Merging and Renaming in Revision Control Software

Freddie Albertsman
c03fan@cs.umu.se

February 4, 2008
Master's Thesis in Computing Science, 30 credits
Supervisor at CS-UmU: Jürgen Börstler
Supervisors at ÅF: Pontus Nyman,
Daniel Ashhami
Examiner: Per Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

Modern projects often involve geographically spread people working on the same project simultaneously. Thus the contents of the project are likely to be read and edited by more people than the original authors of the content. That's why refactoring becomes an important part of the project development since refactoring, amongst others, enhances the readability of the material.

Since revision control systems are widely used in bigger projects, they must thus provide essential tools such as good renaming functions, which is important for refactoring, as well as good merging support which enables several developers/editors merge their work without doing much manual work.

This thesis presents and discusses different merging techniques as well as different approaches to conflict detection, avoidance, and reduction. Moreover a solution to some renaming problems in Subversion is presented and another is discussed.

Contents

List of Figures

Chapter 1

Introduction

Large-scale projects often involve many geographically distributed groups where face-to-face communication is hard. These groups must be organized in order for the project to succeed. That is, the contributions of the developers must be put together into the main project stream. Moreover, each change to the project must be tracked to a developer or perhaps many in order for the team, involved in the project, to understand why the change was made and by who. To be able to manage such projects, revision control systems are used. With such systems, developers are able to contribute to a project remotely and simultaneously and their contributions may be integrated to the main project stream by means of merging. Moreover, in software projects in particular, refactoring is used to enhance the structure of the project rendering the source code more readable without changing the external behavior. Refactoring involves amongst others renaming of source code files so they can project their nature or function more clearly to the reader. This is very important since in large-scale projects there are many people involved and thus it is more likely that the source code is read by more persons than the original writer.

The goal with this thesis is to implement *true renaming* in Subversion. Subversion is an open-source Revision Control system that adapts the client-server architecture. More about Subversion is described in chapter 2.2. In Subversion, renaming of files consists of removing a file and adding another with a different name while with true renaming, the name of a file becomes just an attribute which can be changed without changing the whole file. Another goal with this thesis is to describe and discuss several merging techniques as well as techniques to detect conflicts resulting from merging two artifacts. Merging is the process of integrating the contents of two or more files into one final file while artifacts are units in a project that may be for example source code files, documentation, or design documents depending on the nature of the project. Moreover, approaches to merge reduction are described and discussed.

Since subversion is free of charge and uses well known and proven to be stable components, such as Apache[1], it is considered one of the good free revision control software available. That is one of the reasons why many companies in the software industry, such as ÅF, are interested in it. Some companies in the industry are involved in developing Subversion by giving funds and aiding the developers by providing technical support[2].

[1] `http://www.apache.org/`
[2] `http://www.collab.net/products/subversion/`

1.1 ÅF

This thesis was written and performed at ÅF which is a multinational technical consulting company dealing with amongst others, large-scale software projects.
ÅF believes that Subversion has huge potential in being one of the best free revision control software. Unfortunately Subversion lacks some features that are essential and required by the industry such as a good renaming function, which is essential for refactoring.

1.2 Goals

There are two goals with this thesis. The first is to design and then implement better renaming support in Subversion which will make it possible for developers to run different renaming scenarios without losing information in the renamed files. The second goal is to describe and get a better understanding for different merging approaches as well as conflict detection that results from merging two artifacts.

1.3 Structure of the report

The layout of this paper is divided as follows:

- Chapter 2 talks about RC[3] in general, what it is, how it works, and approaches to design RC systems. Moreover, Subversion is described in details: its advantages and disadvantages, its architecture, and how it is built.

- Chapter 3 discusses the rename problem in Subversion along with scenarios to show where the problem is and what parts of renaming Subversion can handle.

- Chapter 4 covers different merging techniques, conflict detection, conflict resolution as well as conflict reduction. Moreover, the importance of atomicity of operations in RC systems is discussed along with failure scenarios.

- In chapter 5, a solution to the rename problem presented in chapter 3, is presented and discussed.

- Chapters 6 and 7 cover the accomplishments and results of this thesis and the solution presented in chapter 5 is discussed.

- Chapter 8 presents the conclusions for this thesis, limitations to the solution presented in chapter 5 as well as another solution that would solve more of the rename problem in Subversion.
  In appendix A, several abbreviations used in this report are explained.

[3] Revision Control

Chapter 2

Revision Control

In this chapter RC is explained and discussed along with two approaches to design a RC system. Moreover Subversion's architecture, which is a centralized RC system, is explained and discussed.

Before talking about revision control one should clarify what a revision is. A revision, also called version, is an instance of an artifact at a certain time point. Imagine how articles are written and published in newspapers; First a reporter writes the initial draft of an article. This draft is then passed to an editor which may add, change, or perhaps remove some parts of the draft thus creating a *revision* of the original draft. This revision is sent to a final editor who may still find errors and correct the article creating a new revision before sending it for publication.

Revision Control is the management of multiple revisions of the same information unit (artifact). As with the article example above, during the development of a project, artifacts are constantly added, removed, and edited in the project. RC systems store and manage multiple revisions of project artifacts and thus preserving the evolution of these projects. Moreover, modern systems provide sharing capabilities and support for simultaneous editing thus making them perfect with large-scale projects. RC systems are usually used as a stand-alone tool but may also be embedded in Software Configuration Management (SCM). SCM is a "set of activities designed to control change by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms for managing different versions of these work products, controlling the changes imposed, and auditing and reporting on the changes made" [27]. In other words, SCM is a methodology to control and manage a software development project. Embedded RC in SCM is the tool that provides software control. Software control is the discipline to save information about each artifact: What changes are made to it, by whom, and when. In that manner, software evolution is preserved. As software projects become larger in time, the need to manage their evolution consequently increases.

2.1 Revision Control Systems: Structure and Architecture

RC systems save and control artifacts (code files, documentation, test results, etc..) and the changes made to them in a *repository* (or many depending on the architecture of

the system, see next sections). Moreover, developers may edit, add, or remove artifacts from the repository. The repository may be saved as a set of files on a normal filesystem (Subversion/CVS) or in a database (Subversion with BerkeleyDB support). When an artifact is created, it is given a *version number* which is usually global to all other developers. If the artifact is edited, it is given a new version number and the differences made to it are saved either as a new artifact or as a *delta*. A delta is the difference between two successive versions of an artifact. Saving the new version as a new file is easy yet impractical in large projects since all files must be saved when edited and since the number of changes made to artifacts increase with the number of developers, a lot of disk space is required. Saving changes in deltas saves a lot of space since only the changes are saved instead of the whole artifact. Moreover there are several tools, that can generate a delta, such as the UNIX diff [5] command, thus it is easy. Deltas can be backward or forward deltas. When using backward delta, the newest version of the file is saved as a whole file while all previous versions are saved as delta. With forward deltas on the other hand, only the first version of an artifact is saved as an artifact while all newer versions are saved as deltas. With deltas, the evolution of software projects is preserved since an artifact can be restored to any time point by simply applying one or several deltas to the original artifact. Moreover deltas are more practical than saving a whole new file since in many RC systems, changes are sent to remote developers and where network speed is not always optimal, sending a delta, which is smaller in size than a whole artifact, is more optimal.

There are two approaches to design a RC system: *centralized* and *distributed*. In the Centralized approach, all users have a working copy that they work on, and this working copy, known as *WC*, can be integrated to the repository (using different merging techniques). The distributed architecture on the other hand uses several repositories which are synchronized. These two approaches are described in details below.

2.1.1 Centralized Revision Control

The centralized model of a RC system uses the server-client architecture (see figure 2.1): First a repository is created (server side). Information may be put in the repository, which other users (clients), may then check-out into their WC (client-side directory). Moreover, clients can make changes to their WC and then integrate their work with the repository. The WC and repository may sometimes be a copy of the other and in worst case, they might differ completely (when the users for example changes everything in the WC). This model seems to be the perfect one for a RC system and in some cases it is. Sometimes, certain projects need to be in a centralized repository where the progress is easily controlled since the server manager can control which user may do what. For example, in some open source projects, all users are allowed to *check-out* a project. But when integrating their WC into the repository, users must have special permissions to do so.

It is generally easier to have the information in one spot where all users have easy access to it rather than having the information scattered in many repositories. In other words, one server is easier to maintain. This is, on the other hand, one of the disadvantages in centralized RC; the server communication becomes the bottleneck. In order to be consequent, one has to commit his work quite often. Committing regularly is a must when many users are working on the same artifact because it makes merging later on easier. Moreover, one can supply information (in the *log*) that describes every significant change which in turn makes it easier for the other developers to understand that change,

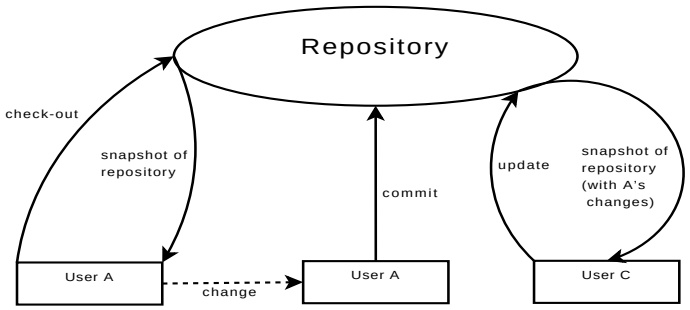

Figure 2.1: A centralized RC system model

why it was made and by whom.

RCS [29] -The software- was one of the first centralized revision control software widely used and it consisted of UNIX commands. Users could check in artifacts, store multiple versions of an artifact, and later retrieve the artifacts. Today there are many centralized RC systems that are dominating in the market, such as Perforce [10], which is a commercial application, and Subversion(see chapter 2.2) which is an open source project.

