University of Minho

Formal Methods in Software Engineering

The Truth About Git

Understanding Git with Alloy

Authors:

Cláudio Lourenço
belolourenco@gmail.com

Renato Neves
nevrenato@gmail.com

Supervisors:

Eunsuk Kang
Massachusetts Institute of Technology

Manuel Alcino Cunha
University of Minho

José Bernardo Barros
University of Minho

July 17, 2012

Abstract

It is unthinkable to think about the world of software with no version control systems. *Git* was born in 2005 and since then it is having a great success. The problem is that there are not any formal specification about its internals. With this manual, we try to give a semi-formal specification of *git* and explain the behavior of some operations. This manual is based on a formal specification defined using the formal modeling language Alloy.

Contents

Chapter 1

Introduction

1.1 Git

A Version Control System (VSC) 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 pushes 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 [1, 2, 3, 5, 6] 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 at first glance, it looks easy to understand and to use. However, When the users start diving into git, they are sometimes faced with unexpected behavior that cannot be explained very well. There is not any formal specification of the operations,

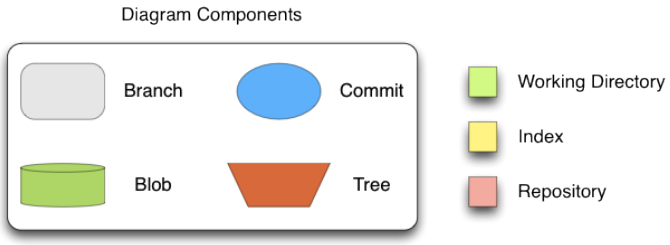

Figure 1.1: Convention used in this manual

like for example, what are the pre-conditions and what is the result of performing an operation. Normally, when the users are fronted with a behavior they do not understand, they just learn how to avoid it.

Alloy [4] is a formal modeling language, used for describing structures in a 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 purpose of this project is to formally model the *git* core using Alloy, analyze it, model some *git* operations and then check which properties the model does (not) guarantees. 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 guarantees.

The difference between this manual and others [2, 1], is that, before writing it, we tried to understand and model the *git* concepts from a formal perspective using Alloy. As result we think that our understanding on some key parts of *git*, is more precise and rigorous. In this manual we try to convey some of that knowledge.

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 the object model in the index and in the repository. 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.

For a more intuitive explanation, diagrams will be used following the convention specified in figure 1.1.

Chapter 2

The Git structure

Git, as said before, is a distributed VCS. 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: each user is able to fetch or push updates from other any other user 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 2.1. Local and remote repository have the same internal structure. What is a local repository for an user is a remote repository for another user.

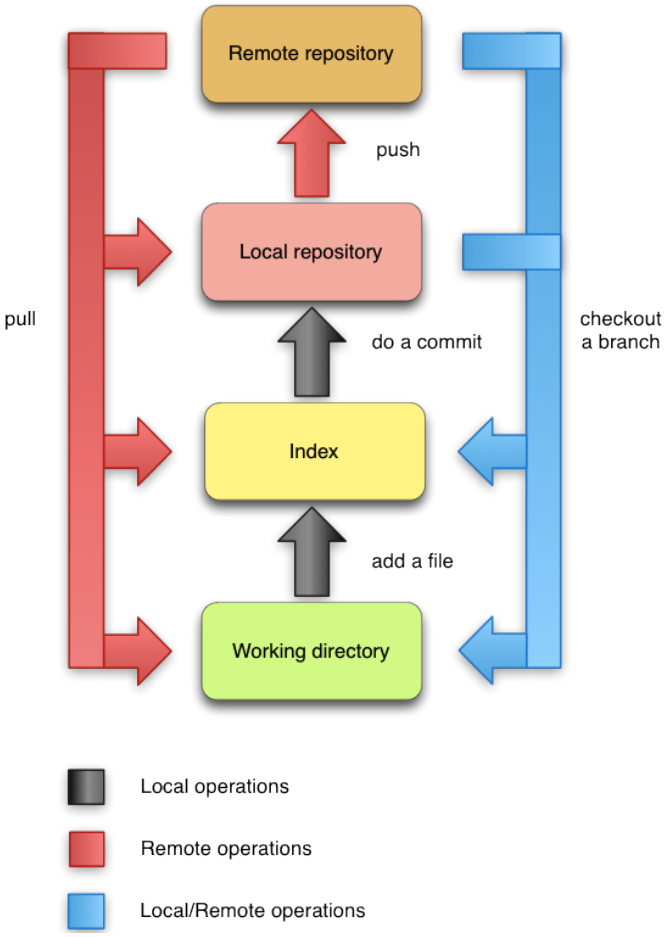

Figure 2.1: Git Workflow

2.1 The Repository

Git repository mirrors the evolution of a project during its lifetime. It is the place where the snapshots taken during the project construction are kept, as all the necessary information to allow the users to navigate 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 sections we present both of them.

2.1.1 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 at that moment. So a tree contains others trees and/or blobs. In *git* the files are not stored as such. What *git* keeps is an 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.

Figure 2.2 presents an example of two snapshots. We have two branches (the gray round square) pointing to two different commits (the blue circles). The "master" branch is pointing to the current commit, because it is identified by HEAD. It has as parent the commit "ff12" that is also pointed by the branch "Foo". We can see from the current commit that when it was performed only the files "file1.txt" and the "file3.txt" were in the index. It is also possible to see that in the root folder there was a file called "file3.txt" and a folder called "conn". Inside this folder there was a file "file1.txt".

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 comparing the hash and if two objects have the same content then only one of them is kept.

Blob

A blob object, as said before, represents the content of a file and the blob identifier is calculated from the blob content. So if we have two files exactly with the same content, only one blob will be stored, even if, they have different names. This happens because the blob is not directly associated to

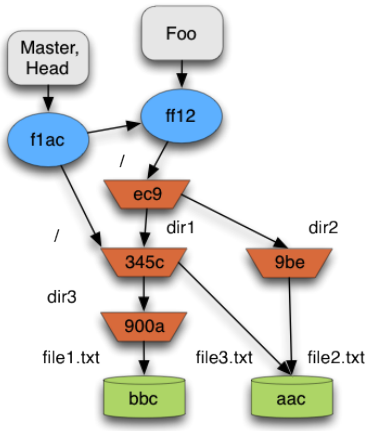

Figure 2.2: A Snapshot

a file. The relation between a path (of a file) and a blob is kept in the tree object. Figure 2.2 shows an example of a blob representing two files.

