University of Oslo
Department of Informatics

SIC

A Version Control System

Siri Spjelkavik

Cand. Scient. thesis

28th July 2003

Abstract

This thesis describes a small version control system, its design and implementation.

The version control system, *sic*, is implemented in Ruby. Its goal was to be a small system, featuring basic functionality, but at the same time solve some of the flaws CVS is suffering from. The system use common data structures such as graphs as internal representation of the data. The Ruby library PStore is used to store the objects. Access across different network is offered using a library called distributed ruby.

Sic is based on the optimistic copy-modify-merge concurrency model and offer basic version control system functionality such as checkin, checkout, update, add and delete in addition to restructuring commands such as move and rename. It implements a simple merging system and remote access to the repository. It does not support branches and other more sophisticated operations.

Preface

This thesis is submitted to the Department of Informatics at the University of Oslo as part of a candidata scientiarum degree.

Thanks

I would like to thank everyone who has helped me with answering all my questions. Professor Hans Petter Langtangen, my supervisor, for being enthusiastic and never giving up on me. I would also like to thank my colleagues at abel, abelvaktene, for never getting tired of all my questions and for all the time we have wasted in search for our motivation. My friends, family and Kristoffer Gleditsch have all been very helpful, reading through parts of my thesis and offering help.

Contents

1 Introduction **1**
1.1 Why Ruby? . . . 2
1.2 . . . so what *is* wrong with CVS then? . . . 2

2 Introduction to version control **5**
2.1 What is version control? . . . 5
2.2 Functionality offered by systems . . . 6
2.3 Terminology . . . 8

3 Existing software – common problems **11**
3.1 Common problems . . . 11
3.1.1 Concurrency and parallel development . . . 11
3.1.2 Parallel development . . . 12
3.1.3 The branch and merge problem . . . 13
3.1.4 Keyword substitution . . . 15
3.2 Existing software . . . 15
3.2.1 Revision Control System – RCS . . . 15
3.2.2 Concurrent Version System – CVS . . . 15
3.2.3 Subversion . . . 17
3.2.4 The Project Revision Control System – PRCS . . . 19
3.2.5 Commercial systems . . . 19

4 *Sic*’s design **21**
4.1 Model . . . 21
4.2 Data structure . . . 21
4.3 Storage . . . 23

4.4 Client/server architecture and modularising 26
4.5 Concurrency model 27
4.6 Algorithms 27
4.6.1 Status and up to date 27
4.6.2 Merge algorithm 29
4.6.3 Delta algorithms 34
4.6.4 The class of insert/delete algorithms 34
4.6.5 Delta compression tools, copy/insert algorithms . . . 35
4.6.6 Delta generators examples 36
4.6.7 Determining file content 37

5 Functionality 39
5.1 Create a project 39
5.2 Obtain a working copy (check out project) 41
5.3 Element types 43
5.4 Working on the project 43
5.4.1 Reorganising the project 44
5.4.2 Difference between a local file and the current 46
5.4.3 Status of your project 46
5.4.4 Update your files 49
5.4.5 Sharing your work (commit) 51
5.5 Reading and creating a log message 52
5.6 History 53
5.7 Export 53
5.8 Configuration files 53
5.9 Working off line 55
5.10 Getting help 55

6 Conclusion and future work 59
6.1 Comparing *sic* against other systems 59
6.2 Shortcomings and future work 61
6.2.1 Configuration file 61
6.2.2 Commit after conflict 61
6.2.3 Graphical User Interface 62

6.2.4 Current version and branch 62
6.2.5 Rsync 64
6.2.6 Sub-projects 65
6.2.7 Stand-alone server 65

A Configuration file **i**

B PStore objects **ii**

C Protocols **vi**

D Reference **viii**
D.1 SIC viii
D.2 Create a project in sic xi
D.3 Retrieve a copy of a project xii
D.4 Update the working copy xiv
D.5 Commit the changes to the repository xv
D.6 Reorganising commands xvi
D.7 Display differences between two items xvii
D.8 The status options xviii
D.9 The log message xix
D.10 Sic's configuration files xx

Bibliography **xxi**

Index **xxiv**

List of Tables

4.1 The different states an element may be in. 27
4.2 Table showing the communication between client and server for the status operation 2
4.3 The development of the fields 31
4.4 The notation used in the discussion of the merge algorithm. . 32
4.5 The files used in the merge example 33
4.6 The two files used in the diff example 36

5.1 Table showing the different states a file may have. 47

C.1 Table showing the communication for the update option . . . vi
C.2 Table showing the communication for commit vii

List of Figures

3.1 A sample version tree diagram for a project. 14

4.1 NIAM-model of a version control system 22
4.2 Sketch of internal representation. 24
4.3 Outline of the organisation of a repository 25
4.4 The communicating models 26
4.5 State diagram over the process to decide the current state of an element. 28
4.6 The changes and development of a typical project. 30

5.1 The initial layout of the project. 56
5.2 The actions from the example in figure 4.6. 58

6.1 Examples showing how the different GUIs may look. 63

List of Examples

4.1 The sicmerge 33
4.2 Diff3 -E -m 33
4.3 CVS delta, RCS format 35
4.4 diff -n 36
4.5 Vcdiff format 37
5.1 Create command 40
5.2 Checkout command 41
5.3 More options to checkout 42
5.4 Element_type 43
5.5 Add 44
5.6 Move 45
5.7 Delete 45
5.8 Deleting a directory 46
5.9 Undelete 46
5.10 Diff command 47
5.11 Status command 48
5.12 Lstatus command 49
5.13 Update command 50
5.14 Conflict 50
5.15 Commit command 52
5.16 Failing commit 52
5.17 Log command 53
5.18 History command 54
5.19 History command with options 54
5.20 Export command 54

5.21 sicrc file 55
5.22 Invalid argument(s) 56
5.23 Help 57

Chapter 1

Introduction

The use of version control systems is widespread and their applications are many. These systems have evolved over time and newly developed open source version control software tends to focus on supporting distributed projects. The issues important in these cases are intelligent merging of branches and a suitable client-server model. The earlier systems such as RCS [33], the Revision Control System, were designed for use by single-users or small projects.

This thesis describes a version control system implemented in Ruby [26], intended to be small and easy to install and maintain. The system, called *sic*, focuses on the needs of small projects and single users. Clearly, a small project using a version control system to synchronise the work has other demands to the system than big, distributed projects.

Although version control systems have evolved to concentrate on the demands for parallel development and project teams (advanced project management), many use such software as an incremental backup facility. CVS [16] is, for instance, a very good tool to synchronise work between different computers and laptops. This kind of usage has other demands to the software than advanced project management. The most important features will be the possibility to synchronise fast and easy, reorganising structures and general file system commands. Features like advanced branching, release tracking and naming is not important and might add complexity to the system, which are not wanted. *Sic*'s goal is to offer a simple tool for backup and at the same time provide support for simple project management.

Thesis outline

In chapter 2, the reader is introduced to the theory of version control and the terminology used in this thesis is defined. The thesis then goes on with

describing existing software and problems these are facing in chapter 3. Chapter 4 and 5 discuss *sic*, first the design and then the practical use. Chapter 6 sums up the thesis, comparing *sic* with other systems and looking at further work.

Chapter 5 is written as a stand-alone tutorial and may be read out of order with the rest, although it is recommended to read the thesis from the beginning until the end. In the appendix, a complete reference to *sic*'s functionality is found, together with definition of the class used to store the revisions.

1.1 Why Ruby?

The system presented in this thesis is implemented in the programming language Ruby [26]. Interpreted languages like Perl [40], Python [39] and Ruby are often called scripting languages. In the beginning, they were partly designed for text processing. Since a version control system processes text it was natural to use such a language. The external libraries use either C or C++, and SWIG [6] generates C extensions for Ruby.

Ruby was chosen as programming language because it has good support for object-oriented programming, which was one of the factors disqualifying Perl. Perl do support object-oriented programming, but the syntax and the implementation are somewhat alternative. Python also supports object-oriented programming, but Ruby was chosen because of personal preferences.[1] Ruby is quite easy to modularise and the language itself is easy to install, which is important.

1.2 . . . so what *is* wrong with CVS then?

In the UNIX world, the system Concurrent Versions System (CVS) [16] has been very popular, but in the last couple of years, different projects have started to develop replacements. This happened mainly because important requirements have changed, making new features more important [7]. The nature of the development process has changed; distributed development has for instance become more common. Another reason is a couple of flaws in CVS, for details read section 3.2.2.

CVS was, and still is, a very popular system, due to different circumstances. It is free software, so it is free of charge, which is quite important. It also introduced the term 'optimistic concurrency control'. Optimistic concurrency control is a mechanism to prevent situations where one developer

[1] The author prefers languages where she is able to choose how to indent her code.

has checked out a file (with a lock) and then gone off for a two month holiday trip to the North Pole. The CVS solution to this problem is now adapted by most version control systems. Although this is a nice feature, there are certain aspects of CVS that are troublesome.

Missing functionality includes a 'logical lock' [9], which enable certain files to be locked before any changes taking place. Other concepts lacking are relations between files and directories and the ability to reorganise the structure of a project, like renaming and moving files between directories [38, 11, 22]. Lack of support of repository replication, local repositories, atomic commits and build-in support for client/server architecture and access control lists are also mentionable. Most of the missed features arise from the needs of distributed developments.

Why just not wrap some scripts around CVS?

So, CVS is not perfect, but why would we want to develop something new, instead of just making some wrapper scripts to improve the usage of CVS?

There are different reasons for starting all over. The main thing to remember is that CVS tracks file content, not the structure or names of the files. In fact, it treats every directory by itself; one file at a time, ignoring relationship between files and directories that the user might believe is there. This 'one item at a time' approach leads to consistency problems while a commit is performed, enabling other users to update a half-updated project. It is also important to have in mind that CVS was developed as a wrapper to RCS. Both RCS and CVS are quite old, and limitations of the hardware have changed a lot since these projects were started, making other factors more important when new systems are designed.

CVS has indeed changed a lot since the start. It does not depend on RCS anymore, although it still uses RCS algorithms. Most of the new functionality has been add-ons to the existing code, making it hard to sustain an overall design philosophy. Every software project has a limit for where it would be easier start all over again than to change anymore. In our opinion, CVS has crossed this limit.

To achieve a system with the ability to track the structure of a project, it is necessary to build a layer on top of CVS. This layer could be working in the client, maintaining a database tracking name, id and revision. The TODO-file of CVS suggests three different ways of solving the renaming problem [42, number 189].

The Meta-CVS [21] project implements a version control system on top of CVS. The added features are storage of links, and versioning of directory structures. They have also simplified the branching and merging scheme.

However, Meta-CVS does suffer from flaws like non-atomic commit, and this problem is difficult to solve without changing CVS itself.

Chapter 2

Introduction to version control

> *Version control is the art of managing changes to information. It has long been a critical tool to programmers, who typically spend their time making small changes to software and then undoing those changes the next day. Imagine a team of these programmers working concurrently, and you can see why a good version control system is needed to manage the potential chaos.*
>
> *– Subversion: The Definitive Guide [10, page 1]*

2.1 What is version control?

A version control system is a program assisting in the development process of a project. The system stores the different versions of documents, defines the ability to support more than one developer and lets the users attach messages to the different versions. Different systems support different working models and functionalities, but most of them are designed to ease the burden of project management. The main functionality is record keeping, storing the different versions.

In almost every project there will, at some time or another, become necessary to look at an older version of the project, perhaps to figure out what went wrong. Version control systems store the different versions of the project files, and let the developers go back and choose whichever version they need. Most systems offer additional functionality. The systems also need to solve the problem of multiple developers and concurrency. The optimistic concurrency control solution introduced by CVS, encourages the decentralised collaboration work model, and is well suited for use in open source projects [38]. A version control system will most likely offer lots of other

functionality, such as log messages, access control lists (ACL) and the ability to perform distributed development, but the main function is storing and retrieving of the project history.

Software Configuration Management

Software Configuration Management, SCM, or just CM, is the managing of evolution of large and complex software systems, supporting both the management and the development perspective [12, 4]. Version control has an important role in CM, so theories and articles about such systems are quite relevant, although they are partly outside the scope of this thesis.

CM should support identification, control, audit and status accounting of configuration in addition to version control, concurrency control and functionality to select associated versions of documents [4].

2.2 Functionality offered by systems

What kinds of functionality should a version control system implement? An unordered list of the functionality most systems support:

- The possibility to store and retrieve data from the system, also known as checking in and out files and directories.
- Add, delete and move files and directories.
- Support both ASCII and binary files.
- Remote access.
- Storing of history and log messages.
- Retrieve old versions (a entire project or a specific file).

This is the most basic functionality; in addition, there are several other features, which may be implemented:

Access control list (ACL): ACL control which users may read and write different files, either in entire projects or files within a project. This functionality is most important in big projects with different groups of developers, like designers, programmers and even management. It should be configurable on a per-group basis as well as on an individual basis (e.g. let all programmers have read and write access, and the management only read access to the code).

This may also be used to let external users download source code directly from the repository, without having to worry about them writing to it.

Annotate: The annotate command should show which line was created or changed by which user, and in which revision this happened.

Diff: Diff shows the difference between the local revision of a file and the up-to-date version in the repository.

Export: Export will retrieve a complete project, but it will not be a working copy. A working copy demands the local configuration to be present, and these configuration files will not be created by an export. The intent of this option is to support distribution of projects to users not part of the development group. For example, export will be used when preparing a project for public download.

Exporting single files: Sometimes it may be necessary to import part of a project into another, and you want to preserve the development history. The files should be duplicated and imported into the other project (in the same repository or another repository).

Filter or trigger: A filter is the possibility to specify certain commands to be executed when a specific action occur. This may for instance be used to send a mail to a given group of users whenever a commit occurs. The user should be able to plug-in any executable program, binary or a script. There is no need to define an own executable syntax for the version control system.

History: The version control system should log some of the actions occuring in the repository. This will enable users to figure out which action different users have done.

Keyword substitution: Keyword substitution is a replacement of certain sequences in the file placed under version control. It is used to place tags in the text, for instance which revision the file is in, the author, or the last time of change.

Keyword substitution should only be done on plain text files, not on binary data, since such substitution may ruin binary data. The version control system must therefore be able to recognise whether the file is in binary or ASCII format.

Local repository: From time to time, developers may need to have some sort of revision control on their own, local version (between the 'real' commits). To solve this problem, the system could support some sort of local repository, where developers may play around until they decide to share their work. Arch [2] supports this feature.

Logical file lock: The most common behaviour for current systems is to not lock files on check out. It would sometimes be situations and files that only one person should change at the time, and merges and conflicts are unwanted. To support this in a system using an optimistic concurrency model, a logical file lock could be available.

A logical file lock will require everyone who wishes to change a file to get 'permission' from the repository, stopping other people from change this specific file. The logical part implies that it is not a real lock, just a property in the version control system; ensure a lock-change-unlock-model.

2.3 Terminology

This section defines the term and expression in this thesis:

Concurrency control: In projects there will often be more than one developer contributing to the project. The system needs to define a model of concurrency control; how to deal with the case when two or more developers wants to work on a single file at the same time. The concurrency control is often said to be either optimistic or pessimistic. Optimistic concurrency control will not prevent developers from changing one file at the same time, using the fact that most of the time it does not cause any trouble. Pessimistic concurrency control prevents developers from changing one file at the same time.

ACID: ACID is abbreviation for atomicity, consistency, isolation and durability. These are the four criteria (database) transactions should support [14].

- Atomic means that the whole transactions should be treated as one unit, and the whole united is either processed or not.
- The database should be consistent after the transaction has finished.
- If more than one transactions are running at the same time, the transactions should be isolated from each other; the changes done by one of the transaction should not affect the other transaction.
- Data put into the database should stay there; durability.

Three-way merge: A three-way merge [12] involves three files. One file which are a origin of the two others. The latter files are merged into one comparing what has changed in each of them from the origin file.

