

FoCAL Project
A conflict free version control system

Petter Börjesson, Anders Karlsson
{widmanbo, andekar}@student.chalmers.se

December, 2008

Abstract

Today most software projects make use of version control systems in order to divide work among many people working on the same project and still being able to glue it together to a whole. In this project we have looked further into the different systems available and observed some cases where most systems do either something wrong, something that could be a lot simpler, or forces the user to intervene in cases where it should not be necessary causing extra work for the user. A prototype has been implemented that has some of the regular features that one expects to find in a version control system as well as avoiding most of the faults found in existing system.

Contents

| | | |
|----------|--------------------------------------------------|-----------|
| 1 | Background | 5 |
| 1.1 | Version Control System | 5 |
| 1.2 | Distributed VCS | 5 |
| 1.3 | State-based VCS | 6 |
| 1.4 | Operation-based VCS | 7 |
| 1.5 | Scenarios | 8 |
| 1.6 | Suggested VCS | 9 |
| 2 | Theory | 10 |
| 2.1 | Domains and operations | 10 |
| 3 | Implementation | 14 |
| 3.1 | Internal representation | 14 |
| 3.1.1 | Directory level | 14 |
| 3.1.2 | Content level | 14 |
| 3.1.3 | Description | 15 |
| 3.2 | External representation for repository | 17 |
| 3.3 | Commands | 17 |
| 3.3.1 | Local commands | 17 |
| 3.3.2 | Distributed commands | 19 |
| 3.4 | Network Interface | 19 |
| 4 | Results | 21 |
| 4.1 | Working prototype | 21 |
| 4.2 | Limitations and future work | 21 |

List of Figures

| | | |
|---|------------------------------|----|
| 1 | A server based VCS | 5 |
| 2 | A distributed VCS | 6 |
| 3 | Criss cross merge | 6 |
| 4 | Tricky move | 8 |
| 5 | Pull over network | 20 |
| 6 | Push over network | 20 |

1 Background

1.1 Version Control System

A Version Control System (VCS), or Revision Control System, is a system that helps to manage multiple versions of documents or information and is today an integral part of many different kind of projects. Almost always the rule is to use a VCS during software development to keep track of different versions of application source code and other files and to allow different people in the project to work on the same files simultaneously. Some benefits of a VCS are that it becomes easy to track who made what changes, revert to previous versions and merge different branches of a project.

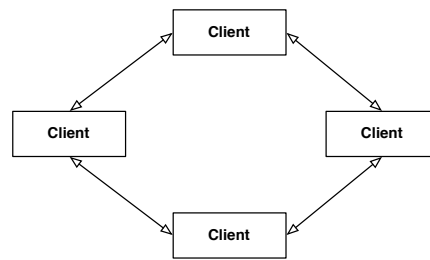


Figure 1: A server based VCS

Version control systems can be categorized into two different classes. The traditional VCS use a centralized server (fig. 1) which have a repository where all files are stored and all VCS functions are performed. This server needs to have some method of managing access to the controlled files, otherwise users trying to update files simultaneously might overwrite each other's work. A simple way to manage this is to simply lock files so that only one user has access to the file at a time. This means that a user “checks out” a file and while he has control over the file only he can write to it, though other users can still read it. When the user is done updating the file he performs a “check in” and the file will be updated. Another method is to allow multiple users to edit the same file simultaneously. When the first user “checks in” the file can be updated as before but when the next user “check in” the file the VCS will have to merge the different versions of the file. If the two users has changed the same piece of information a conflict might occur that the user must resolve.

1.2 Distributed VCS

Another approach is to use a distributed VCS (fig. 2). Instead of using a client-server approach with a centralized server, a peer-to-peer approach is used. This means that instead of having a single centralized repository for the files each user will have his own working copy of the repository. Also, it means that every repository is a potential branch. When changes are made to a repository they are saved and can then be exchanged between users, thus repositories can be synchronized.

In a distributed system the users choose whom to grant access to commit their code to the repository instead of letting a centralized server have a list of allowed users that can commit their changes to the server. One of the most well-known distributed VCS currently is Git (Torvalds, 2008). Since everyone has their own repository locally on their computer, they will be able to keep on contributing to the project even when no internet

connection is available. Also since everyone has their copy of the repository, it does not matter much if some hardware problems occur and data is lost, if it happens one can just “pull” a friends repository and continue from their version.

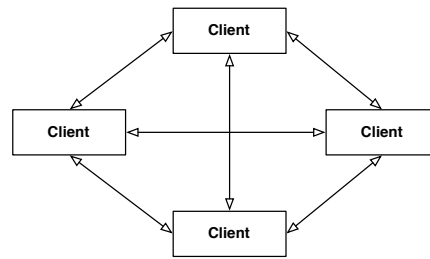


Figure 2: A distributed VCS

1.3 State-based VCS

A state-based VCS saves the current and previous versions of a file and when a file is updated it merges the old version with the newly received one to create a new version of the file. An example of a conflict is if two different persons change the same line in the same file and then when the system tries to merge the two versions it will not be able to decide which line to use. Merging is commonly done with the 3-way-merge algorithm (Khanna, Kunal and Pierce, 2006) where the merge is based on three copies, two new and one old common ancestor.