Tree

A tree is nothing else than a map from names to blobs and other trees. 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. Having the same content means having 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.

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 mandatory, when creating a commit, to write a comment about the commit. Such comments should contain details about the changes that were done in the project.
- 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.

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 chapter.

2.1.2 Git References

Basically a reference is just a pointer to a commit. There is much more about references, but in this manual we will focus on Branches and the special reference HEAD.

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, branch 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.

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 identify directly a commit, like for example, when a checkout is performed

using a commit hash. But in this document lets keep it simple, HEAD will identify always a branch.

2.2 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? That is the role of index which we present next.

2.3 Index

The index is something in between the working directory and the repository. It has mainly two functions: to control which files are being tracked, and to stage a file with the respective content to be committed.

When a file is created, if the user wants that file to be tracked it has to be added to the index. Each time the user modifies the file's content and does not add it to index, the content in index will diverge from the content in the working directory. The file staged in index (file and respective content) is what is stored when a commit is created, irrespective of its current content in the working directory.

It is important to notice that the index contains just files. Directories are not tracked or staged, even if, the user can still use the add operation with a folder as parameter: instead of adding the folder to index, *git* recursively adds all the files inside that folder to index.

Chapter 3

Git operations

In this section we explore some of the git operations. We try do so, in a different way, when 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 the description of an operation in four parts. The first part, gives an high level description of the operation. In the second part, we give the pre-conditions or, in other words, we explain in which conditions the operation can be performed. In the third part, we show what is the result of performing such operation. Finally, in the last part, we show some examples of the operation being performed.

3.1 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 operation does not simply refers of adding a file to the file. Instead it refers of adding a file with a certain content to the index. If we have a file in the index and we modify it on the working directory, for this modification to be visible on the next commit, the file must be added again to the index.

The remove operation removes a file from index, so it will not be present in the next commit and will stop being tracked. The removed file besides of being removed from index, will also be removed from the working directory (if it still exists there).

3.1.1 Pre requisites

The add operation has only one pre-condition. The file has to be in the working directory. It means that any file that is in 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 current commit (the commit pointed by the branch identified by HEAD) 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 a file's contents. It is possible to have a file in the 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.

3.1.2 Result

After performing the add operation there is something that will be always observed. The file will be in index. Besides, the add operation is also used to resolve conflicts. If a file was marked as unmerged, when performing an add operation, the file with the current content is added to index and the unmerged mark is removed. A file is marked as unmerged when it results from a conflict in a merge operation. 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. If the file was marked as unmerged, remove resolves the conflict by deleting the file.

3.1.3 Examples

Figure 3.1, shows a simple case of adding a file to the index. It can be seen, that no changes are made in the repository nor in the working directory. The only effect is that the index will contain the new file. The dots means that the content is kept.

Figure 3.2, contains the case of updating the content of a file. Assuming that the file is already in the index, changing its content in the working directory, will not change the content in the index. We need to explicitly add

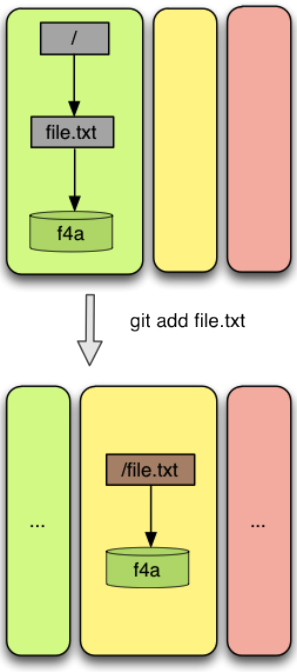

Figure 3.1: Adding a file

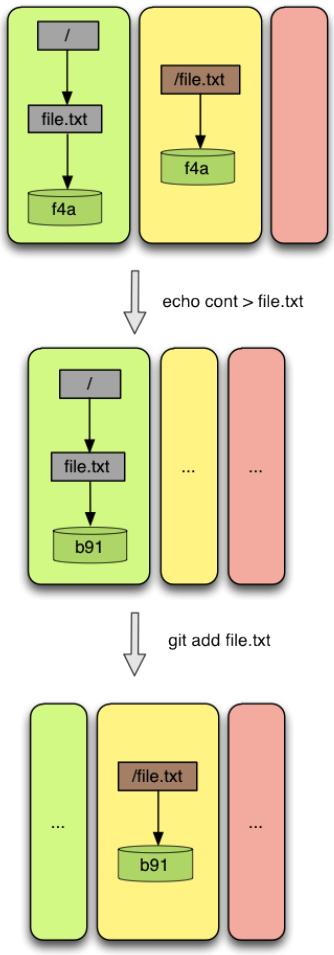

Figure 3.2: Updating a file

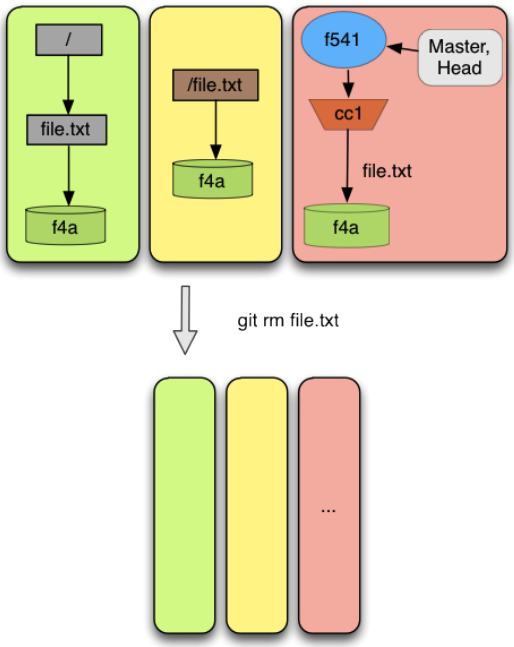

Figure 3.3: Removing a file

the file with the new content in the index.

Now lets loot at the remove operation. Figures 3.3,3.4 show a simple case of removing a file from the index. As the reader can see the file will be removed from the index, but also from the working directory. There are no changes on the repository.

3.2 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.