In version control the origin file is a common previous revision, the other two files are from the working copy and the current revision in the repository.

Branch: A branch is when more than one line of development exists, also referred to as codelines. Supporting branches adds a lot of complexity to the version control system, since the systems must be able to merge files with possible overlapping content.

When a split of development occurs, the reason will often be releases or developers wishing to test ideas while not interfering with the main development. The new line of development is called a branch.

Repository: All information about a project is stored in a central *repository*.

Version: A version of a project is a label identifying the state of all the elements at a given moment.

Revision: The revision number is a label used for internal purposes, to show how many times one element has been changed. Revision belongs to each element, and the version number belongs to project.

Delta: Delta is also known as diff, and is the difference between two files or, in our context, between two revisions of the same file.

The deltas are either **forward** or **backward** deltas. A backward delta is when the newest version is stored "as is", and you have to apply the deltas to retrieve the older versions. Forward delta is the opposite. RCS (and CVS) store the deltas as backward deltas, because the retrieval time for the newest version will then be the shortest possible, and this is the most common checkout action.

Server: The server is running on the computer where the repository is kept and performing all the communication with the repository. The client will always only communicate with the server.

Client: The application program run by the user is the client. The client is responsible for updating the working copy, including the configuration files, and for communicate with the server in order to retrieve all necessary information.

Working copy: The working copy is the local copy a developer does his work in. All changes are done in a working copy and then shared with the repository. Changes to a working copy are not shared with other members of the development team until the changes are committed to the repository, so the working copy is local and private for each developer.

Commit: Commit is the way to share work with the repository. When a developer requests a commit, the changed files are getting new revisions and a new version is created and stored.

Update: The update action is run from within a working copy and updates the working copy to be synchronised with the newest version in the repository. If there are local changes to a file, and this file is also modified in the repository, the software will try to merge the contents. A merge will most likely succeed if the files are modified in different places. If the automatically merge does not work, the developer must manually merge the contents. This is a conflict, and the section in question is marked to make it easier to identify and resolve the conflict.

Element/item: The two words, item and element are used about a file or directory under version control. They are used whenever it is not necessary to distinct between files and directories. When it does matter, the words *file* and *directory* are used to clarify.

Chapter 3

Existing software – common problems

There are a lot of version control systems available today. This area has gone through rapid changes in the last years. Apart from the concurrency problem, the merge-and-branch problem is one of the biggest problems of the older software. This chapter will discuss important issues in version control, and then present some systems and their solutions.

3.1 Common problems

As mentioned before, older software has problems with merging, and which concurrency model to choose.

3.1.1 Concurrency and parallel development

Different users have different behaviours and needs, and one version control system often try to support several individuals with their requirements. Users who use such software to synchronise or back up their own work do not have problems with concurrency. Such users will often be the only developer at a project. It is quite common to use version control systems like CVS to synchronise work between different computers and networks. Many prefer to have important work more than one place, in the case of data loss, and use version control systems to make sure that all the different computers have a new version. Another issue is back-up: If a user has access to one computer with regular back-up, he may choose to create a repository at that computer, and synchronise the projects at for instance laptops and home computers through the version control system.

Whenever a project has more than one user changing the files, the system needs to define behaviour when more than one wants to change a file at the same time. There are two main models in use today: copy-modify-merge and lock-modify-unlock [10]. The copy-modify-merge is an optimistic concurrency model.

3.1.2 Parallel development

Most systems support some sort of collaborative working, i.e. more than one developer at the time. As already mentioned, version control systems need to define a concurrency model that decides how parallel development should occur. It is important to avoid situations where developers overwrite each other's changes. Consider an example where two developers, Bob and Jane, have a common pool where all their code goes. Whenever one of them wants to change a file, the file is copied to his or her home directory and then copied back when they are finished. So, both Bob and Jane decide to change a file, 'Kampeknerten'. Bob is somewhat faster than Jane is, so he copies the file back to the pool and continue his work other places. In the meantime, Jane works with many projects, but after some time she copies the file back. The only problem is that Jane's changes overwrote Bob's changes, so all his efforts were waste of time. Clearly, the system needs to prevent this from happen.

One solution would be to force a serialisation of the process, disallowing more than one developer to change one file at the time. Under such a scheme, the developers have to request the pool for permission to work on the file, and this is only admitted if no one else already is editing the file. This would most likely be achieved through the use of a lock, and the scheme is called **lock-modify-unlock**. If Bob and Jane had used this solution, two different things may have happened. If Bob was the first to request the file for change, he would have gotten the file, done his work and put the file back into the pool, releasing the lock. Meanwhile, Jane is not able to do anything on 'Kampeknerten', but as we remember, she had lots of work to do, and is not bothered. The other situations that might occur would be if Jane got the file first. Then Jane would have held the lock for several days, maybe weeks, since she was working on all those projects at the same time. Bob needs to fix one little line in the file, but is not allowed to do that, since Jane holds the lock, and must wait until Jane remembers to unlock the file.

This scheme works well for single-user projects or projects where the developers are close together, and communication is easy. This model is quite simple, and the software does not need to implement code to deal with conflicts and merge; this burden is on the users. For the users it may be irritating and troublesome to remember to lock and unlock files, leading to

situations where someone goes off for a two month trip, counting penguins with a file locked.

Developer Bob and Jane both want to be able to work independent of the others, and at the same time they want to make sure that changes are not lost. Most of the time, their changes does not overlap, so the changes could just be merged to one document, automagically. The principle to *optimistic concurrency control* builds upon the fact that mostly, developers changes different part of a file. Using a tool supporting this way of working, Bob may submit his version to the pool at any time, independent of Jane's work. Jane can handle all her pending work and submit the file whenever she wants. Most likely, she would be notified that Bob has changed the file. Before Jane is allowed to submit her work, she needs to merge her changes with Bob's. After the merge Jane may submit her work. From the developers' point of view, the **copy-modify-merge** model is easier, conflict should not occur too often, and the developer does not need to remember locking and unlocking files. The model is more complex to implement, especially the merge algorithm may be quite difficult to implement correctly.

3.1.3 The branch and merge problem

In section 2.3, the term branch is defined as a split or fork in the development line to the project. There are several reasons why a fork is created; major changes in design, a developer wants to test some new features or perhaps a new release. Work made in a branch is not visible in the others codelines. Changes may be shared between codelines performing a three-way merge. The success to a three-way merge algorithm depends on the ancestor file used. The version control system must carefully choose which ancestor file to use in each merge.

In figure 3.1, a typical version tree diagram is shown, using the same notation as [1]. Solid lines represent codelines, branches are marked with a preceding slash in front of the name ('/a'). Revisions are surrounded by a circle and branches are surrounded with a box. There are, as mentioned earlier, different types of branches; concurrent development may be achieved using branches. This is called an activity branch [1], and each branch is merged back into the codeline. In the figure, these are shown with dashed lines in the start, and are omitted after version 2. Dotted lines represent merges back into the codeline.

In the figure 3.1, the node marked A and B represents two developers working just like Bob and Jane from section 3.1.2. When B is going to commit his work, he needs to merge it into the development line since A already has committed her work. This result in the version labelled 2. In the three-way merge, the latest common ancestor is S, and the three files used in the

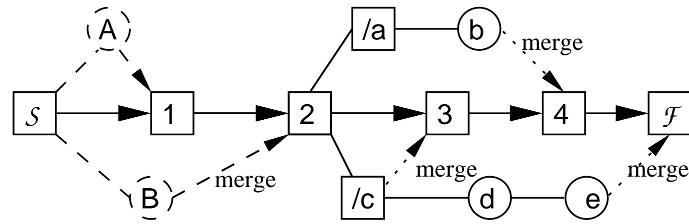


Figure 3.1: A sample version tree diagram for a project.

merge are S, 1 and B's file.

The two next merges, resulting in version 3 and 4, is just as simple as the preceding one: in both cases the common ancestor file would be 2, the point where the branches occurred. Picking the correct ancestor file for the last merge, from e to F is not that simple. Systems like CVS and Subversion will choose 2 as the ancestor file used in the merge, but the changes done in '/c' are already incorporated in version 3. When 2 are used as the ancestor file, it will try to merge the changes from '/c' twice, an act which most likely will produce bogus conflicts [28]. The best choice for ancestor file would have been version 3, which are the latest common ancestor.

The consequence of all this, is that version control system should choose the *latest* common ancestor file. This may be shown using set notation. A revision's set consist of all preceding revision, since the common ancestor used. In this example, this would be the point where the branch occurred. The intersection of two sets to be merged should then be empty.

We illustrate the different sets and the merge, using set notation, starting at version 2:

$$
\begin{aligned}
\text{First merge: } & [\{\}, \{/c\}] \Rightarrow 3 \\
\text{Second merge: } & [\{/a, b\}, \{3\}] \Rightarrow 4 \\
& \{/a, b\} \cap \{/c\} = \phi \\
\text{Third merge: } & [\{/c, d, e\}, \{4\}] \Rightarrow F \\
& \{/c, d, e\} \cap \{/c, /a, b\} = \{/c\}
\end{aligned}
$$

If the software detects that the common ancestor for the merge between 4 and e is 3 and not 2 then the intersection would have been empty. This problem will not be addressed any further in this thesis, since the goal is emphasised on small projects.

3.1.4 Keyword substitution

Keyword substitution is a nice feature, enabling the user to put information about a version into the documents. The problem with this functionality is when a file containing a substituted keyword is merged. In a merge, the lines with keywords will differ from each other and result in a bogus conflict. This may be solved either in the merge library or let the client perform the substitute, so all files in the repository only contain the keyword, and not the substituted values. This will be a question about efficiency, since the client will need to substitute before and after all communications with the repository.

3.2 Existing software

3.2.1 Revision Control System – RCS

The Revision Control System (RCS [33]) manages multiple revisions of single files. It handles storing and retrieval, history of changes, identification and merging of revisions. The revisions are arranged in an ancestral tree, and all revisions are identified with name and revision number. In addition other information such as creation time, author and log message are stored.

This information is stored in a single file called the RCS file, complete with all information about each revision including the backwards delta. The concurrency control used is the lock-modify-unlock scheme. If a user wants to acquire a file which already is locked, he needs to use a special command to steal the lock before modifying the particular file. RCS only works on single files, and is not very well suited for managing large collections of files or projects with a file hierarchy.

3.2.2 Concurrent Version System – CVS

> *CVS and several other systems are designed around the idea that version control for a group of files can be achieved by grouping together many single-file version-control operations.*
>
> *– Joshua MacDonald [18, page 1]*

The Concurrent Version System, CVS [16, 7, 15], is probably one of the most used free systems for version control.

It started as many shell scripts, which together accomplished version control. It was built on top of RCS, although it does not use RCS anymore it

still use the algorithms and RCS. Even though CVS has developed a lot since the start, it still suffers from the limitations of RCS as discussed under, such as non-atomics commits [11, 38].

CVS is a more complex system than RCS, in the way it operates on hierarchical collections of directories consisting of version-controlled files instead of treating each file separately. It does not have any knowledge of connections across different directories. It treats each directory independently from each other, which leads to non-atomic commits.

An important extension to RCS is the client-server architecture, which provides remote access to the repository. This was designed as an add-on, and is quite a good example of how CVS has developed.

Common problems with CVS

- ACID

  This is a 'problem' caused by the locking-scheme used by CVS. It does not follow the ACID properties as defined in section 2.3. A DBMS, DataBase Management System, should support five criteria: persistence, secondary storage management, concurrency, recovery and a query facility [5]. Some of these also apply to a version control system. All actions that may change the contents of the repository should be treated as a transaction, which is not done by CVS.

  CVS deals with one directory at the time (and lock only this directory), so each directory is committed in one go, but not the whole project [16]. However, if anything fails under commit, previously committed files are not rolled back; the changes will stay. Consequently, users will be able to check out or update their own repositories, while some of it is changed, but not all; the system is inconsistent. This means developers will not know that their working copy is not up-to-date, unless this is communicated in other ways or the code fails to compile because of unmet dependencies. The system is inconsistent only while the commits are performed, but this may lead to confusion. In the terms of ACID, it also violates the demand for isolation. Isolation means that two operations should perform isolated from each other, so if an update is performed while a commit is running, the update should not see the changes performed by the commit.

- Renaming

  CVS control file contents, not directory contents [11], neither does it track changes between directories nor the names of the elements. It simply does not store such meta data. This implies that if a file should

be renamed or moved to another directory, there is not any command performing this, keeping the information about the move.

There are a couple of ways to achieve this, the file can be added with an appropriate name in the wanted directory, but you will loose the file's history. This may be solved by putting a message in the file, containing information about the original name and placement, but this must be done manually. Another solution is to manipulate the repository itself [15]. None of these is satisfying and is merely a work-around than a solution to the problem.

- Merging

  CVS suffers from the branch-and-merge problem described in section 3.1.3.

- Handling of binary data and delta algorithm

  CVS use the GNU diff-library [25] to generate deltas. The method used by diff to generate the deltas is not suitable for binary data since the algorithm works on lines. CVS requires the users to check in binary files with a special flag, so CVS avoids to perform delta generation and keyword substitution on the file. If the user put a binary file under version control without flagging it as binary, the file is ASCII-fied by inserting newlines before a delta is computed. At retrieval time, the newlines are removed, but it is not guaranteed that the data is not corrupted.

3.2.3 Subversion

Subversion [10] is a project intending to replace CVS. It was designed with some simple goals; it should be a functional replacement for CVS – supporting all of CVS's features, use the same development model, and solve many of the problems CVS is suffering from [31].

Subversion intends to put everything under version control, including directories and metadata and thus enabling renaming. Other main goals have been atomic commits and better handling of binary files.

There are many differences between Subversion and CVS, some of the main issues compared to CVS are:

Version numbering

In Subversion, only the whole project has a unique identification number, called a version. The term revision is not in use. Each version is a snapshot of the files at the moment, so one file may be equal in version two, three and four. In CVS, each file has its own revision number, and

the project consists of a set of elements in different revisions. In Subversion, there is no simple way for the user to figure out how many times a single file has been changed.

This implies that you do not have revision 'n' of a file, but instead have version 'b' of the project.

Subversion is capable of updating just parts, leading to situations where the project may be in different versions. A project is not up-to-date before update has been performed from the root of the project [10].

Local cache

Subversion stores a copy of the last version from the repository for each user. This enables the status command to run without communicating with the repository, which is comfortable if you are offline. This allows the client to send deltas to the server, reducing network latency, and the ability to let commands like 'diff', 'status' and 'revert' to be available without network access.

Update and status

Many people using CVS run the command `cvs update` to figure out what have been changed since last time. This action has the secondary effect of merging files changed locally and in the repository. CVS does have a `status`-option, but merely no one use it, since the output is overwhelming. The `status`-option to Subversion has similar output as `cvs update`.

Changing your copy

Subversion controls whole projects, including directories, enabling the user to change the structure and not only the file contents. This means it is possible to move, copy, delete and rename files and directories.

Commits and binary handling

Commits are atomic, meaning that either the complete set of changes goes into the repository or not at all. Deltas between revisions use a binary diffing algorithm, working just as well on binary files as text files.

At the moment, Subversion does suffer from the branch-and-merge (or repeated merge) problem. The goal for Subversion is that version 1.0 should support and solve all CVS's features. When this is achieved, the goal will be to improve upon other features, as solving the branch-and-merge problem and internationalisation.

3.2.4 The Project Revision Control System – PRCS

PRCS [18] is a version control system which is quite small. The main motivation was the dissatisfaction with existing systems; commercial systems tend to be big and feature-laden, while CVS has a lot of small flaws, and could be described as 'simple yet complex'.

