Modelling the Git core system with Alloy

Cláudio Lourenço, Renato Neves
Universidade do Minho

July 8, 2012

Contents

1 Introduction **2**
1.1 Git . . . 2
1.2 Motivation . . . 2
1.3 Manual Structure . . . 3

2 The Git structure **3**

3 The Repository **4**
3.1 Git Objects . . . 5
3.1.1 Blob . . . 5
3.1.2 Tree . . . 6
3.1.3 Commit . . . 6
3.1.4 Tag . . . 7
3.2 Git References . . . 7
3.2.1 Branches . . . 7
3.2.2 HEAD . . . 7

4 Working Directory **7**

5 Index **8**

6 Git operations **8**
6.1 Add and Remove . . . 8
6.1.1 Pre requisites . . . 9
6.1.2 Result . . . 9
6.1.3 Examples . . . 9
6.2 Commit . . . 10
6.2.1 Pre requisites . . . 10
6.2.2 Result . . . 12
6.2.3 Examples . . . 13
6.3 Branch Create and Branch Remove . . . 14

6.3.1 Pre requisites 15
6.3.2 Result 17
6.3.3 Examples 17
6.4 Checkout 18
6.4.1 Pre requisites 18
6.4.2 Result 19
6.4.3 Examples 20
6.5 Merge 20
6.5.1 Pre requisites 21
6.5.2 Result 22
6.5.3 Examples 22
6.5.4 Git Pull and Git Push 23

1 Introduction

1.1 Git

A version control system is a tool that records changes on your files over time. The main idea is to keep track of the changes on your files and to retrieve them later. There are mainly three kinds of version control system. Local, centralized and distributed. On a local VCS it is not possible to collaborate with other users and the structure that keeps track of the files is kept only locally. The need for collaborations with other users brought the centralized VCS. On the centralized version control systems there exists a server that keeps track of the changes on the files and each user pulls and checks out files from this server. This approach has mainly two problems. The first is to have a single point of failure, if a server goes down during a certain time, nobody can check out or pull updates. The second problem is when you are not connected to the network, you cannot pull or commit your changes. To solve this problems, the distributed VCS appeared. Here each user keeps a mirror of the repository and there is not a central server. An user can always commit and when connected can push/checkout the changes.

Git is a distributed VCS. It was created in 2005 by Linus Torvalds, for the Linux kernel development. The main difference when comparing to others distributed VCS is that *git* keeps snapshots of how your system looks like on a certain moment in time instead of keeping track of changes.

1.2 Motivation

Nowadays *git* is having great success. It is efficient, fast and in the begin, it looks easy to understand and to use. When the users start diving into git, they are faced with a behavior that nobody can explain very well. There are not any formal or even informal specification of the operations, like what

are the pre-requisites and what is the result of performing an operation. So normally the users when fronted with something they do not understand, they avoid it.

The purpose of this project is to formally model the *git* core using a tool called Alloy, analyse and model some *git* operations and then check which properties the model does (not) guarantee. This manual presents to the reader semi-formal description of *git* internals, the description of some operations and it ends with some properties that the model guarantee.

Alloy is a declarative specification language, used for describing structures in a semi-formal way. After modeling the structures, it is possible to check for some properties, look for counter examples and to visualize instances of the model.

The difference of this manual from the others, is that, before writing it, we tried to understand and model the *git* concepts from a semi-formal perspective using Alloy. As result we think that our understanding on some key parts of *git*, is more precise and rigorous, than most known manuals. Thus, we try with this manual to pass the knowledge obtained.

1.3 Manual Structure

This reports starts by explaining the organization of *git*, how it is structured and then we cover each component of *git*. Working directory, index and repository. For matters of time/space and complexity we focus our model in the index and in the repository, more precisely in the object model. After the presentation of the structure we give the semi-formal specification of some operations. We finish the report with the analysis of some properties.

2 The Git structure

Git, as it was said before, is a distributed version control system. It means that each user has a mirror of the repository (local repository) and in the case of *git*, there is no need for a central server, even if, each user is able to fetch or push updates from other user's repository (remote repository). *Git* is divided mainly in three components. The working directory, the index and the repository. The connection between these components can be seen in Figure 1. Local and remote repository have the same internal structure. What is a local repository for an user is a remote repository for another user.

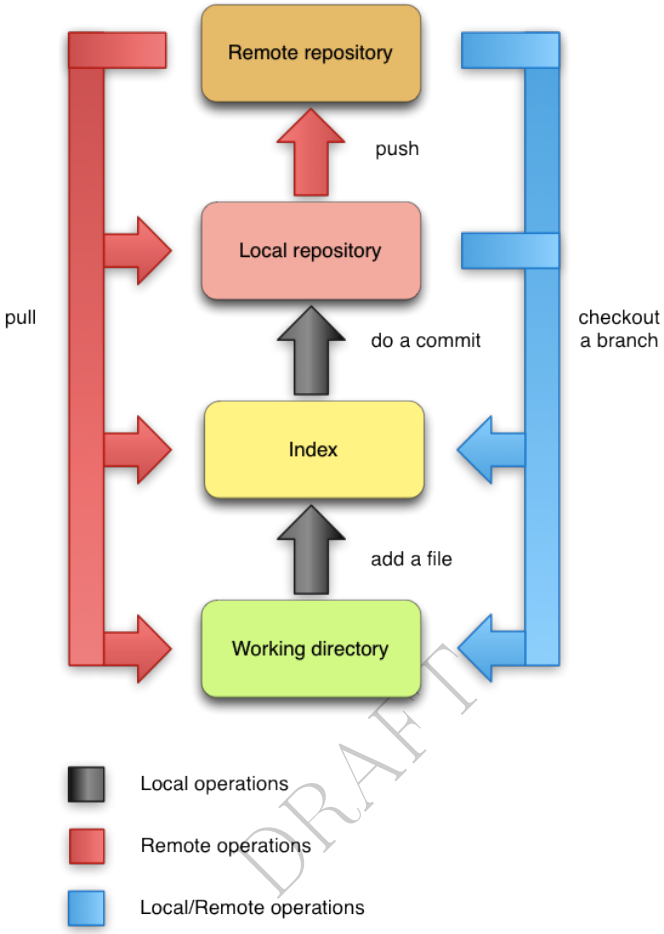

Figure 1: Git WorkFlow

3 The Repository

Git repository is like a database of a project along construction time. It is the place where the snapshots taken during the project construction are kept, as well as, all the necessary information to allow the users to go through all the snapshots. The snapshot's structure and snapshots' relations are kept using, what is called, *git objects*. The information needed to go through the project is kept in, what is called, *git references*. In the next subsections we present both of them.