3.2.1 Pre requisites

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

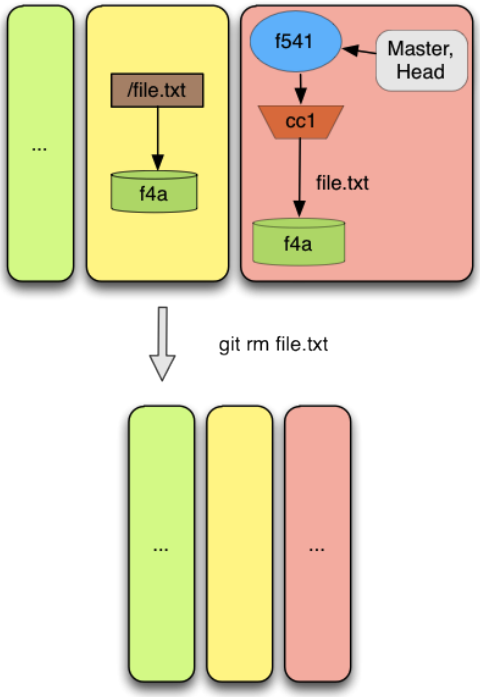

Figure 3.4: Removing a file (alternative)

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

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

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.

3.2.2 Result

The result of performing a commit can be different if it is the first commit or if there are already some commits 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 is 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. 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 point 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 commits in the repository, then it is guaranteed that there is at least one branch and the HEAD is identifying some existing branch. The branch that is identified by the HEAD, is pointing now to the commit created. This commit have now as parent the previous commit (the commit that was previously pointed by the branch identified by 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.

3.2.3 Examples

The figure 3.5 shows a typical process where two files are added to the index and then a commit is performed. It can be seen that the new commit reflects the working directory's structure with these two files. The interesting point is that, because these two files have the same content, they will share the same Blob. This is a case where *git* shows its efficiency.

The previous commit illustrates the case in which the first commit of the repository is being performed. In Figure 3.6 it is possible to see an example of a commit operation, in which the only difference from the previous commit is a removed file. This figure is interesting because it shows which changes are made in the repository when a new (non root) commit is created. The new commit points to the previous commit and the branch (which is identified by HEAD) points to the just created commit. As expected the new commit reflects the working directory's structure for the existing files.

Figure 3.7 is just a concrete example of the second pre requisite referred above. In this case the commit cannot be performed.

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

3.3.1 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.

The second pre-condition is kind of obvious. We cannot create a branch with the same name as one already in the repository, otherwise we would have conflicts. The pre-condition 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.

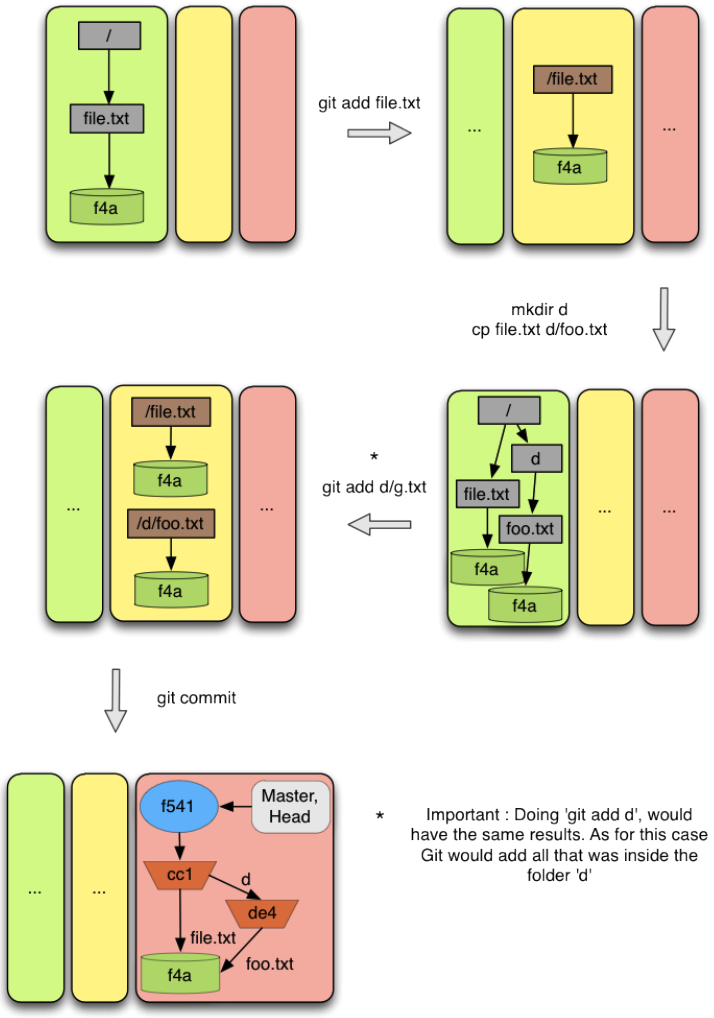

Figure 3.5: Doing a commit

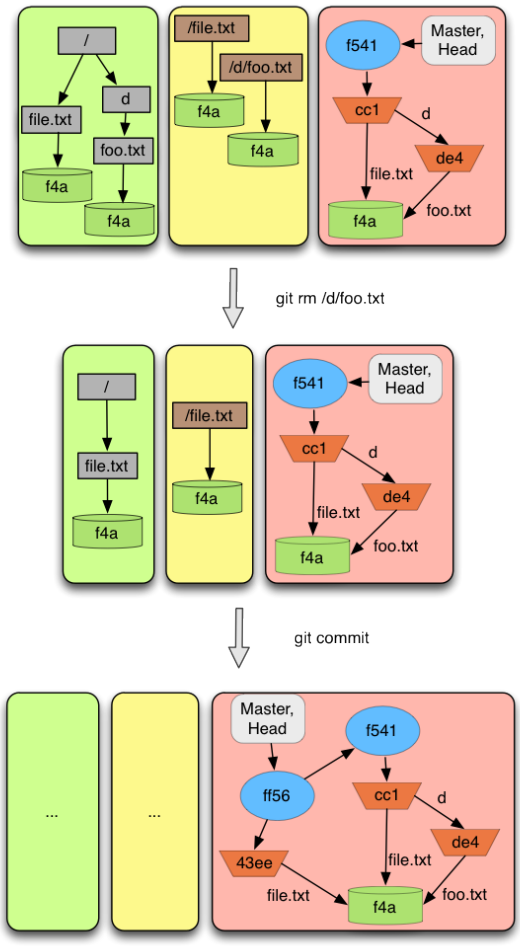

Figure 3.6: Doing a second commit

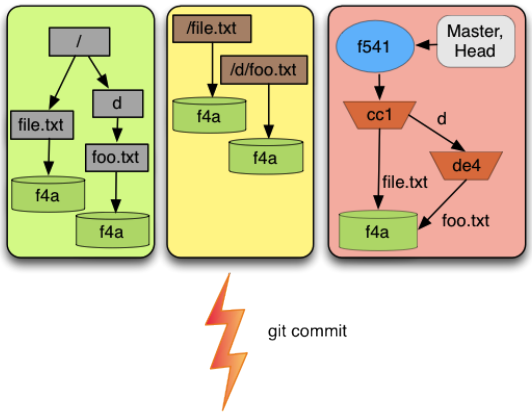

Figure 3.7: An index which does not satisfies the commit pre-conditions

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 point from now on to some branch that does not exists anymore. Thus, there would not exist a current commit. The last pre-condition exists to enforce that 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 all commits only accessible from this branch.