Each version is a labelled snapshot of the project at the creation time. All the files have its own, unique identifier; enabling the system to track changes of names and structure within the project. It uses the same optimistic concurrency control as CVS, the copy-modify-merge model. PRCS uses, for the time being, RCS as a back-end storage, but the goal is to use xdelta/xdfs in the future. For minimising file I/O, it uses MD5 checksums together with timestamps, an idea that was adopted in *sic*.

PRCS implements a smart three-way merge, finding the latest common ancestor to be used. As discussed in section 3.1.3, this is quite important to avoid spurious conflicts.

3.2.5 Commercial systems

There are a lot of commercial systems, some of the most common being Microsoft SourceSafe, ClearCase by Rational Sotware Corporation, Perforce by Perforce Software and BitKeeper from Bitmover Inc. Technical documentation for commercial systems seems to be hard to get by, most companies publish advertisement papers with the intention to sell their products.

ClearCase

ClearCase [13] was originally designed for use on local area networks (LANs) but has been extended to support geographically distributed development as well.

ClearCase stores the data in versioned-object bases (VOBs) [3]. This is an implementation of the NFS protocol and the filesystem is called 'mvfs'. The VOBs are NFS-mounted and appear as ordinary files and directories.

Each user acquires a view into the file database by creating a special mvfs mount point on their machine. Each view has a configuration specification containing a set of selection rules specifying the particular version of each file to make visible in the specific view. The view is similar to the work area in CVS, with one big difference; the files do not exists on the local disk until they are modified, which conserves disk space.

The files in the view appear like normal Unix files so standard Unix commands may be used to do whatever the user wishes, for example diffing versions, and reading them.

Versioning is achieved by the means of different branch-types. Each branch-type is identified with a name and is shared for all files in a VOB. The initially branch-type is 'main'. Versions are created in the usual way of checking in and out. One file may exist on several branch-types at the same time.

A branch named *main/release2/bugfix/3* represents revision number three on the bugfix-branch of release two. The system records the highest delta which has been applied between branches. This may be a problem if merges are applied out-of-order [28].

Perforce

Perforce [27] provided by Perforce Software is a version control system featuring basic version control functionality. Perforce stores the deltas using the RCS format, and offers functionality similar to the one provided by CVS and other open-source projects.

The repository, called a *depot*, resides on one central server and the users request *client workspaces*. The server stores metadata and tracks which clients who currently have which files. At the same time, it also defines *client views*, determining which level of access each user has. The data is stored in a relational database.

Which items to be committed is send to the server, using changelists, containing all the changes to be done. The server also makes sure that the changelist is treated atomically.

The naming scheme used in Perforce for branches is similar to the one used in ClearCase. The names are chosen by the user, and the names have no influence on which versions are used in a merge [28].

BitKeeper

BitKeeper by Bitmover Inc. [8] is a version control system designed to support a true distributed work-model and is customised to the needs of the Linux Kernel developer community. It is possible to obtain and use BitKeeper for free under special conditions, refer to the projects web page: `http://www.bitmover.com/bitkeeper`.

BitKeeper does not have a central server, but defines every working area to be a repository itself, letting different users retrieve changesets (sets of patches) between each other's trees. This is what is meant by 'true distributed model'.

Additionally, BitKeeper handles distributed renaming and changes in structure correctly, to fully support the distributed model.

Chapter 4

Sic's design

4.1 Model

When *sic* was designed, there were many aspects to take into consideration. The amounts of resources available in projects like this are limited, so we will never be able to implement all the features we would like to. Nonetheless it was important to create the overall design to be as flexible and general as possible. A NIAM-diagram of the system is presented in figure 4.1. NIAM is a common tool to model database systems, and it was appropriate to use it as modelling language here too. The figure shows the relationship between the different pieces of information needed by a version control system. A project may consist of several versions; each version consists of any number of elements. As defined in section 2.3, an element is a file or directory under version control. A revision belongs to an element and an element may have any number of revisions. A revision has a set of properties attached, such as log message, date and name. Note that this model shows the relationship between elements and properties; the application will not necessarily implement the entire model.

4.2 Data structure

The system tracks two different kinds of history information: The change history to each element (files and directories) and of entire projects. Each history item is tagged with a unique id number. A file may be moved or renamed in the project hierarchy, and be a part of different versions. The version control system must be able to track and recreate any version the project has been part of.

A file hierarchy is often visualised as a tree structure. Each version consists of a file hierarchy, and is stored as a tree. Trees and graphs are common

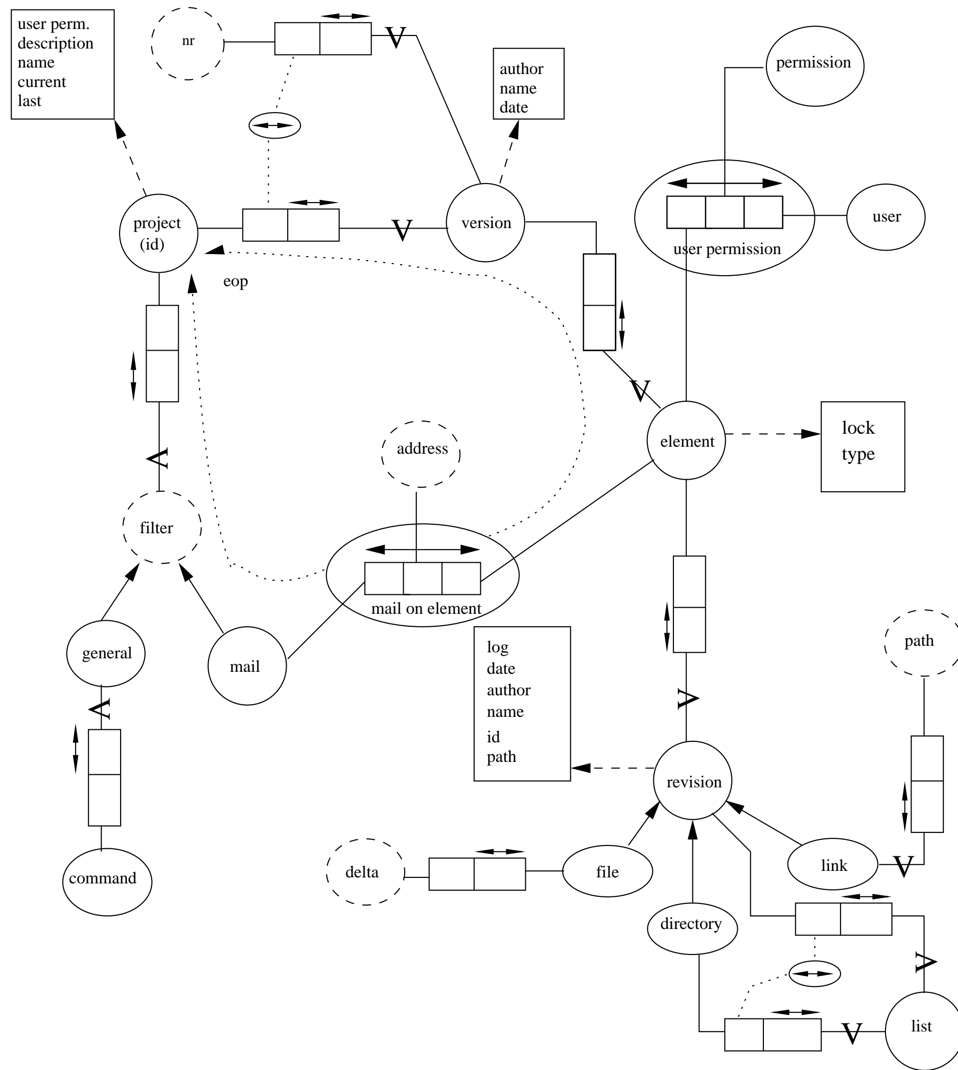


Figure 4.1: NIAM-model of a version control system

data structures with well-defined algorithms for insertion and parsing of the structure. The history to one element may be represented by a list if branches are not supported. Branches require the structure to be a graph, and a list is just a special case of a graph. The internal representation of the data in *sic* is:

- Each version is stored as a tree, with different nodes representing directories and files. Files are represented by leaf-nodes, meaning that it can not hold any other element.
- Each element is represented by a graph, where each node must have either one or two parents (except the root node which has none) and any number of children.

The internal structure is a tree consisting of elements. Each of these elements is also a part of another graph. A sketch of this is shown in figure 4.2. The elements store all necessary properties for each revision, as shown in figure 4.1.

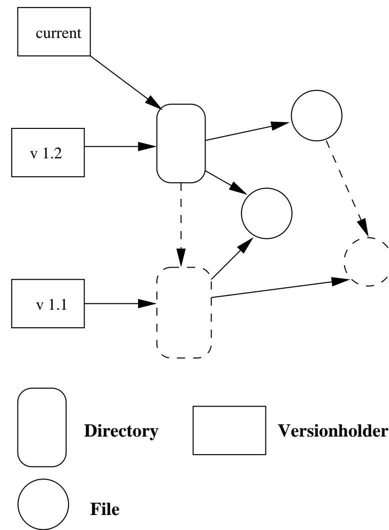
4.3 Storage

Two of the libraries offered by Ruby for storing the state of objects are the Marshal module and the PStore module [32]. The Marshal module offers simple object persistence[1], and the PStore library is a more high-level way to achieve the same (depending on the Marshal module). PStore provides a transactional, persistent file-based storage of Ruby-objects. One PStore-object may store several hierarchies of objects, but each hierarchy must have a unique key (id). The unique key in *sic* is the version number. In the start of a PStore transaction, these hierarchies are read from disk and they are written back in the end of transaction.

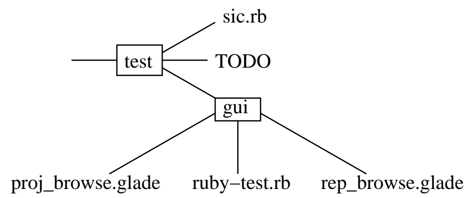
PStore was chosen as storage method because of simplicity; it is a standard library in Ruby and it makes the model easy to implement. A repository consists of a directory for each project; each project has at least one PStore-file, a history file and backup files.

The entries in the PStore hash correspond to the versionholders in figure 4.2. The newest version contains the whole file content, while older versions store only the backwards delta. The latest version stores only the full path of the file; the whole files themselves are stored as ordinary files in the file system in the repository. This is done in order to avoid reading large objects into memory and in order to be able to retrieve the file even if the PStore-file

[1] Java supports this ability and call it serialisation.



(a) Outline of the internal presentation. Version 1.1 is shown in dashed lines.



(b) A practical example of a project.

Figure 4.2: Two examples of the structures, 4.2(a) is a sketch of the two graphs, and 4.2(b) shows a project with files and directories. Note that one node may be a part of different versions.

is corrupted. In addition, every fifteenth revision of each file is also stored separately. This method is an optimalisation, guaranteeing that no more that 14 deltas need to be applied in order to retrieve any revision of a file. This subject is further discussed in [23].

In addition to the versions, the PStore-file also contains some meta information. The parsing of the data structure is mainly done by looking up keys in hash tables, since a traversal of the whole graph will take too much time. The graph is only parsed once each time a new version is created.

A disadvantage of storing the revisions as binary data is the possibility of data corruption. In CVS, which uses text deltas kept in ordinary text files, the files will always be accessible through the file system. *Sic*'s use of binary deltas makes a binary storage method a natural choice. Since every fifteenth revision is stored as an ordinary file, users will be able to restore some of the history even if parts of the repository are corrupted.

The only way to access the deltas themselves is to access the PStore-file. The definitions of the classes used are given in appendix B. In figure 4.3, an outline of the repository organisation is shown.

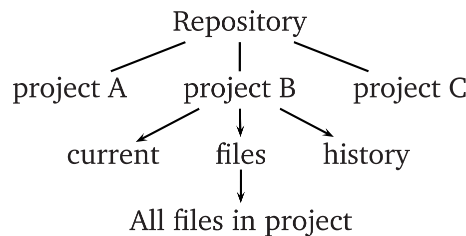


Figure 4.3: Outline of the organisation of a repository

An alternative storage method is to use a database. Most databases offer transactional access to data, and are optimised for retrieving, storing and searching. The two most interesting candidates for use by *sic* are (g)dbm, GNU database manager [43], and Berkeley database[2] (bdb) [29]. The reason for not choosing systems like Mysql or Postgresql is accessibility. If such a system were chosen, we would have required users to have access to a database. *Sic*'s target is to be as stand-alone as possible. Both dbm and db are libraries used by an application, and the application choose how and where to store the data. Dbm is a library managing data files containing pairs of keys and data, offering storing, retrieval, deletion and a traversal of all keys. Db is more advanced and offers both hash-like storage, tree storage and transaction based access. The main reason for not using this was some initial problems and lack of documentation for the ruby interface, and PStore is a standard library in Ruby.

[2]Berkeley DB is copyrighted by Sleepycat Software.

4.4 Client/server architecture and modularising

Design and implementation of this system have focused on re-usability. All the different functionality is implemented as stand-alone libraries and modules; this should make it easy to replace certain parts of the system, without re-implementing everything. API-details to the different modules are documented in the distribution of the program.

Another requirement to *sic* is the possibility to install the application without root access. As a result of this, *sic* can not depend on an up-and-running server where the repository resides. It is also important to offer remote access to the repository, not putting any constraints on the user about the layout of the network he is using. As mentioned earlier, some use CVS to synchronise and back up their work, so it is important to preserve the possibility for remote access in order to synchronise and distribute the code.

Sic use the distributed Ruby library (drb or druby) to implement a client/server-architecture. When the application is invoked, a special server-class is called upon. The server-object is responsible for starting a drb-server and returning a client object which is connected to the server. If the repository is on a remote computer, the server-object uses a third-party program, SSH [30], to connect to the remote computer and start the drb-server. The client will not know or does not need to know, if the server is local or remote. The objects communicating with each other are shown in figure 4.4.

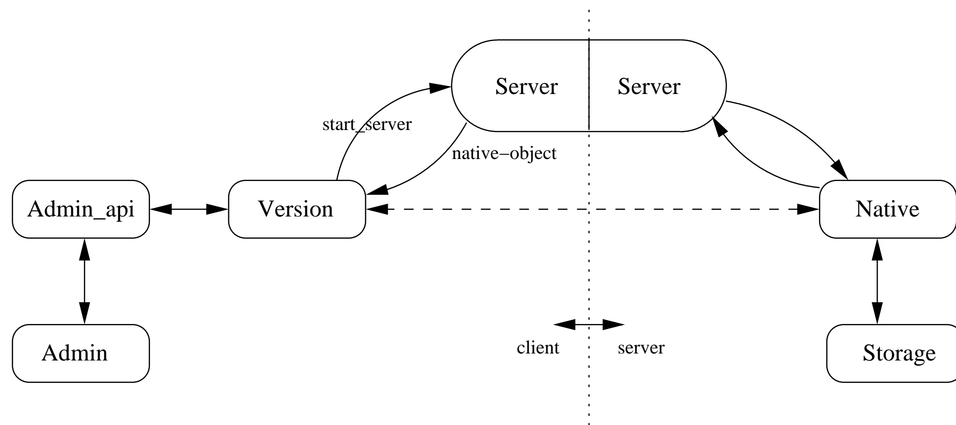


Figure 4.4: The arrows show who communicate with whom. A Version-object retrieves a Native-object from the start_server-function in the Server-class. The labels show the different classes.

Distributed Ruby (drb) is a distributed object system, such as Java's RMI (remote method invocation) or CORBA, although it does not have all the

functionality offered by those systems. The server code starts a service by associating an object with a given port [32]. The client creates a drb-object and associates it with the object on the server.

4.5 Concurrency model

Sic implements a copy-modify-merge concurrency model. This enables more than one developer to make changes to one file at the same time as previously discussed in section 3.1.1. Hence, *sic* also implements a simple merging algorithm. The work cycle will be quite similar to the one provided by CVS, the developer acquire a copy, does his work and commit.

4.6 Algorithms

This section describes the different algorithms used in the implementation.

