Modelling the Git core system with Alloy

Cláudio Lourenço, Renato Neves
Universidade do Minho

July 4, 2012

Contents

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 appear. 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 pull/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 likes 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 next time.

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 formal way. After modeling the structures, it is possible to visualize instances of them.

So, the difference that this manual has from the others, 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, 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 *git* is structured and then we cover each component of *git*, working directory, index and repository. For matters of time and complexity we focus our model on the index and 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 as a mirror from the repository (local repository) and in the case of git, there is no need for a central server, even that, 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 other user.

2.1 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. This 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. It is possible to have in the working directory files that are not being tracked. This files are just ignored

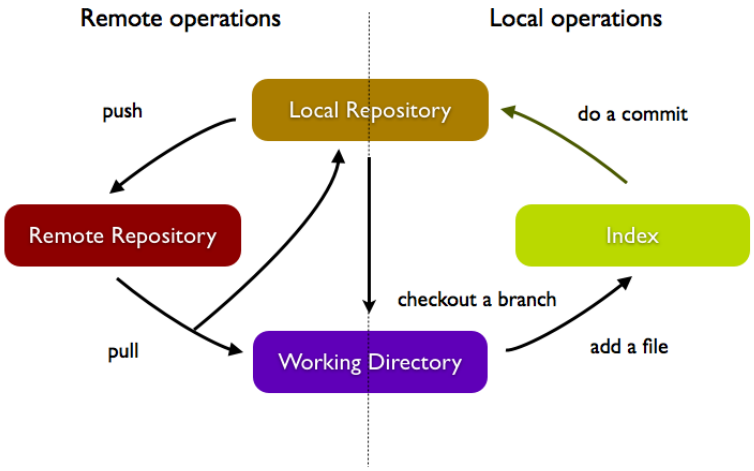


Figure 1: Git WorkFlow

2.2 Index

The index is a component that contains all the files that will be committed on the next commit (the next snapshot).

2.3 Repository

As said, Git is a distributed VCS. So we can work not just with a local repository, or remote repository, but we can work with both types at the same time. One of the advantages of Git is that we can work off-line and then at the end of day, we update the remote repository with our work.

In fact, the definition of local/remote depends only on the perspective of the user, as there are no structural differences between local and remote repositories. Imagining two users, where each one has his local repository, one will see the other as the remote repository and vice-versa.

A repository contains all the information needed to store, and stores all snapshots taken.

3 What lives in the repository?

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. A commit points to a tree, that represents the structure of your system on that moment. So a tree contains others trees and/or blobs. In git the files are not kept. What git keeps is a 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.

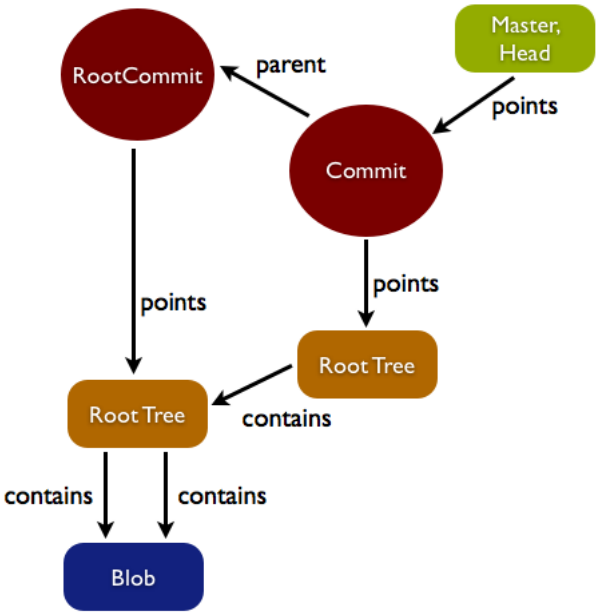
3.1 Objects

All objects are identified by an hash defined by their contents. Thus, when two files have the same content, they will have the same identification.

There are the following types of objects in the world of Git:

- A Blob that stores file data and can be seen as a file
- A Tree that stores blobs and other trees, thus can be seen as folders.
- A Commit that points to a snapshot of the repository at a given moment. Also points to previous commit(s) and identifies the user that made the commit.
- A Tag points to special commits. For example, a commit that has the beta version of some program.

Figure 2: A commit object pointing to a tree



3.2 Branches

One of the trump cards of Git it's his definition of branches. Other VCS have a copy of the entire repository for each branch made. Thus, branches

operations in those VCS are slow and hard to use. Now Git, just uses pointers for branches. In fact, when creating a new branch the only thing done is the creation of new pointer that will point to the current commit.

So, the branches operations in Git turn to be fast and easy to use.