3.3.2 Result

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

3.3.3 Examples

As we wrote before, the changes of creating a branch are minimal. The only thing changing in the repository is the new branch pointing to the current

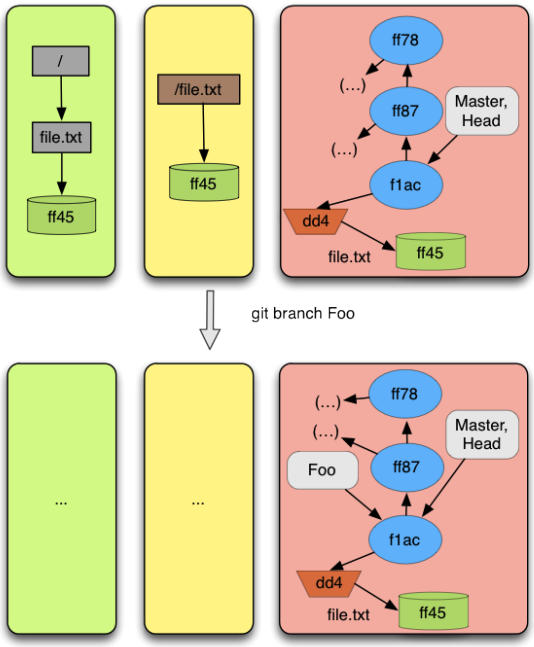

Figure 3.8: Creating a branch

commit, as can be seen in Figure 3.8.

When removing a branch the changes are also minimal. The only thing changing in the repository is that the branch referred must be deleted. If instead of looking at Figure 3.8 from top to bottom, we look at it from bottom to top, it shows the case when a branch is removed.

3.4 Checkout

The checkout operation 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.

3.4.1 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 first 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 last 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 git does not warns the user.

3.4.2 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 target commit object. 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 are in $(IA - CA)$, they 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.

3.4.3 Examples

As referred above checkout behaves in a non-intuitive way. We recommend that before performing a checkout, the user ensures that all the files from the index are committed, otherwise it is not be easy to guess what will happen.

Thus, the example shown in figure 3.9, contains an example in which all content in the index and working directory are committed (the safe case). When performing a checkout we can see in this figure that the repository and index are updated to reflect the branch checked out, more concretely the commit pointed by the branch. Something the reader can also observe is

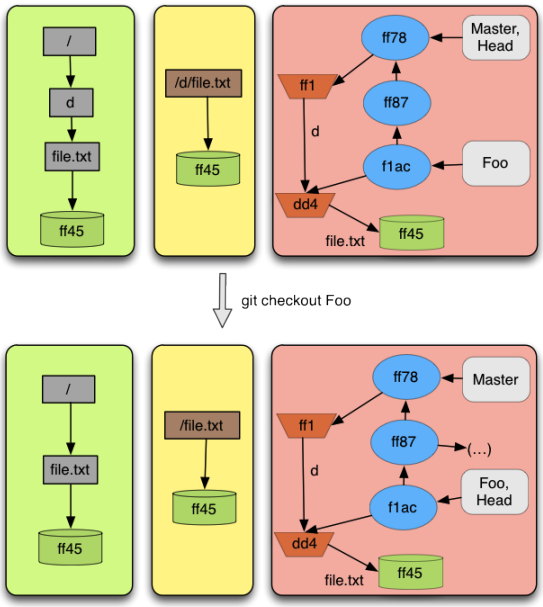

Figure 3.9: A simple checkout

that, after the checkout is performed, the HEAD identifies the branch that was checked out.
The last example in figure 3.10, shows the case where the file 'bar.txt' is in the index, but it's not committed. What happens is that the file will travel to the branch 'Foo' and will stay in the working directory and in the index. As if the file were created and added to the index there. So we have were a case where a file it's in the index but not committed and a checkout is permitted. The file, in fact, is passed to the other branch. In our opinion this is a behaviour that could lead to some problems when checking out.

3.5 Merge

The merge operation is one the of most important operation in *git*. It consists on trying to combine information from two commits. The result of that is a new commit with information from both commits. However there are some cases where this is not true. One of this cases is when the current commit is a precedent from the commit being merged. In this case, it is not created a new commit. The current branch (identified by HEAD) is updated to point to the commit being merged.

Sometimes this operation of joining commits is not possible to do automatically because there are conflicts, or in other words, when the same file

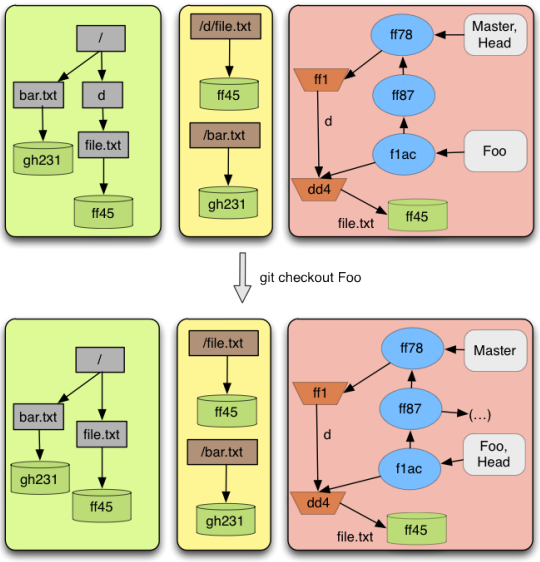

Figure 3.10: A checkout with a file not committed

exists in both commits it is not possible to say which one is more recent. In this case the merge operation has to be finished by the user. The user has to resolve the conflicts and then perform a commit.