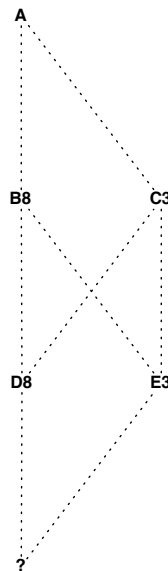


Figure 3: Criss cross merge

With the 3-way-merge algorithm there occur some conflicts that can be hard to solve automatically in a desired way. An example of such a conflict that can easily occur is the classical criss-cross merge. fig. 3 represents a snapshot of the history of a file where time goes downwards. Each letter is what was changed in that version and the number after the letter denotes which line was edited in this snapshot. The question mark should be a merge between the D version of line 8 (version D had the B8 version in history) and

the E3 version of line 3 (it has version C3 in history). Unfortunately there is no single ancestor for the merge that will avoid an unnecessary conflict when 3-way-merge is used.

If the ancestor chosen is B, there will be a merge conflict at line 3 since D has version C in history and E has version E but B has none of them. If the ancestor is C then there will be the same unnecessary conflict at line 8 because D will have version D and E will have version B but C will have none of them. If the ancestor chosen is A there will be a conflict on both lines.

A problem that is worse than the already mentioned conflict can also occur, if the later updates(D8,E3) were undo's of the earlier updates(B8,C3), then picking either B or C as ancestor will merge something wrong. If A is used this will not happen, but there will be a maximized number of unnecessary conflicts instead. In summary the state-based system often gets unnecessary conflicts that requires user intervention.

1.4 Operation-based VCS

In contrast to state-based VCS an operation-based VCS does not keep track of different versions of a file. Instead it captures all changes made to a file as operations. A version of a file is then simply the composition of all operations that have been applied to the file in that version. Instead of only considering the end results of a version as in state-based systems, an operation-based builds up the current version by considering all operations that has been performed on each line of every file. A merge between two branches in a operation-based system is only the union of the set of changes in the two branches.

An operation in operation-based systems is often associated with changing something in a fixed place in a file and so the result of applying many operations might be dependent on which order they are applied. For example if two operations change the same line of text, the end result will be that of the last applied change. It might even be the case that the data an operation is intended to change does not even exist yet if operations are applied in a different order. When sending many operations between repositories this can lead to problems as it is not always possible to tell in which order the patches should be applied.

Conflicts occur also in operation-based systems, one way to find possible conflicts is to check whether there exists operations that does not commute locally, if two operations do commute locally they are considered a good merge candidate (Lippe and van Oostrom, 1992).

Two operations $O_a O_b$ with starting state δ commute locally if they have an equivalence relation \approx such that $O_a(O_b(\delta)) \approx O_b(O_a(\delta))$. But two operations that do not commute locally does not necessary imply a conflict, only that it might be a conflict, most systems let the user resolve these possible conflicts. Although involving the user when a conflict like this appears can be quite problematic, in some operation-based systems it requires changing the history of the repository. One change that can be necessary is to erase a previously recorded change that placed the system in the conflict. In a better operation-based system it should not be allowed to change the history of a repository, it should never be allowed to remove something that has been changed, instead the inverse of the change that is to be undone should be added.

1.5 Scenarios

There are many problematic situations that can and quite often do arise while using a VCS. In fig. 4 is a tricky case where Alice creates a file that she pushes to Bobs repository. Alice decides to move the file to location C, but Bob moves the file to location B. When the two of them decide to merge their branches, most version control systems will have an unnecessary conflict or simply do the wrong thing, since it cannot decide where the file should exist after the merge.