3.1 Git Objects

As we have presented before *git* keeps snapshots of how your system looked like on a certain moment in time. This moment in time is represented in *git* by a commit. Each commit points to a tree (corresponding to a directory), that represents the structure of your project on that moment. So a tree contains others trees and/or blobs. In *git* the files are not kept. What *git* keeps is a object called blob that represents the content of files. The relation between the path of a file and the content of that path is kept on the trees corresponding to that path.

In Figure 2 is possible to see an example of a snapshot. We have a commit that points to a tree. This first tree corresponds to the root directory of the project. All the other trees (and blobs) corresponds to some directory (or file) preserving the relation between them.

In the next sections a detailed description about each object is given, but for now something that is useful to know is that each object is identified by a 40-character string. This string is calculated by taking the SHA I hash of the contents of the object. This approach has many advantages, being the two more important, in our opinion, the fact that it is possible to compare two objects only by the name and if two objects have the same content then, only one of them is kept.

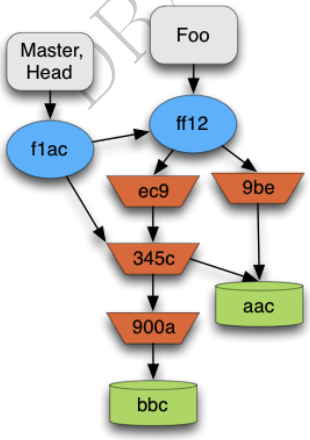

Figure 2: A Snapshot

3.1.1 Blob

A blob object, as it was said before, represents the content of a file and the blob name is calculated from the blob content. So if we have two files exactly with the same content, only one blob will be kept, even if, they have different

names . This happens because the blob is not directly associated to a file. The relation between a path (of a file) and a blob is kept in the tree object.

3.1.2 Tree

A tree is nothing else than a map from names to blobs and other trees. This map represents the contains relation. Also here, a tree is not associated to any specific directory. If two directories have exactly the same content they will be represented by the same tree object. Here have the same content means that it should have exactly the same relations, or in other words, we have exactly the same names in both trees and each name corresponds exactly to the same objects in both trees.

3.1.3 Commit

The commit object, is like a snapshot of the project on a certain moment in time. A commit object has more variety of information than the two objects addressed before. Looking a commit, it is possible to find out the following:

- Author - The person responsible for the change on the project.
- Committer - The person which actually created the commit. It is possible that the committer is different from the author. An author can create a patch, send it to the committer, which will create the commit.
- Parent - The previous commit, or, the commit from which this one was created. It is possible for this field to be empty, when the commit is a Root Commit (the first commit to be done). It is also possible to have two parents, when the commit is a result from a merge operation.
- Comment - It is possible, when creating a commit, write a comment about the commit. This comments can contain details about the changes that were done in the project, as well as, something that the Committer wants to write.
- Tree - It is a pointer for a tree, or in other words, it is a pointer for how the project looked like on a certain moment in time.

In our model, we concentrated our efforts in the Parent and in the Tree fields. We just care about the structure of the object model, so, we removed everything else that does not influences such structure. One property that we observed when modeling, is that, we cannot have cycles in the parent relation. It means that if we are on commit C1 and we go through the parent relation, we will never reach C1.

3.1.4 Tag

At last we have the tag object. The tag object is just a pointer to a commit with some more information. It can be used to mark a special commit, like a new version of the project. The structure is quite similar to the commit object. It has a tagger (the person who created the tag), a date in which the tag was created, a message (some comment from the tagger) and it points to a commit object. In our model we did not model this object, because when abstracting it looks like a branch, that we will see in next section.

3.2 Git References

Basically a reference is just a pointer to a commit. There is much more about references, but in this manual, lets keep it simple and concentrate on the object model. In this manual, we just speak about Branches and the special reference HEAD.

3.2.1 Branches

One of the trump cards of *Git* is its definition of branches. In other VCS each time a branch is created a new copy of the entire repository has to be done. Thus, branches operations in those VCS are slow and hard to use. In *git*, when a new branch is created the only thing that *git* does, is to create a new pointer to the current commit. The current commit is marked by a special reference called HEAD, that we present next.

3.2.2 HEAD

HEAD is maybe the most important reference in the repository. It indicates where you are situated in the repository. When a commit is done, *git* has to know which commit is going to be the parent. It does so, looking into the HEAD. HEAD normally is a pointer to a branch, so when the commit is done, the HEAD is kept, but the branch points to the new commit. HEAD can point directly to a commit, but lets keep it simple.

4 Working Directory

The working directory is basically a subset of a file system that contains the files of the project you are currently working on. These files can be the current files, files retrieved from an old snapshot or even files that are not being tracked. When retrieving an older snapshot of the project, the working directory is updated to reflect the project in that state. The untracked files in the working directory are just ignored, unless there is a conflict when retrieving files from an older snapshot.

When a user starts a repository, all the files are untracked. When a new file is created it will be untracked. So, how does *git* know which files are tracked or not? There exists the index which we present next.

5 Index

The index is something in between the working directory and the repository. When a file is created if the user wants that file to be on the next commit, it has to be on index. Even if the user just modifies a file and the user wants that change to be reflected on the next commit, it must be added to index, otherwise, what will be committed is the older version of the file. So, basically the index contains all the files that will be in the next commit.