4.6.1 Status and up to date

Sic must be able to decide if a file is changed in the repository and not in a working directory. It must also detect locally changed files, so only the changed files are committed to the repository. An element may be as shown in table 4.1 in, four different states. An element may be locally changed or unchanged, and may origin from the newest revision in the repository or be outdated. Whenever an element originates from the newest revision in repository, the file is said to be up-to-date.

	Newest revision	**Old revision**
Local modification	newest	possible conflict
No local modification	up-to-date	old

Table 4.1: The different states an element may be in.

The determination process

Sic uses a mix of md5 checksumming and timestamps to determine if a file is locally modified or not. In table 4.2, the necessary communication between the client and the server is sequentially shown.[3] The details of the different steps are discussed underneath, but note that most of the processing is computed by the client. The server retrieves a list of elements and returns the

[3] In appendix C similar tables for other operation is shown.

revision number belonging to each element. The revision number is sufficient, since each change in the repository is reflected in a new revision, and the checksum at the time of retrieval of a file is stored in the configuration file.

Client	Server
a) find status, compare timestamp and checksum against the version on disk	
b) ask the repository for information on those files	
	i) send the revision number for each requested element
c) Compare the information from server and the local information to determine each file status.	

Table 4.2: Table showing the communication between client and server for the status operation

The checksum is stored in the repository for each revision, and in the local configuration file, together with the timestamp. The checksum stored in the local configuration file is used by the client to decide the local state to a file. If the timestamp in the configuration file and the current timestamp of the file itself differ, a checksum of the file is generated and compared to the checksum recorded in the configuration file. The decision diagram showing the process is shown in figure 4.5.

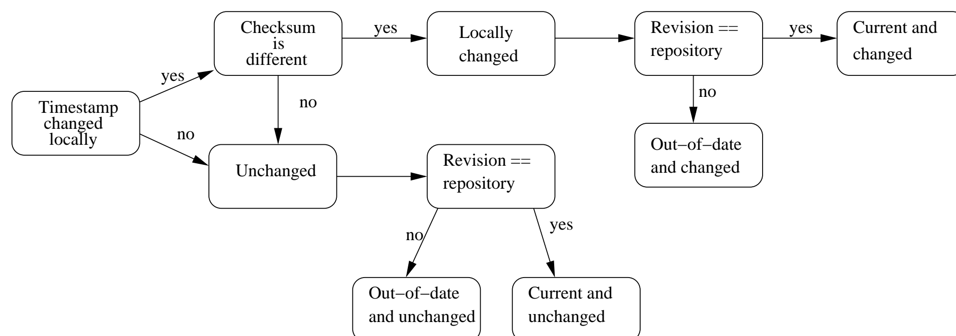


Figure 4.5: State diagram over the process to decide the current state of an element.

The special case of directories

The algorithm is a bit different for directories. A directory is up to date if and only if all the sub-elements are up to date, including names, position in the hierarchy and number of elements. This is necessary because of the possibility of adding, deleting and moving elements. We use the checksums to determine if a file’s content is changed since retrieval.

Consequently, if an element changes revision, all the directories in a straight line upwards in the hierarchy are also changed. This is illustrated in figure 4.6. This shows the three first steps of a project, from the initial step (figure 4.6(a)). A directory-node is marked with a box; all others are files. Files which are changed during the shown step are marked with an oval, the other files are just shown with a label.

Initially, all the items are in revision 1 and the project label is 1.

Four actions are performed on the project, in the following sequence of order:

1. A new file, ’AUTHOR’ is added in the root of the project (together with ’README’).
2. The file ’Kyrkja’ is appended to the directory ’Jotunheimen’.
3. The file ’Kikut’ is deleted and ’Flatbreen’ is moved to ’Jostedalsbreen’.

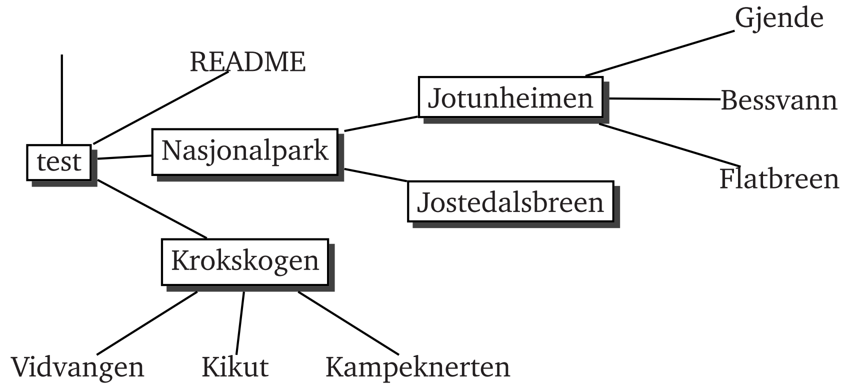
This example shows how a change to a file in a directory propagates to the directories, and which actions create a new revision. In table 4.3, the revision number for each element is shown (for each version). Neither ’Vidvangen’ nor ’Kampeknerten’ are changed after the initial commit, so they both are in their first revision, shared by all the versions.

4.6.2 Merge algorithm

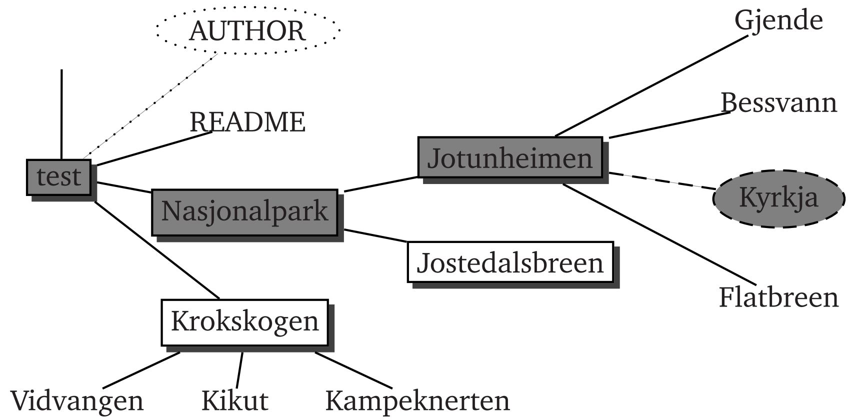
The merge algorithm is quite basic; it takes the latest common ancestor file (A), and computes two delta sets: One against the current file in the repository (AC) and one against the working copy file (AW). These two sets of deltas are then merged into one delta set and then applied to the ancestor file.

The delta sets are computed using a Ruby diff library. This library returns an array of triples: `[+|-], <line-number>, <sentence>`. The ’<sentence>’ was added (’+’) or deleted (’-’) at line ’<line-number>’. The notation used in this section is described in table 4.4.

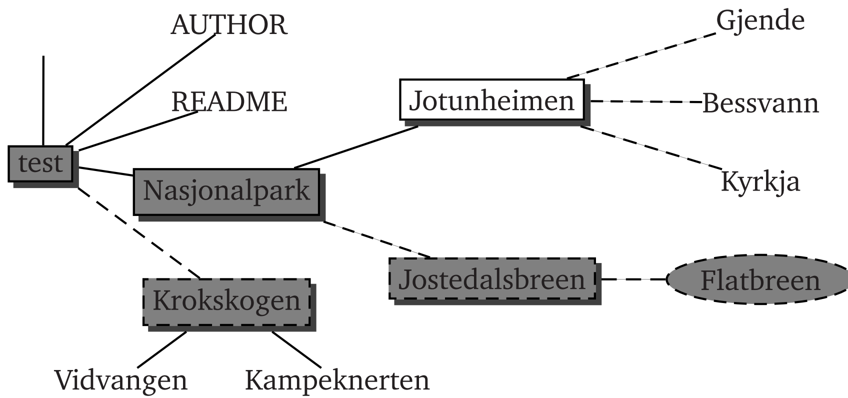
Using the AW delta set as base, the two delta sets are merged using the following algorithm:



(a) The initial version 1.0



(b) The second step is marked with gray, and the first step is shown with dotted lines.



(c) Directories changed in this last step is marked

Figure 4.6: The changes and development of a typical project.

Version	test	README	Nasjonalpark	Krokskogen	AUTHOR
1	1	1	1	1	–
2	2	1	1	1	1
3	3	1	2	1	1
4	4	1	3	2	1

(a) The elements at root level

Version	Vidvangen	Kikut	Kampeknerten
1	1	1	1
2	1	1	1
3	1	1	1
4	1	–	1

(b) The elements in the directory 'Krokskogen'

Version	Jotunheimen	Jostedalsbreen	Gjende	Bessvann	Flatbreen	Kyrkja
1	1	1	1	1	1	–
2	1	1	1	1	1	–
3	2	1	1	1	1	1
4	2	2	1	1	2	1

(c) The elements in 'Nasjonalpark'

Table 4.3: The development of the fields

Notation	Description
Offset	A counter containing the offset to be added to each line number in the resulting set
AW	The delta set between the ancestor and the working copy file.
AC	The delta set between the ancestor and the current (repository) file.
MR	The merged set resulting of merging AW and AC
*.t	The examined triple to delta set * (* is AW, AC or MR).
*.a	The current triple's actions, '+' or '-'.
*.s	The current triple's sentence.
*.l	The current triple's line number.

Table 4.4: The notation used in the discussion of the merge algorithm.

1. Compare AW.l with AC.l. If AW.l is less than AC.l, AW.t is added (pushed) to MR.
2. If AW.l equals AC.l then the AW.s is compared with AC.s. If they are equal, then one of them is added to MR, else a conflict is found and the following action occurs:
   - The triple [+,line-number + offset,<conflict markers + AW.s>] is added to MR.
   - Offset is increased by one.
   - The triple [+,line-number + offset,<conflict markers + AC.s] is added to MR.
   - Offset is increased by one and AW.t is set to next triple in the delta set.
3. If AW.l is bigger than AC.l then AC.t is added to MR, AC.t is set to next triple, while AW.t is untouched.[4].

Conflict

A conflict occurs whenever both the current repository version and the working copy have changed the same part of a file. The file created by the merge process marks the conflicting areas. The user will also have access to three files the repository file, the ancestor file and the merged file. The user must then resolve the conflicts in the merged file. In table 4.5, the three versions used in examples 4.2 and 4.1 are listed. The repository file and the working

[4]This is achieved by using the Ruby functionality of 'redo'.

Original	Repository	Working copy
The Attack Drink with me Turning	On my Own The Attack A little fall of rain Drink with me Turning Finale	On my Own The Attack Drink with me Bring him home Turning Empty chairs

Table 4.5: The files used in the merge example

copy file are almost identical, but the bottom lines differ. This is a conflicting change; the other changes merge in just fine.

The sicmerge

```
On my Own
The Attack
A little fall of rain
Drink with me
Bring him home
Turning
<<<<<<<<<< working
Empty Chairs
==========
Finale
>>>>>>>>>> repository
```

Example 4.1

Diff3 -E -m

```
On my Own
The Attack
A little fall of rain
Drink with me
Bring him home
Turning
<<<<<<< work
Empty chairs
=======
Finale
>>>>>>> rep
```

Example 4.2

In example 4.1, the result of a merge from the build-in library is shown. This may be compared with example 4.2, which shows the result of a GNU diff3 merge.

4.6.3 Delta algorithms

Data differencing is the process of computing an invertible encoding of a target file, given a source file. The encoded data, called a delta, tend to be smaller than the original file, and is fine for storage and transmission. Some delta-generators also compress the encoding to achieve even smaller deltas.

Compressing and differencing improve the efficiency of storage and transmission of data. Recently there has been several projects aiming to improve HTTP performance [34] by transmitting deltas between servers and clients. If web servers could transmit deltas instead of whole web pages to browsers, the network load would be reduced. Even though their efforts have been directed at improvements of HTTP, the work done is also relevant for others who need an efficient delta algorithm.

There are two main classes of algorithms used in delta computations, although some have experimented with other solutions such as semantic diff achieved with suffix tree and history keeping [22], the **insert/delete** class and the **copy/insert** group. The former represents the target string as a sequence of insert and delete instructions, while the latter produces a delta consisting of copy and insert instructions, both from the source string.

The insert/delete class, implemented by GNU `diff` and RCS [33], produces a delta readable for humans as shown in example 4.3. The other class, copy/insert, produces binary deltas not readable for humans.

Vdelta and vcdiff [20, 19], zdelta [34] and xdelta [24] all use algorithms belonging to the copy/insert class.

4.6.4 The class of insert/delete algorithms

As previously mentioned, insert/delete algorithms create a sequence of insert and delete instructions. These algorithms implement the longest common subsequence algorithm, lcs. Computing lcs is a NP-complete algorithm. To be able to compute a delta, the implementations compare whole lines (separated with newlines). The algorithm does therefore not work on binary data. Another shortcoming of the algorithms is the failure to notice patterns which are repeated or reorganised [34, 23].

The UNIX diff-tool[5] [25] compares two files line-by-line, searching for differing groups of lines. The default diff format uses three change commands to describe the changes: 'a' for lines added, 'c' for changes or replacements and 'd' for deleted lines. The RCS format does not use the 'c'-command, creating a pure sequence of insert/delete commands. In figure 4.3 a typical delta from RCS is shown. Line numbers 3, 9, 47, 48, 50, 51, 52 and 93 are

[5] I refer to the *GNU* implementation.

CVS delta, RCS format

```
d3 1
a3 1
# $Id: delta.tex,v 3.9 2003/07/17 22:48:34 siri Exp $
a5 1
require 'rcs'
d9 1
d47 2
d50 3
d93 1
a93 1
    d['log-1.0'] = "msg"
```

Example 4.3

deleted and new lines are added at line 3, 5 and 93. Since two of these lines first where deleted and then added, those lines were really just changed.

As previously mentioned in this section, these algorithms do not work on binary data. This applies for *diff* too, and is the reason why CVS struggles with binary files as mentioned in section 3.2.2. It normally does not make sense to compare binary files on a line basis, *diff* itself try to check if the files are ordinary text files or not, by looking at the first few thousand bytes. If there are no null-bytes in that part, the file is defined to be text. Another problem with diff is the common problem with newlines, which is not the same characters at different operating systems (For instance, windows use a carriage return followed by a newline, but POSIX-compliant systems – like UNIX, uses just the newline. Mac use only the carriage return).

4.6.5 Delta compression tools, copy/insert algorithms

By tradition, differencing and compression has been two separate forms of data processing. In the last years, there have been attempts to combine these two, for instance vdelta/vcdiff,[6] zdelta and xdelta.

As mentioned before, these algorithms implement a copy/insert algorithm. The Lempel-Ziv string compression algorithm compress a string by substituting prefixes with a reference to the compressed data. This may be applied to the computing of a delta – the reference data would be the already referenced data, and the algorithm may be performed on the target data. The most basic algorithm of this is greedy; it passes through the file and picks the longest available match at any offset. This is not very efficient for lookups and there has been two data structures used to make this process more efficient; suffix tree and hash functions [23]. A suffix tree has one major problem, it requires linear space that may cause problems for large files,

[6] vcdiff is an improvement of vdelta [19].

Examplefile 1	Examplefile 2
foo bar zot	foo zot bar ssss

Table 4.6: The two files used in the diff example

diff -n

```
d2 1
a3 2
bar
ssss
```

Example 4.4

and thus a hash is the most common structure in use. The algorithms compute a hash value for each substring in the source version, and need to have a routine for dealing with collisions. To solve the problem with large files, many use a variant of sliding window, e.g. only hash a part of the file at any time. The data is already compressed by the delta algorithm, but vcdiff and zdelta applies a compression on top of that to achieve even smaller deltas. Zdelta uses the Huffman encoding provided by zlib; in fact, zdelta is a modification of zlib. Xdelta is based on rsync [36] and use an adler32-algorithm to compute the hash values (same as gzip).