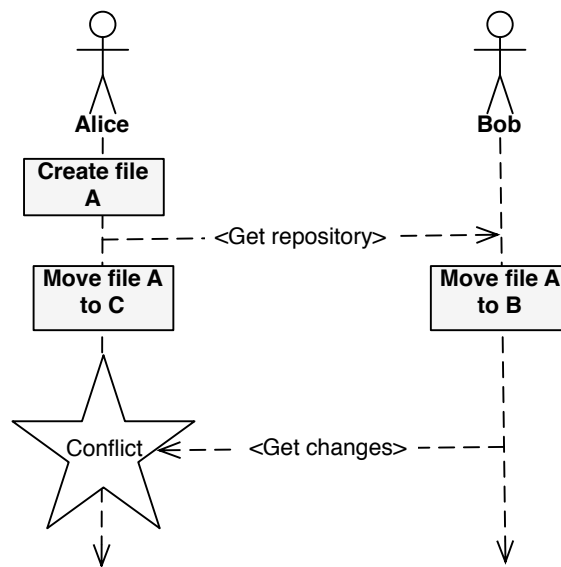


Figure 4: Tricky move

Probably most state-based VCS will do similarly as Subversion (CollabNet, 2008), the repository simply deletes file A and saves file B and C as new files without a connection to each other, even if B and C should be handled as the same file. This scenario will not be flagged as a conflict since Subversion does not recognize that this happens. If only Alice decides to move the file, and Bob decides to edit the file, a conflict will occur when they try to merge the repositories since Subversion does not recognize that the file edited by Bob is the same file as the one Alice has moved.

When this scenario is tested in an operation based VCS like Darcs (Roundy, 2008) there is actually still a connection between file B and C so that when a change occurs in file B and this is pushed to Bob who owns file B the change from file C will be merged with file B. But only Bob will have the file B and only Alice will have file C, the last applied patch will be the one that decides where the file should exist. Darcs will flag this scenario as a conflict, i.e. it cannot represent such a merge. The result of a merge in Darcs is dependent on the order that the changes are applied, Darcs will try all possible “non conflicting” permutations of the changes which will result, in the worst case, in exponential complexity.

1.6 Suggested VCS

The idea behind the suggested system is that it should be an operation based VCS that avoids unnecessary conflicts by using well documented ideas and a theory that is easy to understand. A repository is a set of patches where each patch is a set of changes or operations that have been performed on the files controlled by the system. The system should be conflict free, i.e. all patches are globally non-conflicting, which informally means that it should not make any difference to the result in which order patches are applied to a repository. If the order of the patches has no meaning i.e. they are globally non-conflicting, the system will avoid the problem that Darcs suffer from when merging, it will have a complexity of $O(n)$ instead of a exponential worst case.

The system will only handle text files and the idea is to give each piece of info an identity. This means that each line will have a position in a file that never changes and this makes it possible to refer to a line of text independently of the history. This is achieved by saving text as a tree where lines are indexed with rational-numbers. When an insertion is made that text is assigned an index between the positions it were inserted at.

Some of the expected benefits and assumptions of the suggested VCS are:

- Distributed
 - Every person involved has her own branch/es
 - Security (you only pull from the ones you trust)
- Most of the unnecessary conflicts occurring in state-based can be avoided
- Merging becomes fun (not as many unnecessary conflicts)
- No combinatorial explosions when applying patches like in Darcs
- No need to go back in history when a conflict occurs, all conflicts can be solved by adding a change
- Solves the problem of tracking moved files
- No need for a server constantly running
- Simple to reason about
- Simple to use
- Simple to implement

2 Theory

The idea behind this VCS is fairly simple and easy to understand. The system manages repositories, which is a collection of changes. A change can be described as one or more operations that change the state of a repository, each change also has an unique identifier. With this information we can describe a repository in a more formal notation as:

$$Repo = (Set(C_{ID} \times C))$$

The empty change, a change that does not perform any operation and intuitively does not change the state of a repository, is represented by c_0 .

A repository can be represented in two different ways. First, there is the representation which is used in the system and second, a repository can also be saved on disk in the Unix file system. Similarly to (Löh, Swierstra and Leijen, 2007) we will refer to the data structures used in the system as the internal representation (I), the directories and files where the data is saved on disk as the external representation (E). Where $I = Repo$.

When a user start the program, it needs to be able to read the external representation and when operations are performed it need to be able to update the external representation with the new information. To accomplish this the program needs some function to perform these actions. There should exist a function:

$$\rho :: I \rightarrow E$$

Which writes a Repo to disk and also the inverse of this function, which read a Repo from disc, is needed:

$$\rho^{-1} :: E \rightarrow I$$

Ideally, ρ should strive towards upholding $\forall i \in I : \rho^{-1}(\rho(i)) = i$ so that we always get back the same I after writing and reading a repository to disk without any added noise or data corruption. It would be very convenient to have this property but it will most likely not be possible to guarantee that this holds. For ρ^{-1} on the other hand, it should hold that $\forall e \in E : \rho(\rho^{-1}(e)) = e$. This is very important because if this holds it means that every possible I can be represented by the system without changing the meaning of the external representation. It is also important to notice that E is not always the same as the last time the program saved I to disk as E can also be modified by a user with the standard Unix tools.