6 Git operations

In this section we explore some of the git operations. We try to do so, in a different way, if comparing with other manuals. Normally the manuals we see around just explain how to perform an operation. We try to go further, so in this manual, we have divided our operations in three parts. In the first part, we give a description about the operation. In the second part, we give the pre-conditions, or in other words, we explain in which conditions the operations can be performed. Finally, in the last part of each operation, we show what is the result of perform such operation.

6.1 Add and Remove

The operations add and remove are used to add and remove content from index. As we have said before the index contains all the files and contents to be added on the next commit.

Git add does not refer directly to the file. Instead it is like a map from file to content. If we have a file on the index and we modify it on the working directory, for this modification to be visible on the next commit, the file has to be added again to the index.

The remove operation removes a file from index, so it will not be present if we perform a commit exactly after the remove. The removed file besides of being removed from index, it will also be removed from the working directory (if it is still exists there).

6.1.1 Pre requisites

The add operation has only one pre-condition. The file has to be in the working directory. It means that a file that is on the working directory, can be always added to index.

Relatively to the remove operation there are a few conditions that have to be satisfied for the remove operation be performed, which are:

- The file that is being removed is currently in index;
- The file that is being removed is in the previous commit, exactly with the same content.

The first restriction is quite obvious. It is not possible to remove a file if it does not exists. The last restriction exists, to avoid the accidental deletion of file's contents. It is possible to have a file in index with a certain content that does not exists in the repository and since the remove operation removes the file from the index and from the working directory, the content would be lost.

6.1.2 Result

After performing the add operation there is something that will be always observed. The file will be in index. Besides, if the file was marked as not merged then the mark goes away. More details about merged and unmerged files will be given later.

The result from performing the remove operation is that the file is not anymore in the index. Also here the file will not be marked as unmerged anymore.

6.1.3 Examples

The figure 3, shows a simple case of adding a file to the index.
It can be seen, that no changes are made on the repository or in the working directory, just in the index that will contain the new file.

Figure 4, contains the case of updating the content of a file.
Assuming that the file is already in the index, changing it's content in the working directory, will not change the content in the index. We need to explicitly add the file with the new content in the index.

Now for the remove operation. Figures 5,6 show a simple case of removing a file of the index. As the reader can see the file will be removed from the index, but also from the working directory, if it's still there. The repository stays the same in both cases.

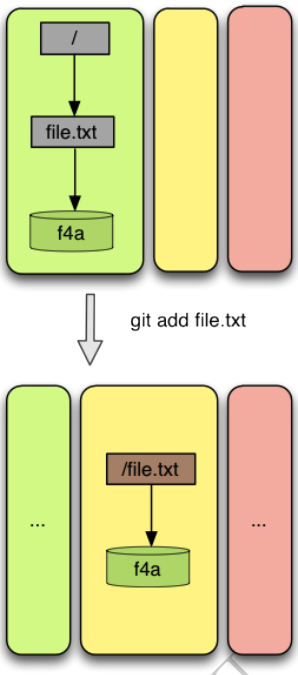

Figure 3: Adding a file

6.2 Commit

The commit operation takes the current index and builds a commit object. Basically it takes the files from the index, builds a structure using trees and blobs objects and creates a new commit object pointing to the tree that corresponds to the project's root directory. This commit object will have as parent the current commit, the commit pointed by the branch indicated in the HEAD. When the first commit is created, a branch called "master" will be created and it will be marked as the current branch, or in other words, it will be indicated by HEAD. Also, the first commit of the repository is called a Root Commit, because it has no parents.

6.2.1 Pre requisites

There are three pre-conditions that are required to perform this operation:

- The index is not empty;
- The commit object that will be created is different from the current commit;
- There are not unmerged files.

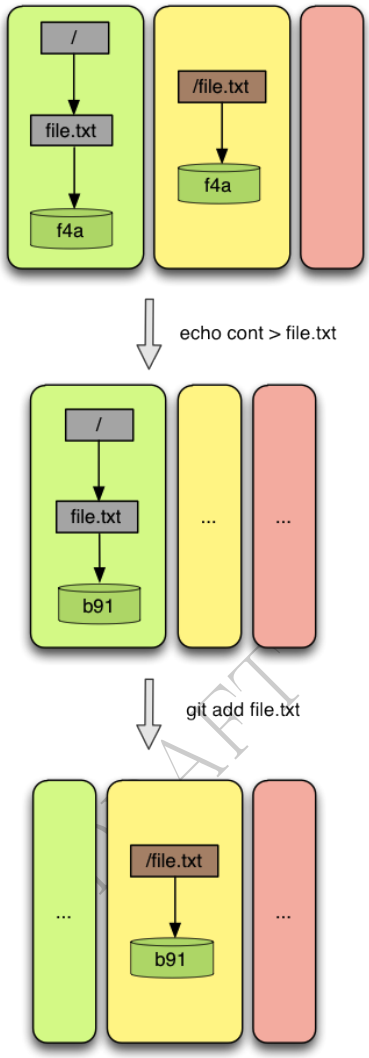

Figure 4: Updating a file

The first pre-condition is obvious, because if we do not have files in the index, there is anything to commit.

The second condition says basically that we cannot have two commits objects pointing to the same tree object in a row, or in other words, if we have the same files and the same content in the index and in the current commit, a commit operation cannot be performed.

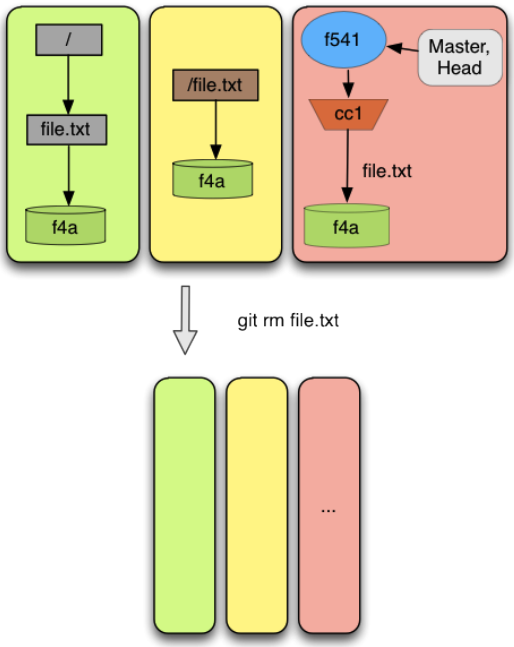

Figure 5: Removing a file

The last condition just says that it is not possible to perform a commit, if there are unmerged files in index. As we have said before, we will speak about this later.

