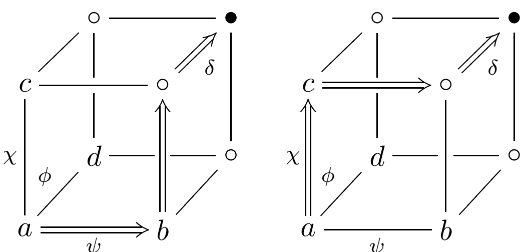


Figure 2: Cube identity in effect, δ can be written in two ways depending on the path preceding it.

The remaining three identities can be understood fairly easily:

- The residual 'ϕ after ϕ' is equivalant to the *trivial* step 1 (second identity); performing that part of ϕ that is not in ϕ means that we do not have to perform anything at all. Hence it is called the *trivial* step.
- The residual 'ϕ after the *trivial* step' is equivalent to ϕ (third identity); performing that part of ϕ that is not in 1 means that one has to perform ϕ in its completeness, since 1 is inherently an empty step.
- The residual of 'the *trivial* step after ϕ' simply is equivalent to the *trivial* step; the *trivial* step is empty, and therefore one does not have to perform anything at all.

Still, before being able to perform a three-way merge in a residual system one require the ability to calculate the *difference* between two objects (as the objects may be present, but not necessarily the corresponding steps). Therefore, a new kind of system is introduced: an abstract version control system which is defined as $\langle \mathcal{A}^{\mathcal{R}}, \overline{}, - \rangle$, where:

- $\mathcal{A}^{\mathcal{R}}$ is an abstract residual system;
- $\overline{}$ is the *converse* function from steps to steps;
- $-$ is a binary function from objects to steps called *difference*.

This is an extension on residual systems, because we add the *converse* and the *difference* function. The converse function allows one to calculate a converse step from a step. Also, the converse of the converse step is the step itself. For example, if ϕ is a step, then $\overline{\phi}$ is its converse, where $\mathsf{src}(\overline{\phi}) = \mathsf{tgt}(\phi)$ and $\mathsf{tgt}(\overline{\phi}) = \mathsf{src}(\phi)$. This automatically means that they are *composable* and therefore the following holds:

$$\mathsf{src}(\phi \cdot \overline{\phi}) = \mathsf{src}(\phi) = \mathsf{tgt}(\overline{\phi}) = \mathsf{tgt}(\phi \cdot \overline{\phi})$$

It should be noted that this composition is not equal to $1_{\mathsf{src}(\phi)}$[12], but it is equivalent in its source and target, meaning that it is a step from an object to itself.[13]

The *difference* function is required to be able to deduce or calculate the step from one object to another. The function is such that for any two objects $a, b \in A$, it

[12] After all, we *are* performing ϕ, followed by $\overline{\phi}$.

[13] The reader should note that not all details concerning the *converse* and *difference* functions have been studied fully, as they are beyond the scope of this thesis. Therefore, it is advisable to do a more thorough study on the interactions between the various operators in the future.

can derive the step $a \rightarrow b$. This can be represented as $(b - a)$. This system also introduces a new identity, listed in table 2. The *difference* between a and b is equal to the *converse* of the *difference* between b and a.

$$(a - b) \quad = \quad \overline{(b - a)}$$

Table 2: Version control identity

The system makes it possible to have the possibility to move back and forth between objects using the converse function. The *difference* function is available, which allows one to calculate steps between objects and the *converse* function is available, which allows one to calculate a step $b \rightarrow a$ from a step $a \rightarrow b$.[14] Now the three-way merge can be described in terms of our operators. The three-way merge is a ternary function from objects to steps (see also Figure 3, where the dashed arrow is the resulting (reduced) step):

$$b \sqcup_a c =^{\text{def}} (b - a) \cdot ((c - a)/(b - a))$$

The alternative (*projection equivalent*) step $(c - a) \cdot ((b - a)/(c - a))$ could have been taken as well, since three-way merging was defined as a residual system.

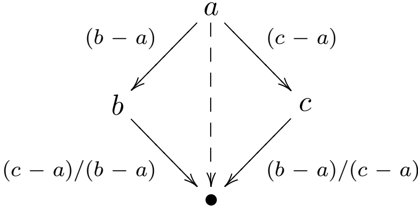


Figure 3: Three-way merge

$(b - a) \cdot ((c - a)/(b - a))$ is of the form $\phi \cdot (\psi/\phi)$. According to the laws of residual systems [Ter03] this is by definition equivalent to $\phi \sqcup \psi$, where $\sqcup$ stands for *designated join* and should not be confused with the three-way merge, which is a function from objects to steps, whereas the *designated join* is a function from pairs of co-initial steps to steps. Thus, the three-way merge can also be written as:

[14] One might wonder why the converse step is relevant at all. After all, it was stated that one can use the difference operator to determine the step between any two objects. This matter will be addressed in the next section, as it turns out this will come in handy if one doesn't have all the objects available.

$$b \sqcup_a c =^{\text{def}} (b - a) \sqcup (c - a)$$

Perhaps this is a more "natural" notion, as this states that the resulting step is the *join* of the changes between a and b and a and c, which is intuitively correct.

The aforementioned *projection equivalence* ($\simeq$) and *projection order* ($\lesssim$) relations for any pair of co-initial steps ϕ, ψ are defined as:

$$\begin{aligned} \phi \lesssim \psi & \quad \text{if } \phi/\psi = 1 \\ \phi \simeq \psi & \quad \text{if } \phi \lesssim \psi \text{ and } \psi \lesssim \phi \end{aligned}$$

Projection order can be read intuitively as: "ϕ is completely contained in ψ". In the case of version control, this means that one set of modifications to a file is completely contained in the other. *Projection equivalence* therefore means that both modifications are completely contained in each other and are thus equivalent, but not necessarily identical.[15] Again, according to the laws for residual systems the following holds:

$$\begin{aligned} \phi \sqcup \psi & \simeq \psi \sqcup \phi \\ (\phi \sqcup \psi) \sqcup \chi & \simeq \phi \sqcup (\psi \sqcup \chi) \end{aligned}$$

Having defined our three-way merge of the form $\phi \sqcup \psi$, it may therefore be concluded that:

$$\begin{aligned} b \sqcup_a c & \simeq c \sqcup_a b \\ (b \sqcup_a c) \sqcup_a d & \simeq b \sqcup_a (c \sqcup_a d) \end{aligned}$$

This presents us with two properties for our system:

1. Owing to the commutativity of the three-way merge the order in which two versions are committed does not matter for the end result;

2. Owing to the associativity of the three-way merge the order in which multiple (two or more) merges are applied does not matter for the end result.

In other words, the order in which two or more users commit their versions is not relevant.[16] It should be noted that we are only stating this for the cases where there is automatic merging possible. If automatic merging is not possible due to overlapping

[15] The main point is that they contain the same modifications, but not necessarily in the same order.
[16] This point was also noted when explaining the *cube identity*.

steps, then it is *assumed* that these properties still hold, even though manual, human, intervention is required. This intervention typically involves a social process in which the committers all need to agree on which merge is the correct one.[17]

[17] Yes, *all* of them should agree.

3 On text

The purpose of this section is to show that the model as introduced in section 2 actually applies to the current version control systems such as CVS, Subversion, Darcs, and so on. The ability of these VCSs to operate on texts will be discussed. I will argue that the version control system introduced in section 2.2 is sufficient for this purpose.

3.1 Subversion

Subversion [CSFP06] and CVS[18] are fairly similar in the way they operate. For instance, they are similar in the way they treat files (per default) as ordered lists of lines and similar in the way they represent *changesets*[19]. Subversion will be the main focus here, because it is designed as a replacement for CVS, comprehending the basic functionality CVS has to offer as well as extending it and being more modular in design. Following the lines of the previous section, it may be said that:

- A file can be seen as an object in a rewriting system. In this thesis text files will be discussed[20], which are represented as ordered lists of lines;
- The difference between two versions of a file, a and b can be represented as a step $a \rightarrow b$;
- A three-way merge of three versions of a file, a, b and c leads to a new file which incorporates all changes;
- Only *delete* and *insert* operations are available. Line modifications, for example, changing a few words, are represented as a line deletion and insertion. The same holds for line moves, if a line is moved from position 4 to position 10, then this is represented as a deletion at position 4 and an insertion at position 10.

How does this work in actual practice? Typically, a user creates a *repository* in a central place. This repository will hold all of the files. When a file a gets created (possibly on another machine, but not necessarily), it optionally receives some initial modifications and is added to the repository. It is this last step, adding the file to the repository, that is of interest. The creation of a repository or the creation of a

[18] Concurrent Versions System

[19] Changesets are comparable to steps.

[20] As opposed to arbitrary data structures, which VCSs can also handle, but not so well. In fact, this thesis is all about the ability of VCSs to be able to operate on more complex data structures.

file is not of particular interest. The file is added to the repository by transmitting it entirely to the repository. As this is the first time that something is added, one can't help but send over the entire data structure to the repository. Before the user can take advantage of all the features that version control offers, the file first has to be *checked-out* of the repository. This creates a *working copy* on the users machine. The reader should note that the *working copy* and the *repository* need not be on the same physical machine (but they may). The *working copy* is said to be *local* in the sense that it is on the users machine. The *repository* is commonly referred to as being *remote*. After checking-out, some changes may be made to the file, leading to local modifications that are not present in the repository. If the user would *commit* these changes to the repository, then the following sequence of actions would be performed:

- The modified file is compared to a locally stored file (in the `.svn` directory of the working copy which contains various metadata, including a pristine, unaltered copy of the file);
- If there are local modifications (and there are in this case) the difference between the modified version (b) and the original version (a) is computed (using some differencing algorithm, our $-$ function) as $b \rightarrow a$;
- This difference is then sent to the repository where it is stored.

The reader should note that not $a \rightarrow b$ is used here, but $b \rightarrow a$. The step required to go from b to a is known as a reversed step (or the *converse* step). For performance reasons the new version b is stored full-text in the repository with a reversed changeset $b \rightarrow a$ to obtain older versions, because most development on a file is assumed to occur in the most recent versions. This guarantees that the newest version can be extracted very fast, but older versions can be extracted in near-linear time (linear in the number of versions you are going back). In the terms used before: b would be stored, alongside a $b \rightarrow a$. Combining b with $b \rightarrow a$ gives us a again. For now, it is claimed that $b \rightarrow a$ can be constructed from $a \rightarrow b$, but this claim will be further developed later on.

For example, suppose there exists some file "test" containing three lines (aptly named "First line", "Second line" and "Third line"). Deletion of the second line results in a changeset that resembles the following:

```
Index: test
===================================================================
--- test	(revision 1)
+++ test	(revision 2)
@@ -1,3 +1,2 @@
 First line;
-Second line;
 Third line;
```

Similarly, the reverse changeset, obtained by running `diff` with its input files swapped, is[21]:

```
Index: test
===================================================================
--- test	(revision 2)
+++ test	(revision 1)
@@ -1,2 +1,3 @@
 First line;
+Second line;
 Third line;
```

It should be noted that this is the actual output that Subversion generates if you issue a *diff* command to determine the differences. Enough information is preserved to be able to reconstruct all pieces of a file from a single version, be it the oldest or the newest, plus its changesets. A slight elaboration on the syntax is in order. The first four lines of the diff are primarily for display purposes. It contains some relevant information, namely the name of the file ("test") and the two revisions that the *diff* is calculated for (revision 1 and 2). The "---" and the "+++" indicate which revision is the *source* ("---") and which is the *target* ("+++"). The fifth line indicates which portions (or *hunks*) in the files are modified. The fifth line of the second changeset can be read as: in revision 2 lines 1 through 2 are considered modified and in revision 1 lines 1 through 3 are considered modified. This information (which lines are considered modified) is not used for conflict detection/resolution, which occurs in a different phase (more on that later).