The three implementations do differ in performance, but research shows that this is mainly in the area of compressing the delta [34]. They also differ in implementation, using different methods of achieving the optimal set of instruction and representation.

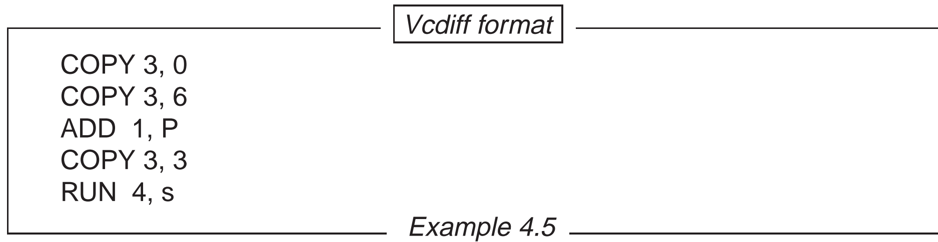
4.6.6 Delta generators examples

In table 4.6, the two files used in the diff examples are shown. The diff sequence describing this is shown in example 4.4.

First, starting at line number two, delete one line, this would be the 'bar' line. Then, from line three, add two lines, 'bar' and 'ssss'.

For the demonstration of a copy/insert-algorithm, two strings is defined; "foobarzot" and "foozotPbarssss". A copy/add instruction sequence is shown in vcdiff-format, in example 4.5.

Starting from position 0, copy three characters into the target string. Continue with copying three characters from position six in the source string (zot). Add one character, P, and copy three characters starting at position



Example 4.5

3 (bar). Last add four s, run means a repeating sequence. As shown, this algorithm detects and use the fact that parts of the string was just moved.

4.6.7 Determining file content

In order to offer keyword substitution the system needs to be able to determine if a file contains binary data or plain text. This is done by reading the 2048 first bytes of the file, counting all null-characters. At the moment, a file is marked as binary if any of those bytes are null-characters. This approach is quite similar to the one used by GNU diff [25].

Chapter 5

Functionality

This chapter describes the basic usage of *sic*, and is somewhat similar to a tutorial. The system is an ordinary version control system, featuring the most important commands such as commit, checkout, log and diff. The examples shown are taken from the case used in figure 4.6. Each example may contain more than one example, showing different options. Note that this is not a complete manual explaining all the different options. In appendix D the complete list of references are listed.

Almost all options take an optional list of files and directories. If no options are given, the system will take all elements in and underneath the current directory. All directories will be treated recursively: a directory's content is the elements belonging to that directory. This was discussed in section 4.6.1. Commands may be given at any stage in the project hierarchy, and will work from that directory. This may lead to situations where some part of the working copy is up to date and other parts are not. A project is only fully up-to-date, when *update* has been run from the project's root.

The change from version$_n$ to version$_{n+1}$ is defined to be the complete set of changes, including both changes in contents and changes in structure. The only command making changes public is *commit*, all other commands works just locally, and do not need to be online with the repository at the moment of typing.

Sic commands are typed like *this*, arguments and options are written like *–this*.

5.1 Create a project

To begin with, you will need to create a project in *sic*, unless someone else already has done this. This is the time to figure out a (preferably) meaningful name to the project and which repository to use. There is always the

Create command

```
sic -d /home/siri/sicrep create test
Create project:
Repository: /home/siri/sicrep, name: test
I'd like you to answer some questions for me.
Do you want me to do the initially check in of this project? [Y/n]
Do you want to check the contents of the configuration file
before the check in? [y/N]
Do you like this to be a working area after the check in?
  (Should the configuration file be kept in-place?) [Y/n] n

Thank you!
```

Example 5.1

possibility to create a new repository. The project should be prepared for check-in, for instance, generated files should be deleted prior to creation time. *Sic* will, however, skip certain files based on their file extension. When the preparation is finished, run *sic* with the *create* option. To determine which repository to use, *sic* runs a list of tests, the first match is used:

1. Option given at run-time.
2. The environment-variable SICREP.
3. The option sicrep in the personal .sicrc-file.
4. If none of these have a value, the program will check if there is a 'sicrep'-directory present in the user's home directory.

If none of these tests found a valid value, the program will stop execution with an error message.

The *create*-command should be run from the projects' root directory. *Sic* will then loop through all directories underneath adding elements to the project. It discards files it guess is generated, and this is decided from the file-extension. This list may be specified by the user, the default list of extension are: .bak, .old, , core, .o, .exe, .so, .SIC, CVS and .a. The system will ask three questions: If the hierarchy should be kept as a working copy, if the system should create the project now[1] and if the user wants to check and perhaps modify the list of elements to be included in the initial project. An example of this process is shown in example 5.1.

[1]Actually, the project will be created anyhow, but if the user answers no, the project will be empty.

Checkout command

```
sic -d /home/siri/sicrep co test
-*- Checkout -*-
Fetching version 1.
N   test/Krokskogen/Kikut
N   test/Krokskogen/Kampeknerten
N   test/Nasjonalpark/Jotunheimen/Flatbreen
N   test/Krokskogen
N   test/README
N   test
N   test/Krokskogen/Vidvangen
N   test/Nasjonalpark/Jotunheimen/Gjende
N   test/Nasjonalpark/Jotunheimen
N   test/Nasjonalpark/Jostedalsbreen
N   test/Nasjonalpark/Jotunheimen/Bessvann
N   test/Nasjonalpark
```

Example 5.2

5.2 Obtain a working copy (check out project)

When a project is created, everyone who wants to work on it needs to obtain a working copy. All work is done in a users working copy, and the user will from time to time share his work with the others, making his work public. To obtain a working copy, the user checks out the desired project, as demonstrated in example 5.2. The *checkout* command automatically, unless otherwise is asked for, chooses the latest version. There are two properties the user must know in advance: in which repository the project is located, and the project's name. To determine which repository to use, the same list as defined in section 5.1 is used.

Obtaining a specific version, example 5.3

Sometimes you want to obtain a specific version, either because there is something interesting in that version you would like to glance at or maybe because your program suddenly stopped working and you know it worked three days ago.

Sic have a special option, *–revision/-r* to specify the version (note the inconsistency in terminology). This option takes as argument a string, which is a pair of keyword and data, delimited by =. Valid keywords are 'version', 'date' or 'label'. If the keyword is omitted, 'version' is assumed. Some of the different possibilities are listed in example 5.3

More options to checkout

```
sic co --revision 3 test_run
-*- Checkout -*-
Fetching version 3.
N   test_run
N   test_run/README
N   test_run/Krokskogen/Kikut
N   test_run/Krokskogen/Kampeknerten
N   test_run/Nasjonalpark/Jotunheimen/Bessvann
N   test_run/Nasjonalpark
N   test_run/Nasjonalpark/Jostedalsbreen
N   test_run/AUTHOR
N   test_run/Krokskogen
N   test_run/Nasjonalpark/Jotunheimen
N   test_run/Nasjonalpark/Jotunheimen/Flatbreen
N   test_run/Nasjonalpark/Jotunheimen/Kyrkja
N   test_run/Krokskogen/Vidvangen
N   test_run/Nasjonalpark/Jotunheimen/Gjende

sic co -r 'date=2003-05-23' test_run
-*- Checkout -*-
Fetching version 6.
N   test_run
N   test_run/README
N   test_run/Krokskogen/Kampeknerten
N   test_run/Nasjonalpark/Jotunheimen/Bessvann
[Output removed]

sic co -r 'date=2003-05-19' test_run
-*- Checkout -*-
Fetching version 1.
N   test_run
N   test_run/README
N   test_run/Krokskogen/Kikut
N   test_run/Krokskogen/Kampeknerten
N   test_run/Nasjonalpark/Jotunheimen/Bessvann
[Output removed]
```

Example 5.3

Element_type

```
sic element_type
-*- Element types -*-
README                                  contains ASCII text
Nasjonalpark/Jostedalsbreen             is a directory
Nasjonalpark/Jotunheimen                is a directory
utsikt1.JPG                             contains binary data
Nasjonalpark/Jotunheimen/Bessvann       contains ASCII text
Nasjonalpark/Jostedalsbreen/Flatbreen   contains ASCII text
Krokskogen                              is a directory
Krokskogen/Kampeknerten                 contains ASCII text
Krokskogen/Kikut                        contains ASCII text
Krokskogen/Vidvangen                    contains ASCII text
Nasjonalpark/Jotunheimen/Gjende         contains ASCII text
Nasjonalpark                            is a directory
```

Example 5.4

5.3 Element types

Certain actions, for instance keyword substitution and diff require the data to be ASCII text. A keyword substitution on a binary file will most likely ruin the file, making it useless. The consequences for the diff are not so serious, but the diff algorithm depends on the data to be ASCII-text.

The software defines its own function to determine if a file consists of ASCII or binary data, and then whether a diff or merge should be performed. To examine the result of the build-in function, *sic* support an *element_type* command as shown in example 5.4.

5.4 Working on the project

After obtaining a working copy, the file and directories are all yours for editing and reorganising. Modifications are not shared with the others on the project until they are made public. The changes are made public when the developer runs the commit command. *Sic* uses the optimistic concurrency model, and it might have been changes in the repository. The possibility will be that your working copy is not up-to-date compared with the repository. Often, the developer also wants to know which changes he has made to the project. *Sic* offers a diff command, which will display the difference between the local copy and the revision in the repository.

Other useful information, aiding in the daily work on a project is status commands, comparing your working copy with the status in the repository. This information is quite useful when preparing a commit and you like to know what would happen. The commands *lstatus*, *status* and *update* are

Add

```
sic add AUTHOR
-*- Add -*-
Adding '/home/siri/fag/hfag/src/test/AUTHOR' to project.

sic add AUTHOR
-*- Add -*-
'/home/siri/fag/hfag/src/test/AUTHOR' is already in project.
```

Example 5.5

useful. The two first commands show the status for each element in the project (*lstatus* works only locally), while the last one updates your working copy to be up-to-date with the repository.

5.4.1 Reorganising the project

One of the more important goals in the design of *sic* was the ability to reorganise a project hierarchy. Whenever an element is moved or added, *sic* make sure that there do not exist any element with an equal name. The name is defined to be the absolute path within the project. *Sic* supports the reorganising commands add, move, delete and undelete, which are thoroughly explained in the following paragraphs.

Add, example 5.5

To add an element to the project, the command *add* followed by a list of items to be added. When the *add* command is given, the changes are just local and the changes will not be public before a commit[2].

Moving or renaming, example 5.6

An element may be moved inside the project. A move is defined as a change in the absolute path; it could be either a change in the directory it exists in or only the name of the file. A simple renaming of an element is also a move, so there is no special rename command. If the element in question is a directory, the directory with all its sub elements is moved. Theoretically, every single element underneath that directory is moved[3].

[2]There is one command which makes changes public, commit. All the others change only the working copy, enabling the system to group different changes into one set of changes. This also has the advantage of avoiding too much communication with the repository.

[3]The repository does not store the full path to each element, so the repository does not need to change every element.

Move

```
sic mv Jotunheimen/Flatbreen Jostedalsbreen
-*- Move -*-
Moving Jotunheimen/Flatbreen to Jostedalsbreen

sic mv AUTHOR author
-*- Move -*-
Moving AUTHOR to author
```

Example 5.6

Delete

```
sic rm Krokskogen/Kikut
-*- Delete -*-
The item 'Krokskogen/Kikut' (and all it's possible childrens will be
deleted next time you perform a commit.)
Do you want me to delete the locally file? [Y/n] n
```

Example 5.7

The *move* command needs at least two parameters, the name of the element to be moved, and the new name. The command may, however, get more than two parameters, provided that the last element in the list is a directory. If the last parameter is a directory, the files to be moved will be moved into the directory keeping the name. When more than two parameters are present, all elements are moved into the directory while preserving their base name. The latter are shown first in 5.6, where the file 'Flatbreen' is moved from the Jotunheimen directory to Jostedalsbreen, preserving its name. The second one is a rename of the file called AUTHOR.

Delete, example 5.7

The *delete/rm* option will remove element(s) from the project. It will still be present in all versions prior to the delete, but will be removed from later releases, unless someone adds it again. If the element is a directory, the directory including all its elements will be deleted, as shown in example 5.8

At delete time, the program will ask if you would like to keep the element or not (the element will anyhow be removed from the project on next commit).

Undelete, example 5.9

Undelete is the opposite of the 'delete' and will only work before the delete is committed to the repository. It assumes that the element was kept at deletion time. This is possible to bypass through the update command. The

Deleting a directory

```
sic rm Krokskogen/
-*- Delete -*-
D   Kampeknerten is marked for deletion
D   Vidvangen is marked for deletion
The item 'Krokskogen' (and all it's possible childrens will be
deleted next time you perform a commit.)
Do you want me to delete the locally file? [Y/n] n
```

Example 5.8

Undelete

```
sic undel Krokskogen/Kikut
-*- Undelete -*-
Krokskogen/Kikut is not marked for deletion anymore.
```

Example 5.9

file will be marked as missing and the newest revision in the repository will be delivered from the server.

5.4.2 Difference between a local file and the current

The *diff* operation will show the difference between the local file and the file in repository. It is useful to be reminded what work has been done. At the moment, *sic* calls upon the status method if a directory is given to diff. This will show if there are any changes in the structure and revision, but will not show differences for single files. The syntax for showing the differences are similar to the one provided by the `UNIX diff`-command, shown at the end of example 5.10.

5.4.3 Status of your project

The *status* option lets you check the current status of your working copy compared with the repository. Nothing is actually changed in the repository or the working copy. The option is added to fulfil the need for overview over the project and its files. This option is meant to avoid the common usage of update in CVS; users perform update to get an overview over which files are changed. The option take an arbitrary list of files and directories, if none is given, all elements underneath current directory is assumed, as shown first in example 5.11.

A file may be in one of the states given in table 5.1.

Diff command

```
sic diff README
-*- Difference -*-
README is equal with the repository.

sic diff Krokskogen
-*- Difference -*-
Project test_run is in version 5:

   Krokskogen/Kikut                              ASCII text
   Krokskogen/Vidvangen                             ASCII text
L  'Krokskogen/Kampeknerten' is locally modfied.      ASCII text

sic diff Krokskogen/Kampeknerten
-*- Difference -*-
0d 1
< Litt nord for Kampen
```

Example 5.10

Code	Explanation
	The file is current
L	File is locally modified
O	File is out-of-date and locally unchanged
B	The file is out-of-date and locally changed (changed both places).
A	The item is added locally (and not in repository)
M	The item is missing
D	The item is marked for deletion locally
R	The item is moved or renamed locally
D(r)	The item is deleted in the repository, but not locally
A(r)	The item is added in repository and not locally
R(r)	The item is moved or renamed in the repository

Table 5.1: Table showing the different states a file may have.

Status command

```
Krokskogen>sic status
  -*- Status -*-
Project test_run is in version 5:

  Kikut                                   ASCII text
  Vidvangen                               ASCII text
L  'Kampeknerten' is locally modfied.           ASCII text


>sic status
  -*- Status -*-
Project test_run is in version 5:

  Nasjonalpark                            Directory
  README                                  ASCII text
  Nasjonalpark/Jostedalsbreen                     Directory
  Nasjonalpark/Jostedalsbreen/Flatbreen           ASCII text
  AUTHOR                                  ASCII text
  Nasjonalpark/Jotunheimen                        Directory
  Krokskogen                              Directory
  Krokskogen/Kikut                          ASCII text
  Nasjonalpark/Jotunheimen/Kyrkja                 ASCII text
  Nasjonalpark/Jotunheimen/Bessvann               ASCII text
  Krokskogen/Vidvangen                          ASCII text
  Nasjonalpark/Jotunheimen/Gjende                 ASCII text
L  'Krokskogen/Kampeknerten' is locally modfied.      ASCII text
```

Example 5.11

Lstatus command

```
sic lstatus
The local changes:
A    AUTHOR
R    Nasjonalpark/Jostedalsbreen/Flatbreen
D    Krokskogen/Kikut
```

Example 5.12

Local status, example 5.12

Another way of getting overview over the working directory is the *lstatus*-command. This option shows only changes applied locally and does not communicate with the repository. This speeds up the process and use the same status codes as *status*, though only those which apply to local changes.