2.1.2 Distributed Revision Control

In the distributed model of a RC system (figure 2.2), all users' WCs act as repositories instead of a sandbox. There is no central repository that users can synchronize their WC with. Instead, each user can make changes into his local repository and then publish the change sets so others can *pull* these changes and then synchronize their repositories. This has both advantages and disadvantages. Since the repository is local, the user does not have to be connected in order to commit the changes. Instead, the user may commit locally and when connected, the user may publish his changes so others can pull them. Since users might be geographically spread, connection quality is not always the same for all users or not even available for others at a certain time. Thus the distributed approach is suitable for such cases because users may connect to a server where connection conditions are better. In other words, not all users are bound to commit to a central server.

On the other hand, this can lead to problems: Since the RC system is most likely to be used by many users, and since there is no central repository that *a*ll users can synchronize from and since the number of repositories increases as the number of users increases, there will likely be more than one copy of the core project itself. This will lead to the project having different states if not all repositories are synchronized properly since the complexity of synchronization will increase with the number of users. Moreover, there will be more overhead since all repositories must synchronize every time a change set is available. Moreover, the distributed approach may encourage users to take on a different direction of a single project. That is, developers may disagree on a certain bug fix or feature, and can end up with working on two or more versions of the same project each with the respective group's terms.

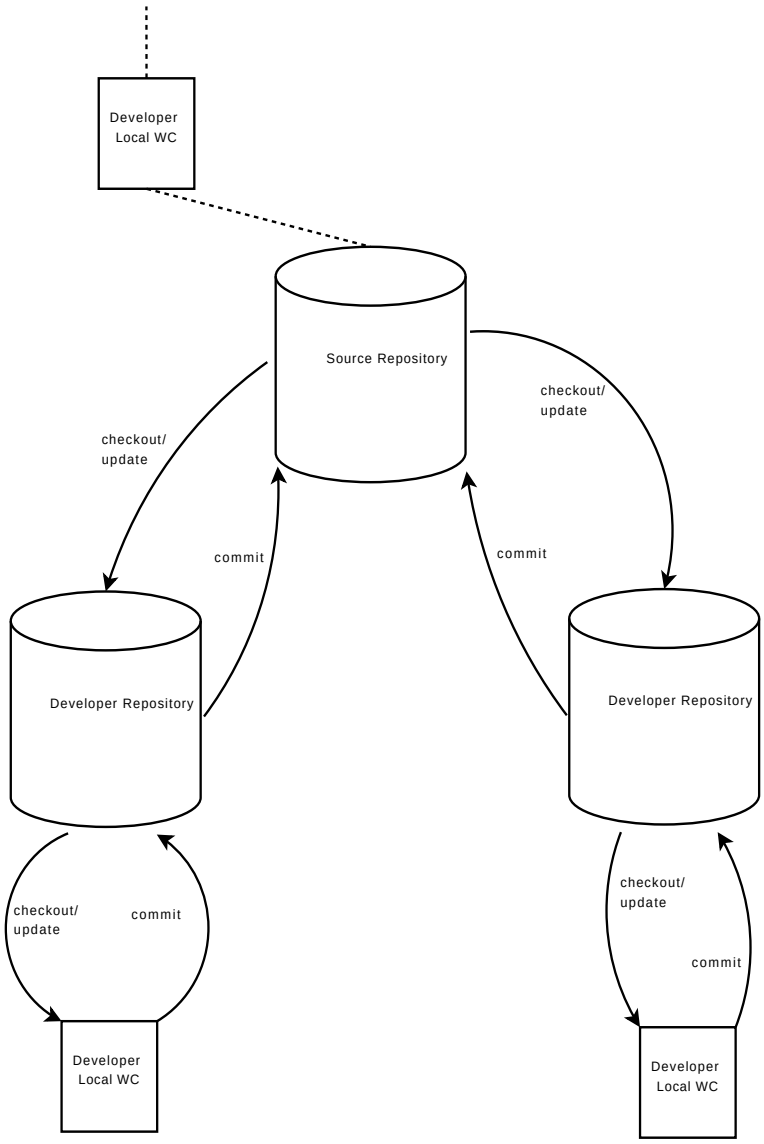

Figure 2.2: A decentralized RC system model

2.2 Subversion

In this section, Subversion's architecture, features, and components will be examined closely.

Subversion is an open-source centralized RC system that uses many well developed and well known applications such as Apache. It is very close to CVS [2] but has far more features and well defined structure. Since CVS was widely used due to its simplicity, Subversion gained popularity because it was similar to CVS in its simplicity yet provided much more features and was more stable than CVS.

2.2.1 Architecture

Subversion uses layered libraries (see figure 2.3). Moreover, it supports both offline and online operations since most operations are done locally on the user's WC. The only operations that need network connection are the `update` and `commit` operations.

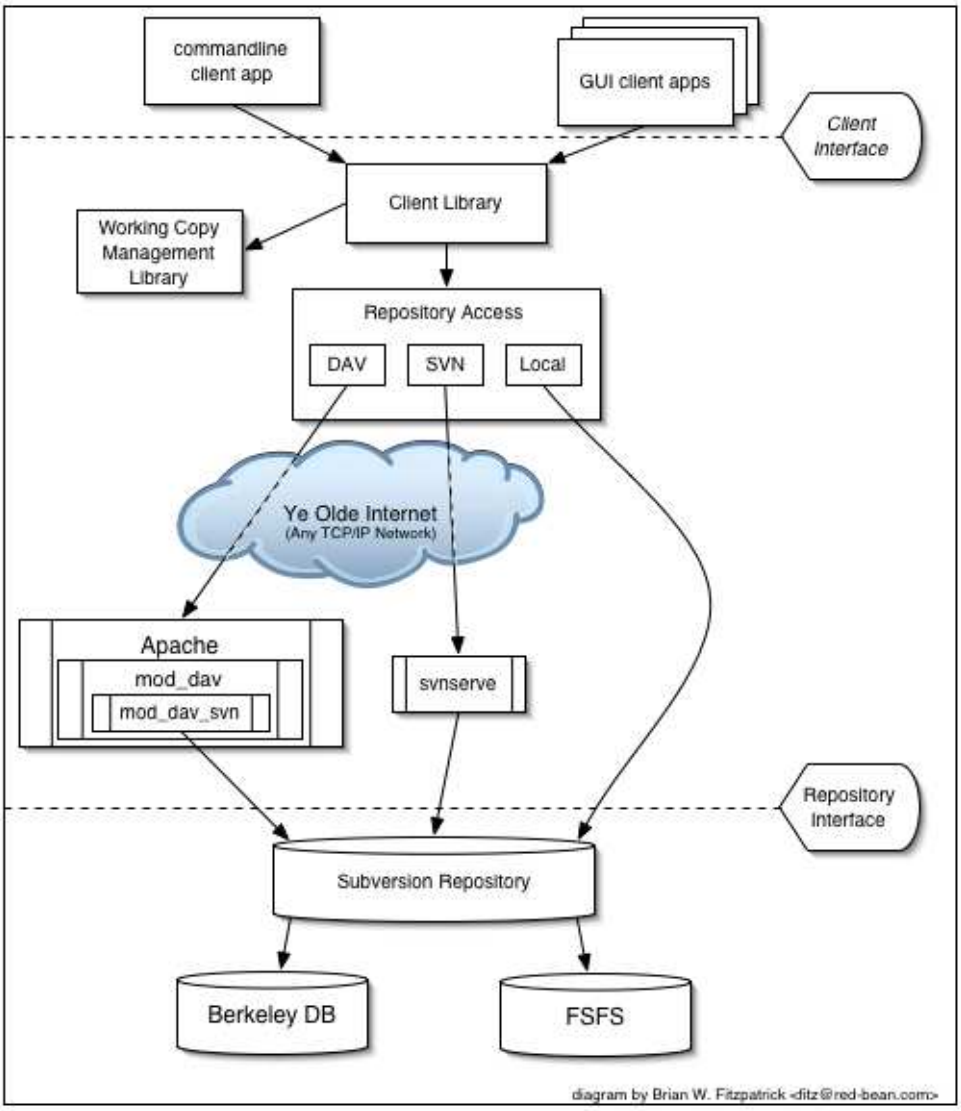

Figure 2.3: Subversion: Architecture

There are three main layers in subversion [26]. Each layer is described below.

1. **Repository Layer**
   In this layer, the subversion's file system is implemented. It provides mechanisms for storing the *versioned* files as well as mechanisms for reporting.

The subversion file system is not at kernel-level but rather a virtual file system. Kernel-based file systems such as FAT32 or ext3 are two-dimensional; the first dimension is directories and the second is files in these directories (see figure 2.4).

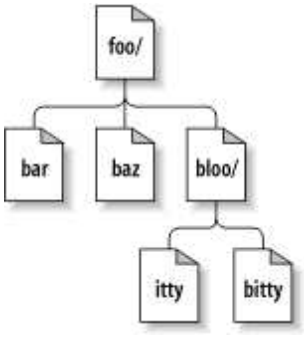

Figure 2.4: A kernel-based file system model

Subversion's file system on the other hand is three-dimensional. That is, the first two dimensions are as a normal kernel-based file system, and the third dimension is for versioning (see figure 2.5). In other words, there are directories and files and then there is the time dimension (keeps information about the state of files and directories at different time points). But that does not mean that subversion stores a copy of each file at every time point (snapshot). Instead, it stores the original file and then stores *deltas* of the file in the time dimension. Delta is the difference between a file at time t and at time $t+1$. In this manner, a lot of disk space is saved especially in the case of big projects where there are many developers committing.