Git tries to be as most autonomous as possible when merging. Thus, it has three types of merge:

- **Fast-forward merge** (figure 3.11): It happens when the commit being merged is accessible from the current commit by the parent relation. If that happens, then the commit being merged is more recent than the current commit. The result will be the HEAD pointing to the commit being merged. A new commit is not created;
- **3-way merge** (figure 3.12): It 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.
- **2-way merge** (figure 3.13): It happens when there is no common ancestor, thus it will be more likely that conflicts will appear.

The reason why a 3-way merge can solve automatically more conflicts than a 2-way merge is because it can verify if one of the versions was not modified relatively to the common ancestor. For example, if we have a file in both commits with diferrent content, but one of them is equal to the common ancestor, then the 3-way tactic assumes the file was not modified. Thus, the file from the other commit takes precedence.

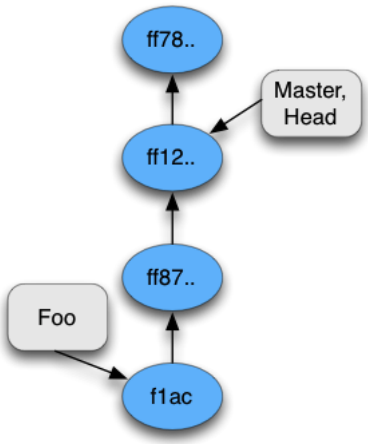

Figure 3.11: A fast-forward case

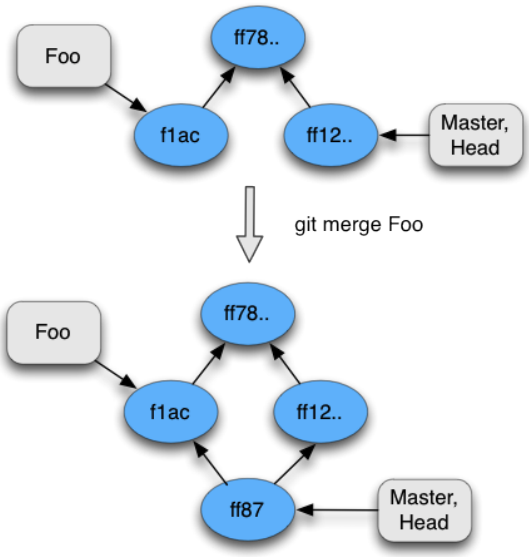

Figure 3.12: A 3-way merge case

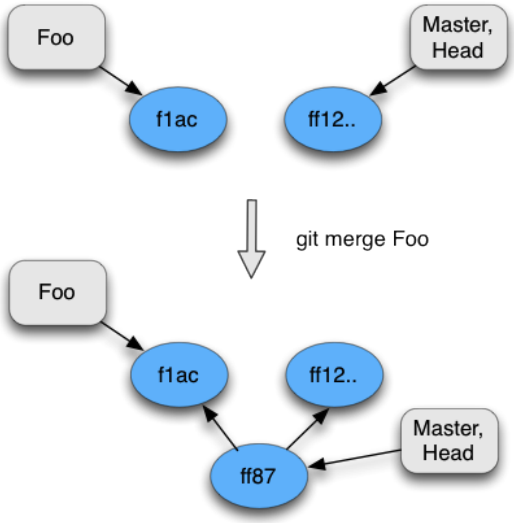

Figure 3.13: A 2-way merge case

3.5.1 Pre requisites

The merge operation, as all other operations, has some pre conditions that must be satisfied in order to be performed. As in the checkout operation, *git* lets the user to perform a merge without all information in the index being committed:

- A merge cannot be performed if the current commit is more recent than the commit being merged. As the current commits is more recent, no changes would exist.
- There cannot exist an unfinished merge on the repository. If the user performed a merge previously and that merge was not automatic, then a commit must be performed before performing a new merge.
- When it is detected that the fast-forward merge will take place, the index cannot have uncommitted files that would would conflict with files from the resulting merge.
- When it is detected that a 2-way or 3-way merge will take place, then uncommitted files cannot exist in the index.

3.5.2 Result

As already said when performing a merge, one of three tactics can be applied. In the case of a fast-forward merge, the operation will be always automatic. It

means that no conflicts will ever be found. In the case of a 2-way merge or 3-way merge, the merge operation sometimes cannot be resolved automatically because conflicts are found. Next, we give a small description of what changes after a merge operation is called.

- If it is 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 there are no conflicts. Of course, the current branch and HEAD will point to the newest commit.
- When it is a 2-way or 3-way automatic merge, the index and the working directory will reflect the new commit (the created commit). Here, the question of uncommitted changes does not exists because *git* does not allow uncommitted files when applying these tactics. In the repository a new commit is created that has the combined changes of the two (possibly three, if it is a 3-way merge) commits involved in the merge. That new commit will have as parents those two commits and it will be pointed by the branch identified by the HEAD.
- If when performing a merge, it cannot be performed automatically, git will throw a message saying that the automatic merge failed and the user has to finish it. In other words it leaves to the user the work of resolving the conflicts that *git* could not resolve. At that moment, the index and working directory will contain all the files that were merged. The working directory will also contain the unmerged files, with marks for the user to solve the conflicts. The index contains the 2 (or 3 in the case of a 3-way merge) versions of the files. As soon as the user solves the conflicts and adds it to the index, this file will be marked as merged. After all files are merged, a commit can be performed. In this manual process, git keeps somewhere in the repository that a manual merge is taking place and it also keeps which commits are being merged. This information is used when the commit operation is performed, mainly to know which commits will be the parent from the created one.

3.5.3 Examples

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

Figure 3.14 is a simple case of a merge operation with the branch "Foo". The figure shows that the Master branch (which is identified by HEAD) points to a commit that is accessible by the commit pointed by 'Foo' by the

parent relation. 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 new created commit now pointed by 'Foo' and Master.

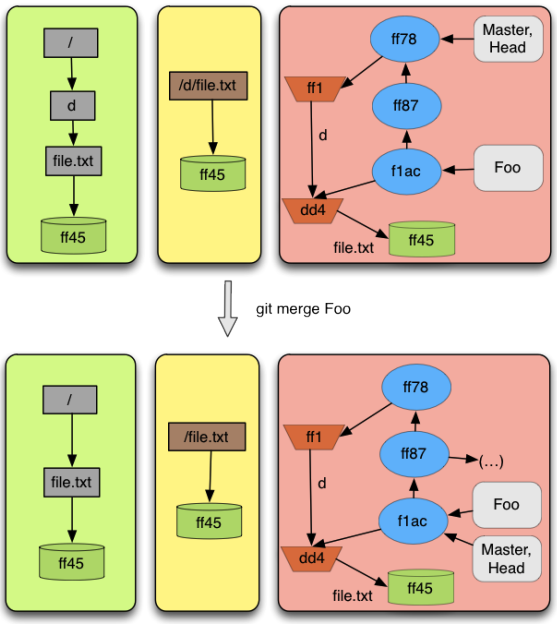

Figure 3.14: A fast-forward merge case

Figure 3.15 shows a concrete example of a state in which it is not possible to perform a merge. This happens because the commit pointed by Master (which is identified by HEAD) takes precedence over all files.