2.1 Domains and operations

To summarize, the notation used is the following:

- External representation: (E)
- Internal representation: (I)
- Change: (C)
- where $I = Repo$: (Set ($C_{id} \times C$))
- c_0 is the empty change

To be able to work with a repository we have the following operators:

- $\rho : I \rightarrow E$ and $\rho^{-1} : E \rightarrow I$
 where:
 $\forall i \in I : \rho^{-1}(\rho(i)) = i$
 $\forall e \in E : \rho(\rho^{-1}(e)) = e$
- $\oplus : C \rightarrow C \rightarrow C$
 $c_1 \oplus c_2 = c_1$ composition c_2
 where:
 Identity: $c_0, \forall c \in C : c \oplus c_0 = c$
 Symmetric: $c_1 \oplus c_2 = c_2 \oplus c_1$
 Associative: $(c_1 \oplus c_2) \oplus c_3 = c_1 \oplus (c_2 \oplus c_3)$
- $\ominus : C \rightarrow C \rightarrow C$
 $c_1 \ominus c_2 = c_1 \oplus$ invert c_2
- invert : $C \rightarrow C$
 invert $c = c_i$ where
 $\forall c \in C : \exists c_i$ st. $c \oplus c_i = c_0$

Here we have the most important operations that a user need to be able to do on a repository. Every operation that adds a new change to a repository also generates a new unique ID for that change. Having unique IDs for all changes are important so a repository can keep track of all changes and this makes it possible to perform the same change more than once.

$$init : \begin{cases} I := \{\} \\ E := \rho(c_0) \end{cases}$$

The init operation creates an empty internal representation of a repository.

$$whatsnew : \begin{cases} I_{new} := \rho^{-1}(E) \\ c_1 := \bigoplus_{(c_{id}, c) \in I} c \\ c_2 := \bigoplus_{(c_{id}, c) \in I_{new}} c \\ c_n := c_2 \ominus c_1 \\ \text{Print } c_n \text{ to the user} \\ I \text{ unchanged} \\ E \text{ unchanged} \end{cases}$$

To see what has been changed since the user last recorded to a repository two changes are created. One from the current external representation and one from the current internal representation. The changes that are in E but not in I is calculated and printed to the user.

$$\text{revert} : \begin{cases} I \text{ unchanged} \\ E := \rho \left(\bigoplus_{(c_{id}, c) \in I} c \right) \end{cases}$$

It takes the unchanged internal representation and replaces the old external representation with the current internal representation, changes in the external representation that are not recorded will be lost.

$$\text{record} : \begin{cases} c_1 := \bigoplus_{(c_{id}, c) \in I} c \\ c_2 := \rho^{-1}(E) \\ c_n := c_2 \ominus c_1 \\ I := I \cup \{(c_{id}, c_n)\} \\ E \text{ unchanged} \end{cases}$$

It creates a change from the external representation, by subtracting the internal representation from the current external representation. Finally this change is added to the internal representation together with a new unique identifier.

$$\text{add } f : \begin{cases} I := I \cup \{(c_{id}, a_f)\} \\ E \text{ unchanged} \end{cases}$$

Where a_f is the change that adds the file f . The change is added to the internal representation and the external representation is left unchanged since the file has already been created.

$$\text{remove } f : \begin{cases} I := I \cup \{(c_{id}, \text{invert}(a_f))\} \\ E := \rho \left(\bigoplus_{(c_{id}, c) \in I} c \right) \end{cases}$$

It will find the change that adds the given file to the internal representation, it will then create an inverted change and by adding this to the internal representation, the file will be removed.

$$\text{move } f_0 \text{ } f_1 : \begin{cases} I := I \cup \{(c_{id1}, a_{f_1}), (c_{id2}, \text{invert}(a_{f_0}))\} \\ E := \rho \left(\bigoplus_{(c_{id}, c) \in I} c \right) \end{cases}$$

A move operation is just removing an item and then adding it at the new location. As two changes are added to the repository each change need a fresh ID, c_{id1} and c_{id2} . $c_{id1} \neq c_{id2}$

$$\text{unrecord } c : \begin{cases} I := I \cup (c_{id}, \text{invert}(c)) \\ E := \rho \left(\bigoplus_{(c_{id}, c) \in I} c \right) \end{cases}$$

An unrecord of a change, c , is performed by adding the inverted change, $\text{invert } c$, to the internal representation with a new fresh ID. Unrecord is needed because it is nice to be able to undo an add or record operation.

$$\text{pull } I_{remote} : \begin{cases} I := I \cup I_{remote} \\ E := \rho \left(\bigoplus_{(c_{id}, c) \in I} c \right) \end{cases}$$

A pull operation calculates the difference of the remote and local internal representations and adds the missing changes to the internal representation.

3 Implementation

The implementation of the system follows the theory closely and as such there are two ways to represent a repository. The internal representation, which is the data structures used in the program to represent a repository, and the external representation, which is the way a repository is represented in the Unix file system.

3.1 Internal representation

The implementation of the internal representation follows the idea that a repository is a set of changes. The following Haskell data structures are used to represent a repository in the system. The implementation is split in two parts, one that handles the directory level of a repository and one that handles the content of added files.