The three-way merge in Subversion is accomplished as follows: suppose a user makes some changes to a file that was obtained (checked-out) from the repository. Before it is committed back to the repository, somebody else already made some

[21]There exists no software tool or argument for `diff` to calculate the reverse diff from the original output, but a tool that can parse the output and return a reverse diff should not be hard to write.

other (disjoint) changes to the file and committed it to the repository. As the users attempts to commit the changes, a notification comes from the repository that the version that the modifications were made in is outdated and the local copy should be updated. With disjoint changes we mean that both versions contain modifications on different lines, i.e. no overlapping edits. This is the standard case, we will see a counter-example later on.[22] At this point the three-way merge takes place on the client, not on the machine that hosts the repository. So, the user has three versions at his disposal: the version that's the most current version in the repository (*rep*), the version that he has made his changes to (the *working copy*: *wc*) and the version that both versions originated from (*org*). Three-way merging can then be expressed as:

$$\text{wc} \sqcup_{\text{org}} \text{rep}$$

The graphical representation can be seen in Figure 4.

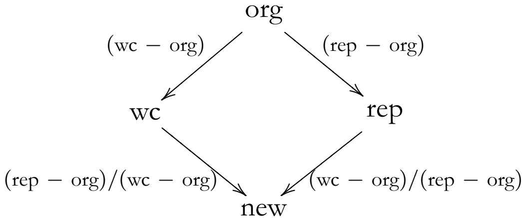


Figure 4: Three-way merge in Subversion

As can be seen, the end result is a new version "new". This new version is not automatically committed to the repository, but remains in the users working copy. The user can choose to inspect it (to verify that all changes have been correctly incorporated), possibly modify it and commit it to the repository.

The only changesets that are used here are the changesets $(\text{wc} - \text{org})$ and $(\text{rep} - \text{org})$. Is has not yet been specified how they were obtained. In fact, a client application may choose to record all changes to a file as they are made. This means that the changeset does not have to be computed at commit-time, but is already available for use. However, the standard Subversion client does not do this as it requires support in the text editor one uses. So, the standard Subversion client simply uses a differencing algorithm to obtain this changeset known as *xdelta*.

[22] This notion of overlap probably has some resemblance to the notion of *critical pairs* in term rewriting systems [Ter03]. In fact, version control on texts could probably be represented as a string rewriting system. This direction will not be pursued, but it may be interesting.

What remains to be defined is what the / does in this case. Recall the definition from the previous section: / is the residual operator and ϕ/ψ can be read as ϕ after ψ. So, at some point ψ has been performed and next ϕ is performed minus any steps that have been taken by ψ. In the terminology of Subversion this means that $(\text{rep} - \text{org})/(\text{wc} - \text{org})$ can be read as performing that part of $(\text{rep} - \text{org})$ that is left after $(\text{wc} - \text{org})$. The reader should recall that both versions of the file (the working copy and the one that's currently in the repository) have disjoint modifications. Therefore, in this case $(\text{rep} - \text{org})/(\text{wc} - \text{org})$ is equivalent to $(\text{rep} - \text{org})$ (and likewise for the complementing step). However, difficulties may arise if this is claimed, because the sources and targets for composition are now off. We cannot say that they are truly equivalent. Recall Figure 4 which implies the following:

$$\begin{aligned}
\mathsf{src}((\text{rep} - \text{org})/(\text{wc} - \text{org})) &= \text{wc} \\
\mathsf{tgt}((\text{rep} - \text{org})/(\text{wc} - \text{org})) &= \text{new} \\
\mathsf{src}((\text{rep} - \text{org})) &= \text{org} \\
\mathsf{tgt}((\text{rep} - \text{org})) &= \text{rep}
\end{aligned}$$

These steps are however almost equivalent in terms of specifying which actions need to be taken to get from “wc” to “new”. For example, suppose that in $(\text{wc} - \text{org})$ some lines (between lines 3 and 4) were inserted into the data structure. That means that $(\text{rep} - \text{org})$ might not be able to be applied directly if it is located after these lines, as it could have the wrong line numbers associated with it. Therefore, the step requires some *post-processing* and one could say that the residual operator is exactly *that* post-processing.

As said, in some cases, however, it is possible that there are overlapping edits. Some of these can be automatically applied, for example, suppose a users creates some file consisting of the following lines:

```
First line;
Second line;
Third line;
```

He commits this file to the repository. Someone else checks out this file and makes some modifications to it, resulting in a new version:

```
First line;
Second line, with edits;
Third line;
Fourth line;
```

This version is committed to the repository. Meanwhile, the user has made a modification, which yields in the following version:

```
First line;
Second line, with edits;
Third line;
```

That is, he has made the same edit to the second line, but has omitted adding the fourth line. When he tries to commit this, he gets the "out of date" message and is required to update his local version. Subversion is smart enough to notice that even though there are overlapping edits (the second line), both edits are the same and automatic three-way merging occurs, yielding in a version where both the edit of the second line and the adding of the fourth line are present. He is then free to commit this version.

Alternatively, suppose the repository contains the original file as listed previously (which consists of three lines). Now, again assume that two people create their own versions:

```
First line;
Second line; (modified)
Third line;
```

```
First line;
Second (inserted) line;
Third line;
```

These versions cannot be merged automatically, as both persons have made different edits to the second line. This is where *granularity* (also) comes into play. For Subversion, the granularity is at the level of individual lines. That means that the smallest modification it can observe is a line modification. It does not matter if only one letter is changed in the line, or the whole line is modified, it will only notice that the

line has been modified. The granularity could be increased to for example the level of individual words (white-space separated entities). But for most general purposes, granularity at the level of individual lines is sufficient. In this example one would be able to make use of an increased level of granularity as modifications were made to different parts of the line, so they could be consolidated into a single new version:

```
First line;
Second (inserted) line; (modified)
Third line;
```

So far, we have seen how the system introduced in the previous section can be mapped onto Subversion. However, one problem remains, that is how to define the semantics of the "/" operator, which can be roughly translated to the question: which three-way merges can be performed automatically and how are they performed?

- ϕ/ψ can be performed automatically, if:
  - There is no overlap between ϕ and ψ.
    This cannot be expressed formally as there is no way to correctly represent overlap between ϕ and ψ. For text it suffices to say that each step modifies its own distinct part of the file: ϕ modifies some set of lines (a continuous block of text), ψ modifies some set of lines, with $\phi \cap \psi = \emptyset$.
  - $\phi \simeq \psi$
    If ϕ and ψ are equal at least up to *projection equivalence*, then the same edits have been made, and there exists no conflict.
  - $\phi \lesssim \psi$ or $\psi \lesssim \phi$
    Recall that $\phi \lesssim \psi$ if $\phi/\psi = 1$. This means that every change made in ϕ is also in ψ. Therefore, no conflict exists. Likewise for $\psi \lesssim \phi$.
- ϕ/ψ cannot be performed automatically in all other cases.

Combining all this, we can start to model Subversion using the systems introduced in the previous section. Let $\mathcal{A}$ be a rewriting system $\langle A, \Phi, \mathsf{src}, \text{tgt} \rangle$ such that:

- A is a set of key-value pairs $(x : a)$ where x is some natural number $x > 0$ and a is an ordered list of text lines. x will be known as the *revision number* and can

be used to identify a version;[23]

- Φ is a set of tuples (ϕ, x, y) where ϕ is …and x, y are two natural numbers $x, y > 0$ pointing to versions in A, not unlike *keys* in a database or *pointers* in programming. x, y are the *src* and *tgt* of the step respectively;
- src and tgt are two functions $\Phi \rightarrow A$, able to extract the source and target objects from a step by *dereferencing* (looking up) the labels x, y in a step.

It is important to bear in mind that rewriting systems implicitly have a way of combining an object with a step to get a new object. The presented system requires some mechanism to do this and that mechanism will be the traditional UNIX `patch` tool. It takes an object and a step as its arguments and returns the *patched* object.

Let $\mathcal{A}^{\mathcal{R}}$ be a residual system with composition $\langle \mathcal{A}, 1, /, \cdot \rangle$ such that:

- $\mathcal{A}$ is the rewriting system that was just created;
- 1 is the trivial (empty) step;
- $/$ is the residual operator for any pair of co-initial steps $\Phi \times \Phi \rightarrow \Phi$;
- $\cdot$ is the composition operator for any pair of composable steps $\Phi \times \Phi \rightarrow \Phi$.

This extension allows one to speak of empty steps, the use of the residual operator (which we will use in modelling the three-way merge) and the composition of steps (more on that later).

Let $\mathcal{A}^{\mathcal{VC}}$ be a version control system $\langle \mathcal{A}^{\mathcal{R}}, -, ^{-} \rangle$ such that:

- $\mathcal{A}^{\mathcal{R}}$ is the residual system introduced above;
- $-$ is the *difference* operator $A \times A \rightarrow \Phi$, for convenience we could just use the UNIX *diff* tool for this;[24]
- $^{-}$ is the *converse* operator $\Phi \rightarrow \Phi$.

[23]Caveat: this only applies to the situation where we have one file in our repository. If we are adding more files, then we need to add these to the existing rewriting system, or create a new rewriting system for that file. That means we have to account for interactions between these files and use a different identifier than revision numbers since revision numbers in Subversion apply to the entire repository and not in a file-by-file fashion where each file has its own revision number. CVS, however, does allow files to have its own revision number.

[24]Note that this is not the default tool Subversion uses, it uses a custom algorithm called *xdelta*. However, Subversion allows you to choose your own *diff* tool, so we can use `diff` for it.

Again, this is an extension of our system. The difference between two objects can now be calculated. This is necessary because the step may not always be available. Furthermore, the requirement that steps can be reversed can now be fulfilled.

For illustrative purposes it will be shown how a typical workflow in Subversion is handled by the formal model. The following actions will be performed:

1. Create a repository.
   This is as easy as instantiating the aformentioned model using $\mathcal{A}, \mathcal{A}^{\mathcal{R}}, \mathcal{A}^{\mathcal{VC}}$. This results in an empty repository where $A = \emptyset, \Phi = \emptyset$.

2. Import a file into the repository.
   This is done in Subversion using the `svn import` command. Suppose a file a is imported. This updates the repository to $A = \{(1 : a)\}, \Phi = \emptyset$.

3. Check-out the repository.
   One can use the `svn checkout` command. This results in a *working copy* on the client machine.