Instead of storing files and directories, the file systems uses BerkeleyDB [1] or a flat-file representation. The file system API provides essential functions for adding, removing, and renaming files and directories. In addition to that, the API allows users to modify file meta data (properties). Whenever a user changes the directory tree, the file system remembers the structure of the tree prior to change so it is easy to restore a state of the tree to any time point.

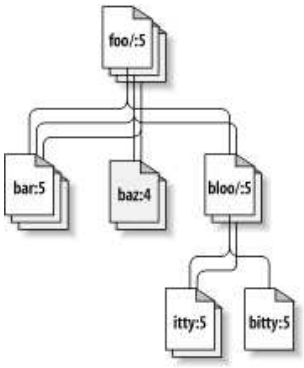

Figure 2.5: Subversion's file system model

This layer is connected to the *Client Layer* via the *Repository Access Layer*.

2. **Repository Access Layer**
   This layer acts as a mediator between the *Repository Layer* and the *Client Layer*, marshaling data between the two. It includes modules that shape up the connection protocol between the client and the repository. Since it is modularized, it is easy to implement any communication protocol using existing network functions provided by this layer. Each installed module publishes a list of protocols it *speaks* so the *RA*[1] loader would know what protocol to use in the task at hand. There are many implemented *RA* modules in subversion that makes it possible to access the repository in many different ways, such as `file://` for local access, `http://` and `https://` for network access, and even `ssh://` for secure access.

   The API exported by this layer provides functionality for sending and receiving versioned data to and from the repository.

3. **Client Layer**
   The core duty of this layer is to provide support and operations on the client side WC (working copy) such as propagating information to and from the *Repository Access Layer*. It has libraries, such as `libsvn_wc`, that are directly responsible for managing data in the WC. In order to manage data at the WC, subversion stores administrative files in a directory called `.svn` at each directory and subdirectory in the WC. These administrative directories contain information about each file such as name, change date, author. Moreover, prior to a commit, subversion keeps information about each operation done on files such as modify, delete. If the file is modified, a delta is included, which shows what changes are made to a file and where. When the developer commits, this information is marshalled into commands which are then submitted to the Repository Layer. There, operations are performed and a feedback is sent to the client layer. This provides offline functionality; prior to a commit or update, the developer may work offline, adding changes in the local directory, and consequently, the server is only contacted when the developer commits or updates. This prevents congestion at the server side.

   In addition to the above mentioned functionalities, this layer provides highest-level API to any application that wishes to use general revision control operations. For example, calling the function `svn_client_checkout()` takes an `url` as an argument, and then contacts the repository (opening a session if necessary) to retrieve the tree at the specified url into the WC, writing all administrative information in that WC.

2.2.2 Limitations

Subversion has its limitations. For instance the lack of merge-tracking support makes developers involved in bigger projects reluctant to using Subversion since a lot of manual book keeping has to be done. Moreover since refactoring has become more important, renaming, amongst others, has become an important feature that a RC system must have. The lack of true-renaming support in Subversion pushes developers of bigger projects farther off from using it.

[1] Repository Access

Chapter 3

Problem Description

The practical part of this thesis deals with the problem of renaming in subversion. Originally, Merge-Tracking support for subversion was to be implemented but that changed due to the fact that the open-source community that is responsible for the development of subversion laid plans to implement merge-tracking support and the implementation part is almost over. One issue that the community did not address was renaming.

Although renamed files and directories retain full revision history in subversion, renaming still causes problems in some scenarios. In this chapter, the problems along with the scenarios causing this problems will be looked into and discussed thoroughly in order to show what the problem is, how it happens, and where it happens.

3.1 Renaming in Subversion

Renaming in Subversion consists of two operations: *Copy* and *Delete*. This may be found to be inappropriate because of the fact that the renamed file would then be a copy of the original file with a different name and not the same file with a different name. That is quite true, but nevertheless, this works fine in most cases because Subversion keeps the properties of the *renamed* file and copies them as the new file's properties (except for the name). Below are some common rename scenarios that happen quite often when dealing with bigger projects. The first two scenarios have two developers (A and B) working on the same developing branch (trunk). When several developers are working on the same branch, each actually works on a *checked out* version of the branch. All changes prior to a *commit* command are stored in the respective developer's working copy. After committing, the developer's working copy is synced to the main repository where changes are available for other developers to *update*. After a commit, a new revision is created. In the below scenarios, all changes are made locally until a commit or an update is called. Note that the time axes are directed from top to bottom in the first two scenarios whereas in the other scenarios, it is directed from left to right.

The first scenario (figure 3.1) shows that A checks out the trunk directory from the repository and then modifies a file (`foo.txt`) and then commits that file back into the repository. In other words, A checks out a *revision* of the trunk, modifies that revision, and then commits it back into the repository so that the repository now includes the changes made by A. B starts by checking out the trunk directory containing `foo.txt`(containing A's modifications). B then adds another line (*WORLD*). After that B commits his changes. A tries to commit his changes, after having modified the file

adding a new line (*CRUEL*), but fails to do so because subversion detects that a newer reversion is present at the repository (B committed before A did an update) so A has to do an update before a commit. When A updates, he gets B's line (*WORLD*). Immediately after the update, A commits the file. At the same time, B is updating and he gets A's version (*HELLO CRUEL WORLD*). Now both files have the same content, but B suddenly decides to change the name of `foo.txt` to `bar.txt` and then commits the change. A does an update, and `foo.txt` in A's WC changes name to `bar.txt`. Now both files in both users' WCs have the same content and file name.

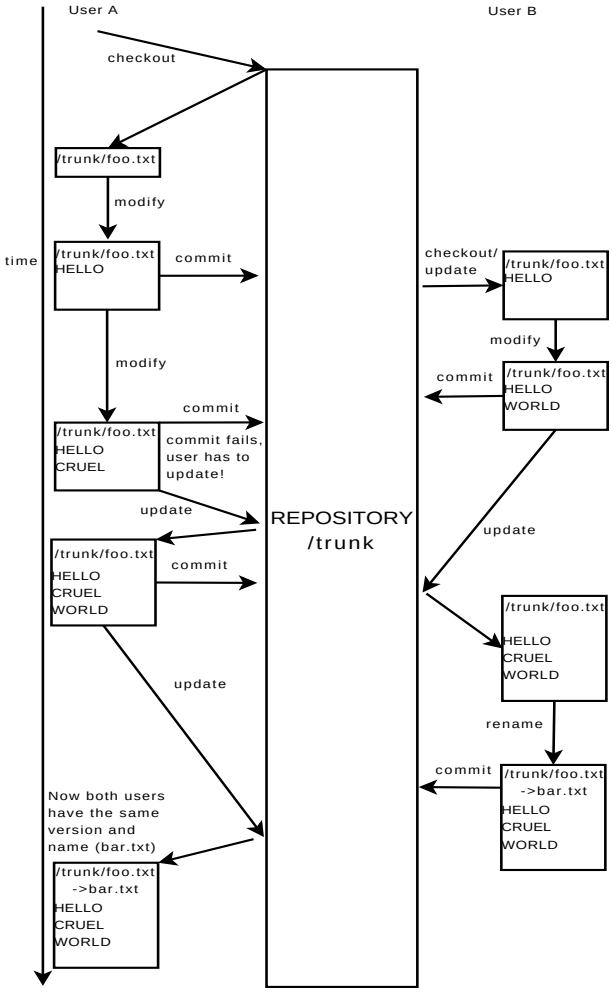

Figure 3.1: A rename scenario where Subversion handles the problem

In the next scenario (figure 3.2), users are not as lucky as in the previous one. This scenario starts in the same way as the previous one. Both users have the file (`foo.txt`) with content (*HELLO*). The difference here is that both users commit less often; after having added a line to `foo.txt`, B does a rename directly without committing. A, on the other hand, adds another line and does an update after B has committed. This is where we see subversion failing during renames. One would logically assume that what happens is that A would get B's updates (both rename and file update) so that the

result would be similar to that in the first scenario. What really happens here is that A gets a file called `bar.txt` with 2 lines (*HELLO* and *WORLD*). Moreover, `foo.txt` is excluded from version control (i.e removed from the repository), yet it still exists as a normal file in A's WC. Thus the *lost update* problem occurs because A's modification, *CRUEL*, is not included in the new renamed file. One might argue that this is not a big problem because it is obvious that a new file, `bar.txt`, replaced `foo.txt` and one can merge the two files together thus preventing the lost update problem. This is true if there are few files in the repository, where it becomes obvious which files are new and which are not. Often, this is not the case. Revision control is a must for bigger projects including not only files, but directories and subdirectories.

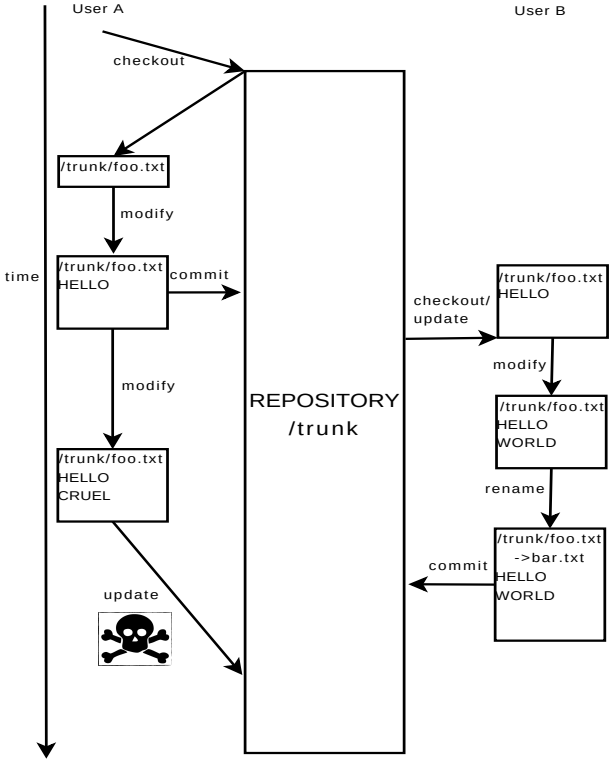

Figure 3.2: Subversion fails in this rename scenario

The next scenario is similar to the previous but with three developers A, B, and C. B and C each perform a rename in their WC and A performs a change to the file. Assume that the repository initially contains `trunk/foo`, that is a directory `trunk` and a file in it (`foo`), at start. Note that the arrows whose directions are horizontal, are local operations which means no connection with the repository.