6.2.2 Result

The result of performing a commit can be different if it is the first commit or if there are already some commit in th repository. We start by exposing the observed result when the first commit is being performed and then we show the case when there are already some commit in the repository. In the end we show some properties that are observed in both cases.

The first commit object that is being created is a commit with no parent. It means that it does not have a parent. There cannot be branches at this moment, because they cannot point to any commit, as there are none to be pointed, so a new branch called "master" is created. This branch is going to be pointing to the commit that has been created. Also the HEAD, that previously was not defined will identify this "master" branch.

When there are already some commit in the repository, then it is guaranteed that there are at least one branch and the HEAD is identifying some existing branch. The branch that is identified by the HEAD, is pointing

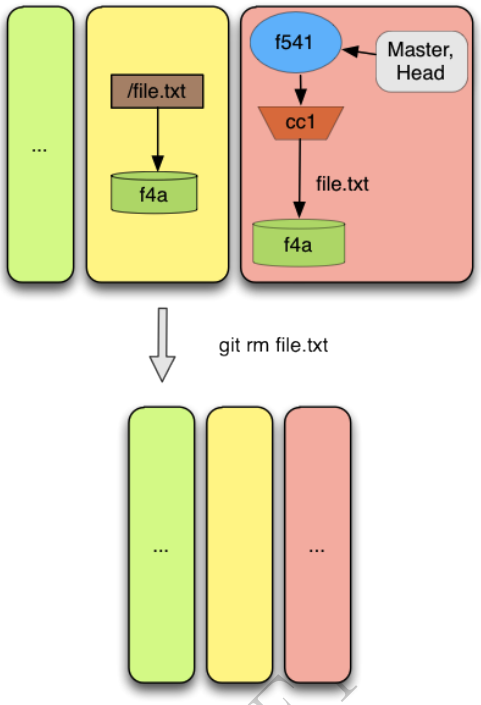

Figure 6: Removing a file (alternative)

now to the commit created. This commit have now as parent the previous commit (the commit pointed by the branch identified by the HEAD), unless it is a result from a merge. In this case the commit must have two parents. The previous commit and the branch that is being merged.

The common properties that are observed are that all the files and content that were in the index, are now in the current commit with the structure from the file system. For this new blobs and trees were probably created. The index keeps exactly the same content.

6.2.3 Examples

The figure 7 shows as typical process where there are added some files to the index and then is done a commit. As can be seen the new commit reflects, the structure of the working directory at that point. The interesting thing here is that, because the two files have the same content, they will share the same Blob. This is a case where *git* shows it's efficiency.
Figure 8 has an example of a commit operation, where in already exists a commit object in the repository. This figure is interesting because it shows what changes are done in the repository when a new (not RootCommit) is

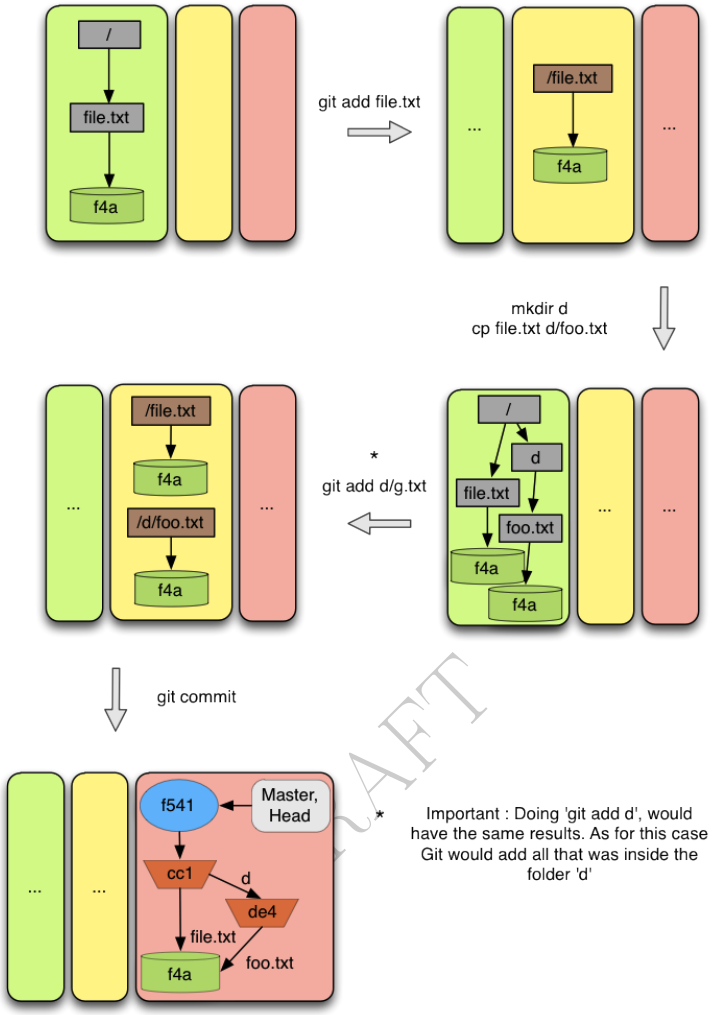

Figure 7: Doing a commit

done. The new commit points to the previous commit, and the branch and HEAD point to the new commit. As expected the new commit reflects the working directory structure at the point of commit.
Figure 9 is just a concrete example of the 2nd pre requisite referred above.