3.1.1 Directory level

- **data** $Repo = Repo [String] DirChange$

In theory, a repository is a set of changes paired with change IDs, $Repo = Set(C_{ID} \times C)$. The implementation follows model theory but handles the information a bit differently. A *Repo* consist of a list of *String* which are the IDs of all applied patches and a *DirChange*.

- **type** $DirChange = [(Integer, (Atom String), FileChange)]$

A *DirChange* is the model of a change and it is a list of the files that have been added to the repository. Each file is a tuple that has a unique identifier, a list of tuples with paths where the file exist along with the weight for that path, and a *FileChange*.

- **type** $FileChange = Tree String$

This is just a synonym for a *Tree String* and it is a change to the content of a file.

- **type** $Atom a = [(a, Weight)]$

An *Atom* is a list of tuples that pairs some *a* with a *Weight* of type *Int*. *Atom* is used both to keep track of file paths in *DirChange* as well as text in *Tree* which can be seen below. The weight in the *Atom* is needed since the repository should be globally non conflicting, by having weights the order that the changes are applied in does not matter, the result will be the same no matter in what order they are applied. So if the same change is applied several times, the weight of the change will be larger than one. The weights makes it easy to satisfy the properties of the set of changes that we saw in the theory. For example, to invert a change, the same change is used but with the inverted weight.

3.1.2 Content level

- **data** $Tree a = Tree (Atom a) (Chunk a)$

A *Tree* that holds some data *a*. In the case of this system *Tree String* is used.

- **type** $Index = Rational$

A *Rational* number is used as *Index*, so that a node always can be inserted between two others.

- **type** *Chunk* $a = [(Index, Tree\ a)]$
Chunk can be seen as a list of lines with an index, where each line can have sub lines.
- **type** *Path* = [*Index*]
All lines of text in a file can be found by a unique path in the *Tree* of a file. This type represents the path of *Rational* which determine the position of a line in the file.

3.1.3 Description

In the system, information is exchanged between repositories by sending patches. In the theory chapter we saw that repositories exchange information by sending changes to each other and so it would make sense to say that a patch is just another name for a change. Because of this a repository can be seen as a set of patches just as well as a set of changes and this is the way the system is implemented.

A repository is a set of patches paired with their IDs and this is the only information that needs to be saved to correctly represent a repository. In practice it is inefficient to only save patches separately as this means that a repository state needs to be recalculated very often while the system is running. Because of this, the system caches the latest version of the repository at all times to speed things up. The *Repo* structure keeps track of all the patches it consist of at the moment with a list of patch IDs and it has a *DirChange* which is the result of applying the \oplus operator on all patches of the repository to obtain one change that represent all the content of the repository.

As a patch is a change and a repository is a set of changes, a patch can be seen as a repository with only one change. Because of this it makes sense to say that a patch is represented by a *DirChange* and a unique patch ID. With this representation we can implement the \oplus operator as a function $DirChange \rightarrow DirChange \rightarrow DirChange$. With this function in place, the *Repo* structure becomes very easy to obtain and update. When a patch needs to be added to a repository, the patch ID are just added to the list of patch IDs and the *DirChange* of the patch is merged with the *DirChange* of the repository with the \oplus function.

In this implementation, as we have seen, the *DirChange* of a repository is the combination of all the *DirChange* of its patches but, as we also need to be able to send patches to other repositories, we need to be able to find the *DirChange* of any single patch. To do this we also save the *DirChange* of each patch to a separate file named with the patch ID and so we can easily find any patch again.

Just as the content of a repository in theory is saved in a lot of different patches the same goes for the content of files. But when the current *DirChange* of a repository is cached we handle the *FileChange* of each element in the *DirChange* separately and cache the current content of each file. For each file handled by the repository a new file is created in the file system that caches the current *FileChange* for that file and this new file is named by the file ID.

When a file is added the content is represented by an empty tree, $Tree [] []$, and when something is recorded, $Tree [] Chunk$. This $Chunk$ will contain all the text of the file where each element of the $Chunk$ can be seen as one line of text. Each element of the $Chunk$ has a $Rational$ which is the index of the line it represents, a $Tree$ that holds the line itself and, possibly, sub chunks.

We have now seen that both patches and repositories are represented by a $DirChange$ and that the \oplus function can merge two $DirChange$. Combined with the fact that both file paths and all lines in the content of files are tagged with weights, this gives the system some good properties. First, this ensures that a patch can always be applied to a repository and it will make sense independently of the context. Second, it will not lead directly to conflicts, instead there will only be de-normalized files or trees in the repository.