Subversion fails totally in this scenario since it does not know that all renamed files, `bar` and `baz`, are actually the same (`foo`). That's why at the end, two files, `bar` and `baz`, reside in the repository while `foo` and its changes are ignored. The expected result should be one file named `baz` (since C was the last to commit) with A's changes merged in it.

The next scenario is about renaming in different branches and then merging these branches. Assume that from the beginning there is `trunk/foo` in the repository. Assume

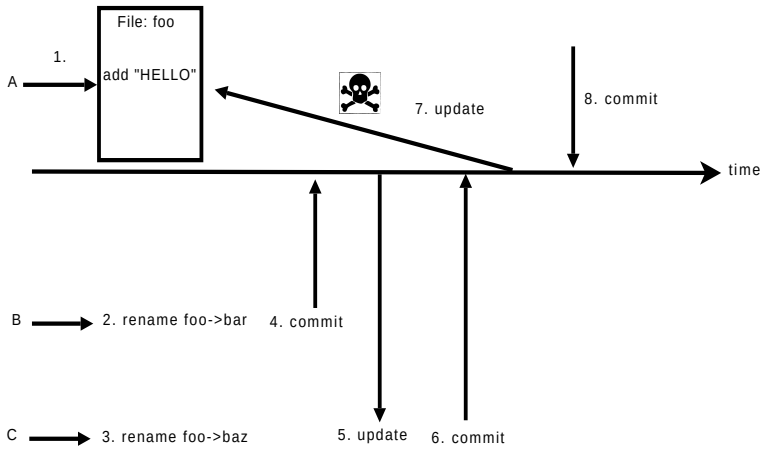

Figure 3.3: Subversion fails again in this rename scenario

further that two developers A and B are working on the repository. B starts by branching `trunk` to (`branch A`). A branch is a usually a copy of a directory in the repository, in this case `trunk`. In this case, when B creates a branch, he gets the "directory" trunk and its contents (in this case a file `foo`) copied to another directory (`branch A`). Now B has `Branch A` in his WC while A still has `trunk` in his WC.

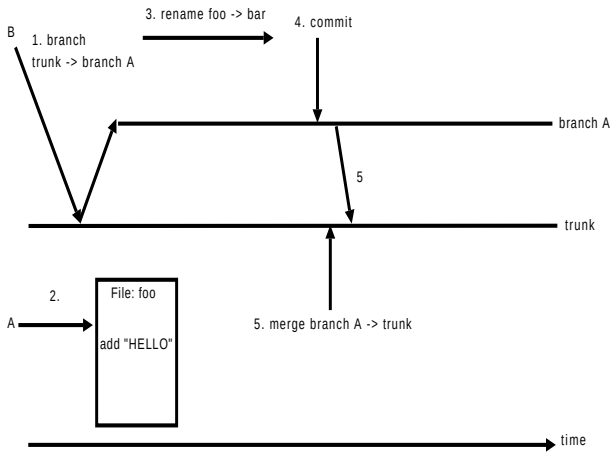

Figure 3.4: Subversion fails in the cross-rename scenario

Note that in step 5, the merge command is called from A's WC which is a "trunk wc". The result is that `foo` along with the updates (`HELLO`) disappear from both A's WC and the repository. The result should have been the renaming of `foo` (in the trunk) to `bar` with the changes made by A (`HELLO` token).

3.2 Problem Statement

Refactoring is a "series of small steps, each of which changes the program's internal structure without changing its external behavior" [14]. Refactoring in other words, as the title of the book suggests, is to enhance the structure of existing projects rendering the code more readable. This is very important because the code is likely to be read by more people than the writers.

As refactoring has become an important factor in software development, renaming and merging become necessary and important tools in revision control software. Refactoring includes amongst others changing the names of files in order to give the code reader better understanding of what the file represents. Since in modern development the code is changed more often than it is written, and since renaming is an important part of that, revision control software are bound to have good renaming capabilities in order for software projects to advance and for developers not to *step on one another's toes* during development. Subversion, as mentioned in section 2.2, is considered one of the good free RC systems available. The reason to why it is not best is amongst others the lack of a *good* renaming feature which renders it less favorable among developers when dealing with important and complicated projects. That is because the developers then have to spend a lot of time tracking renames, and dealing with bigger projects, that is almost impossible.

Chapter 4

Merging in RC systems

This chapter represents the theoretical part of this thesis. As the title reveals, the in-depth study will deal with different merge techniques, algorithms, conflict detection, and approaches to conflict reduction. Moreover, approaches adapted by some RC systems will be discussed to give insight about solving the rename problem in subversion.

The merge tool is perhaps the most important tool in a RC system. As developers work concurrently on a software artifact using optimistic RC, their work has to be properly merged. This is actually the cost of using optimistic RC. With pessimistic RC, developers do not have to merge their work since only one developer, at a time, may work on an artifact. This approach though hinders parallel development.

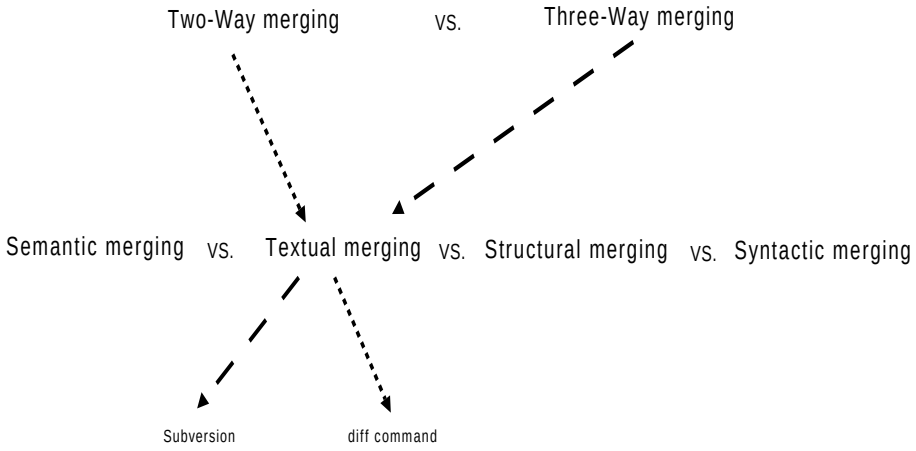

Figure 4.1: Merging approaches

4.1 Merging Techniques

There are many merging techniques and algorithms, some of which are easy to implement and suitable for certain purpose while others are more general and robust. Below are the common techniques that most RC systems use. Figure 4.2 shows a common scenario of two users (a and b) working on the same file, whose work is then merged. One

talks about using three-way or two-way approach with textual, syntactic, semantic, or structural merging (see figure 4.1). Subversion for instance uses three-way textual merging, which means that it uses two revisions of an artifact along with a common ancestor to compare *textual* differences between those revisions. The UNIX *diff* [5] command on the other hand uses two-way textual merging which means that two files are compared for textual differences without comparing them to a common ancestor.

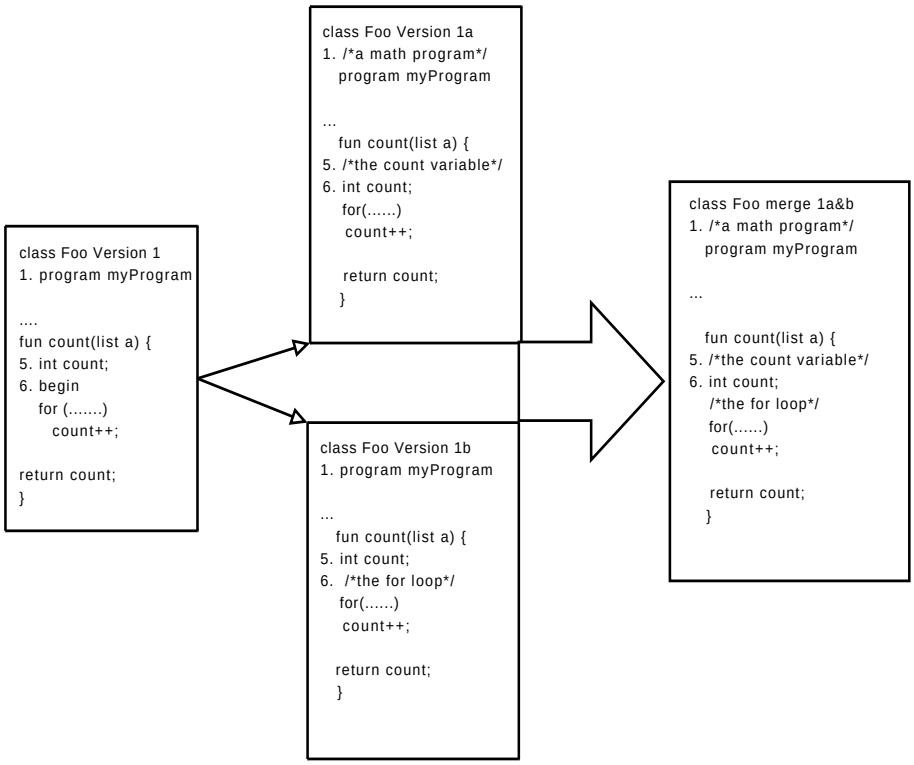

Figure 4.2: A possible merge scenario

4.1.1 Two-Way Merging

In this approach, two files are compared, line by line, for differences and then merged together. This is similar to the UNIX *diff* utility. Two-Way merging does not depend on a common ancestor from which two files are originated. Unfortunately this approach is not robust enough to detect which changes are done by who. For example in figure 4.2, this approach would not be able to tell whether (line 5) in `Foo version 1a` is added in 1a or removed in 1b, or even modified by both in the final version (`Foo 1&b`).