The two last figures 3.16 and 3.17 demonstrates the difference between a 2-way and a 3-way merge commit. When using 2-way merge there are conflicts that cannot be solved automatically, in the 3-way merge the conflicts are resolved automatically by looking at the common ancestor. Because the 3-way merge was automatic, a new commit was created that represents the merged information. Also, the working directory and index were updated accordingly. In the 2-way merge the system is left in a state for the user to solve the conflicts manually by updating the unmerged files and adding it to the index. Finally, when 'git commit' is performed a new merge commit is created and the merge process terminates.

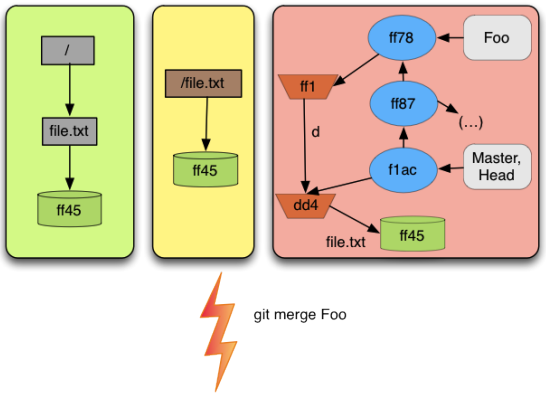

Figure 3.15: A fast-forward case that can't be done in *git*

3.5.4 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 they are used to get content from a remote repository or to push local content to a remote repository.

If a pull operation is performed, the branch from the remote repository will be fetched and it will be merged with the local branch. So, it is expected from this operation to behave exactly as a merge operation and consequently everything that was referred in the merge operation applies also here.

If a push operation is performed, the local branch will be uploaded to a remote repository. A pre-condition of this operation is that the local branch contains the remote branch. This happens to ensures that the local user resolves the conflicts.

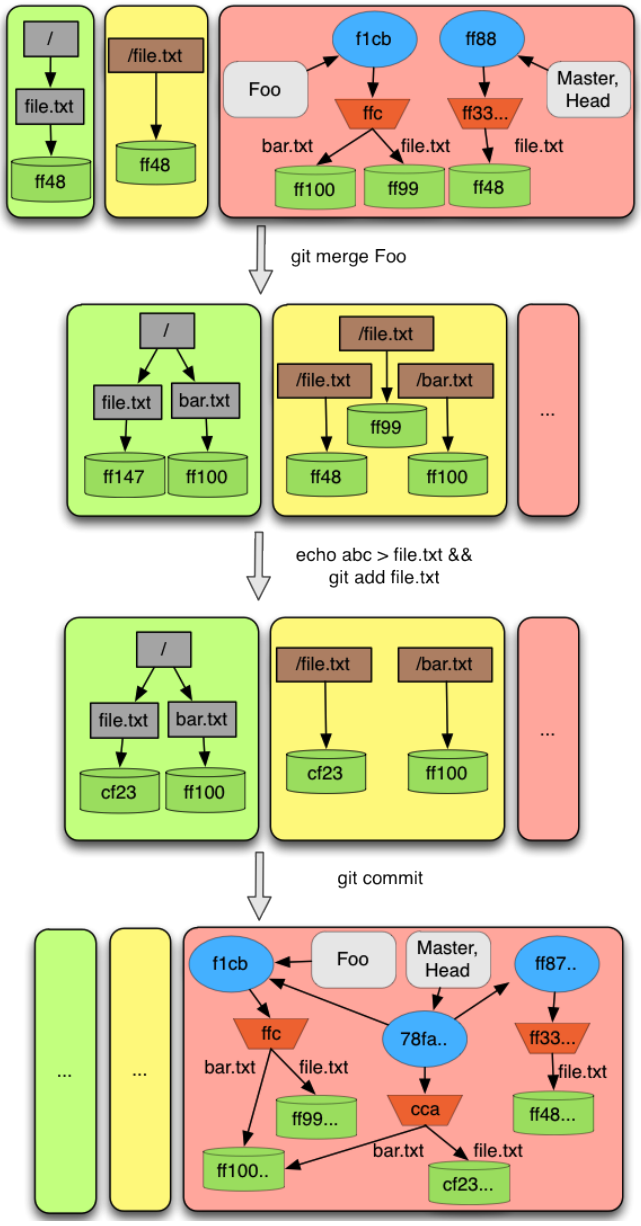

Figure 3.16: A typical 2-way merge

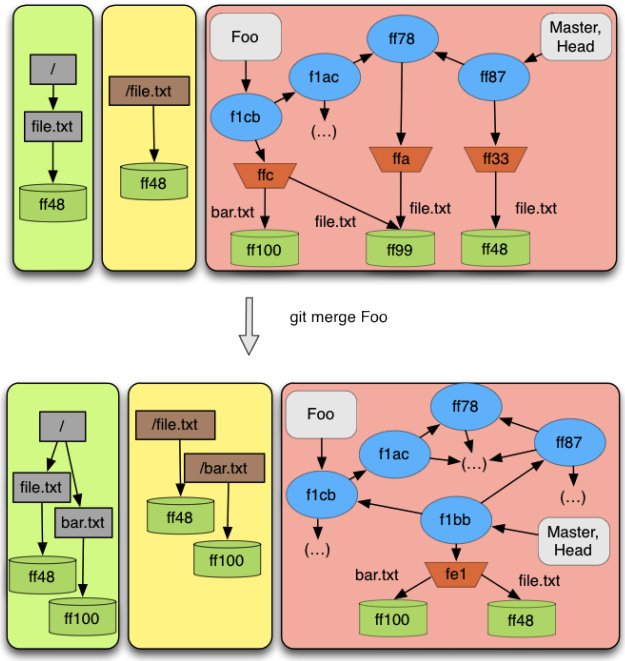

Figure 3.17: A typical 3-way merge

Chapter 4

Properties

After the *git* internals and some operations were modeled, we checked for some properties. These properties are important to prove the correct behavior of the model. Such properties were checked using Alloy Analyzer, which tries to find counter examples for the properties. This analysis is bounded, which means that it is not a complete analysis, but instead it analyses the model up to *"n"* objects.