The lstatus-method use the status flag in the configuration file together with timestamps and checksums to determine if a file is changed or not. Whenever the content of a file is a result of a merge, and may possible contain a conflict, the checksum in the configuration file is not changed according to the changes done in the merge. The reason for this behaviour is the need of detecting if the file is changed compared to the repository, and needs to be checked in at next commit.

5.4.4 Update your files

The working model this system supports, assumes that developers share their work through a central repository. The process of obtaining others work are invoked with the *update* option. As for almost every option, you can choose to update only a given set of files; either by specifying files at command line or run the command from a subdirectory.

If the hierarchical structure is changed the update function will move the files accordingly. It will also detect if any files are missing, and retrieve these once more from the repository. This is shown in example 5.13.

When a file is changed both in working copy and in the repository, the program will try to merge the changes, unless the *–nomerge* option is true. This option causes the program to bypass the merge function and stores the current version and the common ancestor file as *filename-revision*. The user has the choice to use a third-party program to perform the merge, or not merge at all.

The merge function used by *sic* is based on the diff-library. It will try to merge changes to different parts of the files. If the changes apply to the same sentences, the library will mark these lines as conflicts.

Update command

```
sic up
-*- Update -*-
A  AUTHOR is added and not in repository yet.
Up-to-date: Krokskogen
Up-to-date: Krokskogen/Kampeknerten
Up-to-date: Krokskogen/Kikut
Up-to-date: Krokskogen/Vidvangen
Up-to-date: Nasjonalpark
Up-to-date: Nasjonalpark/Jostedalsbreen
Up-to-date: Nasjonalpark/Jostedalsbreen/Flatbreen
Up-to-date: Nasjonalpark/Jotunheimen
Up-to-date: Nasjonalpark/Jotunheimen/Bessvann
Up-to-date: Nasjonalpark/Jotunheimen/Gjende
L  README
M  'utsikt1.JPG' is missing, need to get the latest version from repository.
utsikt1.JPG is updated to be in revision 1
```

Example 5.13

Conflict

```
sic up
-*- Update -*-
C      AUTHOR

>less AUTHOR
<<<<<<< working
author: Siri Spjelkavik
=======
Author: Siri Spjelkavik
>>>>>>> repository
```

Example 5.14

5.4.5 Sharing your work (commit)

After working on the working copy, the developer will eventually want to share his work with the rest of the project group. This is achieved through the *commit* command. The only preparation required before a commit is making sure that all elements to be committed are up-to-date with the repository. If this is not the case, those changes must be incorporated, refer to section 5.4.4. It would anyhow be wise to examine the changes done prior to the commit. As mentioned earlier, the commands *status* and *diff* aids in this quest, refer to the section 5.4.3 and 5.4.2.

Because of *sic*'s concurrency model, the user's working copy may not be up-to-date with the repository. If so, the working copy needs to be updated and the modifications must be incorporated before the commit.

The process of committing work is divided into the following steps:

1. Check the status of the files to be committed. There might have been other changes to these files, if so, these changes needs to be integrated before the commit. If not, proceed to the commit part (last point on this list). At this stage, useful commands will be 'diff' and 'status'. If the files need to be updated, the commit will fail, as shown in example 5.16.

2. If there are modifications in the repository, those changes need to be included. The 'diff' command will show which changes are made, and the 'update' command will retrieve the newest version from the repository and try to merge this into the local document. The merge function will notify if it managed to merge the contents, otherwise the conflicts is marked in the document. The three files involved in this process, the common ancestor file, the repository file and the local file will all be available. After a merge it is recommended that the developer checks the document, in case the merge did something unexpected. All conflicts must be resolved by the user.

3. At this stage, everything should be in order to perform a successful commit. A commit is an atomic operation, which means that either everything or nothing is stored. The client will transport the changed files to the server and the server will return the revision numbers and the project version number. These values will be updated in the local configuration file, together with timestamp information and md5 checksums.

Commits works recursively, just like most of the others commands. If a directory is given as an argument, all elements belonging to that directory are added to the list of elements to be committed. If no elements are given

Commit command

```
sic ci
-*- Commit -*-
Project is now in version 4
'/hom/siri/test/Nasjonalpark/Rondane' is added to the repository (revision is 1).
```

Example 5.15

Failing commit

```
sic ci
-*- Commit -*-
Nasjonalpark/Jotunheimen/Kyrkja
B AUTHOR
'AUTHOR' must be updated. Run sic update
NotUpdate
```

Example 5.16

as argument to the command, commit will pick all elements in the directory the command was issued, including all underneath elements.

If any of the files have an associated log message (see section 5.5), this will be used for the specific file, all the others will get the optional log message, given as part of the command.

5.5 Reading and creating a log message

Creating a log message

The user can either associate a message to selected elements or give a general log message for all files at the time of commit. If provided, the general message will be used for every changed file that does not have an associated message.

To associate a message to a file, use the *logmsg* command which take as arguments the message and a list of files. To set a general message at time of commit, use the *–message/-m* argument. The associated message will be stored in the configuration file until next commit is performed.

If neither an associated message nor a general message is present, the element will not have any message.

Retrieving the log information, example 5.17

To retrieve the log information for elements in the project, the *–log* option is used. As almost all the others commands, an arbitrary list of elements may

Log command

```
sic log README Krokskogen/
-*- Log -*-
Project: test
Repository: /home/siri/sicrep

Working file: README
Current revision:  1          Number of revisions: 1
==========================================================

Working file: Krokskogen
Current revision:  1          Number of revisions: 1
==========================================================
```

Example 5.17

be given, unless all the files in the project are wanted. If a revision does not contain a log message, the revisions information is not shown unless the *show-all* option is given to the log command.

This command does not work recursively on directories, but will take all elements underneath if no arguments are specified.

5.6 History

The history command will print a list over all logged commands performed on a project. The default behaviour is to print user, date and the name of action. It is also possible to limit the list, at the moment each line is matched against the expression given by the user, as shown in example 5.19

5.7 Export

The export option does an ordinary check out, but it does not create a configuration file, so the received version is not a working copy. This option is useful if you want to distribute the project.

5.8 Configuration files

Sic uses two different configuration files. Each working project has a configuration file situated in a directory, .SIC, which is in the root directory to the project. This file stores meta information about the working directory, including information belonging to each element. The information stored is

History command

```
sic history
-*- History -*-
siri   Thu Mar 13 16:27:21 UTC 2003   create
siri   Thu Mar 13 16:27:27 UTC 2003   checkout
siri   Thu Mar 13 16:27:58 UTC 2003   commit
siri   Thu Mar 13 16:36:03 UTC 2003   status
siri   Mon Mar 24 16:50:13 UTC 2003   update
siri   Mon Mar 24 16:50:52 UTC 2003   update
siri   Mon Mar 24 16:51:11 UTC 2003   checkout
siri   Mon Mar 24 16:51:28 UTC 2003   update
siri   Mon Mar 24 16:52:46 UTC 2003   checkout
siri   Mon Mar 24 16:52:46 UTC 2003   checkout
siri   Mon Mar 24 16:54:37 UTC 2003   export
siri   Mon Mar 24 16:56:08 UTC 2003   checkout
siri   Mon Mar 24 16:56:15 UTC 2003   history
```

Example 5.18

History command with options

```
sic history --limit checkout
-*- History -*-
siri   Thu Mar 13 16:27:27 UTC 2003   checkout
siri   Mon Mar 24 16:51:11 UTC 2003   checkout
siri   Mon Mar 24 16:52:46 UTC 2003   checkout
siri   Mon Mar 24 16:52:46 UTC 2003   checkout
siri   Mon Mar 24 16:56:08 UTC 2003   checkout
```

Example 5.19

Export command

```
-*- Export -*-
sic export
Version is: 2
The project 'test' is retrieved.  This is not a working copy,
use the 'checkout' command for that.
```

Example 5.20

sicrc file

```
# This is a .sicrc-file.

$SSH="/local/ssh/bin"  # note the envrionment variable is SIC_SSH
$RUBY="/hom/siri/bin/ruby"
$EDITOR="emacs"
```

Example 5.21

name, identification, revision, checksum, timestamp and status messages. It should not be necessary to edit this file manually. The file is only text, meaning that this may be done if the format is held in a consistent state. Lines which do not correspond to the given format will be skipped. In appendix A a complete configuration file is shown.

The other configuration file is global for one user and will apply every time *sic* is run, independent of repositories and projects. *Sic* tries to evaluate a file called '$HOME/.sicrc'. If another file is wanted, use the *–sicrep* option to specify the file to be used.

The syntax of this file is 'variable=value' and must contain valid Ruby code, since the file is only evaluated through the Ruby parser. All global variables may be defined in that file, but the most common are $EDITOR, $SICREP, $UMASK and $SIC_IGNORE. The precedence of order is values given at command line, values from .sicrc, .SIC and at least environments variable. An example file is shown in example 5.21

5.9 Working off line

Sometimes you work on computers not connected to the computer the repository resides on (the server). This may be if the server goes down, there are problems with the network, working off-line or perhaps at home with only a modem available. In these situations it may be useful for some options to still be functional. All reorganising commands as add, delete and move works only locally, so it is possible to work on the structure of the project and not only the contents. The *lstatus* command shows which files you have changed since last time you updated. *Sic* does not keep a local cache like Subversion does, so diff is not available.

5.10 Getting help

In appendix D a complete list of all the commands and options available in *sic* is given. If there is anything wrong with the combination of argument given, *sic* will print an error message, as shown in 5.22.

Invalid argument(s)

```
sic undel                    # undel should be called with an argument
Sorry, I need some elements to restore.

Usage:
sic [nsdmlegqv] command [command options and argument]

For more info, try --help option
```

Example 5.22

```
.
|-- Krokskogen/
|   |-- Kampeknerten
|   |-- Kikut
|   `-- Vidvangen
|-- Nasjonalpark/
|   |-- Jostedalsbreen/
|   `-- Jotunheimen/
|       |-- Bessvann
|       |-- Flatbreen
|       `-- Gjende
`-- README

4 directories, 7 files
```

Figure 5.1: The initial layout of the project.

Sic also support the help-option, printing a short usage help.

The example used in figure 4.6 is shown in figure 5.2, and the initial layout of the directory is shown in 5.1

Help

```
sic --help
This is sic, Sic Is not CVS
Usage:
sic [nsdmlegqv] command [command options and argument]

For more info, try --help option
    --name,    -n
    --sicrep,  -s   Repository
    --dir,     -d
    --message, -m   Log message on this action
    --logmsg,  -l   Associate a message to a file
    --server,  -e   Name of server
    --help,    -h   Print this message
    --queue,   -q   Queue message
    --version, -v   Print version number
    --force,   -f   Force creating of files

command is one of the following:

        commit/ci     Commit
        create        Create a project
        update/up     Make this project up-to-date
        status        Show the status to each file
        lstatus       Show the local status to the files
        history       Show the history to the project
        checkout/co   Check out a copy of a project
        add           Add a element to the project
        log           Show the log messages
        export        Export this project
        move          Move/rename a element
        delete        Delete element from project
        undel         Restore a element that are marked as deleted
        diff          Show difference against the repository

The options 'add', 'move' and 'delete' does not occur in the
repository before a commit has been performed.
All options communicates with the global repository, except
'add', 'move', 'delete', 'undel' and 'lstatus'.

Files may be added more than one at a time.
```

Example 5.23

```
kampeknerten: ex> sic create test
Create project: test
Repository: /home/siri/sicrep, name: test
I'd like you to answer some questions for me.
Do you want me to do the initially check in of this project? [Y/n]
Do you want to check the contents of the configuration file
before the check in? [y/N]
Do you like this to be a working area after the check in?
  (Should the configuration file be kept in-place?) [Y/n] n

Thank you!

kampeknerten: src> sic co test
N   test/Krokskogen/Kikut
N   test/Krokskogen/Kampeknerten
N   test/Nasjonalpark/Jotunheimen/Flatbreen
N   test/Krokskogen
N   test/README
N   test
N   test/Krokskogen/Vidvangen
N   test/Nasjonalpark/Jotunheimen/Gjende
N   test/Nasjonalpark/Jotunheimen
N   test/Nasjonalpark/Jostedalsbreen
N   test/Nasjonalpark/Jotunheimen/Bessvann
N   test/Nasjonalpark

kampeknerten: test>sic add AUTHOR
-*- Add -*-
Adding '/tmp/test/AUTHOR' to project.
albino: test>sic ci -m "Added a file for the authors" AUTHOR
-*- Commit -*-
Project is now in version 2
'/tmp/test/AUTHOR' is added to the repository (revision is 1).
kampeknerten: Nasjonalpark/Jotunheimen>sic add Kyrkja
-*- Add -*-
Adding '/tmp/test/Nasjonalpark/Jotunheimen/Kyrkja' to project.
kampeknerten: Nasjonalpark/Jotunheimen>sic ci Kyrkja
-*- Commit -*-
Project is now in version 3
'/tmp/test/Nasjonalpark/Jotunheimen/Kyrkja' is added to the repository (revision is 1).
/tmp/test/Nasjonalpark is updated to be in revision 2
kampeknerten: test>sic rm Krokskogen/Kikut
-*- Delete -*-
The item 'Krokskogen/Kikut' (and all it's possible childrens will be
deleted next time you perform a commit.)
Do you want me to delete the locally file? [Y/n] n
albino: test/Nasjonalpark>sic mv Jotunheimen/Flatbreen Jostedalsbreen/
-*- Move -*-
Moving Jotunheimen/Flatbreen to Jostedalsbreen
albino: test>sic ci
-*- Commit -*-
Project is now in version 4
/tmp/test/Nasjonalpark is updated to be in revision 3
/tmp/test/Nasjonalpark/Jostedalsbreen is updated to be in revision 2
/tmp/test/Nasjonalpark/Jotunheimen is updated to be in revision 3
/tmp/test/Nasjonalpark/Jostedalsbreen/Flatbreen is updated to be in revision 2
```

Figure 5.2: The actions from the example in figure 4.6.

Chapter 6

Conclusion and future work

6.1 Comparing *sic* against other systems

It is not easy to create a test suite or to benchmark this program. Some systems like PRCS have been using a couple of versions of emacs and gcc to compare efficiency. The design of *sic* has been concentrated on usability and friendliness, not efficiency.

The easiest way to compare and judge this program is to look at how different systems have implemented various functionalities. The systems chosen were CVS, because of its widespread use, and Subversion, the system intending to replace CVS.

Since *sic*'s functionality is based upon CVS, CVS was a natural system to choose. Subversion tries to replace CVS and not add any new functionality in the first version. These two systems were chosen because of the similarity and relevance compared to *sic*.

Versioning model:

As discussed previously in section 3.2.3, Subversion only defines one single version number to the whole project, each item does not have its own revision tag. CVS has no version scheme for projects, since CVS does not have any concepts of projects. Each file is given a new revision number for each change made.

Sic combines these two schemes by giving each file a revision number and each version a unique number. In this model, a version consists of a set of elements in different revisions. One revision may be part of many versions, but one version has only one revision for each element.

Network access:

Subversion keeps a local cache for each working copy, containing the last version retrieved from the repository. This enables certain commands to be computed in the client without communicating with the repository. This has various advantages such as reducing the network access and possibility to work on systems which are not connected to a network at the time. Another benefit occurring from this is the possibility for the client to send deltas to the server. In version control systems without this cache, the server dispatch deltas while the client transmits whole files. The Subversion method reduces network latency and the system does not need to wait for the server to respond, and consequently is more efficient.