4.1.2 Three-Way Merging

With three-way merging, two files are compared along with a common ancestor from which these two files originated. Information form the common ancestor is used during the merge process to determine what information is added/removed/modified by who. This approach is more powerful that the previous one in the sense that more conflicts

can be detected. This is similar to the UNIX *diff3* [6] utility. This technique is used in most commercial RC systems. For example in figure 4.2, line 5 in version a is a comment, while in b it is (`int count;`). Looking at the common ancestor, line 5 is the same as in b, which means that in a, another line, the comment, was added.

4.1.3 Textual Merging

In textual merging, software artifacts are considered as text files. Lines in these text files are the smallest *merge-entities*. That is, lines in *text* files are compared for addition, deletion, or modification. This approach however is not fine grained in the sense that modifications in the same line cannot be handled easily. For example, if two files have the same line number and contents such as : "This is a comment" and "This is a good comment". These lines can be logically merged into a single line: "This is a good comment", but text-based merging is unable to identify the differences within a single line; only a single change can be selected, but not both. The user has to manually merge these two lines.

Despite the disadvantages, text-based merging is very useful because of its efficiency, scalability, and its satisfactory accuracy. Moreover, according to a study [25] performed on a three-way text-based merge tool, about 90 percent of the changed files could be merged without asking any questions, that is the files could be merged automatically without human intervention. That's why three-way text-based merging tools are commonly used in commercial RC systems.

4.1.4 Syntactic Merging

Text-based merge tools often give raise to unimportant conflicts, such as a line break or a comment. With syntactic merging, the syntax of the software artifact is taken into consideration and thus a conflict is reported *only* when the syntax of the software artifact is defected. Moreover textual merging tools focus mainly on software systems at the source code level which makes it harder to detect inconsistencies at higher level of abstraction [23]. That's what makes syntactic merge tools more powerful that text-based ones. General syntactic merge tools can be categorized according to their underlying data structure: parse trees or graphs. An example of tree-based merging tool can be found at reference [11] where programs are divided hierarchically into classes, procedures, and functions. Each program is represented as a tree where nodes (leaves) contain the actual data and the path to that node represents the type (class, procedure, etc...). Another example of tree-based merge tool is the *Cdiff* [18] tool which uses parse trees to find differences between C++ programs.

4.1.5 Semantic Merging

Semantic-based merging takes the semantic of the software artifact into consideration. Some frequent conflicts can not be detected with syntax-based merging. Consider the merge scenario in figure 4.3. Two developers, A and B, working simultaneously on a software artifact (`Foo`). A decided to change the variable name `count` into `cnt` for shortness. B on the other hand decided to change the function `fac` into a procedure. Moreover he decided to add another variable `arg` to optimize the "procedure". A possible merge result is represented in `Foo version a&b`. This version is syntactically correct. However there is a semantical conflict; the "`if (arg==1)`" statement is added to the result, but the used variable `arg` does not exist in the final merged version and thus

it is semantically incorrect. Tree-based merge would not be able to detect this conflict while graph-based approaches like [21] would because a graph representation maintains an explicit link between a definition and its invocations thus making it easy to detect incompatibilities between them.

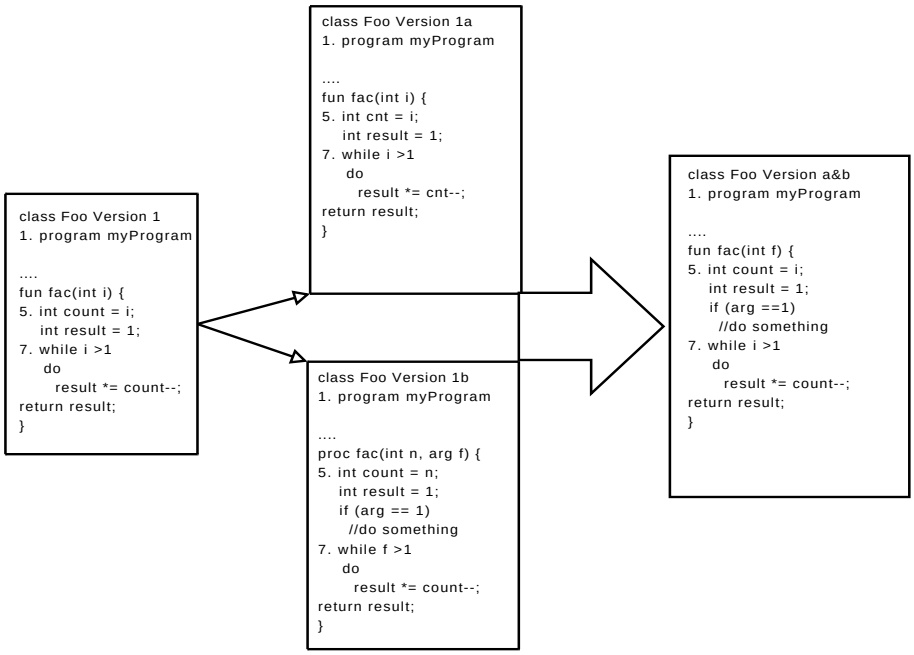

Figure 4.3: A possible merge scenario..2

4.1.6 Structural Merging

Refactoring and restructuring are important techniques that prevent software aging and erosion. As explained in section 3.2, refactoring is property preserving, i.e it does not change the behavior or rather the semantics of the software. However, the structure of the software is changed significantly. Structural merge conflicts arise when the change done is of a structural character and the merge algorithm does not know which structural changes are to be applied. As an example of such a conflict, consider the dilemma in figure 4.4. The first class, *Vehicle*, consists of three subclasses *Car*, *Lorry*, and *Bicycle*. The first user chooses to insert a new subclass, *Motorcycle*. The second user chooses to refactor *Vehicle* in order to categorize it. A structural merge may arise when another class, *Motorcycle*, is to be added: Should it be added directly to vehicle or should it be added as a subclass to *Motor*? It is easy to determine that it should be added as a subclass to *Motor* but it is not easy to determine that if the class to be added is a *Tricycle*. These kinds of merge conflicts must be solved manually. Unfortunately, in most commercial merge tools, there is no support for structural merge. Moreover, it is not easy to design such tools due to the nature of the conflicts since those conflicts are

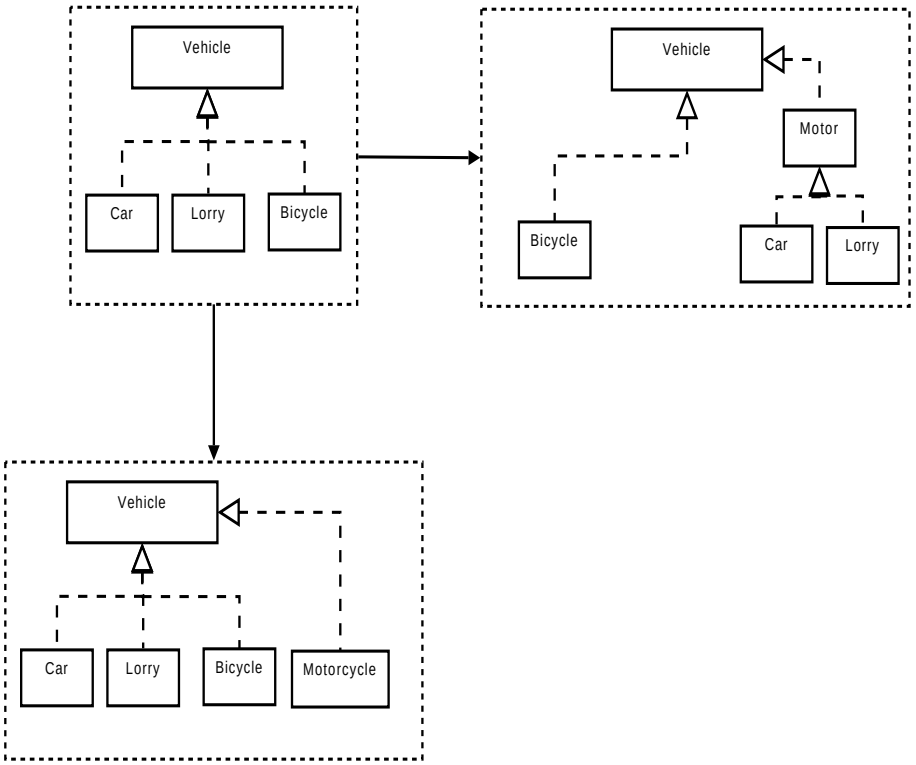

Figure 4.4: Structural conflict

implicit.

4.2 Merge Conflict Detection

Most RC systems are only able to detect the most simple types of conflicts: physical changes, that is changes made on top of other changes [25]. Physical changes are not the only ones, there might be changes that indirectly conflict with each other. For these, more advanced tools that use software analysis are required. In this section, several conflict detection approaches and tools will be discussed.