In the next sections we present some of the verified properties. We start with the invariant preservation, then we show which operations are idempotent and then we check for operation's specific properties.

4.1 Invariant Preservation

An invariant is a property that is always true in the model. In this section, we show that the operations we modeled preserve an invariant when they are performed. Our invariant says the following:

1. There is a commit if and only if there is some created branch and HEAD is known;
2. The HEAD identifies an existing branch;
3. Every branch points to a commit;
4. Each branch identifies only and only one commit;
5. There is not any object that does not descend from an existing commit;
6. Every Tree has content (Blob and Trees) and these objects exists in the repository;
7. Every commit object points to an existing tree and if it has a parent it exists in the repository;

8. There are not two consecutive commits pointing to the same tree.

Most of these properties are trivial. The first property holds because before the first commit is created it is not possible to create branches, and after that it is never possible to remove the branch identified by HEAD. The property number 6 holds because *git* does not allow empty Trees and every object contained in a tree exists in the repository. The last property guarantees that each existing commit has a different snapshot from its parent.

To check if an operation preserves an invariant, that invariant must be valid after the operation is performed, assuming that is valid before. We have checked for each operation and we conclude that our operations preserve the invariant.

4.2 Idempotent Operations

An operation is said idempotent if after being performed, repeating it does not change the state of the system. It does not make sense to check if this property holds for some operations. For example after removing a file or a branch, it cannot be removed again. The idempotent operations in our model are: commit, add, and checkout. Next we try to justify such property.

The commit operation is idempotent because, if we perform a commit which is equal to the previous, git will not create a new commit and nothing is changed in the repository.

The add operation is also idempotent because while the file in the working directory keeps its content the add operation, after performed the first time, does not brings any differences.

The last operation we proved to be idempotent and maybe the less intuitive is the checkout. As we have seen in 3.4, the checkout operation updates the index and the working directory to reflect the commit pointed by the branch being checked out. Even if there are some "weird" pre-conditions this operation is still idempotent.

4.3 Add, Remove and Commit Operations

For this operations we have checked the following properties, which were valid. Notice that when we write "new file", we mean a file that does not exists in index.

1. Adding a new file to the index and removing it, brings us to the initial state;

2. If a commit is performed after an add operation, the added file with the respective content will be in the commit pointed by the branch identified by HEAD;

3. If a commit is performed after an remove operation, the removed file will not be present in the commit pointed by the branch identified by HEAD;

4. If a commit is performed, a new file is added to index and a new commit is performed, then the first commit and the second commit will point to different Trees;

5. Assuming that the invariant is verified and the following operations are performed: commit, add a new file, commit, remove the file added previously, and commit, then the first commit created and the last one will point to the same Tree object;

4.4 Branch and Checkout Operation

For the branch operations (add and remove) we have checked two properties. We have checked if after creating a branch and then removing it we reach a state with the same branches as in the begin. We have also checked if when a branch is created it points to the same commit as the branch identified by HEAD. In both cases no counter examples were found.

Relatively to the checkout operation it was verified the following:

1. If when the invariant is true and a checkout from a branch "b" is performed, then all content from the commit pointed by "b" will be in index. Such property is not true, due, the strange pre-condition we described in 3.4.1. As it is shown next, if the checkout operation is performed exactly after a commit, the property holds.

2. If a commit is performed, a checkout from a random branch is performed and after that a new checkout from the initial branch is also performed, then the index content after the commit will be the same as in the end.

Chapter 5

Conclusion

In this project we started by modeling the git internals using Alloy [4]. When we reached a stable model, we started modeling the most used operations: add, rm, commit, and checkot. Such model was defined based on the documentation in some popular git manuals [2, 1], the git man pages, and by performing additional tests using git. After some operations were modeled, we checked for some properties, like invariant preservation or idempotence.

We had many difficulties when modeling the operations due to the lack of precise descriptions. Many tests had to be performed to find a general behavior for each operation. During those tests a bug in git was found. In appendix A and B, we describe a sequence of operations to reach a state were information is lost in the checkout operation. This bug was reported in the Git Mailing List and by the replies we received many people seemed confused about the intended behavior of this operation. First, we received some answers saying it was the normal behavior and then the *git* maintainer concluded that it was indeed a bug.

This report, as well as http://nevrenato.github.com/CSAIL_Git/, show a detailed description from the work we have done analysing the model we built. We wrote the report without presenting the Alloy formalization, so that readers not familiar with Alloy can also benefit from it. The full Alloy model can be found in the above URL.

The time we had to work on this project was not enough (not even close) to go through the whole *git*. Some choices were made trying to simplify the model and choosing the most important operations. The model is considered by us, to be stable and ready to be extended with more operations.

Bibliography

[1] Scott Chacon. *Pro Git*. Apress, Berkely, CA, USA, 1st edition, 2009.

[2] Git community. *Git Community Book*.

[3] Charles Duan. Understanding git conceptually. Website. `http://www.sbf5.com/~cduan/technical/git`.

[4] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012.

[5] Tommi Virtanen. Git for computer scientists. Website. `http://eagain.net/articles/git-for-computer-scientists/`.

[6] Denis Zack. A walk-through of the commands used in git workflows. Website. `http://www.mutuallyhuman.com/blog/2012/06/22/a-git-walkthrough/`.

Appendices

Appendix A

Bug found in *Git*

```
teste$ git init
   Initialized empty Git repository in /home/smooke/Dropbox/teste/.git/
teste$ touch f
teste$ echo a > f
teste$ git add f
teste$ git commit -m 'first commit'
   [master (root-commit) dab04b9] first commit
   1 files changed, 1 insertions(+), 0 deletions(-)
   create mode 100644 f
teste$ git branch b
teste$ touch something
teste$ echo b > something
teste$ git add something
teste$ git commit -m 'something added'
   [master 9f2b8ad] something added
   1 files changed, 1 insertions(+), 0 deletions(-)
   create mode 100644 something
teste$ git rm something
   rm 'something'
teste$ mkdir something
teste$ cd something/
something $ touch f1
something $ echo c > f1
something $ cd ..
teste$ git add something/f1
teste$ git checkout b
   Switched to branch 'b'
teste$ ls
   f
teste$ git checkout master
   Switched to branch 'master'
teste$ ls
   f  something
teste$ cat something
   b
```