4. Modify the file.
   Use your favourite text editor.

5. Commit the file.
   This is where it gets interesting: at the moment the `svn commit` command is issues, the difference between the modified file (let's label it "b") and the pristine version "a" is calculated using $(b - a)$. This results in the step (ϕ, a, b). This step ϕ (which is shorthand for (ϕ, a, b)) is sent to the repository, where it is used to construct b from a. b is stored full-text in the repository, a is deleted and $\overline{\phi}$ (the converse) is stored in Φ. That results in the following: $A = \{(2 : b)\}, \Phi = \{\overline{(\phi, a, b)}\}$.[25]

6. Again, modify the file.
   Use your favourite text editor once more.

7. Let someone else commit a (different) modified version.
   Suppose someone committed a new version c before you were able to commit, that leaves the repository as: $A = \{(3 : c)\}, \Phi = \{\overline{(\phi, a, b)}, \overline{(\psi, b, c)}\}$.

8. Attempt to commit the modified version, resulting in a three-way merge.
   Trying to commit version d, the user gets notified that the version he started working on (b) is outdated and superceded by c. This requires that he updates

[25]The observant reader probably notices that a was deleted, but a step $\overline{(\phi, a, b)}$ is still available from which one should be able to extract the *source* and *target* objects. The *source* is not particularly hard, as that is b, which is in A. The *target* however is a which was just deleted. So, the *pointer* analogy is not entirely correct, because the pointer is still there, but the object no longer is. Of course this can be repaired by augmenting the `src` and `tgt` functions in such a way that they can apply the needed steps to the object that we *do* have in the repository to (re)construct the requested object.

his local copy. That results in a three-way merge $c \sqcup_b d$. See Figure 5. Recall that the three-way merge was defined as a concatenation of two steps, in this case one can use: $(d - b) \cdot ((c - b)/(d - b))$. Conveniently, $(d - b)$ is already available at the user's disposal, it was calculated when first trying to commit version d. What happens is that when updating d (using the residual operator) the step $(c - b)/(d - b)$ is calculated and applied to the working copy d. $(c - b)$ is available because that step can simply be retrieved from the repository. Applying the new (concatenated) step effectively leads to a new version e (assuming that the merge was applied succesfully). This new version can then be committed, and the difference $(e - c)$ gets calculated (which is equivalent to $(d - b)/(c - b)$, mind you, so this can be calculated by using $/$ or by using $(e - c)$) and added to the repository. The new repository then looks like:

$$A = \{(4 : e)\}, \Phi = \{\overline{(\phi, a, b)}, \overline{(\psi, b, c)}, \overline{(\chi, c, e)}\}$$

That indeed means the working copy d cannot be extracted anymore. It has never entered the repository, and therefore one cannot return to that state. Therefore, it is advisable that one verifies the merged version before committing after a three-way merge (if at all possible).

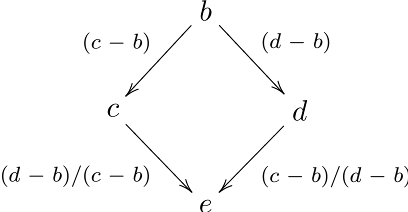


Figure 5: Three-way merge in Subversion

This example should give the reader a bit of understanding how version control actually works. As said, storing the reverse steps has the nice property that it is very fast to check-out the latest revision (linear in object size: $O(n)$ where n is the size of the object). Older versions can be checked-out in $O(n \cdot \sum m)$ time where $\sum m$ is the combined size of the steps that are required to be applied.[26]

Three things have not been explained yet: how the $/$, $\cdot$ and $^{-}$ work.

- The residual operator operates on two co-initial steps and returns a new step. Assuming the *unified diff*[27] is used for this, one can simply determine which

[26]This can probably be studied a bit better, however it is meant illustrative: no nasty complexity issues arise.

[27]The *unified diff* can be calculated using the `-u` flag on the `diff` tool. This diff is reversable, and uses a

lines have changed between the source object and the two derived objects. That results in two options:

1. There are no overlapping changes in the two derived versions;
2. There are overlapping changes in the two derived versions.

If there are no overlapping changes, then the residual operator can simply return the (reduced) concatenation of the two co-initial steps. If there are overlapping changes then there are (again) two options:

1. The overlapping edits are identical;
2. The overlapping edits are not identical.

If the edits are identical, then one can simply pick one and use that. If they are not identical, there is a conflict and one cannot continue. Having a conflict requires that you are required to manually merge the files and mark the result as *resolved* before anything else can be done with the file. Intuitively this means that the three-way merge is not finished until it has been manually resolved.[28]

- Assume there exist two *composable* steps ϕ, ψ, where *composable* means that $\mathsf{tgt}(\phi) = \mathsf{src}(\psi)$. Composition can be achieved by simply performing the first step and then the second. Reduction of two composable steps is somewhat harder, but not too hard, there are again two possibilities:

  1. The steps are non-overlapping[29];
  2. The steps are (partially) overlapping.

  In the first case, the reduction is simply a new step with its *source* the source of the first step and its *target* the target of the second step. The new step can be described by taking the *union* of the two steps, that is, a *unified diff* with its begin position the smallest begin position of any of its composable steps and with end position the biggest end position of the two steps. For example, assume we have two composable steps ϕ and ψ. ϕ has start position line 5 and end position line 8, ψ has start position line 15 and end position line 20. The resulting step will have start position 5 and end position 20.

block-like notation to highlight the changed parts of a file.

[28] Recall the remarks about manual intervention. We still assume the residual properties to hold even after manual resolution.

[29] In a slightly different way from the residual case: composable steps that have overlapping edits do not cause conflicts, they are sequential in nature, not parallel, meaning that they can always be applied in order.

In the second case, two edits may have to be combined into one. Suppose the following composable steps are available:

```
[...]
-Second line;
+Second line, modified;
[...]
```

```
[...]
-Second line, modified;
+Second line, altered;
[...]
```

These can be combined into a new reduced step that looks like:

```
[...]
-Second line;
+Second line, altered;
[...]
```

- The converse of a step can be done by switching its *source* and *target* and by taking the *unified diff*, changing the order of line deletions and insertions (deletions always precede insertions in this format) and replacing the $+$ and $-$ line flags with $-$ and $+$.[30]

In conclusion, it has been shown that the version control system Subversion can be (partially) modelled by applying the model introduced in the previous section. Of course some caveats apply. As said, we are operating only on one file here. VCSs typically operate on multiple files. A design choice can be made here. One can choose to combine all files in the same rewriting system. However, that may result in unexpected results. For example, three-way merges can only be applied to versions that originate from the same source object. There currently is no way to determine that this is indeed the case as any step can be generated using the *diff* operator, not just steps between corresponding versions. All that is stated is that there are some objects that can be combined to form a new object. More on using multiple files (not just in Subversion, but more generic) will be presented later on in the conclusion.

[30]Diff does not recognize line moves or updates. Everything is represented as a deletion and/or an insertion, hence these are the only actions that Subversion detects.

3.2 Darcs

Another interesting version control system that shall briefly be mentioned is Darcs [Rou06]. It is interesting in that it takes a slightly different approach from the traditional VCSs such as Subversion and CVS.

Roundy takes certain ideas from quantum mechanics and applies them to version control. He speaks of changes to the tree (the file tree) as patches.[31] These patches can either be primitive patches (file add/removal, directory rename, *hunk* replacement[32] within a file), or they can be composite patches. The formal model that was introduced does not make any distinction between primitive or composite changesets, but it does share some similarities with the system used by Darcs. For example, one could say that steps always represent a composite patch which may only contain a single patch, such as a line insertion.

This *theory of patches* that Darcs thrives on enables it have some "advanced" features that other VCSs do not possess, such as:

- Every patch is invertible. Moreover, this inverse can be computed from knowledge of the patch only;
- Sequential patches can be reordered, although this reordering can fail, which means the second patch is dependent on the first;
- Patches which are parallel can be merged, and the result of a set of merges is independent of the order in which the merges are performed.

A slight elaboration on these properties is in order. The first property (invertible patches) is something that is also proposed in the formal model. Steps can be converted and the converted step can be obtained from the original step alone. The second property (reordering of patches) can in fact be seen as a mechanism to signal conflicts. A reordering can and will fail due to a conflict, but also due to modifications that occur on lines that were introduced by some other patch. Roundy call this a dependency. This can mean that the two patches (or in our case changesets) contain conflicting actions or that one of the patches modifies pieces of other patches. The third property states what we try to obtain using a residual system: merging of two parallel patches can be obtained by taking any of the possible routes (applying the first and then the second, or applying the second and then the first). The result will have

[31] Comparable to how other VCSs speak of *changesets*.

[32] The modification of one or more consecutive lines in a file.

to be the same, no matter which route has been taken. Again the independence of the patches plays a role here.

Another design feature of Darcs is the fact that every check-out is in fact a new repository on the user's system which is a branch in the central repository. This allows users to take advantage of version control features inside their local repository without disrupting the central repository. In this sense Darcs is a distributed version control system. Darcs does not store reverse patches (or changesets) in its repository (as Subversion and CVS do). As noted, this may turn into a computational nightmare. Currently[33], Darcs (the program itself) consists of more than 3800 patches. An initial check-out takes considerable time, because every patch first has to be downloaded (in total more than 500MB worth of patches) and then applied. To compensate for this Darcs allows for checkpointing along the way. So, at given points one may choose to create a snapshot of the files and store these in the repository. After that, upon check-out (and provided one enters the correct command-line switch) one will get the latest snapshot with any patches that may have been applied later. There is a drawback: one can't go back further in patches then the checkpoint; it is only a partial repository. However, one can *pull* additional patches from the central repository if needed.

Parallel patches in Darcs have a striking resemblance to our residual system. Darcs' definition of merging two parallel patches to a single patch is:

$$P_2||P_1 \Rightarrow {P_2}'P_1 \longleftrightarrow {P_1}'P_2$$

This reads as that two parallel patches P_1 and P_2 can be merged to a patch ${P_2}'P_1$ which commutes with ${P_1}'P_2$. The resemblance here is that in our model we can have two changesets derived from a single source; performing a three-way merge results in a new changeset which incorporates both changesets and is also independent in applying the first after the second or vice versa. In that sense, they also commute.

Furthermore, Darcs has the property that a merge of two parallel patches cannot fail. This seems rather odd, for there are merges that cannot be done automatically (i.e. when we are in conflict). Darcs remedies this by constructing a special kind of patch called a *merger* which still satisfies the commutation property and tries to resolve this. In the case where there is no automatic resolution possible, Darcs uses a CVS/Subversion like mechanism by inserting conflict markers (more on this later) and requiring the user to resolve the conflict manually. This is actually quite close to the assumption that conflicting merges are still a residual system, even if it requires

[33]As checked on june 13, 2006.

manual merging. Recall that the assumption is made that if there is a conflict, all committers should agree on a resolution. If they all agree then the one can say that it is still a residual system.

Darcs has a slightly different view on conflicts than Subversion. For instance: in Subversion it is possible that a line has been modified in both versions of a file. However, if that change is identical in both versions, then there is no conflict and there can be automatic resolution. Darcs does not have this functionality. Instead, it ignores the parts that are in conflict. Other parts that may be included in the patch are performed as usual, only the conflicting parts[34] are ignored. A conflict can be resolved by introducing a new patch which has a dependency on both the conflicting patches.