6.3 Branch Create and Branch Remove

Branches in *git* are very efficient. This happens because a branch in *git* is just a pointer to a commit. So, when a branch is created or removed there is only a new pointer created or a pointer that is removed. As we will see later, there are some restrictions when removing branches. When we are referring

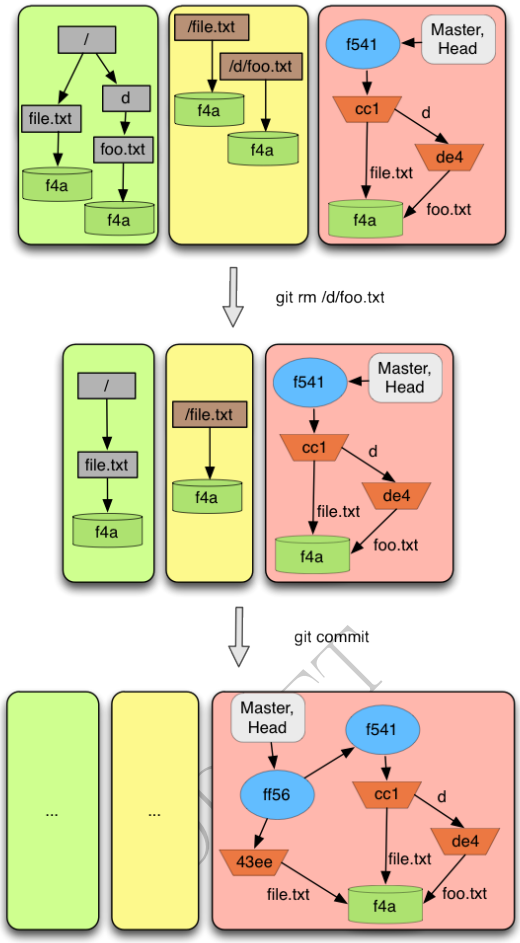

Figure 8: Doing a second commit

to creating or removing a branch, we mean the operations *git branch* and *git branch -d*.

6.3.1 Pre requisites

When creating a branch there are only two restrictions:

- Some commit must exist, and consequently the HEAD reference must point to some commit;
- There is not any branch with the same name.

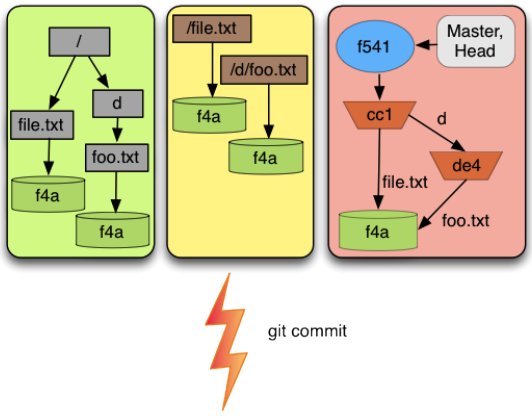

Figure 9: This commit can't be done in *git*

The second pre-condition is kind of obvious. We cannot create a branch with the same same name as one already in the repository, otherwise we would have conflicts. The pre-conditions that could bring some doubts is the first one. *Git* does not allow the creation of branches before the first commit is done. When the first commit is done, a branch called "master" is created and after that it is possible to create as many branches as the user wishes.

When removing a branch there are also some restrictions:

- The branch exists;
- The HEAD is not identifying the branch;
- The commit pointed by the branch is accessible from the current commit (the commit pointed by the branch identified by the HEAD).

The first pre-conditions says that it is only possible to remove a branch if it exists. The second says, that the branch being removed is not the one identified by HEAD, otherwise the HEAD would be point from now on to some branch that does not exists anymore. Thus, there would not exist a current commit. The last pre-conditions exists to check the user does not deletes a branch that is not yet merged with the current branch, or in other words, the user does not delete a branch that is not accessible from the current commit, through the parent relation. If this pre-condition did not exist it would be possible to loose the commit pointed by the branch, as well as, some commits from the parent relation.

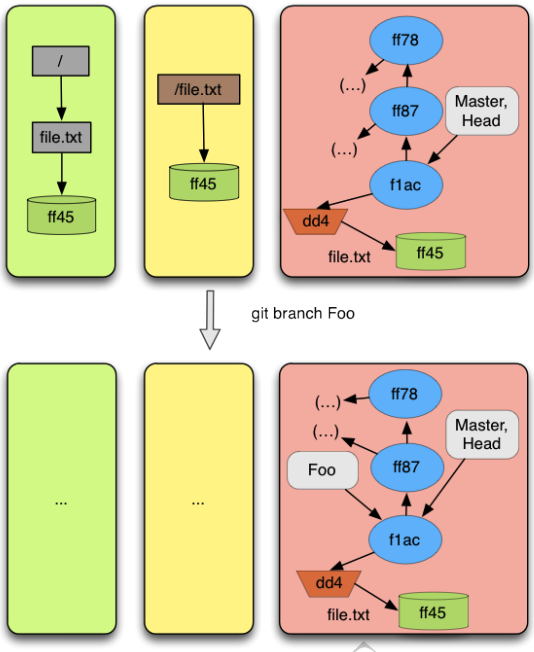

Figure 10: Creating a branch

6.3.2 Result

The result of performing this operations are expected. When creating a branch, the result is a new branching pointing to the same commit as the branch identified by the HEAD. When removing a branch, the result is that the branch removed does not exists anymore.

6.3.3 Examples

The changes that happen when creating a branch are minimal. The only thing changing is the appearance of a new branch pointing to the current commit, as figure 10 demonstrates.
The removal of a branch is also minimal. The only thing changing is that the branch referred must be deleted. If in 10 we view the grey arrow as pointing up, instead pointing down, we automatically view the process of deleting a branch.
About the branch removal pre condition, this case is legit because the current commit achieves (it is the same) the commit pointed by the branch deleted. Even if 'Foo' pointed to any other commit, it could be still deleted, as all commits in this figure are achieved by the current commit.

6.4 Checkout

The checkout operations is used to update the working directory and index with a set of files from a commit. In this project, we have explored only the case the checkout operation is used with a branch. In this case, all the files from the commit pointed by the branch are exported to the working directory and index. This was the most difficult operation to explore, because it behaves in a non intuitive way. We will try to explain it and in the examples we show you a sequence of operations that causes the *git* to loose information. This is a proof of the complexity of this operation.