Appendix B

Bug found in *Git*

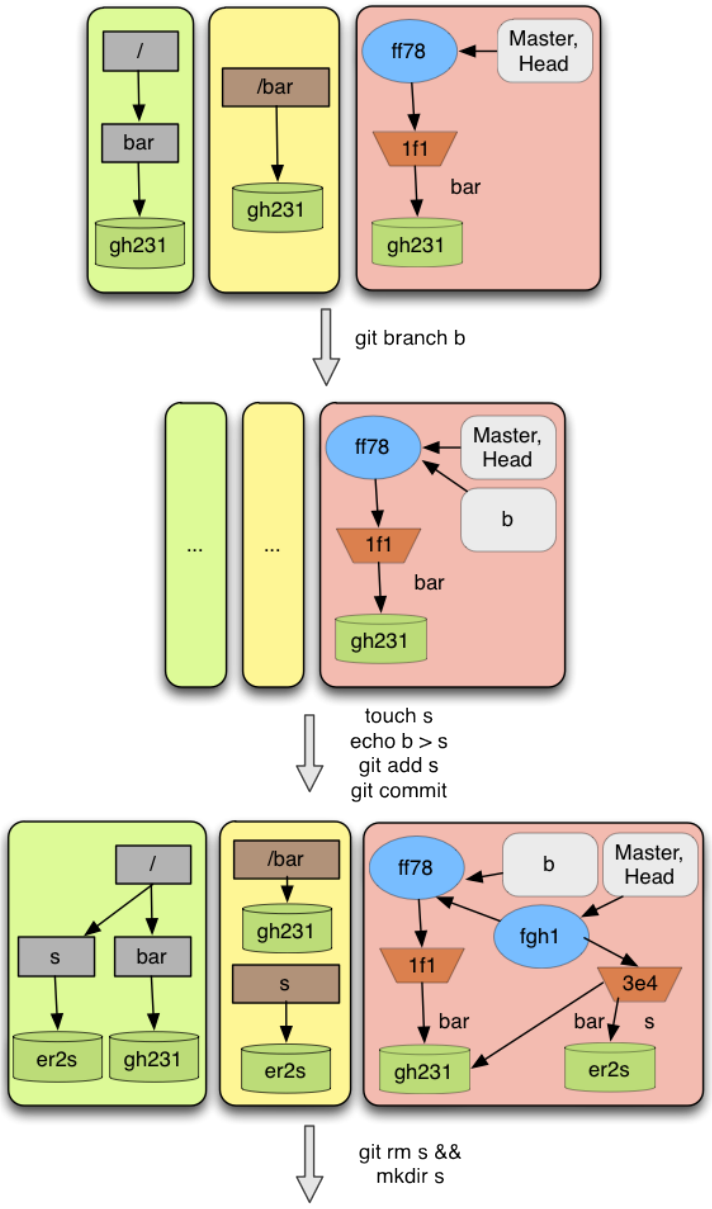

Figure B.1: Showing the git bug (part 1)

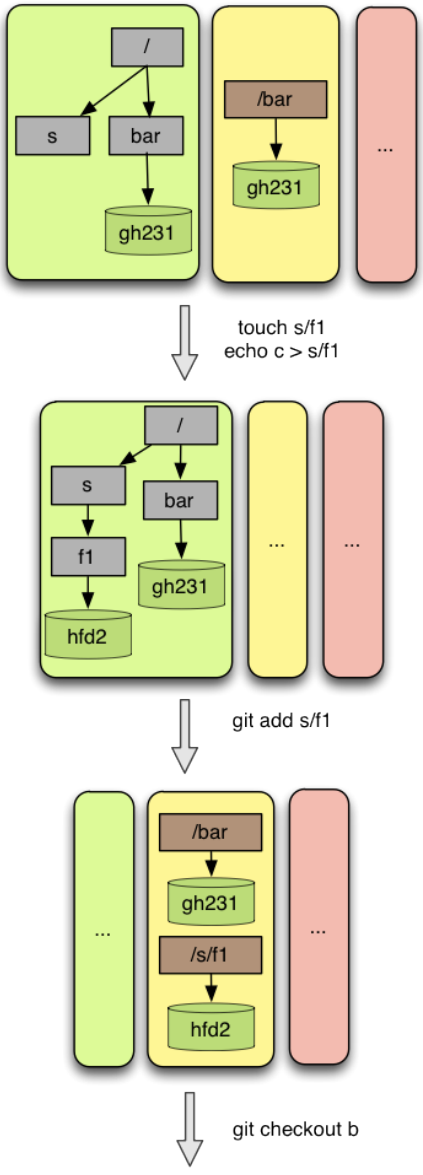

Figure B.2: Showing the git bug (part 2)

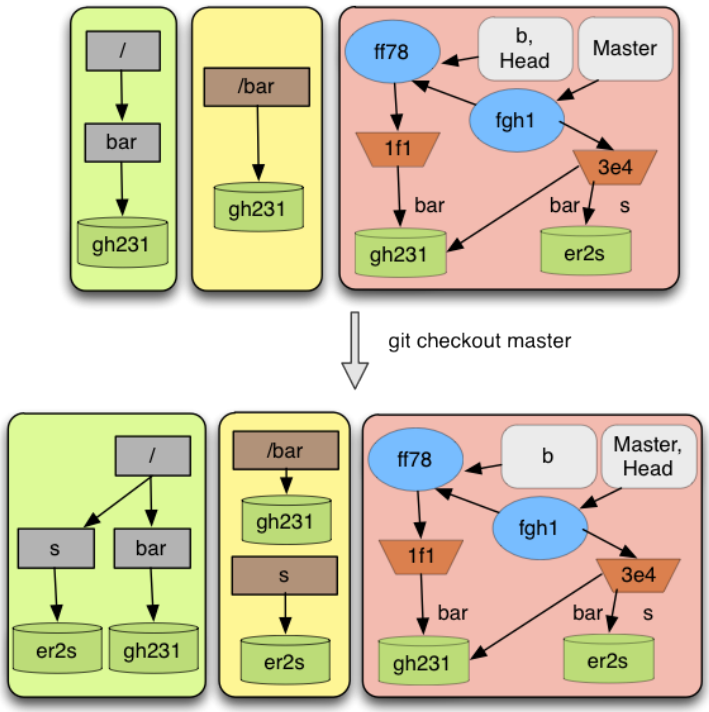

Figure B.3: Showin the git bug (part 3)