Dependency was already briefly introduced, but it deserves some more explanation as it is an intrinsic part of Darcs. Normally, patches are independent of one another. This means that they operate on separate parts of a file or on separate files. Independent patches are very much preferred, because they have the commutation property by definition. This also means that Darcs has no strict notion of ordering (if all patches are independent). This is very much in contrast to "traditional" VCSs, such as Subversion or CVS which induce a strict ordering on the files and the repository. CVS does this on a per-file basis (each file has its own version number which gets increased everytime a change to it is committed to the repository), Subversion does this on a repository basis (each file does not have its own version number, but it is the state of the entire repository that is recorded each commit). However, it is possible (and useful) to obtain versions in Darcs. This is done by grouping a set of patches into a *tag*. For example, if we were to release a software product which is maintained in a Darcs repository, we may want to ship only a subset of the patches because we are currently working on a new feature which is not yet fully implemented/tested.

Dependent patches introduce an ordering of patches. This ordering only holds for the patches that are dependent on eachother. Dependency of patches in Darcs can be seen as a directed graph. We'll restrict ourselves for the moment to *hunk* patches where some text lines are changed in a document. The first patch that enters the repository is often the initial state of a file. After that additional patches may be made that apply to this file. For example, suppose we add a file to the repository that has ten lines of text. Now we add to new patches, one that modified line 5 and one that modifies line 8. These patches are independent of each other, but are dependent on the first patch, as they modify something that was added by the first patch. We

[34] Most likely *hunk* replacements.

can visualize this by making the first patch the initial node of a graph, with edges towards the other two patches. Because the last two patches are not dependent on each other they do not have an edge between them. Now suppose we add a new patch that changes lines 3 through 6. This means that this patch is not only dependent on the first patch, but also on the patch that modified line 5. After that we may decide that line 5 needs another change. This new patch is only dependent on the previous patch that modified lines 3 through 6. See Figure 6 for a visual representation of the dependency graph, the label in the node indicates the order in which they were added, arrows from one node to another indicate that the node is dependent on the other one.

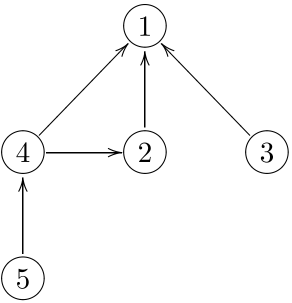


Figure 6: Dependency graph after five patches

This makes Darcs quite a different beast from Subversion. For the formal model this means that we are probably not going to be speaking much of objects, as everything in Darcs is done in terms of patches (or steps). That means we might need some empty object on which the steps can operate. However, the dependency between patches cannot be so easily represented using our formal model as there exists no way of expressing that two steps are dependent on each other (or on more than one step).

It should be noted that the strong notion of independence between the patches in Darcs is a very attractive property to have. Compare this to the notion presented earlier, that (due to the cube law) the order of commits is not relevant for the end result. With additional research into Darcs one might be able to model Darcs using the systems introduced earlier.

Even though there exists no way to indicate step dependence in the formal model, it can be modelled usings objects and steps. The above figure indicates step dependence. To model this using the version control model one needs to translate "step dependence" to "object dependence". If two steps are independent than they can be co-initial. If two steps are dependent then they cannot be co-initial. In the depen-

dency example presented here, one could represent the dependencies using a residual system as shown in Figure 7.

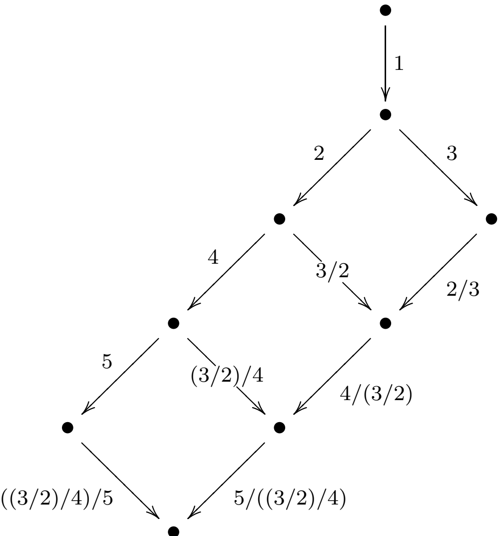


Figure 7: Dependency graph converted for use with residual system

Intuitively, this translates to the fact that steps that are dependent on another step can only operate on the resulting object of the step they are dependent on. Now, if one wanted to release a version by using a tag (a selection of patches) as Darcs does not explicitly have the notion of version, one could simply pick the corresponding object from the graph in Figure 7. It should be noted that because this is now a residual system the cube identity also holds and it is possible to have any number of steps that are independent, thus co-initial. Again, the order of commits does not matter. To make it a bit more concrete why dependent patches cannot be co-initial, assume for a moment that we have two hunk replacement patches, one that inserts a line into an empty document at position 1 and one that deletes the line from that document. If one was to say that these patches are dependent, but also co-initial then the diagram listen in Figure 8.

In conclusion, there is no way that one would be able to perform a delete before an insert[35], therefore these dependent patches cannot be represented as a co-initial pair. It is highly recommended that additional research takes place to find the correct mapping of the formal model to Darcs, as it has an interesting view on version control. To find this mapping would go beyond the scope of this thesis.

[35]For obvious reasons, a line cannot be deleted before it is inserted.

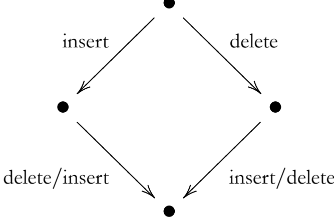


Figure 8: Insert/delete example

3.3 Miscellaneous notes

It should be noted that in the case of a conflict during three-way merging, both Subversion and Darcs exhibit some (unwanted) behaviour: they both insert conflict markers (as does for example CVS) into the file that is being edited. They are inserted to make it more apparent to the user which sections are in conflict. However, in doing so they disrupt the file they are operating on. An example of these markers is listed here:

```
First line
<<<<<<< .mine
Second line, modified by me
=======
Second line, modified by someone
>>>>>>> .r110
Third line
```

This (simple) fragment has conflict markers around the second line, indicated with "<" and ">" and the revisions. The first line after the conflict marker is modified by the user as indicated with "mine". The first line after the separator ("=======") is modified by someone else in revision 110 as indicated with "r110".

The problem may be obvious, the VCS inserts its own text into files that may not see this as valid strings, for example "=======" will not be recognised as a valid statement in many programming languages. What's more, if one decides to move from this line-based data structure to a different structure such as a tree (as we will see later), chances are that these markers will invalidate the file's structure by inserting data into the textual representation of the structure. One might argue that files that are in conflict require attention (to resolve the conflict) and therefore it is not a big problem as the conflict markers will be removed by the user, however one would like to see the conflict information stored *alongside* the original file and not *inside* as the chance of

data corruption is imminent. This calls for another extension on our version control system, however I feel that this is beyond the scope of this thesis (but still worth noting here).

4 On trees

We will now turn our focus to a different kind of structure: a tree structure. The formal model introduced in section 2 will be used in applying it to tree structures. So far we have already seen that what has been introduced can be used to model existing VCSs (up to a certain point). The decision to focus on tree structures has been made because they are the structures underlying the syntax of XML files [BPSM+04], a data format which is open and easily readable. Furthermore, it is very useful for modelling structured data. An XML file basically is a rooted, ordered tree. This means that an XML file is required to have a single root element and all other elements are children to this single root element. Not all XML files are required to be ordered, but we will focus on ordered trees; the unordered case can be achieved by loosening some restrictions on our model. The reader should note that while this sounds harmless, the inherent complexity of comparing/matching trees will increase (and has been shown to be NP-hard [Cha99], [ZWS95]).

While the aim is to eventually deal with XML files, all operations will be performed on tree structures. There are different data formats that also structure data in a tree-like fashion (such as the iCalendar format [DS98]) which we do not want to exclude from the design. This means that if one was to implement the model, one would have to use some parser to read structured files, perform operations on the underlying tree structure, and output the new structure. This does not guarantee that reading and parsing a file, performing no operation, and outputting it will yield precisely the same file (byte for byte) as the input file. However, it will guarantee that the input and output files are at least semantically equivalent (since performing no operations on a tree should yield the exact same tree).

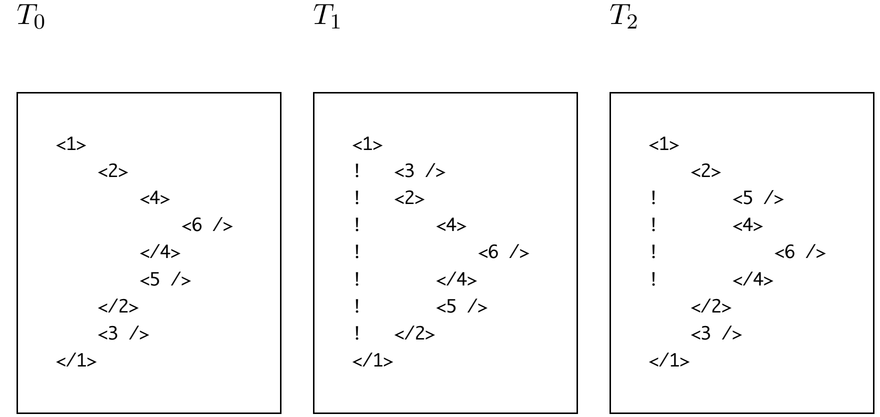
Operation	Description
Insert	Insertion of a single leaf node
Delete	Deletion of a single leaf node
Update	Update of a single node value
Move	Moving of a single node or subtree

Table 3: Operations on tree

First of all, an agreement should be reached on which operations can be performed on trees. These operations are listed (informally) in Table 3 (taken from [CRGMW96]). The first two actions (insert and delete) are pretty straight-forward, they are the basic operations with which one can construct a full step from one tree to another. Update can be expressed as a delete and an insert, but we shall see that it is much more helpful

(in terms of user interaction and structuring) to have a specific action for an update. Likewise, a move is useful as it provides a more detailed view of what the difference is between two trees; it is more useful to know that a node (or subtree) has moved to a different location in the tree then specifying that one (or more) node was deleted and inserted at a new location. Furthermore, it allows more parallel operations on a tree in a single changeset. For example, a move of a subtree could be expressed as a bunch of deletions and insertions. However, this would mark all of the nodes in the subtree as changed and therefore any other actions on this subtree would lead to conflicts. Observe Figure 9: we have a source tree T_0 and two derived trees T_1, T_2. T_M is the merged tree, which features changes made in T_1, as well as changes made in T_2.

In T_1 the nodes labelled 2 and 3 have switched order, and in T_2 the nodes labelled 4 and 5 have switched order. Note that the node labelled 6 automatically also moves, because it is a child of 4. The set of changed nodes in T_1 is therefore $\{2, 3\}$, the set of changed nodes in T_2 is $\{4, 5\}$. This means we do not have a conflict (the changes are not overlapping), and can therefore perform a merge. We could never have done this merge using insertions and deletions on the trees, because the sets of changed nodes would then have been $\{2, 3, 4, 5, 6\}$ and $\{4, 5, 6\}$ for T_1, T_2 respectively. Likewise, if we were using an XML representation of these trees, and tried to perform the merge using the standard line-based merging and differencing tools, we would have run into the same problem. See Table 4 for an example using a textual representation; lines marked with an exclamation mark are modified with respect to T_0.