6.4.1 Pre requisites

As we have said before, this operation is not intuitive. In our opinion this operation should have as pre-condition something like "Everything that is in index should be committed". Instead *git* tries to relax this pre-condition and we have the following:

- The branch we are checking out, exists;
- Everything that is in the index has to be in the current commit with the same content, except if:
    1. The content of a file is the same in the current and destination commit (warning is thrown)
    2. Exists a file in the index, and that file does not exists neither in the current nor in the destination commit (warning is thrown)
    3. Content of the file in the index is the same as in the destination commit (no warning is thrown)

About the pre-condition there are not much to say. If we want to checkout a branch, the branch should exist.

The problem here is the second pre-condition. There are a lot of exceptions when a file is not in the current commit with the same content. To simplify the exposition of this pre-condition, lets assume we have a file f, which is not in current commit, or at least it is not with the same content. All other files from index are in the current commit, with the same content. Relatively to f, if at least one of the exceptions enumerated above is satisfied, then the checkout operation will proceed. Now lets analyse each one of the exceptions. The first says that, f can be in the commit we are checking out with the same content. So, even if f is not commit, but it is in the commit we are checking out, git will proceed with the operation. The exception marked as 2, says that f does not exists neither in the current commit nor in the commit we are checking out. It means that if f has just been created and a

checkout operation is performed then if the file does not exists in the commit being checked out, the operation will proceed. The last exception says that if there exists a file with the same path and the same content in the commit being checked out, then the operation will proceed. In this last case does not warns the user.

6.4.2 Result

As we have said before, in this manual we have concentrated our efforts in the index and repository, simplifying as most as possible the working directory. So, we will just explain what happens with the index after a checkout operation is performed. We do not make any description about the files that are in the working directory but not in index.

The first thing the reader should know is that the HEAD, after a checkout is performed, identifies the branch which was checked out.

There is one more alteration. The index when a checkout is performed has to be updated to reflect the operation. The result of it is not intuitive. To understand it lets assume that the checkout operation has not yet been performed. Consider three relations from file to content.

- CA: Files from the current commit and their respective content;
- IA: Files from index and respective content;
- CB: Files from the commit we are trying to checkout and their respective content.

After understanding this three relations it is simpler to understand what happens with the index. Lets also assume that $(IA - CA)$ is a relation that contains the files from IA which are not in CA with the same content. It mean that $(IA - CA)$ will contain the new files and the modified files relatively to the last commit. Now, consider $CB + +(IA - CA)$ as being all the files from CB, but if the files is in $(IA - CA)$, the file will keep the content from $(IA - CA)$. If when comparing the index with the current commit it contains only new files and modified files (not deleted files) then it is all. $INDEX = CB + +(IA - CA)$. When the index contains removed files, or in other words, files that are in CA but are not in IA ($CA - IA$ is not empty), then case the file that is marked as removed exists in CB it will be marked as removed, otherwise the removed information is ignored. So, the result of performing a checkout is: $INDEX = (CB + +(IA - CA)) - (CA - IA)$. All files from CB are overwritten by the files that are in the index but not in CA and from the result of this, the files that are marked as removed will not be in index.

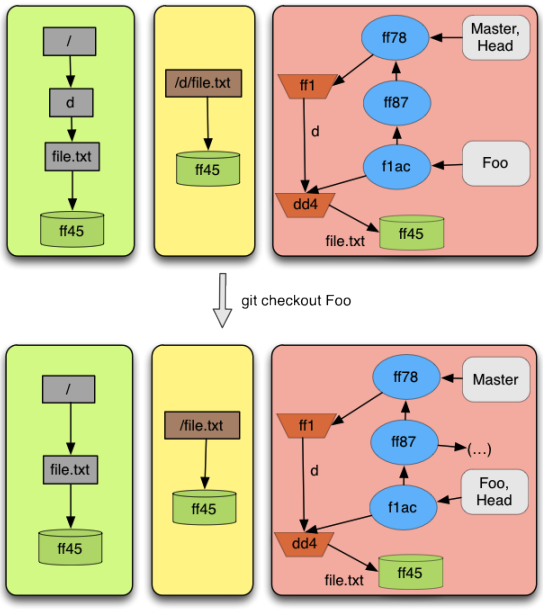

Figure 11: A simple checkout

6.4.3 Examples

As referred above checkout behaves in a non-intuitive way. We recommend that every time, a checkout is done, all things must be previously committed, or else we risk ourselves to pass things that we don't want to other branches, or even loose information.
Thus, the example shown in figure 11, contains the case where all content in the index and working directory are committed (the safe case). When doing the checkout we can see in the figure that the repository and index change to reflect the branch checked out, more concretely the most recent commit of the branch. But also, the commit referred will now be pointed by the HEAD reference. So, all new changes done will belong to the branch checked out.

6.5 Merge

The merge operation is one the of most important operation on *git*. It consists on trying to combine information from two commits. Resulting of that is a new commit with information from both commits, however there are some cases where this will not always be true. A commit information can take precedence over another, thus there is no need for a new commit. Also, the process of joining information cannot be done, because it is not certain which correspondent data in both commits should take precedence (merge conflict).

When a 'git merge <branch>' is done, *git* will pick both involving commits (the current commit, and the commit pointed by the branch referred) and try to merge them into a new commit that must belong to the current branch.
As already said, a merge conflict can happen, and if it occurs *git* leaves to the user the process of resolving the conflicts. When that process is done, the user can notify *git* by doing a 'git commit', and a new merge commit will be made, where it's parents will be both commits trying to be merged.
Git tries to be the most autonomous possible when merging, thus it has three types of merge

- A fast-forward merge (figure 12). Happens when the non current commit, can achieve the current commit by the parent relation. If that happens, then the non current commit is more recent than the current commit, so the merge commit would be equal to the non current commit. Because *Git* tries to be efficient a merge is not made. But, the HEAD will point to the newest commit.
- A 3-way merge (figure 13). Happens when a fast-forward is not possible, but, both commits have a common ancestor. *Git* will use the common ancestor to minimize potential merge conflicts.
- A 2-way merge (figure 14). Happens when both types above are not possible. In this case there are no common ancestor, thus will it be more likely that conflicts appears.