The implementation has functions both to read and write a $Repo$ from and to disc. All file level information in a repository is saved in a repository file and as we just saw the content of each file, $FileChange$, is saved in its own separate file. The reason for this separation, in contrast to saving all information including file content in the repository file, is that it would be a lot of information for the read function to obtain at once otherwise. Haskell uses lazy evaluation and this works in our favor when a $Repo$ is read from disc because of the separation mentioned above. This because the content of each $FileChange$ will only be read from disc when it is needed because of the lazy evaluation and so we can avoid reading unnecessary information.

These functions to read and write a repository from and to disc correspond to the ρ and ρ^{-1} functions from the theory. Unfortunately it is not always easy to uphold the invariants we want for these functions as described in the theory. This because of the differences between how the programs data structures handles information compared to how the Unix file system works. For example it could be the case that if a conflict occurs in the internal representation, it is not always possible to construct an external representation without changing the representation.

With some effort the problems of constructing an external representation can be circumvented and this program strives to do so in as many cases as possible. Currently there is only one known case when this will pose a problem and that is when there is a naming conflict between two files, i.e. when two files have the same name on the same path. This can happen if 2 different branches are merged, in one of the branches a file A is moved to `/tmp/record` and in the other a B which is different from A is moved to `/tmp/record`. When these two branches are merged, both file A and B are located in `/tmp/` and has the name `record`.

3.2 External representation for repository

This is what the structure of a version controlled directory looks like:

- Directory
 - file
 - BaseDirectory
 - .FVSrepository
 - * Directory structure with files you want to control
 - fileContent
 - original Files - “ID”
 - Repository file - “repo.cfc”
 - Cached files - “ID.f_cfc”
 - Patches - “ID.p_cfc”

The hidden “.FVSrepository” directory is created during initialization of a new repository. During the initialization process the “fileContent” directory and “repo.cfc” file is also created in the “.FVSrepository” directory. It is also in this directory all information about the internal representation, patches and cached content files are stored.

The repository file, “repo.cfc”, contains information about what patches are applied, what files are controlled and on what paths these files are found. Basically, all information in the *Repo* structure except for the *FileChange* of each file.

When a file is added to the VCS an unique ID is created for the file. The file is renamed to its ID and moved to the “fileContent” directory. On its original path in the base directory a link (retaining the original name) is created in the file system that points to the new file in fileContent. Two files are then created directly in “.FVSrepository”. First, an empty file with the name “file-ID.f_cfc” is created and second, a patch, with the add operation, is created with a unique ID and saved with the name “ID.p_cfc”. The file “file-ID.f_cfc” is the place where the cached, currently recorded, content of a file will be saved. Finally the repository file is also updated with the file ID and on what path it currently exist as well as the patch ID.

3.3 Commands

The current prototype supports the following commands.

3.3.1 Local commands

These are the commands that are currently implemented and supported in the interface. They follow the specification described in section 2.2. Those that are not present there follows the same principles.

The commands are described with a Haskell inspired notation:

- **command name** :: arguments
 - **init** :: targetRepository
Initialize a repo. An initialization is done by creating all the directories and files the VCS need. A hidden directory is created, “.FVSrepository”, that contains all information about the VCS. In this directory, the main VCS file is created, “repo.cfc”, and a directory containing the content files is created, “fileContent”.
 - **add** :: targetRepository path1 path2 ...
Add files to a repository, for each file, when it is added a unique ID is created for the file. It is renamed to that ID and moved to the fileContent directory. On the original path a link to the new file is created in the file system. Directly in the FVSrepository directory another file is created, “ID.fcf”, which contains the currently recorded content of the file.
 - **remove** :: targetRepository path1 path2 ...
Removes a path to a file from a repository. For each given path, decrement the weight of the file at that path.
 - **move** :: targetRepository currentPath newPath
Move a file from one place to another. Decrements the weight of the file at the current path while incrementing the weight at the newPath.
 - **record** :: targetRepository
Record the changes that has been made to all files in the target repository. Updates all files that holds cached trees.
 - **unrecord** :: targetRepository patchID
Unrecord the changes in a given patch. Creates the inverse of the given patch and applies it to the repository.
 - **whatsnew** :: targetRepository
Prints the changes that has been made to the content of the files in a repository since it was last recorded/updated.
 - **diff** :: targetRepository
Prints all changes of a repository since it was last recorded/updated.
 - **pull** :: targetRepository sourceRepository
Pulls patches from the source repository to the target repository. The pulled patches are the ones that are applied to the source repository. This can easily be changed to a “push” command simply by changing target to source and source to target.
 - **repair** :: targetRepository
Recreates a repository from all existing patches.
 - **revert** :: targetRepository
Reverts a repositories external representation to the last recorded version.
 - **verify** :: targetRepository
Verifies that a repository has the same content as the content of the applied patches.