T_0

```
<1>
    <2>
        <4>
            <6 />
        </4>
        <5 />
    </2>
    <3 />
</1>
```

T_1

```
<1>
!   <3 />
!   <2>
!       <4>
!           <6 />
!       </4>
!       <5 />
!   </2>
</1>
```

T_2

```
<1>
    <2>
!       <5 />
!       <4>
!           <6 />
!       </4>
    </2>
    <3 />
</1>
```

Table 4: Textual representation of T_0, T_1 and T_2

Later on, these operations also included a "Copy" operation [Lin01]. However,

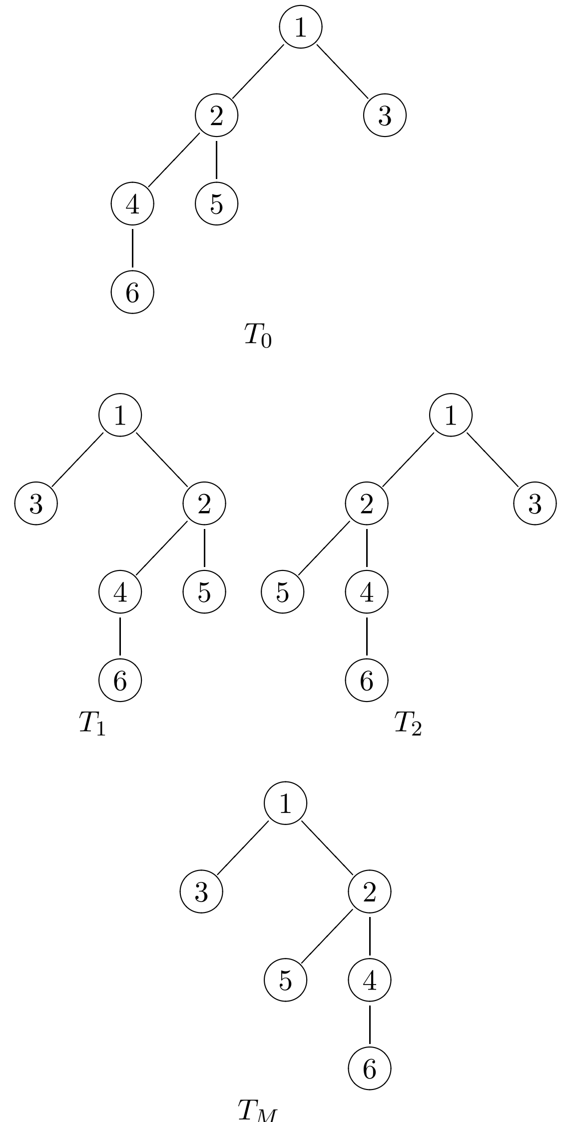


Figure 9: Moving a subtree and reordering its children

in later work [Lin04] the author argued that the "Copy" operation is very uncommon in real-life examples (as opposed to explicitly constructed use cases) and it was too dependent on the semantics of the data. Copying also brings in a plethora of hard to answer questions: are we using some notion of sharing here? Do copied nodes need to be updated when the node they were copied from is updated? Should copied nodes be deleted when the original node is deleted? Should we allow for the "uncopying" of a copied node, where the original and the copy are somehow unified? All of these questions are beyond the scope of our research and we find it is much easier to just say that a new node was inserted, instead of saying that a node was copied. Omitting copying also brings down the complexity of the matching algorithm (more on matching later).[36]

For these operations to be meaningful we have to be a bit more precise about what they do and why. For example when deleting a node with children. Will the children be deleted as well, or will they simply become children of the parent node of the deleted node? This could mean that we run into semantic problems, after all, we have some DTD[37] or schema to adhere to when using XML. Our model design will not factor in these semantics. We will concentrate ourselves purely on the tree structure at hand and leave the checking of (in)correctness to the underlying application. We will be operating under the assumption that if we have T_0, T_1 and T_2 for a three-way merge and they are in accordance with the defined structure of the tree, the resulting tree T_M will also be in accordance with the defined structure.

4.1 Operations on trees

We now turn to define what our actions should look like. Actions may be grouped together in an ordered list (or some other ordered data structure) to produce an edit-script.

For most operations we need to be able to precisely specify at which point in the tree the operation should be performed. For illustrative purposes, we will use a notation resembling XPath notation [CD99].

For example, suppose we have the following XML fragment for a fictious calendar:

[36] This is a trade-off: an increase in number of semantic operations also increases the complexity of the matching algorithm, as more heuristics are needed to extract these semantics from two trees.

[37] Document Type Definition, a typing mechanism commonly used to describe the structure of XML files.

```
<cal>
    <item>
        <date>2006-03-21</date>
        <desc>Appointment with supervisor</desc>
    </item>

    <item>
        <date>2006-03-22</date>
        <desc>Visit dentist</desc>
    </item>
</cal>
```

This can be seen as a root node `<cal>` with two child nodes (the two `<item>` nodes); each child contains a `<date>` and a `<desc>` node. We refer to the `<desc>` node in the second `<item>` node as `/cal/item[2]/desc`. Figure 10 shows this in tree form. Now, performing

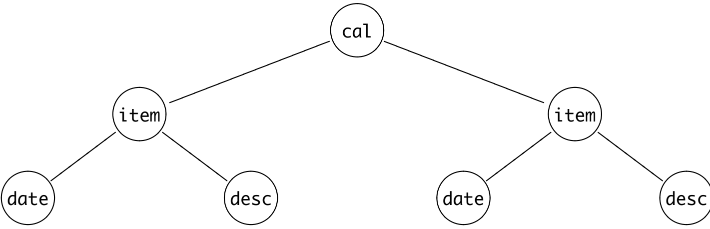


Figure 10: Fictious calendar fragment

operations on this tree instead of the plain text representation will give us a more useful description of the meaning of these operations. For example, assume that your dentist appointment is being rescheduled. Performing an operation on a text file could yield a (human-readable) diff such as "Line 8 has changed from '`<date>2006-03-22</date>`' to '`<date>2006-03-25</date>`'.".[38] Note that it would require knowledge and examination of the text file to determine what kind of item we are actually changing, there is no helpful information in this message other than the line number. If we were to operate on the tree structure, we would be able to generate messages like "The item 'Visit dentist' was rescheduled from March 22, 2006 to March 25, 2006.".

This kind of information is clearly much more useful to the end user as we are giving direct cues as to what is happening, instead of leaving it up to the user to go and see what exactly has changed in the text file. Because we know that the date that was modified is contained in the second item in the calendar, we can grab additional

[38] And this is after post-processing, a normal diff would simply say "Line 8 was deleted. A new line was inserted at position 8 with value '`<date>2006-03-25</date>`'.".

information from that item to present to the user. We are not going to focus on this, as it is the job of higher level tools to generate these messages, but this point is merely made for illustrative purposes; retaining the data structure can provide us with useful additional information that we would lose when ignoring the structure. Of course we could use extra domain-specific heuristics and algorithms to provide meaningful messages to the end user while operating on plain text, but it's easier to provide them using the original structure of the data.

Much of the work in this section will focus on determining proper definitions of the residual (/) operator and the difference (−) function, as they are what makes the model introduced earlier "pluggable".

Insertion / Deletion Insertion and deletion are closely related, as their each others opposites: insertion is the *converse* of deletion. Since we are requiring that the *converse* of a step can be calculated using only the step itself, we will need to make sure that insertion and deletions preserve as much information as possible, regarding the position in the tree, as well possible content of the node. A position in the tree can be precisely identified using a nodes parent (which it is required to have, except the root node), its optional children, and its left and right neighbours.

Update Updating a node requires that in the update operation we maintain the old state of the node, as well as the new state. For example, if we were changing some value in the node (think of an argument in an XML node) we need to maintain the old value of the argument and the new value.

Move Moving a node/subtree requires information regarding the old and new position of the node/subtree. Just like the insertion/deletion, we can pinpoint the position of the node/subtree using parent, children and neighbour information.

4.2 Tree tools

Before turning to the implementation of our model, we want to look at the technology we will be using for our tree operations. We will look at the 3DM[39] tool [Lin01] as a basis for our model and try to determine whether we can leverage this existing technology to suit our needs. Several tools have been used for performing actions on trees (or rather: XML files), but not all of them are suitable for our cause. Possible tools to use include:

[39] **3**-way merging, **D**ifferencing and **M**atching

- Traditional Unix `diff` and `patch` tools, used by the version control systems introduced earlier. As said, they suffer from the fact that they can't operate properly on structured data, as they tend to run into a lot of conflicts, for example when doing something as simple as moving a subtree and reordering some of its children. The `diff3` tool can be used to perform a *merge* between three text documents.[40]

- XMLTreeDiff, a tool developed by IBM Alphaworks[41]. XMLTreeDiff is used in XMiddle [MCZE02], a middleware for portable devices such as PDAs, cell phones, etc. Unfortunately, XMLTreeDiff suffers from two significant flaws: it is closed source, and therefore we cannot study its internals, but more importantly, it has been retired by IBM (for unknown reasons) and is no longer available. XMLTreeDiff appears to be replaced with a different tool called "XML Diff and Merge Tool"[42]. However, this tool also is closed source, only functions as a GUI[43] tool, does not have any API[44]s available to embed/use it in another tool and requires a commercial license for use. Therefore, it is hardly useful for our purposes.

4.2.1 3DM introduction

3DM provides us with two things: tree differencing, and tree merging. Both are very interesting for us: we need to have a good notion of differencing (as to give a good definition of the difference operator, which in turn plays a vital part in three-way merging; recall our definition of three-way merging using the residual, difference and composition operators from section 2). We might be able to use some ideas from Lindholm's merging algorithm in our model.

The 3DM tool consists of three parts: tree matching, tree merging and tree differencing.

Tree matching A matching between two trees T and T' (where T' is some modification of T) is a set of edges or tuples (n, m), where n, m are nodes such that $n \in T$ and $m \in T'$. Matchings tell us which nodes in distinct trees correspond to each other. They are used to detect changes that transform one tree in another, they are not meant as a differencing mechanism, they merely determine

[40] How's that for bad naming!
[41] `http://alphaworks.ibm.com/`
[42] `http://alphaworks.ibm.com/tech/xmldiffmerge/`
[43] Graphical User Interface
[44] Application Programming Interface

which nodes in two trees are related to each other. The matching is used in the process of determining an edit script or the difference between two trees (which we will explain a bit further on). For an example see Figure 11. The dashed lines represent the found matches. Nodes 1 and 4 are matched and nodes 2 and 5 are matched. Node 3 is unmatched.

Tree merging A merge (or three-way merge) between a base tree T_B and two derived trees T_1 and T_2, resulting in a new tree T_M.

Tree differencing An algorithm to encode the difference between two trees, preferably as space-efficiently as possible. The differencing algorithm is based on encoding the matching between two trees.

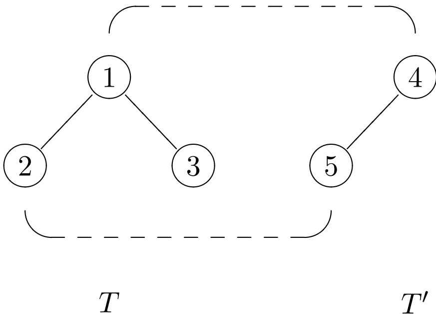


Figure 11: Matching example between T and T'

It should be obvious that the differencing as mentioned above is not quite the differencing we are looking for. We are looking for a *diff* from which in any case the converse can be computed.[45] However, seeing as the difference is based on encoding the matching, we might be able to encode the matching in a different way to be able to fulfil this requirement. Each aspect of 3DM is designed to be pluggable in nature, so modifications can be made to each module (matching, merging or differencing).