6.5.1 Pre requisites

The merge operation, as all other operations has some pre conditions that must be satisfied in order to be possible to do it. As in the checkout, *git* let's the user do merges without all information in the index being committed. However,again, it is recommended that no information is left uncommitted :

- A merge can't be made, if the current commit is more recent than the other. As the current commits takes precedence, no new changes will exist. Thus, the merge is not necessary.
- If *git* can't do the merge automatically because of conflicts, the user must resolve all those conflicts, before doing another merge.
- When it is the case of a fast-forward merge, the index can't have uncommitted files that would be changed by the resulting merge.
- In the case of a 2-way or 3-way merge operation there simply can not be uncommitted files.

6.5.2 Result

As already said when doing a merge, one of three things can happen. A fast-forward, a successful merge or a partially done merge:

- If it's the case of a fast-forward merge. The index and the working directory will contain the information of the newest commit. Also, they will contain all information prior to the merge that was uncommitted, as long as that it brings no conflicts to it. Of course, the current branch and HEAD will point to the newest commit.
- When a successful merge happens, the index will reflect the new commit, and the working directory will contain the information of the commit. Here, doesn't exist the question of uncommitted changes, as *git* forbids that to happen. In the repository a new commit will appear that has the combined changes of the two (possibly three) commits involved in the merge. That new commit will have as parents those two commits, and will be pointed the current branch and the HEAD.
- If when doing a merge, a conflict appears that *git* can't resolve, it will say that the automatic merge failed, and that the manual merge will begin, in other words it leaves to the user the work of resolving the conflicts that *git* couldn't solve. At that moment, the index and working directory will contain all the information that was merged. For all the files unmerged, they will also be maintained in the index and the working directory, but in a state where we can see what differs in them, when they belong to a commit or to the other. Finally, *git* will mark some fields in the repository, so that when the user does the next commit, it will be the son of both parents involved in the commit, where it will be pointed by the current branch and HEAD. When this new commit is done, the merge process is over.

6.5.3 Examples

As usual the next figures will show some concrete examples of what happens in *git* when a merge is done, we don't involve here cases of uncommitted changes, as we don't recommend to do that and things would turn much more tricky to understand and much less intuitive.

Figure 15 is a simple case of a merge operation. It shows that the Master branch and HEAD point to a commit that is achievable by the commit pointed by 'Foo', so this is a case eligible for a fast-forward commit. We can see that the result is that the working directory and index will be updated accordingly to the commit now pointed by 'Foo' Master and HEAD. As known any untracked file will not be changed.

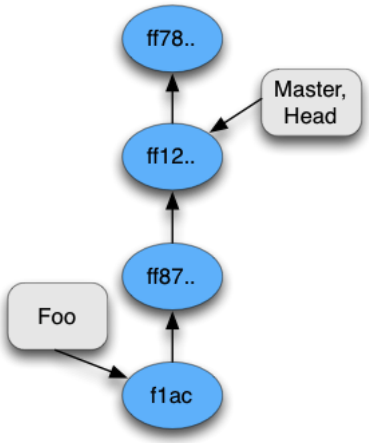

Figure 12: A fast-forward case

Figure 16 shows a concrete example of state in the repository where it is not possible to do a merge. As the commit pointed by HEAD and Master can achieve the commit pointed by 'Foo', thus the current commit taking precedence over any data than the other commit could have.

The two last figures 17 and 18 demonstrates the difference between a 2-way and a 3-way merge commit. When in the 2-way there are conflicts to resolve, in the 3-way the conflicts are resolved by looking to the common ancestor. Because the 3-way merge was automatic, a new commit was created that represent the merged information. Also, the working directory and index were updated accordingly. In the 2-way the user had to resolve the conflict by adding new content to the unmerged file and adding it to the index. Finally, when 'git commit' was called a new merge commit was created and the merge process successfully finished. In this case the index and working directory were updated just before *git* leaved to the user the merging process.

6.5.4 Git Pull and Git Push

When working with remote repositories, these two operations will be vastly used. They are in fact, a sequence of operations, and are used to get content from a remote repository or to upload local content to the remote repository. If a 'git pull' is done, the branch from the remote repository will be fetched, and the local branch and the remote (now, also local) branch will be merged. So expect this operation to behave exactly like a merge operation and consequently all that was referred in the merge applies also here.

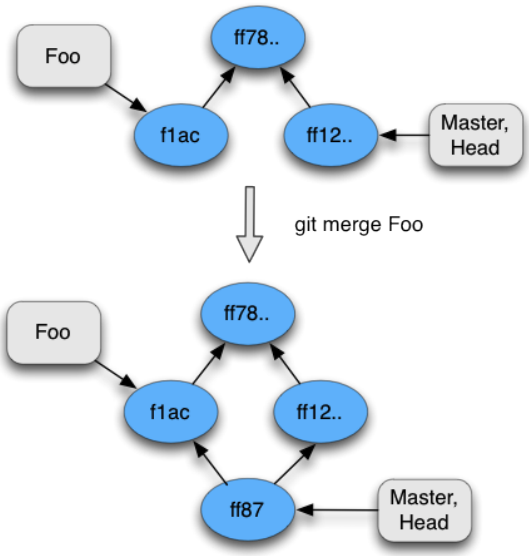

Figure 13: A 3-way merge case

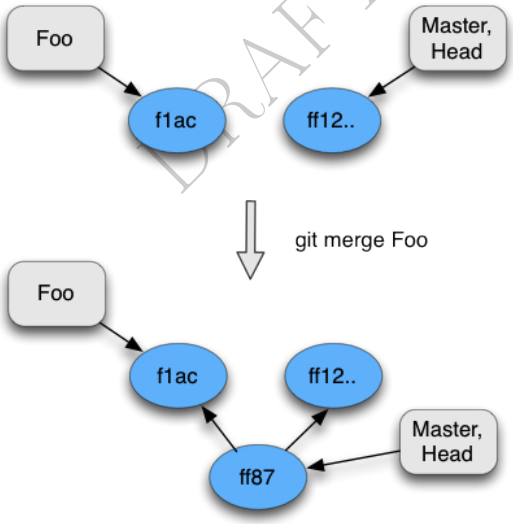

Figure 14: A 2-way merge case

If a 'git push' is done, the local branch will be uploaded to the remote repository and will be merged there.
In Push there is an important restriction. A 'git push' can only be done when the resulting merge is a fast-forward. This happens because, there is no one on the other side to resolve the merge conflicts, and thus *git* must