Sic, like CVS, does not keep a cache and they depend on communicating with the repository to generate diffs and status. *Sic* is, however, able to do some computations without communicating with the repository. The command *lstatus* shows the changes to local files, including deleted, moved and added elements. CVS does not have this opportunity at all, although some scripts which enable queueing of actions like adding files exist.

Binary files handling:

CVS use the GNU diff-library to generate and patch the deltas. The diff-library implements a longest common subsequence algorithm based on splitting the data into lines or segments delimited by a newline character. This approach is not suitable for binary data, since tempering such as inserting newlines may ruin the data. To get binary files treated correctly in CVS, the user must remember to pass the appropriate option to CVS. This action seems to be forgotten by users all the time.

Sic use zdelta to compute the difference, a library which performs well on both binary- and ASCII-data. Another advantage of using this library is the additional compressing of the deltas, implying the deltas to be as small as possible and hence reduce network latency.

To show differences (diffs), *sic* uses a Ruby library implementing the GNU diff-library on ASCII-files. This library is also the base for the build-in merge algorithm. The merge algorithm computes two set of deltas, these deltas are merged to one delta set and then applied to the common ancestor file. This approach is similar to the one used by GNU diff3.

Subversion also uses a binary delta algorithm. All files have a set of properties: one of them stores the mime-type to the file. Keyword substitution, textual merging and displaying differences are just done on files with a mime-type starting with text.

Commits:

CVS's commits happens in pieces and not atomically. *Sic* sends a list of changes to the server, and the server will not write this to file unless everything goes well (this is quite similar to Perforce, section 3.2.5). Subversion also guarantees atomic commits.

6.2 Shortcomings and future work

A system will never be quite finished, and there is lots of functionality you wished you had implemented. There are the usual bunches of more advanced options to the different commands. In chapter 2 a list of wanted functionality was presented. Access control list, filters (mail and general), symlinks, branches and keyword substitution is not implemented at this stage.

6.2.1 Configuration file

The project's configuration file is at the moment based on a home-brew format and is parsed using regular expression. If the format had been something more standard, like XML, it would be easier to further extend and maintain this part of the program. When using element tags, such as used in XML, it is not necessary to specify all values an element may have; it is only necessary to define those values one particular element need.

6.2.2 Commit after conflict

In most projects it is undesirable that files with unresolved conflicts are committed. There are several reasons why this is undesirable: executable programs will most likely not compile nor interpret and conflicts should be resolved by persons involved in the changes made. Often it will also be easiest to resolve conflicts shortly after time of creation. As a result of this and the fact that conflicts should be resolved, *sic* should detect if a file, which are a result of a conflict, is scheduled for commit with unresolved conflicts. The system should not deny developers from doing this, but it should hinder to avoid that this happen unintentionally.

At the moment, a merged file keeps the checksum it had before the merge. This is done so the file will be marked as locally changed. This problem may be solved by storing the checksum after the merge together with the conflict keyword as status. Another approach would be to store the checksum after merge as the files checksum and set status to conflict. Both methods require

the update and commit functions to pay special attention to files which have conflict as status to perform any necessary actions.

6.2.3 Graphical User Interface

> *Thus, an effective navigation tool should provide the ability to select any version and to propagate changes to later versions along the version tree.*
>
> *– Lee, Chang and Narayanan [22]*

There are a lot of interesting issues to be discussed in the matter of graphical user interfaces (GUI). CVS has been used for a long time, and several tools has been developed to aid in the use of CVS. For instance, a module for browsing CVS repositories in a web browser, together and graphical front-ends. New systems will often have some sort of a graphical front end, and this should be offered in addition to a classical command line tool. It is important to offer a command line interface, since it may be situations where it is not convenient to use a GUI.

ICE [41] offers the SKATE browser, which is implemented to be similar to any ordinary browser, letting you browse through your project. Subversion [10] uses Apache as server, hence let the users browse the repositories in a browser. There has been experimented with using druby over HTTP. If this succeed, it should be quite straightforward to support a HTTP front-end to *sic*.

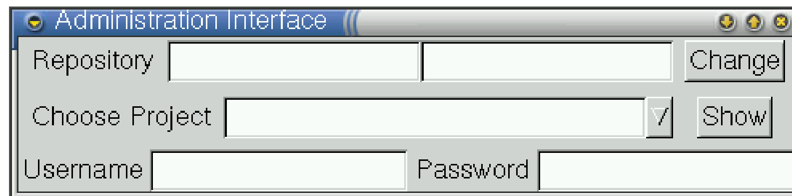
A GUI to *sic* should consist of two different interfaces; one to administer the working copy and one to browse the repository. The GUI to browse the working copy should be an advanced file browser, letting the user reorganise the structure, customising the log message and deleting and adding files. The *add* command should parse the file system, showing all files and let the user only choose files which are not in the project yet. The interface should even let the user choose a file and open this in the preferred editor for editing.

The interface to the repository should be more like a classical configuration tool, letting the administrator customise filters and file permissions.

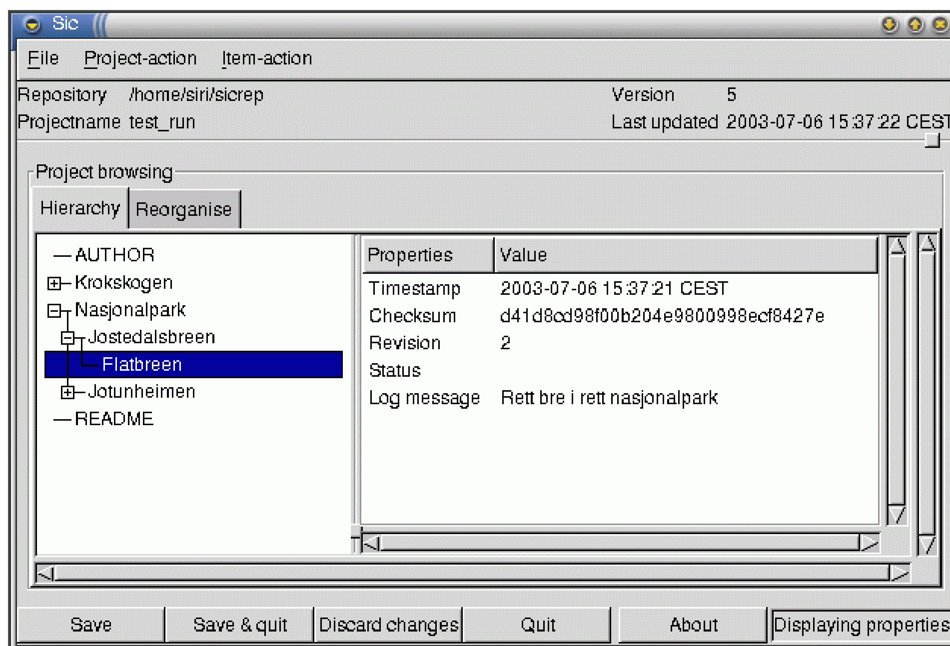
An example of what this may look like is shown in figure 6.2.3 (generated with glade [17]).

6.2.4 Current version and branch

In project running for some time, you may want to change which version the development should continue from. This will mean to 'discard' latest



(a) Repository



(b) Project

Figure 6.1: Examples showing how the different GUIs may look.

changes and go back to an earlier version. This is achieved through a branch, even though the user not explicit requires this. The new branch will be the new main trunk.

This challenge could be solved trough a current pointer, which always point at the current version. Most of the time, it will point at the last version committed. This feature depends on the branch functionality, since this must be implemented through with branches. *Sic* is designed to support branches, but this is not currently implemented.

6.2.5 Rsync

The rsync algorithm [35, 37] deals with the problem of synchronising the content of two files over a slow communication link. A common method for updating a file to be identical with another is to transmit the delta between the two files and using this to update the file. Normal delta algorithms work on local files, thus requiring both files to be present in one end of the connection. Rsync uses two checksums; a weak rolling checksum and a strong checksum, MD4. A rolling checksum is an updateable checksum; the checksum covers just a chunk of the file at the time, sliding through the data, computing each checksum using the previous chunck's checksum as base for the new one [37].

> *The rsync algorithm efficiently computes which parts of a source file match some part of an existing destination file. These parts need not be sent across the link; all that is needed is a reference to the part of destination file.*
>
> *– The rsync algorithm [35]*

Thus, the only data needed to be sent in verbatim is the data which does not exist in the source file. All the other data is just sent as references. This use of references to the data is similar to the one previously discussed in section 4.6.5, about the copy/insert class of differencing algorithms. In fact, the xdelta algorithm [37, 23] is inspired by rsync.

Using rsync for transmitting data is efficient. At the moment, *sic* only sends deltas from the server to the client; the other way whole files is transmitted. Using rsync for the transmission from the clients would speed up commits and receive some of the benefits Subversion has from storing a cache, but not of course the offline advantages such as diff.

6.2.6 Sub-projects

A secondary effect of CVS organisation of repository and the no-existing connections between directories are the possibility of sub-projects. When a check-out is performed in CVS, the user retrieves every files in the directory it asked for and all files underneath. This directory must not be at the top level at all. If an user request for a check out of 'foo/bar', then the directory foo and bar will be created, but the foo-directory will only consist of the bar directory and the CVS information.[1]

Sub-projects like this is quite useful, and are used for instance among students. Often, students attend different courses, all with different assignments and projects. To illustrate, we use three students, Katie, Jane and Bob. Katie cooperates with Jane on the course inf310, while Jane and Bob works together on the course infserv. Both groups use Jane's repository to their work. Jane's repository is organised with a top-level directory called course where all her data regarding her different courses stay. When Katie wants to check out a working copy, she will type *cvs -d <path to jane's repository> course/inf310/project*. Bob will do the same to obtain the infserv code and Jane is still able to store her own notes and other stuff in the inf310-directory, stuff that Bob does not need to read or change. The same applies for Katie and Jane's work. Both Bob and Katie, may if they desire, have their own repository with their personal notes to the courses, since CVS use the repository specified in each directory. Each subdirectory in a project may therefore have different repositories.

This opportunity to specify different repositories and sub-projects is highly appreciated in certain settings, especial those *sic* are written for. This is a feature that really should be implemented.

6.2.7 Stand-alone server

At the moment, *sic* does not support a stand-alone server, which always run letting clients to connect it. The only way to access any repository is by letting the client invoke the server at start time.

There are several issues which becomes more important if such a server should be supported. The server needs to be thread-safe and there are certain security issues involved. Security issues are not addressed in this thesis, since the program demands that all users have access the server computer as well as (obviously) access on the different clients. The connection phase between the client and the server is out-sourced to a third-party program of the users desire, default SSH.

[1]CVS supports modules, which is aliases so users do not need to specify long paths of directories.

Appendix A

Configuration file

```
# --* administration *--

# This is a administration file to the sic version control To support
# offline development, the file will queue certain messages and these
# should be 'committed' next time there is communication with the
# repository.  Another solution would be to queue this to a local
# repository, which could be merged.

repository: '(user)(host)(/path)'
project: 'project-name'
last-updated: 'timestamp'
id: 'version number'

# --* contents *--

# The files and directories belonging to this project
# Status is stuff like delete and add
# Example of status: ('a') -- add

# This is confusion by obscurity, but anyhow, this is not for humans
# its for computers.

A(id(status|message|checksum|timestamp|revision))
fileb(id(status|msg|a64c15b6fae219259256cd2fcf6927e2|tmstamp|2))
directory => (id(status|message|timestamp|revid)) (
   C(id(status|message|checksum|timestamp|revision))
   directoryf => (id(status|message|timestamp|revision)) (
      b(id(status|message|checksum|timestamp|1))
   )
)
```

Appendix B

PStore objects

```
 1 #!/usr/bin/env ruby
 2
 3 # graph.rb, direct acyclic graph
 4
 5 require 'sic/Zdelta'
 6 require 'md5'
 7 require 'parsedate'
 8
 9 class SicNode
10    attr_reader :revision, :name, :user, :ident, :mode, :timestamp
11    attr_accessor :full_path
12
13    def initialize(user, log, name, mode, id, revision)
14       @username = user
15       @log = log
16
17       @name = name
18       @timestamp = Time.now.gmtime        # store time in GMT
19       @next_version = Array.new
20       @prev_version = Array.new
21       @mode = mode
22       @revision = revision
23       @ident = id
24    end
25
26    def add_revision
27    end
28
29    def up_to_date
30    end
31
32    def get_log
33    end
34
35    def parent(item)
36    end
37
38    # Returns true if this name is different from the last revision
39    def changed_name?
40    end
41
42 end
43
44 # Node for all the contents
45 # file is true if delta is the whole contents false if it is a delta
46 class Filenode < SicNode
```

```
47    attr_reader :checksum, :content
48
49    def initialize(user, log, name, new_file, mode, revision, rep, id)
50      super(user, log, name, mode, id, revision)
51
52      @content = File.join(rep, File.join('files',@ident))
53
54      File.open(@content, File::CREAT|File::TRUNC|File::RDWR) {|f|
55        f.print new_file
56        f.flush
57      }
58
59      if revision%15 == 0 then
60        File.copy(@content, "#{@ident}-#{revision}")
61      end
62
63      @checksum = MD5.md5(IO.readlines(@content).to_s).hexdigest
64      @file = true
65    end
66
67    # This is not longer the newest version ;(
68    # Create the backward delta, store in current, create the new
69    # node and store a reference in @next_version
70    def create_new(user, log, name, file, rep, mode = 0644)
71      # First, temporary store the old content
72      to = Tempfile.new("old")
73      to.close
74      File.copy(@content, to.path)
75
76      # create the new and store
77      new_node = Filenode.new(user, log, name, file, mode,
78                              self.add_revision, rep, @ident)
79      new_node.full_path=full_path
80
81      new_node.parent(self)
82      @next_version.push(new_node)
83      if @revision%15 != 0 then
84
85        # compute delta
86        from = File.open(new_node.content)
87
88        del = Tempfile.new("delta"); del.close
89        Zdelta.delta(from.path, to.path, del.path)
90
91        del.open
92        @content = del.gets
93        del.close(true)
94
95        from.close
96        to.close(true)
97
98        # The whole file is no longer stored here
99        @file = false
100     end
101     return new_node
102   end
103
104   # return true or false if this is up-to-date
105   # If checksum is equal we defines the file to be up-to-date
106   def up_to_date(checksum)
107   end
108
109   def ancestor
110   end
111
112   # Returns the revision-item asked for.
113   def get_revision(revision)
114   end
```

```
115
116     # Check out a copy of this version
117     def checkout(projectname)
118     end
119
120     def get_version(original, delta, result)
121     end
122 end # class FileNode
123
124 class Directorynode < SicNode
125     # content contains pointers to the current version of files which this
126     # directory consist of
127     def initialize(user, log, name, delta, mode, rev, id)
128         super(user, log, name, mode, id, rev)
129         @delta = delta
130         @content = Hash.new
131         @link = Hash.new                      # fra til
132     end
133
134     def create_new(children, user, log)
135         node = Directorynode.new(user, log, @name, @delta, @mode,
136                                  add_revision, @ident)
137         children.each {|ident, child|
138             node.add_element(child)
139         }
140
141         node.full_path = full_path
142
143         @next_version.push(node)
144         node.parent(self)
145         return node
146     end
147
148     def add_element(node)
149     end
150
151     def delete_element(node, user, log)
152     end
153
154     def get_contentclone
155     end
156
157     def up_to_date(content_list)
158     end   # up_to_date
159
160     # check out project
161     def checkout(projectname)
162     end
163
164     def build_new(admin, user, log, answer)
165     end # build_new
166 end   # class Directorynode
167
168 class Adminnode
169     attr_accessor :current, :projectname
170
171     def initialize(name)
172         @projectname = name
173         @version = 0                     # version counter
174         @current = @version
175         @version_name = Hash.new       # contain a reference from current name
176         @version_id = Hash.new         # and id to the object
177         @ident = -1                      #
178         @add_element = Hash.new        #
179         @delete_element = Array.new #
180         @labels = Hash.new              # Let useres associate a label to a
181                                          # specific version
182     end
```

```
183
184   def inspect
185     puts "\nInspect of Adminnode−object:\n\n"
186     puts "Projectname: #{@projectname}"
187     puts "Version     : #{@version}"
188     puts "Current     : #{@current}"
189     puts ""
190     puts "Number of elements in last version: #{@version_name.size}"
191     puts ""
192   end
193
194   def set_current(ver)
195     @current = ver
196     @version = @current
197   end
198
199   # Set label.  Note that each label must be unique
200   def set_label(label, ver=@current)
201   end
202
203   # Add one to the revision−counter, so each file have its own id.
204   def get_next_revision
205   end
206
207   # Returns the next version number
208   def get_next_id
209   end
210
211   def get_next_unique
212   end
213
214   # Put node with full_path into the registers (hashes)
215   def add_element(node)
216   end
217
218   # Delete node from the hashes
219   def delete_element(node)
220   end
221
222   def delete_element?(id)
223   end
224
225   def get_element(name, id=false)
226   end
227
228   def get_added_element(rev)
229   end
230
231   def register_delete(node)
232   end
233
234   def register_add(node)
235   end
236
237   def get_label(label)
238   end
239
240   def check_root(id, roots)
241   end
242
243   # Search for the version at a given moment.
244   # The version at that moment is the version which is closest
245   # in time committed before the requested time
246   def get_time(time, roots)
247   end
248
249   private :check_root
250 end
```

Appendix C

Protocols

The commands *status*, *update* and *commit* all have to compare the current state of the repository with the current state to the files in the working directory. This requires some communications between the client and the server. All the three functions use one method which retrieves informations from the server, compare it with the working directory, and then return an array with a state for each element. In table 4.2, C.1 and C.2 the necessary communication betewen server and client is shown.