Using operation-based merge, conflicts can be detected in a more disciplined manner by comparing the modification operations that have been applied by different parallel developers. These conflicts may be stored in a so called *conflict matrices* or *conflict tables*. Then it is easier to know if there is a conflict by doing a simple *look-up*. [13] suggested that instead of storing the conflict themselves, they should be analyzed to determine what changes to specification properties they induce and store that information instead. A more generalized approach is suggested by Mens [21]. He suggested that all software artifacts are to be represented as graphs while software evolution is to be represented by graph rewriting. Moreover, type-graphs are to be used for domain-specific constraints that have to be satisfied by all graphs in a particular domain.

Conflict sets can be used as an alternative to merge matrices. According to Edwards [12], a conflict set is "a group of *types* or *classes* that have the potential to

generate conflicts with each other". He suggested to group potentially conflicting combinations of operations together, based on the application-supplied semantics. These combinations are defined manually and are static. That means that they remain the same until the semantic of the software is changed. This approach addresses scalability because the number of operations is limited to the set of conflicts.

There are also some semantic conflict detection techniques. Horwitz [15] was the first to develop a powerful algorithm for merging program version without semantic conflicts. The programming language used was assignment-based. He used dependence graphs and program slicing [16]. Program slicing is a decomposition technique that extracts from program statements relevant to a particular computation [31]. This algorithm however has its limitations. For example if someone, in a version, changes the way the computations is made without changing the input/output, and someone else merely uses the results of the computation, a conflict will be issued. Moreover, this approach is not general because it is restricted to a certain programming language. Another *more* general approach is the *Semantic Diff* [17]. This tool, used mostly with the *C* programming language, is a two-way merge tool that uses local dependence graphs to identify the differences between two procedures by monitoring their input/output behavior rather that their syntactic behavior. It can also be applied to other programming languages though its support for object-oriented languages is trivial or rather non trivial to apply due to the presence late binding and polymorphism.

4.3 Merge Conflict Resolution

Now that the conflicts are detected, they must be resolved. There are many ways to resolve conflicts depending on their type and the level of accuracy. Some of which must be resolved manually, which is often costly in terms of time and complexity. However, most RC systems provide interactive ways to resolve conflicts by using graphical user interfaces to help the developer to see the conflict, by showing markers and versions of the software artifact in which the conflict exists. An example of such RC system is *Perforce* [10], which is a centralized having a user friendly merge-tool showing, graphically, three versions of the artifact, two of which are the versions to be merged and the third is a possible merge result. The developer may then accept the merge result or perform the resolving manually. Markers are often used to show where conflicts exist, even with no graphical interface, markers are inserted in the software artifact. Such is the case with Subversion, if one is not using a GUI-client. An example of a conflict that must be solved manually is when a procedure/function is renamed differently in two versions. The merge algorithm cannot decide what rename to choose, that's why manual intervention is needed.

On the other hand there are some automatic strategies that the merge tool could use. For instance, in a case of rename. Renaming is a global operation since it may affect many artifacts. That why, when there is a renaming, the parallel operations must be merged *before* any renames to insure that the rename is applied to all changes as well [20]. Another automatic strategy that may be used is when there are two identical operations performed on two versions, the merge-tool may then discard one of the operations.

Edwards [12] suggested a resolution strategy called "The Explosion Strategy". This strategy *forks* branches, or rather calculate possible outcomes of resolving a conflict and present them to the developer. In other words each action, that caused the conflict, is removed in a branch and the outcome is presented to the developer. Another strategy

suggested by Edwards is the "*Promotion Strategy*" where actions causing a conflict are replaced by equivalent actions that do not depend on previous actions and consequently would not cause a conflict. For example in a draw program, where there are actions such as *Draw, Delete, Copy,* and *paste*. If a sequence of actions, such as

$$Draw\ A \rightarrow\ DeleteA \rightarrow Copy\ A\ \rightarrow Paste.$$

Obviously step 2 makes a conflict with step 3 and 4 because after deleting a figure, there is nothing to copy or paste. Thus the strategy *promotes* a stand-alone action, such as *Draw A'* which does not depend on the previous action and thus the conflict is resolved. This strategy on the other hand is not effective in all cases. Another strategy, *The Recursive Acceptance Strategy*, provides an obvious option to resolve conflicts and that is to remove all inconsistent operations. After removing such operations, new conflicts may arise and they can be resolved recursively. An effective strategy suggested by Edwards is the *Quantum Uncertainty Strategy* which allows the use of inconsistent operations. These operations are given a risk value which is recalculated every time the developer chooses to discard/use these actions.

Munson and Dewan [22] propose, among others, two strategies: *Consolidation* and *Reconciliation*. In consolidation merge, it is assumed that each of the two revisions to be merged is a version of a common revision (three-way merge). Moreover, the conflict between changes are assumed to be complementary. In case of overlapping changes, one change is chosen interactively while in the case of deletion, the merge proceeds automatically. In reconciliation merge on the other hand, it is assumed that the reason for merging is to solve conflicts. This is used when it is known that developers change the same piece of code. In the opposite case, i.e when the changes do not overlap, consolidation merge is to be preferred.

4.4 Approaches to conflict reduction/avoidance

A significant problem with software evolution is that small changes can have large impact [30]. This is known as the *ripple effect* [32]. This means that merge tools often give rise to a lot of conflicts being reported. There are certain techniques to reduce that abundance of conflicts.

Asklund [19][11] suggested the approach of fine-grained version control to keep change of fragments as small as possible. In this way, conflicts will be reduced in the future because the changes are minor. Moreover, if conflicts still exist, they will be easier to resolve. He also suggested that the developers are to be aware of each others changes. In that way, unintentionally made conflicting changes are minimized. Merging will be consequently simpler.

Another way to reduce conflicts is by using *information hiding*. This is true on the basis that "changes that do not change the interface of an artifact have no further influence beyond the realm of the artifact itself" [28]. Otherwise this is not sufficient since changes often have effects beyond the imposed encapsulation boundaries.

Mens suggested the use of *graph rewriting* [21]. By representing changes in object-oriented software in graphs, merging becomes easier. Consequently conflicts are easy to detect and resolve.

Graph partitioning [21] may be used to detect local inconsistencies first and global inconsistencies later. By representing software artifacts and their dependencies in graphs, all the nodes in the graph can be partitioned according to their number of dependencies.

As a result, conflicts in certain sub-graphs (having lowest number of independencies) are solved first. Then conflicts between sub graphs are solved to obtain global consistency.

On the other hand, a lot of conflicts, that do not affect the semantic or the syntax of the software artifact, are reported. Example of such is a change in a comment or a line break. One *trivial* way to reduce such conflicts is ignore or rather turn off the detection of such conflicts since they do not change the semantics of the program. Alternatively, the merging of comments may be done by *force*, that is, the comment in the new version is to replace the comment in the old one. In such cases, the developers may have to agree on a common way to write comments. As known, commenting in programming languages is not done in the same manner but on the other hand there are relatively few ways to write a comment in a programming languages so this approach may be implemented in a more general way.

4.5 Atomicity of operations in RC systems

In this section the importance of atomicity is explained along with the importance of RC systems adapting it.

Atomicity is one of the ACID [24] transaction properties in database systems. Since in most RC systems, versions are stored in a database-like structure, atomicity is an important or rather necessary feature that contributes to the completeness of the RC system. Imagine that a commit transaction between a client and a corresponding repository server fails; some corrupt data may be written to the repository which can sabotage it. This may be devastating in terms of losing valuable information and causing disconnection with many developers. The data may be recovered since most users *would* have an updated snapshot of the whole repository but nevertheless certain users may have different *views*(branch, tag) which renders the recovering of the *whole* repository a complex and time consuming operation. Furthermore, since users will have to wait until the repository is recovered (up and running) in order to commit their changes, the collaboration degree is reduced. A more complex and dangerous problem is when the repository behaves as if it was complete, where in reality it is not. Developers then have no problems in contacting the repository. When they commit, this will further corrupt the repository. Recovering the repository becomes then a very complex operation, if not impossible. This may be less harmful if it is a distributed RC system since there is no *single* repository that all users commit to. On the other hand with a decentralized approach, certain repositories may not be up-to-date with each other.

Subversion is one of the RC systems that has truly atomic commits which means that either the transaction is performed completely, or not at all. When a commit operation is performed, first the data is copied to a temporary location at the client side in order to make sure that the data is not corrupt. After that the data is copied to the server side at a temporary location. Then the data is copied to the repository, after making sure that the data is not corrupt. In such manner, the data is checked at different levels to make sure it is not corrupt which minimizes the risk to corrupt the repository.

Chapter 5

A Solution

In this chapter, different approaches in open source RC systems that solve the problem mentioned in section 3.2 are discussed. Moreover a solution to the problems described in chapter 3 is presented.

In order to get a deeper understanding and make use of the knowledge of solving the rename problem, several approaches adapted in some open source RC systems will be discussed in the next section. Moreover the "*update*" command in Subversion is explained since it is the point where Subversion fails to address some parts of the rename problem.

5.1 Renaming in open-source RC systems

There are many RC systems available and each has its advantages and disadvantages. Those who handle renames have different approaches to solve the problem. Subversion handles some parts of the rename problem (see section 3.2) but not in all cases. These cases are described in chapter 3.

GNU arch [8], which is a distributed open source RC system, handles the rename problem in a very good manner. Each file is given a unique ID, rather than a name. Unique ID is a better reference than file name since the name of the file is not stable though out the development life span. That's why files can be merged across branches even if they have a different file name and their history remains preserved. Monotone [9], also an open-source RC system, seems to have the same approach as arch. Git [7] on the other hand does not solve the problem entirely; sometimes directories are duplicated when renamed. Moreover it does not use unique id to identify a file. In its defense, it shows how a revision is created (by copying, renaming, or splitting). Bazaar [3] on the other hand solves the scenario described in figure 3.2 perfectly. It uses unique ID to distinguish between files so the path is not needed to identify a specific file. BitKeeper [4] is a distributed RC system that handles renames well: it can handle renames of files across various trees.