A matching between nodes in original tree T_0 and a modified tree T' (where T' stands for either of the derived trees, T_1 and T_2) is interpreted in terms of edit operations. A node $n \in T_0$ with no match in one of the trees in T' is considered *deleted* in that tree. A node $m \in T'$ with no match in T_0 is considered *inserted* into that

[45]This requirement is in conflict with encoding the diff "as space-efficiently as possible"; being able to calculate the reverse requires that the old and the new situation are available. Encoding as space-efficiently as possible requires storing as little information as possible, so the old situation doesn't get stored. This is a design choice.

tree. Matching is expressed with the predicate $\mathrm{m}(n, m)$, which is true iff n and m are matched. Matching is required to satisfy five requirements:

1. Every node in T_1 is matched to at most one node in T_0 and any node in T_0 may have at most one match in each of T_1;
2. The (artificial) parents of the roots of the trees are matched;
3. A matching is reflexive;
4. A matching is symmetric;
5. A matching is transitive.

Likewise for T_0 and T_2. Note that the first property effectively prohibits the "Copy" operation. The last three properties effectively are a definition of an equivalence relation, which makes the matching relation an equivalence relation between two nodes in separate trees. The second property states that each tree T_k has an artificial parent $\perp_k$ and for all related trees, these parents are matched to each other. It will be explained why we need this artificial parent in a moment. Tree matching generally is one of the hardest things to do in the process of tree merging, as it requires a solid definition of how two nodes can be matched using some matching function. Various literature exists that describes this phenomenon [CGM97, CRGMW96] and it is very important that we are able to find a good matching, because differencing depends on the matching. In fact, differencing can be trivially described once a matching between two trees T_0, T_1 is found using the following rules:

- Any node that exists in T_0, but has no matching node in T_1 is said to be *deleted*;
- Any node that exists in T_1, but has no matching node in T_0 is said to be *inserted*;
- Any node that exists in T_0, that has a matching node in T_1, but has a different parent, or different neighbours in T_1 is said to be *moved*;[46]
- Any node that exists in T_0, has a matching node in T_1, but has a different content is said to be *updated*;
- All other nodes are matched, due to the fact that all unmatched nodes are already filtered out by the first two rules, and are considered unchanged.

[46]Position in the tree for any node is determined by its parent and its direct neighbours. This is a design choice, it suffices to express the parent and the number of siblings left (or right, just make it consistent) of a node to determine position in a child list.

And likewise for T_0 and T_2. Trees can be described by use of *parent-child-successor* (PCS) relations. Many more representations of trees exist, but for our purpose we choose this one. This PCS relation is a ternary relation $\mathsf{pcs}(r, p, s)$ expressing that r is the parent of both p and s, and that s immediately follows p in the child list of r (recall that we are using ordered trees, so the order of the children must be preserved).

The set of relations expressing the tree T_k is denoted $\mathbf{T}_k$. The start and end of a child list are expressed with the symbols $\dashv$ and $\vdash$ respectively. There exists an artificial parent for the root R_k of the tree T_k, designated with $\perp_k$. There exists a PCS relation with an empty child list for each leaf node, thereby guaranteeing that there exists a PCS relation $\mathsf{pcs}(n, *, *)$ for each node n. The artificial parent $\perp_k$ is used to indicate the position of the root node of the tree. Because the root node has no siblings or parent, there would otherwise be no pcs relation for the root node to indicate that it *is* the root node.

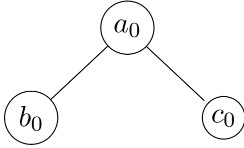


Figure 12: T_0

So, a simple tree T_0 as listed in figure 12 can be represented using PCS relations as:

$$\begin{aligned}\mathbf{T_0} \;=\; & \{\mathsf{pcs}(\perp_0, \dashv, a_0), \mathsf{pcs}(\perp_0, a_0, \vdash), \\ & \mathsf{pcs}(a_0, \dashv, b_0), \mathsf{pcs}(a_0, b_0, c_0), \mathsf{pcs}(a_0, c_0, \vdash), \\ & \mathsf{pcs}(b_0, \dashv, \vdash), \mathsf{pcs}(c_o, \dashv, \vdash)\}\end{aligned}$$

Furthermore, since we are going to deal with XML files, we also need to differentiate between text and element nodes and possible node attributes. To facilitate this, we not only need PCS relations in our tree description, but also something to indicate content. Consider for example the following XML (or XHTML [W3C02], a reformulation of HTML in XML) fragment:

```
<b>
    This text is bold
    <img src="tree.jpg" alt="Inline image of a tree" />
</b>
```

This fragment maps beautifully to the tree in figure 12, the result can be seen in figure 13. Keep in mind that even though the *labels* of the nodes are still a_0, b_0 and c_0; we are showing the *content* in the node now.

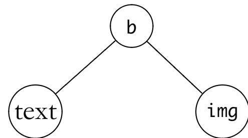


Figure 13: T_0 with XML

So, we can add content relations to our existing PCS relations using the relation $\mathsf{c}(n, \mathbf{c})$ to indicate that the node labeled n has the content **c**. This means that we'll add the following content relations to our PCS relations:

$$c(a_0, \{\text{type: element, name: } \texttt{b}, \text{attributes: } \{\}\}),$$
$$c(b_0, \{\text{type: text, chardata: } \texttt{This text is bold}\}),$$
$$c(c_0, \{\text{type: element, name: } \texttt{img}, \text{attributes: } \phi\})$$

Lindholm does not specify in his paper what the attributes should look like, merely that they are a set of attributes and values, so they are presumably unordered and without repetitions. A logical solution (in line with Lindholm's definitions) would be to let ϕ be an unordered set of key-value pairs $k : v$ where k is the *key* and v is the *value* of the attribute. Resulting in:

$$\phi = \{\texttt{src} : \texttt{tree.jpg}, \texttt{alt} : \texttt{Inline image of a tree}\}$$

This also means that the granularity for attribute changes is placed at the node level: one changed attribute marks the entire node as updated, which is fair enough.

Changes to a tree can be described using these PCS and content relations. For example,

$$c(a_0, \{\text{type: element, name: } \texttt{i}, \text{attributes: } \{\}\})$$

means that the content of the node labelled a_0 is now `i` instead of `b`.

It should be obvious that this notion is not suitable as a correct representation of a changeset for our purposes. For one, it does not fulfill the property that every changeset has a converse, and that this converse can be computed solely from the changeset. The content changes merely state the updated content, and provide no means of reverting to the old situation.

Likewise, for a structural change:

$$\mathsf{pcs}(a_0, \dashv, c_0)$$

means that the node labelled c_0 is the leftmost child node of a_0; no information is retained regarding the old position. Several changes (both content and structure) can be combined into a changeset. This set should be *consistent*, which in turn means that it should be unambiguous. This holds if the changes in the set state no more than one content, parent, predecessor, and successor for each node.[47]

To come up with a suitable implementation of our abstract model, we need to define the following things: objects, steps, src and tgt, 1_a, $\cdot$, $/$, $-$, $\bar{\cdot}$

Starting from the ground up, that means that we let $\mathcal{A}$ be a rewriting system $\langle A, \Phi, \mathsf{src}, \mathsf{tgt} \rangle$ such that:

- A is a finite set of key-value pairs $(a : T)$ where a is a label and T is a rooted, ordered, labelled tree;

- Φ is a finite set of steps, where steps are represented by a triple (ϕ, a, b) where ϕ is a rooted, ordered, labelled tree, and a and b are tree labels, respectively the src and tgt of the step. a and b are not unlike *pointers* in programming, or *keys* in database theory.

- src and tgt are functions of type $\Phi \to A$.

The src and tgt functions work on a step by obtaining the labels from the tuple (ϕ, a, b). With this label they can look up the corresponding object in A. For most practical purposes it would even be sufficient not to look up the corresponding objects; suppose we are trying to determine whether two steps are composable. It would be sufficient to extract the tgt of the first step and the src of the second to determine this. There is no need to actually retrieve the objects from A and compare them node by node, step by step.

Stepping up to residual systems, we let $\mathcal{A}^{\mathcal{R}}$ be a residual system $\langle \mathcal{A}, 1, /, \cdot \rangle$. The residual operator is the most important part here, it is what determines whether and how we can perform a three-way merge. For any two co-initial steps it can determine the cofinal steps. It is this part of our system that also signals conflicts. If the residual is successful, it returns a step, so it's a function $\Phi \times \Phi \to \Phi$. For any

[47] It is not *consistent* because it may state multiple positions and multiple child lists for a node, and it is *ambiguous* because that means it can be read in multiple ways, where there is no way of determining what is the correct reading.

two co-initial steps $(\phi, a, b), (\psi, a, c)$ (recall that the sources need to be identical), $(\phi, a, b)/(\psi, a, c) \Rightarrow (\chi, c, d)$ for some new d (see figure 14). Composition on two steps is fairly easy, as it will probably suffice here to simply concatenate one step to the other.

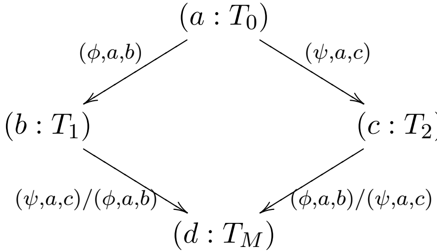


Figure 14: Residual system

Let A^{VC} be a version control system $\langle \mathcal{A}^{\mathcal{R}}, -, ^{-} \rangle$. As said, $-$ is the difference operator, and operates on two objects and resulting in a step: $A \times A \rightarrow \Phi$. This is also a very important operator as it plays a part in the three-way merging process. For this operator we'll use the techniques described by Lindholm for matching and differencing. The $^{-}$ operator is the converse operator, operating on steps: $\Phi \rightarrow \Phi$.

4.2.2 Tree matching

The matcher presented in [Lin01] still allows for copies of nodes, which we will not use. Matching is only touched upon lightly in [Lin04]; there a heuristic matcher is used to build one-to-one matchings between T_1 and T_0, as well as between T_2 and T_0. The resulting match is then extended in such a way that it satisfies the five requirements presented earlier on. The matcher consists of three parts:

Node similarity To be able to match nodes, there needs to exist some way of determining how similar nodes are. Suppose the content of a node is slightly modified, we still want to be able to match the original node to the modified node. Therefore, there needs to be a notion of *node similarity*.

Matching truncated subtrees This is the part that actually does the first stage of matching. A truncated subtree of a tree T is a subtree of T, which have all descendants of some of the nodes in the subtree removed. In short, the matcher recursively traverses T_1 to find matching truncated subtrees in T_B. Once a subtree is found it is marked as such, and matching occurs for its children. The matching occurs in four steps [Lin01]:

1. Finding matches
   Each node in T_1 is traversed and it is attempted to find a match in T_B by first matching exactly on node content, that is, a match is found if its content is exactly the same. If multiple nodes are found in T_B it is heuristically determined which node is the best match (using *node context*) and that node is returned. If no exact matches are found, the node is returned which has the smallest *nodeDistance*, which is smaller than some threshold to indicate the maximum distance for two nodes to be considered matched. A match in T_B is only valid if it is not matched to some node already.[48]

2. Finding matching subtrees
   Subtrees are matched by traversing T_1 depth-first, starting at matched node $n \in T_1$ and $m \in T_B$. Subtrees are recursed as long as the current nodes of the traversal has exactly the same number of children, and the content of the children match exactly.