- **patches** :: targetRepository
Prints the patches that are currently applied in a repository.
- **shape** :: targetRepository
Prints the shapes of the trees in a repository.
- **check** :: targetRepository
Checks a repository for inconsistencies and prints the result.
- **fixRepo** :: targetRepository
Currently removes files from the internal representation if they have no path and no content. Incomplete function at the moment.

3.3.2 Distributed commands

To use the distributed commands the interface is a bit different. Instead of a command you send the flag `-r` to the program followed by some information. `-r` :: input.

The input that the distributed command need has to be given in the specified order: username, hostAddress, port, command, remoteRepository, localRepository.

The client will connect to the `username@hostAddress:port` via SSH and run the given command, automatically passing the right arguments along. Command can be any of the following.

- **init** :: remoteRepository
Perform the same action as a local init but on the remote repository. The user do not need to specify the local repository argument in the case of init.
- **pull** :: localRepository remoteRepository
Performs the same actions as the local pull but on a remote repository.
- **push** :: localRepository remoteRepository
Push changes from the local repository to the remote repository. Tells the remote repository to do a pull from the local repository.

3.4 Network Interface

The prototype has an implemented network interface, making it possible to push and pull patches from remote repositories. As can be seen in fig. 5 the system uses SSH to connect to the remote repository and tells it to start the server version of the program. When it has started, it will contact the client, local, side at the port specified by the client in the first call to the server. By using SSH, only allowed users can modify a remote repository. Also since there is no need for a server constantly listening for a push, pull or init command we can spare the user from having yet another application constantly running.

Commands that are implemented are init, push and pull. Only the push and pull commands will force the repositories to exchange information. init is only a call to the remote repository to do an init on that system, it requires an existing path on the remote repository. When a remote pull command is executed the local repository will send the locally applied patches to the remote repository, the remote repository will then calculate

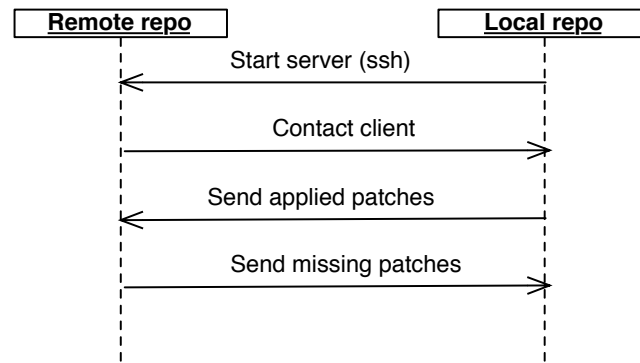


Figure 5: Pull over network

which patches that is applied on the remote, but not on the local, and send these patch names and patches to the local repository. The local repository will simply apply all patches it gets from the remote repository.

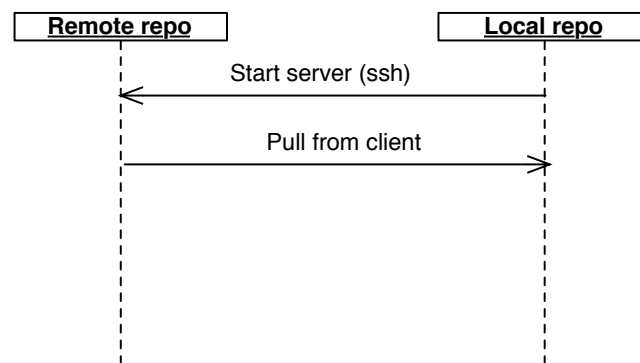


Figure 6: Push over network

A remote push simply sends a command to the remote telling it to pull from the local version and from this point it is the same as the above pull command, only the remote repository is considered to be the local.

4 Results

4.1 Working prototype

We have an operation-based VCS prototype that gives the possibility to test the presented ideas in “real world examples”. The prototype is implemented in the Haskell programming language and is a Unix terminal based program. The system handles distributed repositories which mean that users can access a repository in two ways, either locally where all local commands are available or remotely through the network interface via the commands that are supported there. This is nice as a user can always work with their local repositories even if they do not have an internet connection. The remote version of the VCS is only started when a user tries to connect and so the server does not need to be running at all times. Instead the server is started via SSH when a client tries to connect and then the server shuts down again after the operations are performed.

The security of the remote operations are quite good as it makes use of SSH to connect and start the remote server. This means that only users which have the right to connect with SSH to a computer with the repository as well as Unix rights to edit the files of the repository can push and pull from it. Other than this features the system currently have no way of tracking users and controlling which users can push and pull from a repository.

We have not performed any extensive testing of the system due to the limited time of this project. The tests that have been performed have been small, tests with four to five files, during the development to check that the system behaves as expected. The prototype performs very well on the small tests we have done though and it solves the problems that have been mentioned previously. For example, the criss-cross merge (fig. 3) is handled without producing any conflicts and tracking files that has been moved to different locations in different branches (fig. 4) works as well. It also has more reasonable complexity compared to Darcs as patches are independent of the order they are applied to a repository we avoid the combinatorial explosion when merging two branches.