Client	**Server**
– get the status for each file (look at table 4.2 on page 28) – compare the information and request delta/files for every file which is not up to date. Send local revision together with id.	
	– Compute delta or file and send back. New files must be send whole, and send delete and move messages
– save and store the files, create, delete and move files if those messages are received. – update the local configuration file	

Table C.1: Table showing the communication for the update option

Client	Server
– get the status for each file (look at table 4.2). All files **must** be up to date. – If the old copy is present, send delta, otherwise send the whole content of every file which is modified locally. Also send information about moved, deleted and new files.	
	– Store the new contents, update revisionsids and versionid and the current versionnumber. Return the new revisions and versions id
– Update the local configuration file with timestamp, revision and versionnumber.	

Table C.2: Table showing the communication for commit

Restructuring commands

To show what lies behind the restructuring commands, the *delete*-actions are here shown.

1. User types delete or rm and a list of files

   The program register this in the working project file. The change will not be official until the next commit

2. User commits his changes

   (a) Server notice the delete and adds the file(s) to the list of deleted files. It also creates a new version

   (b) Server respond to the client in the usual way, and also notify that some files is deleted

   (c) Client deletes the item(s) from the project file.

Appendix D

Reference

D.1 SIC

Sic Is not CVS

SYNOPSIS

sic [fhlmnpsv] command [command options and argument]

DESCRIPTION

Sic is a version control system, which allows you to keep old versions of files. It keeps a log over who did what when, and perhaps even why. Sic operates on projects, which are a hierarchically structure consisting of directories and files. Sic allows the user to reorganise and to keep track of renaming of files.

Sic use a single master copy, called the repository. The repository is centralised and contains all the different revision and versions of all items.

OPTIONS

–force

Set the force-option to true. Mostly for debugging purposes. When the force-option is true, existing projects are replaced and certain files would be overwritten.

–help

Display general information and a summary of all commands.

–label 'name'

Associate a label to a version, giving a version a symbolic name that is easier to remember than a generated number.

–message 'mesg'

Use 'mesg' as the general log message. 'mesg' will only be used for those items that has no specialised log message associated.

–name 'name'

Option used together with the –server to specify the name of the project the server should be started on.

–server 'port-number'

Start the program as a server on the given port. If no port is given as argument, the default value '9000' is used.

–sicrep 'path' or 'user:host:path'

Specify the repository to be used. If the repository resides on a local computer, the full path will be enough. Otherwise the user-name, host and path on a remote computer must be given. –sicrep '/home/siri/sicrep' –sicrep 'siri:login.ifi.uio.no:/home/siri/sicrep'

Normally this is only used at create or checkout-time. A working copy has the repository specified in the '.SIC/conf_file' at the root of the project. If the –sicrep-option is used in a working copy, it will override the value given in the conf_file.

–version

Print sic's version number.

Sic commands

The sic-commands are listed underneath. For more information refer to their man-pages:

add

checkout/co

commit/ci

create

delete

diff

export

element_type

history

log

logmsg

lstatus

move

status

update/up

undel

Most of the commands take an arbitrary list of files, if no files are given, the commands will work on all elements underneath the current directory, including all subdirectories.

FILES

.SIC/.conf_file, $HOME/.sicrc

SEE ALSO

sic_diff, sic_update, sic_status, sic_log, sic_checkout,sic_commit, sic_create

AUTHOR

Siri Spjelkavik, <siri.ifi.uio.no>

D.2 Create a project in sic

Synopsis

sic create projectname

Description

Create a project in sic. This command should be invoked from the root-directory of the wanted project.

The repository must be given, either with an environment variable (SICREP), the default ˜/sicrep or as an argument. If the repository is reachable through the local file system, the repository only need to contain this with full path. If not, the host and username must be written in this format: ’<user>@<host>:path’, for instance:

sic –sicrep ’siri@kampeknerten.bofh.no:/home/siri/sicrep’

The username is siri, the host is kampeknerten.bofh.no and the repository is /home/siri/sicrep.

Sic lets you examine what files to be a part of the project at creation-time. This is done by showing the file in the editor specified by the envrionment variable EDITOR, and may be overridden in the local configuration file ˜/.sicrc.

Alternate Names

None

Changes

Create a project in the repository and optional a working copy.

Requires

Access to the repository the project should stay in.

Switches

–message (-m) ’log-message’

–sicrep (-s) ’repository’

D.3 Retrieve a copy of a project

Synopsis

sic checkout projectname

sic export projectname

Description

Creates a copy of the project, either as a working copy (includes the .SIC/conf_file) or as a copy, suitable for exporting.

The repository must be given, either with an environment variable (SICREP), the default ˜/sicrep or as argument. If the repository is reachable through the local file system, the repository only need to contain this with full path. If not, the host and username must be written in this format: '<user>:<host>:path', for instance:

sic –sicrep 'siri:kampeknerten.bofh.no:/home/siri/sicrep'

The username is siri, the host is kampeknerten.bofh.no and the repository is /home/siri/sicrep

It is possible to specify which revision to retrieve. The –revision switch consist of a pair of key and value.

Alternate Names

co

Changes

Create a working copy.

Requires

Access to the repository

Switches

–sicrep (-s) 'repository'

–revision (-r) revision.

version-number

version=<version-number>

date=<YYYY-MM-DD>

label=<label>

D.4 Update the working copy

Synopsis

sic update <files>

Description

Bring the working copy up-to-date by incorporating changes. For each item a line a character is shown reporting which action occured.

L: Locally changed

M: Missing item

A: Added item

R: Moved item

D: Deleted item

C: Conflict

G: Merge

U: Updated

Alternate Names

up

Changes

Working copy

Requires

Access to the local repository

Switches

–nomerge
–revision(?)

D.5 Commit the changes to the repository

Synopsis

sic commit <filelist>

Description

Share the changes in the working copy with the repository by sending these to the repository. An optional message may be given by the –message switch.

Alternate Names

ci

Changes

The items in the repository and the working copy.

Requires

Access to the repository

Switches

–message (-m) 'log-message'

D.6 Reorganising commands

Synopsis

sic add|delete|undel <filelist>

sic move SOURCE DEST

sic move SOURCES... DIRECTORY

Description

Options for reorganising the structure of the working copy. Add adds files and directories, delete removes elements, undel undo a previously performed delete. Move renames or moves an item.

The changes are scheduled for changes in the repository, but not changed there until commit occurs.

If a directory is marked for delete, all elements underneath will also be removed.

Alternate Names

None

Changes

Working copy

Requires

Access to the working copy

Switches

None

D.7 Display differences between two items

Synopsis

sic diff files

Description

Display the difference between two items. For directories, it will only call upon the ’status’-method. For files it will display the differences in a UNIX diff-style notation.

Alternate Names

None

Changes

Nothing

Requires

Access to the repository

Switches

–revision (-r)

D.8 The status options

Synopsis

sic status|lstatus|element_types <files>

Description

Display status information about <files>. The lstatus shows changes done in the working copy, while the status command displays changes both in the repository and locally.

The command element_types will show a list of which type the program treats each element as (ASCII or binary).

Alternate Names

None

Changes

Nothing

Requires

The status-command requires access to the repository, while the lstatus-command and element_type do not require anything else than a working copy.

Switches

None

D.9 The log message

Synopsis

sic log|logmsg <filelist>

sic history

Description

Three commands taking care of the retrieval and setting of log messages and history.

The log command retrieves the log messages for all revisions of the optional list of files or all items from the given position.

The logmsg command associate a specialised log message to the list of elements taken from the mandatory list of files (which may consist of one element). At checkin-time, these elements will get the specialised log message instead of the optional general message.

The history command will display a list of actions occurred in the repository.

Alternate Names

None

Changes

Nothing

Requires

Repository and a working copy. History needs only a repository.

Switches

None

D.10 Sic's configuration files

Synopsis

sic –sicrc configurationfile (default is ˜/.sicrc)

Description

The .sicrc-file is the user possibility to customise certain values every time sic is run.

The variables and their default values:

$SIC_IGNORE = the environment variable SIC_IGNORE || '\.bak|\.old$|˜$|core|\.o$|\.exe$|\.s

$UMASK = File.umask

$HOME = ENV['HOME']

$USER = ENV['USER']

$EDITOR = ENV['EDITOR']

$RUBY = 'ruby'

$SSH = the environment variabel 'SIC_SSH' || 'ssh'

Bibliography

[1] Brad Appleton, Stephen P. Berczuk, Ralph Cabrera, and Robert Orenstein. Stramed lines: Branching patterns for parallel software development. http://www.cmcrossroads.com/bradapp/acme/branching/. An earlier revision of this paper appears in the Proceedings of the 5th Annual Conference on Pattern Languages of Program Design (PLoP'98).

[2] Arch. http://www.hackerlab.com.

[3] Ulf Asklund and Boris Magnusson. A case-study of configuration managment with clearcase in an industrial environment. Department of Computer Science, Lund institute of Technology, Lund University, 1997. LU-CS-TR:97-184.

[4] Ulf Asklund. Configuration managment for distributed development – practice and needs. Technical report, Department of Computer Science, Lund Institute of Technology, Lund University, 1999.

[5] Malcolm Atkinson, Franqois Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The object-oriented database system manifesto. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 223–240, Kyoto, Japan, 1989.

[6] David Beazley. SWIG. http://www.swig.org, 1995. SWIG is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages.

[7] B. Berliner. CVS II: Parallelizing software development. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 341–352, Berkeley, CA, 1990. USENIX Association.

[8] BitMover Inc. BitKeeper. http://www.bitmover.com/bitkeeper.

[9] G. M. Clemm. Replacing version-control with job-control. *ACM SIGSOFT Software Engineering Notes*, 14(7):162–169, 1989.

[10] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. Subversion: The definitive guide. http://svnbook.red-bean.com/, February 2003. Draft, revision 5113.

[11] Ben Collins-Sussman. The Subversion project: Building a better CVS. http://www.linuxjournal.com/article.php?sid=4768, February 2002.

[12] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Computing Surveys (CSUR)*, 30(2):232–282, 1998.

[13] Rational Software Corporation. ClearCase. http://www.rational.com/products/clearcase/.

[14] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of database systems*, chapter 17, pages 537–538. The Benjamin/Cummings Publishin Company Inc., 2 edition, 1994.

[15] Per Cederqvist et al. *Version Managment with CVS*. Signum Support AB, 1993.

[16] Karl Fogel. *Open Source development with CVS*, 2000.

[17] User interface builder for the GTK+ toolkit and GNOME. http://glade.gnome.org.

[18] Luigi Semenzato Joshua MacDonald, Paul N. Hilfinger. Prcs: The project revision control system. Master's thesis, University of California at Berkeley, Department of Electrical Engineering and Computer Sciences and National Energy Research Scientific Computing Denter, Lawrence Berkeley National Laboratory, 1998.

[19] David G. Korn and Kiem-Phong Vo. Engineering a differencing and compression data format. Technical report, AT&T Laboratories - Research.

[20] D. Korn, J. MacDonald, and J. Mogul. RFC 3284 the vcdiff generic differencing and compression data format, June 2002.

[21] Kaz Kylheku. Meta-CVS. http://users.footprints.net/~kaz/mcvs.html, January 2002.

[22] Byong G. Lee, Kai H. Chang, and N. Hari Narayanan. An integrated approach to version control management in computer supported collaborative writing. In *Proceedings of the 36th annual Southeast regional conference*, pages 34–43. ACM Press, 1998.

[23] Joshua P. MacDonald. File system support for delta compression. Master's thesis, University of California at Berkeley, Department of Electrical engineering and computer sciences, 2000.

[24] J. MacDonald. Versioned file archiving, compression, and distribution.

[25] David MacKenzie, Paul Eggert, and Richard Stallman. *Comparing and Merging Files*. GNU, for diffutils 2.8.1 and patch 2.5.4 edition, April 2002.

[26] Yukihiro Matsumoto. Ruby. `http://www.ruby-lang.org`, 1995. Programming Language.

[27] Perforce Software. Perforce. `http://www.perforce.com/`.

[28] Christopher Seiwald. Inter-file branching. `http://www.perforce.com/perforce/branch.html`, March 1998. Perforce Software.

[29] Sleepycat Software. Berkeley db. `http://www.sleepycat.com/`.

[30] Secure shell. `http://www.ietf.org/html.charters/secsh-charter.html`. Active IETF Working Group.

[31] Subversion. `http://subversion.tigris.org`.

[32] David Thomas and Andrew Hunt. *Programming Ruby – The pragmatic programmer's guide*. Addison Wesley Longman, Inc., 2001.

[33] Walter F. Tichy. RCS: A system for version control. *Software Practice and Experience*, 15(7):637–654, 1985.

[34] Dimitre Trendafilov, Nasir Memon, and Torsten Suel. zdelta: An efficient delta compression tool. Technical report, Department of Computer and Information Science, Polytechnic University, Brooklyn, 2 2002.

[35] Andrew Tridgell and Paul Mackerras. The rsync algorithm. Technical report, Australian National University Canberra, ACT 0200, Australia, 2000. `http://rsync.samba.org/`.

[36] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, The Australian National University, April 2000.

[37] Andrew Tridgell. The rsync algorithm. `http://olstrans.sourceforge.net/release/OLS2000-rsync/OLS2000-rsync.html`, July 2000. Presentation from Ottawa Linux Symposium, Ottawa Congress Centre, Ottawa, Ontario, Canada.

[38] Andre van der Hoek. Configuration management and open source projects. In *Proceedings of the 3rd International Workshop on Software Engineering over the Internet*, Limerick, Ireland, June 2000.

[39] Guido van Rossum. Python. `http://www.python.org`, 1991. Programming Language.

[40] Larry Wall. Perl. `http://www.perl.org`, 1987. Programming Language.

[41] Andreas Zeller. Smooth Operations with Square Operators - The Version Set Model in ICE. Technical Report 95-08, Gausstrasse 17, D-38092 Braunschweig, Deutschland, 1995.

[42] *CVS TODO*.

[43] Gnu dbm. `http://www.gnu.org/software/gdbm/gdbm.html`.