3. Selecting the best matching subtree
   If more than one subtree is matched in T_B then the most suitable one is selected as the match. This is determined by number of nodes in the subtree (larger subtrees are considered better matches) and by context (looking at siblings). If there is no matching subtree no match is made.

4. Matching and recursing
   After the best subtree rooted at $m \in T_B$ is matched to some subtree rooted at $n \in T_1$, the nodes of the subtree are matched with their corresponding nodes in the subtree of T_B.

Matching similar unmatched nodes Postprocessing takes place in the form of an algorithm that tries to match any unmatched nodes that may remain. It does this by taking pairs of unmatched nodes $n \in T_1, m \in T_B$ and looking at their position in the tree (whereas matching previously looked more at content than position). A pair n, m of nodes is matched if their parents are matched, and if either the left or right siblings (if they exist) are matched. If neither sibling matches they are still considered matched if they are the left-most or right-most node. To avoid a domino effect this is done bottom-up.

[48] In [Lin01] it is not a one-to-one relation, but a one-to-many. Multiple nodes may be matched. Here we are explicitly denying multiple matches.

In short, matching in [Lin01] and [Lin04] is done by first matching nodes on content, then trying to match maximal subtrees from these nodes and finally try to match any unmatched nodes that remain.

4.2.3 Chawathe's matcher

Lindholm's matcher is derived from the matcher presented in [CRGMW96]. Due to the fact that Lindholm remains rather vague regarding the matcher used in [Lin04], we will also present the matcher due to Chawathe et al.

The assumption is made that all nodes in an ordered tree have a *label* and a *value*. For interior nodes (non-leaf nodes) the value may be *null* or empty. The existance of unique identifiers is not assumed. The label of a node does not have to be unique. For example, suppose we have some XML element `<document>`. Its label is "document". Now consider an element `<paragraph>`. Its label is "paragraph", but multiple paragraphs may exists in a document.

The focus of the paper is finding a transformation from some "old" tree T_1 to a "new" tree T_2. The notion of a correspondence between nodes that have identical or similar values is formalized as a *matching* between node identifiers. Matchings are one-to-one. The *change detection* problem presented here is split into two problems:

- The *Good Matching* problem, concerned with finding a "good" matching between two trees;
- The *Minimum Conforming Edit Script (MCES)* problem, concerned with finding a minimal edit script based on the matching found.

The actions that can eventually end up in the edit script are:

- Insert; The insertion of a new **leaf** node, denoted by $\mathsf{ins}((x, l, v), y, k)$ where x is the node identifier, l is a label, v is a value (assumed *null* if omitted). It is inserted as the kth child of y.
- Delete; The deletion of a **leaf** node, denoted by $\mathsf{del}(x)$ where x is the node identifier.
- Update; The update of the value of a node, denoted by $\mathsf{upd}(x, \textit{val})$, where x is the node identifier and *val* is the new value.
- Move; The move of a subtree (a single leaf node is also a subtree), denoted by $\mathsf{mov}(x, y, k)$, where x is the node identifier, y is the new parent and k is the new position in the child list of y.

Naturally, these matching problems also occur to a lesser degree when we are operating on texts. However, they are much easier to resolve there, as the only actions that we allow are insertions and deletions and we are not retaining any other structural information. In fact, the `diff` command implicitly creates a matching between lines in two texts by finding sequences of lines common to both files, interspersed with groups of differing lines called hunks. From this matching the difference can be easily constructed since we only have to account for deletes and inserts (parts that are unmatched).

Regarding the actions listed above, we feel we should note that there is no record kept of the "old" situation when describing changes. When moving a subtree/node, only its new position is stored in the edit script, not its old position. We need this old position to be able to calculate the converse step. That means that if we are to use this matcher/*diff* generator we will have to augment it to also store the old situation. Likewise for update a node value: we need to maintain the old value in the edit script.

As indicated, the *delete* and *insert* operations operate solely on leaf nodes. This means that if we want to insert an interior node, we first insert it as a leaf node and then move some subtree under it. If we want to delete an interior node, we first need to move its subtree somewhere else and then delete it. Or delete all its children with it (for example, when we mean to delete an entire subtree).

For the MCES there must be a way to express the cost of an edit script: $c_D(x)$, $c_I(x)$, $c_U(x)$ and $c_M(x)$ denote the cost for deleting, inserting, updating and moving respectively. Chawathe et al assume that the cost of deleting, inserting and moving are equal and are unit cost operations: $c_D(x) = c_I(x) = c_M(x) = 1$ for all x. The cost of updating a node is given by a function, *compare*, that evaluates how different the old value is from the new value. The idea behind this is that it should be cheaper to move and update a node, then it is to delete and insert a node. Likewise, just an update on a node should be cheaper than a delete and an insert. This *compare* function takes two nodes as its input and outputs a number in the range $[0, 2]$. This intuitively states that it is preferred to update changed nodes than it is to delete it and insert a new one, unless they are very different (in which case it is unlikely that they are corresponding nodes). The *compare* function should therefore return a cost greater than 1 for very dissimilar nodes and smaller than 1 for similar nodes. It could simply determine the percentage of identical text, or use some heuristics to determine similarity.[49] Or interestingly, we

[49] As it turns out, it is not easy to measure text similarity. For example, constructing a histogram on character usage will yield approximately the same results for a given language, for any text of sufficient size.

might use another version control system there. In theory, there is nothing holding us back from using another (perhaps different) system to provide version control facilities for node values , as version control on node values is separated from version control on the tree structure. We will touch upon this interesting idea a bit later. The edit script generated is supposed to give a transformation from T_1 to T_1', which in turn is isomorphic to T_2, where two trees are isomorphic when only their node identifiers differ.

Finding a MCES given a matching M for two trees T_1, T_2 and an empty edit script E involves five phases:

1. Update phase. In this phase the trees are scanned for pairs of nodes $(x, y) \in M$ whose value differ. If found, an update operation is added to E and applied to T_1.

2. Align phase. In this phase the interior nodes of the trees are scanned. For any pair of matched interior nodes $(x, y) \in M$ we say that their children are *misaligned* if they are in a different order in T_2 than they are in T_1. For each misaligned child a move operation is added to E and applied to T_1.

3. Insert phase. In this phase T_2 is scanned for any unmatched nodes $z \in T_2$ where $p(z)$[50] is matched to some node $x \in T_1$. A new node identifier is generated, an insert operation is added to E and applied to T_1.

4. Move phase. In this phase we scan for pairs of nodes $(x, y) \in M$ such that $(p(x), p(y)) \notin M$. We then look for the matching node to $p(y)$ in T_1, call this node v. We then add a move operation to E such that x is moved to be a child of v with the correct position in the child list (which we can do because we have already aligned the children) and apply it to T_1.

5. Delete phase. In this phase we look for unmatched leaf nodes $x \in T_1$. For each of these nodes we add a delete operation to E and apply it to T_1.

Give a matching M, the MCES algorithm consists of two steps:

1. A breadth-first traversal of T_2 combining the following four phases in a single pass: update, insert, align, move;

2. A traversal of T_1 to delete any unmatched nodes.

[50]The *parent* of z.

After these two runs, we have a minimum cost edit script, a total matching M' and a (copy of) T_1 that is isomorphic to T_2.

For the matching itself (which serves as an input for the MCES algorithm) Chawathe et al. first define criteria that the matching algorithm must satisfy:

1. Two leaf nodes that are "too dissimilar" may not be matched. This is formulated by stating that two nodes may only be matched if their labels are identical and if the difference between the value of the two nodes is between some predefined range, this difference can be calculated using the *compare* function.

2. For interior nodes the comparison using node values is not really helpful, as they often are empty. Again, the labels are required to be identical. Furthermore, similarity is determined using the number of common descendants (children).

Furthermore, two assumptions are proposed, that may or may not hold:

1. There is an ordering $<_l$ on the labels in the *structuring schema* such that a node with label l_1 can appear as the descendant of a node with label l_2 only if $l_1 <_l l_2$.

2. For any leaf node $x \in T_1$, there is at most one leaf $y \in T_2$ such that *compare*$(v(x), x(y)) \leq 1$ and vice versa.

The first assumption imposes the *acyclic labels* condition: certain labels may only occur as children of other labels. The second assumption states that for any leaf node the *compare* function will only return a cost value smaller than 1 for at most one other node in its partner tree.

A matching is said to be *maximal* if it is not possible to augment it without violating the two criteria introduced above. According to Chawathe et al. the matching criteria imply that there exists a unique maximal matching. Furthermore, given the two assumptions, that unique maximal matching is also the best matching.

The matching algorithm presented, "*FastMatch*" uses a function *equal* to compare nodes. Intuitively, a node-pair gets added to the matching iff two nodes are *equal*. Formally, *equal* is defined:

- For leaf nodes, *equal*(x, y) is true iff $l(x) = l(y)$ and *compare*$(v(x), v(y)) \leq f$, where f is a parameter in the range $[0, 1]$;

- For interior nodes, *equal*(x, y) is true iff $l(x) = l(y)$ and $\frac{|common(x,y)|}{\max(|x|,|y|)} > t$, where t is a parameter in the range $[0.5, 1]$ and *common* is a function from nodes to node-pairs, returning the node-pairs of matched children of x and y. $|x|$ is

defined as the number of children of a node x. $|$*common*$(x, y)|$ is the number of pairs returned by *common*. Informally, this means that in addition to having identical labels, more than half of a node's children need be matched to the children of some other node for them to be considered matched (more than half of the children of the node with the most children).

Finally, the actual matchers works in two phases which are identical except that the first operates on leaf nodes, and the second on interior nodes. The algorithm is as follows for each leaf or interior (depending on phase) node label l, with initial empty matching M:

1. Construct a *chain* S_1 of equally labelled nodes in T_1; nodes appearing earlier in the tree (using an in-order traversal of T_1 where siblings are visited left-to-right) will appear earlier in the chain.

2. Construct a chain S_2 of equally labelled nodes in T_2; similarly to the way S_1 is constructed.

3. Find the *longest common subsequence* (LCS) for these chains, using the *equal* function. This return the pairs of nodes that are in this subsequence.

4. Each pair of nodes $x, y \in LCS$ is added to the matching M.

5. For each unmatched node $x \in S_1$, if there is an unmatched node $y \in S_2$ such that *equal*(x, y) then (x, y) is added to M and x, y are marked "matched".

4.2.4 Edit script disambiguation

Lindholm's and Chawathe's matchers both create a minimal edit script. That is, each node that is modified has at most one insert, update, delete or move in that script. However, suppose we have some text editor that does indeed keep track of the changes that are made each save (we'll ignore the case where every keystroke is recorded). If we were writing a document, we may make multiple edits to a particular part, say a sentence, before we commit our document (for example, some latex document, like a thesis). As we now already have the edit script available at commit time (with edits for each consecutive save of the document), it would be a waste not to use this and simply try to calculate a new edit script as we would in a regular *commit*.[51]

[51]Some popular synchronisation tools for PDAs and other handheld devices keep their own edit scripts, especially for the purpose of not have to calculate their own structured *diffs*. This is useful in for example calendaring tools. These devices consequently do not have to compare entire data structures with each other, which is most of the time infeasible due to constraints on CPU power and available memory.