The prototype also behaves as expected in times where conflicts occur and the repository is put in a de-normalized state. There are only one problem we have noticed here and that is that there exist a possibility that the internal representation may not always be possible to convert to an external representation. Thus breaking the invariant of the ρ function. This can happen when there are two files in the same directory with the same name. This could be fixed by for example adding some extension to conflicting filenames until the conflict is resolved.

4.2 Limitations and future work

The prototype produced in this project is just an early version of the system with focus on the most basic functions of a VCS. Because of this there is not much more functionality than the basic VCS functions one can expect and so there are a lot of things to do in the future to improve the program.

One serious problem in the current version is that when a conflict occurs, for example the above mentioned case where two different files exist in the same directory with the

same name, the program provides no possibility to resolve the conflict. The only way to resolve such a conflict between files is to manually write a patch and then add it to the repo file. A more reasonable solution would be to allow the user to move one of the files to another non conflicting location by using the ID of the file. Another way to solve this would be to show the user the conflict by changing the names of both files to something telling the user about the conflict, and then the user can move the files to the selected locations. There is currently no way to represent this conflict in the external representation which should be possible. This could be done by for example giving the conflicting files some extension to their names until the conflict has been resolved. In general, there is no help for the user to resolve conflict or fix de-normalized trees in the prototype which is something that needs to be worked on more if the system should be of any practical use in the future.

Another thing the prototype does not handle is metadata. The system should probably support the option of adding comments like the email of the one responsible for the patch and also to provide metadata in the history like time stamps. It would be good to have the system be able to display when and who is responsible of a certain patch or when it was pushed/pulled to or from a repository but the prototype does not support this yet.

During the development of the prototype the focus has been to get the basic functionality working and providing a prototype which follows the theory and so avoids conflicts as much as possible. Even though some thought have gone into the data structures not much time have been spent on making sure the system is implemented in a fast and efficient way. A few problems that might limit the efficiency of the prototype include for example the way added files are handled. The files that are handled by a repository is stored in a simple list and this might not be the most efficient data structure for this as we need to iterate over this list very often to find and edit certain files. Also the reading and writing of files have not been optimized in any way and so much information are read or written even though it is not strictly necessary. As a repository can and often need to handle vast amounts of files and information and IO operations are slow this is probably something that needs looking at for this system to be viable in real use.

More testing of the system should definitely be performed. It would be interesting to try different merges that causes several conflicts and compare how many this system can avoid compared to the other existing VCS. But also run more extensive tests to check that the program actually does the what it is supposed to do.

Currently the prototype only works in a Unix environment. Partly because we found that it was very hard to get Haskell with extensions working in Windows and partly because we use specific features of the Unix file system. This was not something we took into consideration during the project but the system should be a platform independent program in order to be usable by everyone.

The implementation has many functions in the IO monad, it should be possible to write it in a more functionally “pure” way. This is a problem to look at in the future and it needs to be decided if one want to go for a more functional approach or if the program should focus on speed.

The system uses a hash function to generate “unique” IDs for files and patches. It is based on the content of the file and the current time to minimize chances of generating the same ID twice. The hash function seem to perform quite well and we have not had any problems with clashing IDs but we have not done any real testing of the hash and no analysis of its performance. This is something that would need to be done to be able to give guarantees for how well the hash function performs if it is to be used in a real system.

In order to remove a patch one has to give the whole ID of the patch, this would be easily solvable by adding a feature to give them as regular expressions similarly as in the Unix terminal. To find out the number of a patch one has to either check in the repo files manually or print all changes and read them. To find out what a specific patch changes one has to print all patches and look for the one wanted, also this could be solved by allowing the user to give a regular expression.

References

CollabNet (2008), ‘Subversion’.

URL: <http://subversion.tigris.org/>

Khanna, S., Kunal, K. and Pierce, B. C. (2006), A formal investigation of diff3.

URL: <http://www.cis.upenn.edu/~bcpierce/papers/diff3-short.pdf>

Lippe, E. and van Oosterom, N. (1992), Operation-based merging, in ‘SDE 5: Proceedings of the fifth ACM SIGSOFT symposium on Software development environments’, ACM Press, New York, NY, USA, pp. 78–87.

URL: <http://dx.doi.org/10.1145/142868.143753>

Löh, A., Swierstra, W. and Leijen, D. (2007), A principled approach to version control.

Mens, T. (2002), ‘A state-of-the-art survey on software merging’, Software Engineering, IEEE Transactions on **28**(5), 449–462.

URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1000449

Roundy, D. (2008), ‘Darcs’.

URL: <http://darcs.net/>

Torvalds, L. (2008), ‘The “git” vcs’.

URL: <http://git.or.cz/>