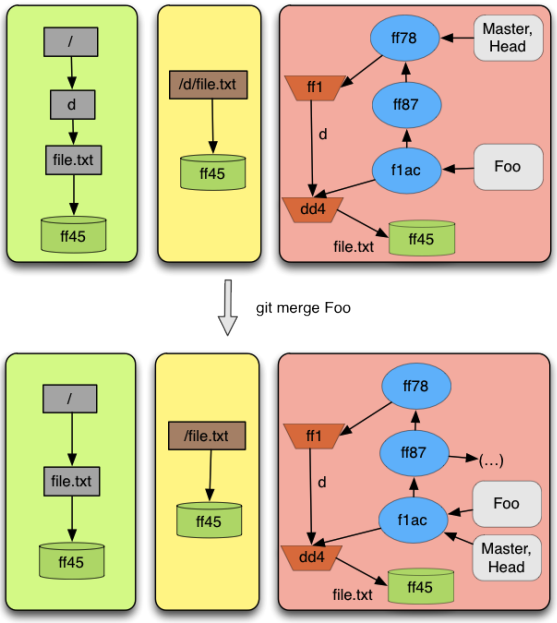

Figure 15: A fast-forward merge case

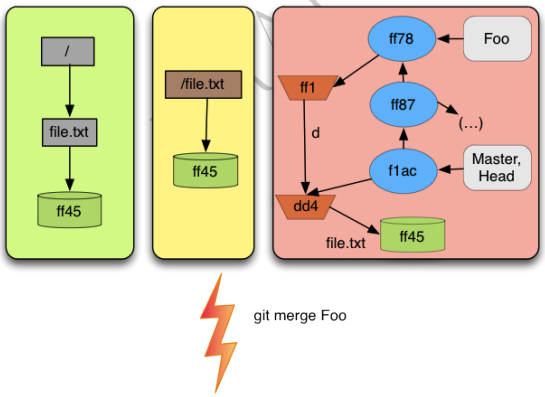

Figure 16: A fast-forward case that can't be done in *git*

guarantee what no conflicts happen.

References

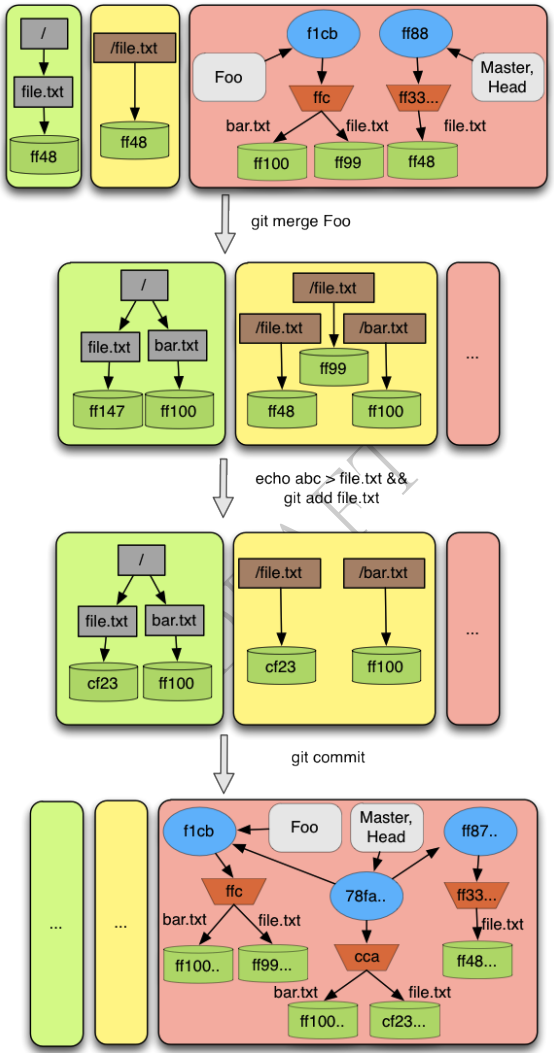

Figure 17: A typical 2-way merge

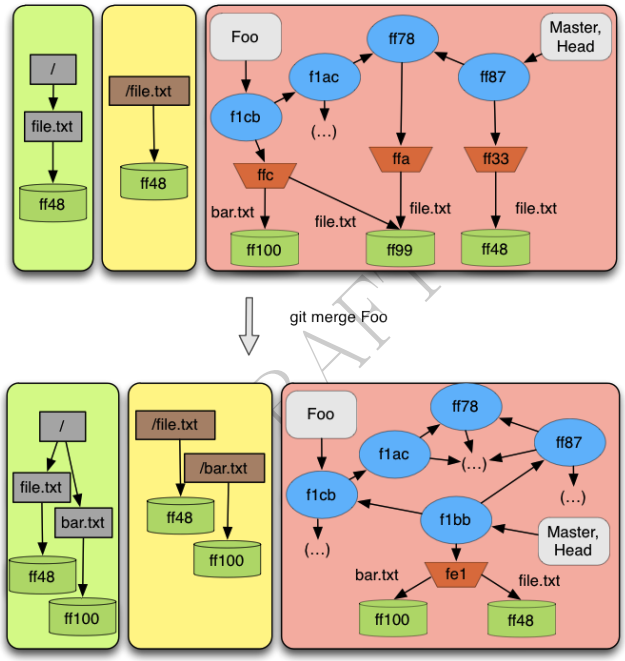

Figure 18: A typical 3-way merge