It seems that the general idea to be able to have true renaming support is to identify files by IDs. In such a way, renaming becomes simpler because the file is not identified by a name, which may change often during the development life span. This idea may be applicable in subversion but then a lot of rebuilding is required since subversion does not use id as a way to identify files. Instead, *properties* may be used to solve the rename problem. In the next section, a design that uses properties is going to be discussed.

5.2 Updating in Subversion

Since all operations in Subversion, including rename, except for the update and commit are done locally, and since all the presented rename scenarios fail at the update command, it is important to explain and discuss the sequence of the update command.

As mentioned above, the rename command, prior to a commit, is performed locally. The data in `.svn/entries` file at the root directory which the renamed file belongs to, changes. That is, a "schedule add" command is placed at the "new" renamed file while a "schedule delete" command is placed at the old one. Below is a sample of the `.svn/entries` file after a file `Foo.txt` is renamed to `bar.txt` and prior to a commit.

```
.
.
.

^L
Foo.txt
file
delete
d41d8cd98f00b204e9800998ecf8427e
2007-12-17T10:04:07.472870Z
4
freddie
has-props

^L

bar.txt
file
add
d41d8cd98f00b204e9800998ecf8427e
has-props
copied
file:///home/freddie/svn/proj/trunk/Foo.txt
4
^L
.
.
.
```

Note how each entry is separated by a single `L` character. Notice further the `copied` information stored at the `bar.txt` entry specifying that it has been copied from `Foo.txt` at revision 4.
When a commit command is issued, the server reads the `entries` and performs the schedule commands (such as add, delete..). The meta information such as the `copied` arguments are transferred to the sever side. When a client, another developer, issues an update command, a reporter object is created that reports the differences between the source (the version that the client has) and the target (the latest version at the repository). After that, each directory is updated along with its contents (files). First is the source directory; each entry (artifact), existing in the report object is updated.

Entries that exist in the source but not target are deleted. In the rename scenario, that's what happens. `foo.txt` exists in the source but not the target and consequently it is removed. That is part of the rename problem and is discussed in the next section.

5.3 Design

The idea presented in section 5.1, unique IDs, would be applicable to Subversion. In fact Subversion has unique IDs for directories but unfortunately not for files. Implementing unique IDs for files would definitely be the ultimate solution for the problem. Consider the scenarios presented in section 3.1. If each file had a unique ID then updating after renaming would become a much simpler task since all files to be merged are compared on a unique ID basis. Name would then be just an attribute and not an identifier. Considering the size of Subversion code-base and functionality, this would unfortunately take quite long time. Moreover this might give raise to backward compatibility problems. Instead one can make use of meta data[1]. Meta data is attributes stored at both client and server side. The information stored can vary from the author of the document to schedule commands (client side). Since copy operations are recorded as meta data one could use this data when merging or updating.

5.3.1 Solving the rename problem in Subversion

As an alternative to the solution presented above, one may use existing infrastructure in Subversion. In other words there is information about "*renaming*" but this is never used. One can use this information to determine if files are copied or not. For example, when a file is renamed (locally, i.e before a commit is performed), meta data is stored in the `.svn` directory at the user's WC which tells whether a file is copied, when it is copied, and what other file it is copied from. After committing, this information is stored at the server side.

When performing an update, Subversion compares what is stored in the target (latest version at server) and source (client version of the repository) for differences which are then handled accordingly. For example, in the scenario presented in figure 3.2, when doing an update the source now includes the file `foo.txt` while the target includes `bar` with its corresponding *"copyfrom"* arguments. First the source is checked (in this case the file `foo.txt`). This file is present at the source but not target since the target contains only `bar`. After that the target is updated (in this case `bar`). This file is now added with its corresponding *"copyfrom"* arguments. But the `"copyfrom"` arguments are useless in this case since `foo.txt`, which is the "copied from" file, is already deleted and Subversion can not find it. Subversion ignores files which can not be found and instead adds a new file `bar` discarding the changes made to `foo.txt`. Subversion has in fact a function (`add_file_smartly`) which can merge the new added file with the old one (if it is found) which in this case is not since it has been removed. To solve this problem the source is checked for special files. That is files that *might* be *copyfrom* arguments to other files. In other words, files who are present in the source and not in the target are files that *may* have been renamed in newer version. If such files exist then they should be deleted **after** checking the target since they must be present in order for the addition of the new files to succeed, i.e so that the old file could be merged with

[1] http://en.wikipedia.org/wiki/Meta_information

its renamed sibling. After checking the target, those *special* files are deleted. In this manner the scenario presented in figure 3.2 works as it should!

The scenario presented in figure 3.3 works with the above solution but not completely. When doing the final update (by A, step 7) the file(s) are merged correctly as a result of the above solution. The problem is that two files (`bar` and `baz`) reside in the repository as two different files while in fact they are the same file since they both are renames of `foo`. The problem is when C does a rename then an update (step 3 and 5), a new file (`bar`) is thrown by the server without checking if that file and any another file in C's WC , such as `baz`, have common *copyfrom* arguments. In fact Subversion discards reporting any files in WC that are scheduled for addition. In this case since `foo` was renamed by C (step 3), it is then scheduled for deletion and consequently a new file, `bar`, is scheduled for addition. That's why when the update takes place in step five, a whole new file is thrown to C. To solve this, newly added files that have valid copyfrom arguments should be included in the report. This leads however to a problem since when comparing directories (source and target), the reporter would not be able to find the newly added file since it does not exist anywhere in the repository yet (rename command prior to a commit). One solution is to check not only for files who have copyfrom arguments but moreover for files *w*ith common copyfrom arguments. In this case, `bar` and `baz` have common copyfrom arguments, `foo`. Then when updating in step 5, the reporter may then discard the latter renaming (at step 3) and merge the files (`bar` and `baz`) into a single file, `bar`. Developer C may then be notified that a file that it has renamed, had already been renamed, and then C may rename the file *a*fter it has done an update.

Chapter 6

Accomplishment

As mentioned in previous chapters, this thesis has two major parts: A theoretical part and a practical part. The theoretical part covers merging techniques, conflict detection, and several approaches to conflict detection and avoidance. That is to show how current RC systems work during merging and what they can and can not do. Moreover, it shows how developers have to be consistent in their merging in order to reduce extra work for other developers sharing the same resources. The practical part however covers the rename problem in Subversion and shows which parts it could solve and which that had to be treated. This chapter explains how the work was done and what difficulties were met during the course of this thesis.

6.1 Preliminaries

As mentioned in chapter 3, merge-tracking support was to be implemented as the practical part of this thesis. That however changed in the very beginning of the course of this thesis due to the fact that the community had already begun implementing merge-tracking and was already in its final implementation phase. Thus the initial goal was changed to address the rename problem in Subversion instead.

6.2 How the work was done

Concerning the theoretical part of the thesis, many scientific papers that treated *Merging techniques, Merging detection and reduction, Revision Control,* and *Software Configuration Management* in general were read. After that a draft, that covered theory part, was written.

The practical part however took longer time and involved many ups and downs. After discarding merge-tracking as the practical part of this thesis, many issues were looked upon as candidates for the practical part. One problem, that many developers complained about and the Subversion community did not really put effort into, was renaming. Then testing began to see what was the real problem and which common scenarios should be working in order to have a good renaming capability. After putting together three scenarios (presented in figures 3.1, 3.2, 3.3, and 3.4), the real work began. First these scenarios were tested on some commercial RC systems, such as Perforce, to

see if they worked as they should on those systems. Perforce performed the scenarios flawlessly.

Since Subversion has over 500,000 lines of code, the work took a very slow pace in the beginning since there was a huge amount of data that had to be processed in order to understand what was really happening. To get a better understanding of what really happened inside Subversion when for example a commit or a rename command was issued, a debug program, the built-in Eclipse debugger, was used to see step by step execution of commands. That was very slow since Subversion has many layers and libraries. At the end, the core problem was found. The initial design to solve the rename problem was to put more meta data at the client's side that could track renaming. Moreover this copy information was to be persistent at the client side after the user issues a commit. As it is now, the *copy* meta data is removed from the client's WC once a commit is issued.

However, after several conversations with the community, the design was discarded since all information needed was available in the sense that one could iterate the repository to track a renamed file. Moreover, the scenarios that were presented in chapter 3 did not require more than one rename and thus implementing yet another `copyfrom` meta data was a waste of time. But implementing the initial design was not really a waste of time since throughout the implementation phase, a lot of experience about data types and functions was gained and that helped very much in the final design. The final design was discussed with the community and a patch that solved part of the rename problem was sent to the community for possible future inclusion in Subversion.

Chapter 7

Results

This chapter describes the results of this thesis. The renaming problem presented in figure 3.2 works with the design described in section 5.3.1. This can be tested by performing the following steps:

```
$REPOS=file:///pathToRepository
Repository contains: /trunk/foo

#Developer A
svn co $REPOS/trunk
cd trunk
echo "HELLO WORLD" > foo

#Developer B
svn co $REPOS/trunk
cd trunk
svn rename foo bar
echo "HELLO WORLD FROM bar
svn commit -m ''

#Developer A
svn update
```

In the older versions of Subversion, the final update would result in removing `foo` and replacing it with `bar`. Moreover, the changes made to foo, `echo "HELLO WORLD" > foo`, would be discarded and not merged with developer B's changes. With the design implemented, the result would be one file, `bar` with both A's and B's changes merged together.