So, suppose we have some edit scripts available containing any number of updates, deletions, insertions and moves. How do we move to a minimal edit script?

For any number of updates, this is fairly easy. Updates are the only type of actions that modify a node value. Furthermore, they don't have any relation to the tree structure, whereas deletions, insertions and moves operate on the tree structure. Remember that edit scripts are in order, so we can simply collapse all updates into one update: the update that changes the value of a node to the value of the last update in the edit script. Consider two consecutive updates of a node value: a to b, b to c. This can be collapsed to a single update a to c.

Unfortunately, for structural updates (such as move, insert and delete) things aren't so easy. By far the easiest method would be to apply all actions in the edit script to the tree and at the end determine what has changed, but that would be calculating the matching which we discussed in the previous sections. The issue at hand is the fact that subtrees may become intertwined at some point in the edit script. That is, one subtree b may be moved such that it becomes a child of node a. This is where it is important to remember that all actions are in order and that this order must not be disturbed. For calculating the MCES this does not apply; there we can use to fixed data structures and determine the order in which actions take place ourselves. Here, we must adhere to the ordering to guarantee that the resulting edit script is equivalent with the original (ambiguous) edit script.

Another thing to keep in mind is that the matcher presented in [CRGMW96] only allows deletions and insertions on leaf nodes.[52] Deletion of a subtree therefore will yield a number of deletions in the edit script. We will assume (informally) that the edit scripts are correct, in that sense that if a node is inserted at some part of the edit script, no operations on that node may precede it. Likewise, if at some point in the edit script a node is deleted, no operations on that node may follow. Using that information, we may decide that the disambiguation algorithm consists of four phases: insert, update, move, delete.

Consider an (ordered) edit script E and a disambiguated edit script E' which is initially empty, then the following holds:

- For any node that is inserted in E, we can place its insertion at the beginning of our resulting edit script E'. As there may be entire subtrees inserted, we must maintain the order in which inserts take place in E. This is the insertion phase.

[52]Insertion may be an ill-chosen term as it implies that the node can be inserted at any point in the tree, addition or appending might be better.

- For any node that has precisely one update in E, we may copy this update to E'. For any node that has more than one update in E, we may collapse all updates into one update, with as its old value the old value of the first update, and as its new value the new value of the last update. This resulting update is added to E'.

- For moving a node or subtree we also need some sort of collapsing mechanism, to create a single move from any number of moves. As said, moves may become intertwined, with which we mean that a subtree may be moved to become a part of another subtree, which in itself may be moved to a new position. Any two nodes a, b that at least have one move in E and at some point in the transformation become each others parent or child or part of the same child list are said to be *move-dependant*. However, for any node that is not *move-dependant* on another node and has one or more moves in E we can determine its final parent by collapsing its moves into one move, similar in the way we collapse multiple updates into one update. The result is that we end up with one move per moved node, regardless of how many times it was actually moved. These moves may then be added to E'.

  For *move-dependant* nodes we need to take extra precautions that their collapsed move actually moves them to the correct position in a child list (as intertwined moves can cause incorrect child lists if we simply collapsed all moves as if they were not *move-dependant*). Harmless as this may sound, it probably is not that easy to implement.

- For any node that is deleted in E, we can place its deletion at the end of our resulting edit script E'. As there may be entire subtrees deleted, we must maintain the order in which deletes take place in E. For any node that is deleted in E and has its value updated at some point in E, we may remove these updates (as the node is deleted anyway and updating its value has no structural consequences). This is the deletion phase.

One might wonder why we should bother with this disambiguation anyway; the extra information that we have available (due to all the changes being available) are basically thrown away by performing this (rather hard) disambiguation. The end result is an edit script that may be a bit closer to what actually happened, but it remains the question if this resulting edit script is a “better” edit script than the one obtained from a generated matching.

In conclusion: two matchers have been presented, Chawathe's matcher and a combination of Lindholm's matchers. A few remarks can be made here. Lindholm's matcher allows for node labels to change from one versions to another, Chawathe's does not. It requires in the matching criteria that for any two nodes to be matched at least their labels should be identical (along with other requirements). This is in favor of Lindholm's matcher. However, Lindholm's matcher (as he describes it in [Lin04]) is never formally presented and we therefore lack the ability to evaluate it.
The work on matchers is nowhere complete and presumably there may be better suited matchers out there. This is where our model (and 3DM) shine: the model allows different tools to be plugged in for its operators, and 3DM allows different matchers, diffs and mergers to be plugged in (in addition to the stuff that is already available).

5 Conclusions & future research

At the end of this thesis I would like to review what has been done and found and discuss what remains as future research. Furthermore, what kind of possibilities are opened due to the research presented?

Firstly, a formal model that uses ideas from rewriting systems was presented with the purpose of modelling a version control system. After using rewriting and residual systems, they were augmented with a new kind of system: an abstract version control system, which added extra functionality (the *converse* and *difference* operators) required for version control. By using a residual system as the basis a significant amount of functionality was obtained "for free". This included one of the necessary requirements for performing three-way merges: the guarantee that for every pair of co-initial steps there is a pair of convergent cofinal steps. The missing link was a way of obtaining an unknown step between two objects, which is why we introduced the *difference* operator. The *converse* operator was added to be able to store information in a way similar to the way Subversion stores reverse changesets.

Secondly, the model was applied to existing version control technology: Subversion and Darcs. Using an instantiation of our model a simplified version of Subversion was fairly accurately modelled. It was shown that existing tools could be used in the model (for example, using the traditional *diff* and *patch* tools).

Finally, the model was used to provide version control on tree structures. Specifically, we looked at rooted, ordered trees for the reason that they are the underlying data structures to file formats such as XML and HTML, iCal, LaTeX etc. The 3DM tool was shown to be very useful for the model, as it provided matching, differencing and merging functionality, which could be plugged into the formal model fairly easily.

5.1 Caveats

Of course the goal is to make this study as complete as possible. However, given the limited scope of this thesis, not all relevant aspects could be extensively examined. Therefore, a few caveats are in order:

- The interaction between various operators, especially $^{-}$, $-$ and $/$ has not been studied fully. The functions $-$ and $/$ are closely related to each other as they both provide some sort of *differencing* function, $-$ between two objects and $/$ between two steps.

- The presented model only accounts for version control on a single data struc-

ture. It must be extended at some point to be able to operate on multiple structures. One way of doing this would be to create a version control system that has sets of version control systems as the objects it operates on.

- In the Subversion example the revision number was used as an identifier for the version. When more files are added to the repository this cannot be used anymore, as Subversion uses revision numbers to indicate the state of the repository as a whole and not the state of each individual file. CVS, on the contrary, does allow individual revision numbers for files, but it would probably be better to assign some other identifiers to the different versions of the files.
- Tree merging has not been discussed.[53] Tree merging allows us to apply a changeset to a tree. This not a very complicated operation as we can just apply the actions in the changeset in order to the tree.
- The research on the tree matcher designed by Lindholm could be expanded to provide a more accurate account of what the current status of this matcher is (due to the differences in matching algorithms in [Lin01, Lin04]).
- Darcs could be studied more. We have seen that the formal model does not map onto it very easily. This is regrettable as Darcs has some promising features such as its strong independence between patches. With additional work I believe that it will be possible to find a good mapping.
- The tree matcher only works on ordered trees and does not take any structural constraints into account. Not all data is ordered per se and on the other hand, various data does have structural constraints.[54] Matchers should respect these structural constraints as it in fact may make matching easier (two nodes may not be matched if that would invalidate the structural constraints).
- Branching has not been discussed, even though branching is something that occurs frequently in version control systems. Branching occurs when some development on files should not take place in the main line of development due to stability issues, testing and so on. Fortunately for us, branching can result in a tree-like structure which once again can be put under version control.[55]

[53] Here merging means applying a changeset to a tree, not three-way merging

[54] For example, in (X)HTML the `<tr>` element (table row) may only occur in a `<table>` element.

[55] In some cases it is more appropriate to look at branching VCSs as a directed graph, but that is also a data structure one might be able to put under version control.

5.2 Future research

The work presented here opens some interesting possibilities for future research. Worth noting are:

- Designing version control systems at file system level. One can see a file system as a tree, representing files with leaf nodes and directories with interior nodes. This means we can easily build a version system that operates on file system objects. File nodes can then have a version control system as their value which in turn permits version control on the contents of a file. Manipulation on the contents of a file therefore has no structural implications for the tree.

- Designing a truly user-friendly/user-centric version control system that does not involve complex manipulations on the command-line. Current version control systems are generally not thought of as being user-friendly. That means that the adoption rate is very low (apart from academic or business uses). The "general public" will only use it if it is available in an easy-to-use fashion.

- Designing new matchers, differencers and mergers for other data structures. These do not need to be limited to "basic" structures as trees, graphs and lists, but may also be designed for a specific file format and make use of the defining characteristics of that format (Java sources, PostScript files, various image formats and so on).

References

[BPSM$^+$04] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (Third Edition). Technical report, W3C, 2004.

[CD99] James Clark and Steve DeRose. XML Path Language (XPath). Technical report, W3C, November 1999.

[CGM97] S. Chawathe and H. Garcia-Molina. Meaningful Change Detection in Structured Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 26–37, Tuscon, Arizona, May 1997.

[Cha99] S.S. Chawathe. *Managing Change in Heterogenous Autonomous Databases*. PhD thesis, Stanford University, 1999.

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

[CSFP06] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*. <`http://svnbook.red-bean.com/nightly/en/svn-book.pdf`>, revision 2191 edition, May 2006.

[DS98] F. Dawson and D. Stenerson. Internet Calendaring and Scheduling Core Object Specification. RFC 2445, Internet Engineering Task Force, 1998.

[Lin01] Tancred Lindholm. A 3-way Merging Algorithm for Synchronizing Ordered Trees – the 3DM merging and differencing tool for XML. Master's thesis, Helsinki University of Technology, September 2001.

[Lin04] Tancred Lindholm. A Three-way Merge for XML Documents. In *Proceedings of 2004 ACM symposium on Document engineering*, 2004.

[MCZE02] C. Mascolo, L. Capra, S. Zachariadis, and W. Emmerich. XMIDDLE: A Data-Sharing Middleware for Mobile Computing. *Int. Journal on Personal and Wireless Communications*, 21(1):77–103, April 2002.

[Rou06] David Roundy. *Darcs 1.0.7.* <`http://www.darcs.net/darcs.ps`>, May 2006.

[Ter03] Terese. *Term Rewriting Systems*. Number 55 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 2003.

[W3C02] W3C. XHTML 1.0 The Extensible HyperText Markup Language (Second Edition). Technical report, W3C, 2002.

[Wei91] Mark Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):94–102, September 1991.

[ZWS95] Kaizhong Zhang, Jason T.L. Wang, and Dennis Shasha. On The Editing Distance Between Undirected Acyclic Graphs, 1995.