As for the scenario presented in figure 3.3, it can be tested with the following steps:

```
$REPOS=file:///pathToRepository
Repository contains: /trunk/foo

#Developer A
svn co $REPOS/trunk
```

```
cd trunk
echo "HELLO WORLD" > foo

#Developer B
svn co $REPOS/trunk
cd trunk
svn rename foo bar
echo "HELLO WORLD FROM bar
svn commit -m ''

#Developer C
svn rename foo baz
svn update
svn commit

#Developer A
svn update
```

The result is not completely satisfying: The problem with double files, for each rename, still exists. The difference is that the "`HELLO WORLD`" token is merged with the new files as a result of the implemented solution. An explanation to why this problem still exists along with a suggestion to a solution is discussed in section 8.2.

Chapter 8

Conclusions

This thesis was originally planned for two students but that changed because the other candidate turned down the thesis for some reasons. The goal of implementing true renaming has not been achieved due to the following reasons: The difficulty of doing so alone and the relatively short time. True renaming may be achieved by implementing unique IDs. Then changing the name of a file simply becomes changing an attribute in that file entry. Implementing unique IDs in Subversion means changing the base structure on which it is build and that's why it needs much more time and definitely more than one person. Most of the time went to understanding how the components of subversion worked and why the problem existed and where it existed in terms of source code.

The goal with implementing true renaming in Subversion was to solve the problems associated with the lack of this feature. Therefor, by solving these problems in a different approach, the goal would still be achieved. In fact, this thesis provides an implementation of a solution that solves some rename problems, partially and/or completely. Moreover, some test cases were published to the community that shed light on issues not known before (the double renaming problem). Furthermore a possible solution, that solves the above mentioned test cases, which could not be solved with the current solution, is suggested and perhaps will get the attention of the open source community at one stage.

Apart from the practical part, several merging techniques were discussed and compared. Moreover, some techniques to reduce or avoid conflicts are discussed which will give the user and even the developer a wider view of how RC works.

8.1 Limitations

The solution presented in this thesis does not fully solve the renaming problem. Cross-branch renaming does not work with the current solution (see fig 3.4). Moreover, parts of the problem presented in figure 3.3 still do not work properly. More specifically when renaming takes place in two WCs simultaneously, two files (the two renames) still reside at the repository as two different files though no lost update problem occurs. In other words, Subversion does not recognize that files with the same *copyfrom* arguments are the same. An explanation and a possible solution to this problem is explained in the next section.

8.2 Future work: *A*nother solution

A solution that solves the remaining issues in the problem presented in figure 3.3 is presented in this section.

As discussed earlier, when an update command is issued, Subversion writes a report of, amongst others, files that exist in the WC that the update command was issued in. However, files that are scheduled for addition are excluded from the report. In the scenario presented in 3.3, two successive renames are performed prior to a commit. In other words, in step 3, `foo` is deleted and `bar` is added. In step 4, `foo` is deleted and `baz` is added. When performing the update in step 5, the reporter excludes `baz` from the report because it is scheduled for addition and therefor Subversion does not take into consideration that files which are scheduled for addition may be renames of other files. In this case `baz` is a rename of `foo`. Consequently, `bar` is added as a new file and not merged with `baz` and thus the problem of multiple files.

To solve this problem, files who are scheduled for addition **and** who have valid *copyfrom* arguments should be included in the report. Valid *copyfrom* arguments are arguments which are other than `null`. That will however lead to a problem since when reading the report, the *s*pecial added file would not be found anywhere in any version in the repository since it has not yet been added (prior to commit).

That can however be solved under the assumption that files who have the same *copyfrom* arguments, that is files who have been copied from the same file, are in fact the same file entity. With this assumption, any file which can not be found at the repository, in this case `baz`, should be checked for in the WC from which the update command was issued. Furthermore, any file in the report which has similar *copyfrom* arguments, in this case `bar`, should be merged with that special file (`baz`) as one file. The name should be that of the latter. There is only one problem left, `baz` is still scheduled for addition and when a commit is performed, it will be thrown at the server as a new file. The entries file should consequently be changed and the "schedule add" should be removed. Moreover, a message that states that the rename (`foo -> baz`) is not performed, should be sent to the caller. Alternatively, a conflict could be issued to inform the caller that a certain file, `foo`, has already been renamed.

Chapter 9

Acknowledgments

First I'd like to thank Jürgen Börstler for his valuable comments on this report and for taking the time to explain essential and important concepts in scientific writing. Moreover I'd like to thank Pontus Nyman and Daniel Ashhami at ÅF for their suggestions and for their comments on this report. Finally I want to thank the open-source community for providing essential feedback, specially Ben Collin-Sussman.

References

[1] Berkeleydb. `http://www.oracle.com/technology/products/berkeley-db/index.html`, date visited: Feb. 2nd, 2008.

[2] Concurrent version system. `http://www.nongnu.org/cvs/`, date visited: Feb. 2nd, 2008.

[3] *Bazaar*, distributed rcs. `http://bazaar-vcs.org/`, date visited: Feb. 2nd, 2008.

[4] *BitKeeper*, distributed rcs. `http://www.bitkeeper.com/index.html`, date visited: Feb. 2nd, 2008.

[5] *diff* utility. `http://en.wikipedia.org/wiki/Diff`, date visited: Feb. 2nd, 2008.

[6] *diff3* utility. `http://en.wikipedia.org/wiki/Diff3`, date visited: Feb. 2nd, 2008.

[7] *Git*. `http://git.or.cz/`, date visited: Feb. 2nd, 2008.

[8] *GNU arch*. `http://www.gnu.org/software/gnu-arch/`, date visited: Feb. 2nd, 2008.

[9] *Monotone*. `http://monotone.ca/`, date visited: Feb. 2nd, 2008.

[10] *Perforce*, centralized rcs. `http://www.perforce.com/`, date visited: Feb. 2nd, 2008.

[11] U. Asklund. Identifying conflicts during structural merge, 1994.

[12] W. Keith Edwards. Flexible conflict detection and management in collaborative applications. In *UIST '97: Proceedings of the 10th annual ACM symposium on User interface software and technology*, pages 139–148, New York, NY, USA, 1997. ACM Press.

[13] M. S. Feather. Detecting interference when merging specification evolutions. In *IWSSD '89: Proceedings of the 5th international workshop on Software specification and design*, pages 169–176, New York, NY, USA, 1989. ACM Press.

[14] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 2002.

[15] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387, 1989.

[16] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[17] Daniel Jackson and David A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *ICSM '94: Proceedings of the International Conference on Software Maintenance*, pages 243–252, Washington, DC, USA, 1994. IEEE Computer Society.

[18] Grass J.E. Cdiff: A syntax directed diff for c++ programs," proc the advanced computing systems professional and technical association (usenix) conf. c++, pp.

[19] Boris Magnusson and Ulf Asklund. Fine grained version control of configurations in coop/orm. In *ICSE '96: Proceedings of the SCM-6 Workshop on System Configuration Management*, pages 31–48, London, UK, 1996. Springer-Verlag.

[20] T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, 28(5):449–462, 2002.

[21] Tom Mens. Conditional graph rewriting as a domain-independent formalism for software evolution. In *AGTIVE '99: Proceedings of the International Workshop on Applications of Graph Transformations with Industrial Relevance*, pages 127–143, London, UK, 2000. Springer-Verlag.

[22] Jonathan P. Munson and Prasun Dewan. A flexible object merging framework. In *CSCW '94: Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 231–242, New York, NY, USA, 1994. ACM Press.

[23] Nan Niu, Steve Easterbrook, and Mehrdad Sabetzadeh. A category-theoretic approach to syntactic software merging. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 197–206, Washington, DC, USA, 2005. IEEE Computer Society.

[24] M. Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[25] Dewayne E. Perry, Harvey P. Siy, and Lawrence G Votta. Parallel changes in large scale software development: an observational case study. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 251–260, Washington, DC, USA, 1998. IEEE Computer Society.

[26] Michael Pilato. *Version Control With Subversion*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2004.

[27] R S Pressman. *Software engineering: a practitioner's approach (2nd ed.)*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[28] Anita Sarma, Zahra Noroozi, and André van der Hoek. Palantir: raising awareness among configuration management workspaces. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 444–454, Washington, DC, USA, 2003. IEEE Computer Society.

[29] W.F. Tichy. RCS — A System for Version Control. *Software — Practice and Experience*, 15(7):637–654, 1985.

[30] Wei-Tek Tsai. Ripple effect analysis. `http://asusrl.eas.asu.edu/cse565/content/regression/rea.pdf`.

[31] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[32] S. S. Yau, J. S. Collofello, and T. M. MacGregor. Ripple effect analysis of software maintenance. pages 71–82, 1993.

Appendix A

Acronyms and Abbreviations

SCM
Software Configuration Management.

RC
Revision Control

svn
Subversion.

Repository
The directory into which users using a centralized RC system check-out and/or integrate their work (from their WC).

WC
The working copy of the repository at the client side.

check out
Users *check out* a copy of a software artifact from the repository into their wc.

commit
A command that allows users to *integrate* their wc into the repository.

update
Synchronize wc with the repository

Change Set
The set that has all changes made on an artifact by a certain user.

artifact
Information unit in a project (source code files, documentation,..).

refactoring
Changing the structure of an artifact without changing its semantics.

branching
The process of copying a trunk into another branch. Usually used for bug fixes or implementing new features. The branch can then be merged to the trunk.

merge
Applying a diff of two or three files to one of them.

trunk
Is usually the main developing branch.

revision
An instance of an artifact. A revision can never be modified, however new revisions can be created from an existing one.

delta
The difference between two successive versions of an artifact.

copyfrom
Meta data provided by Subversion that includes information about a file that has been copied. If for instance foo is copied from bar, then *copyfrom* meta data would include:
`copied`
`path/to/repository/copied_from_file`
`revision number (revision of the copied_from_file)`