It's important to say that exists a special pointer called "Head" that will always point to the current commit.

4 Git operations

4.1 Add and Remove

4.1.1 What they do?

When you use the 'git add <file>' command, a file will be added to the index. The file added can be a new file, or a file already existing in the index. When the latter is true, git just updates the content of that file.

When you add a file, it will be marked to be committed in the next commit.

'git rm <file>' can be seen as the inverse operation of the 'git add <file>'. In this case Git, will remove a file from the index, so it won't appear in the next commit. Besides being removed from index, it will also be removed from the working directory (if it's still there).

4.1.2 Pre requisites

There are a few cases where Git prevents you from removing a file, and they are:

- The file doesn't exist in the index.
- Removing a file, when the file with it's current content doesn't exist in last commit.

The last restriction exists, to avoid the accidental deletion of files. Imagine the case where you have a file 'x' with 10k lines of code. You add it to the index, but, before committing you accidentally use 'git rm x'. Because Git erases from the index and the working directory you would lose it permanently.

4.1.3 Examples

In these figures, it is shown the process of adding a file to the index. Note that both files showing represent the same file, but with different content at a given time. So imagine that, after adding "File1", "File0" is added. "File1" would be removed from the index (because there cannot be a file with different contents in the index at any given state) and "File0" would

be added. What was just described was the representation of the update of the content of a file in the index.

Figure 3: Before the add

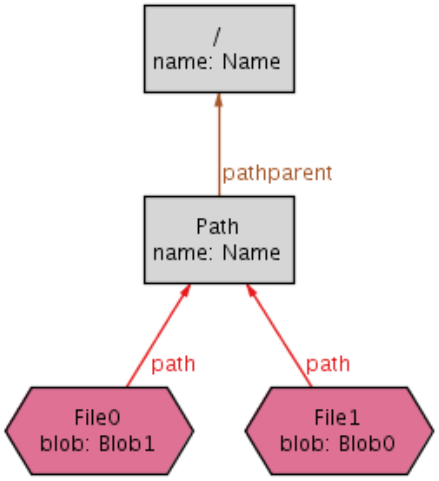


Figure 4: After the add

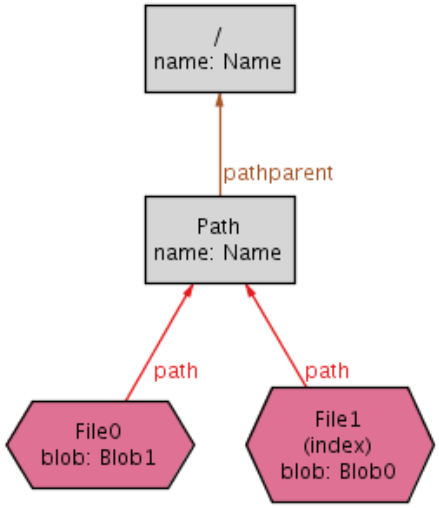
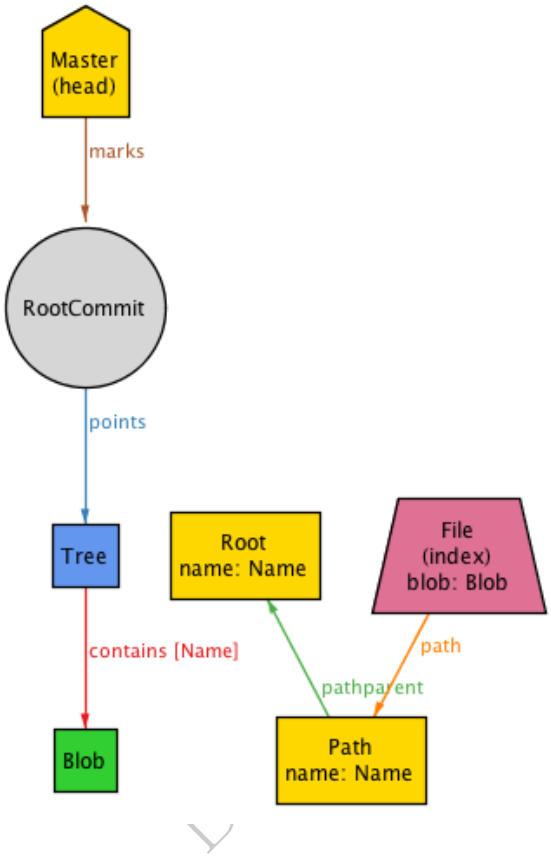


Figure 5: A first commit



4.2 Commit

4.2.1 What it does?

The 'git commit' operation, stores the current content of the index in a new commit.

When the first commit is created, a branch called "Master" will be created and will be marked as the current branch. Also, the first commit of a repository is called a RootCommit.

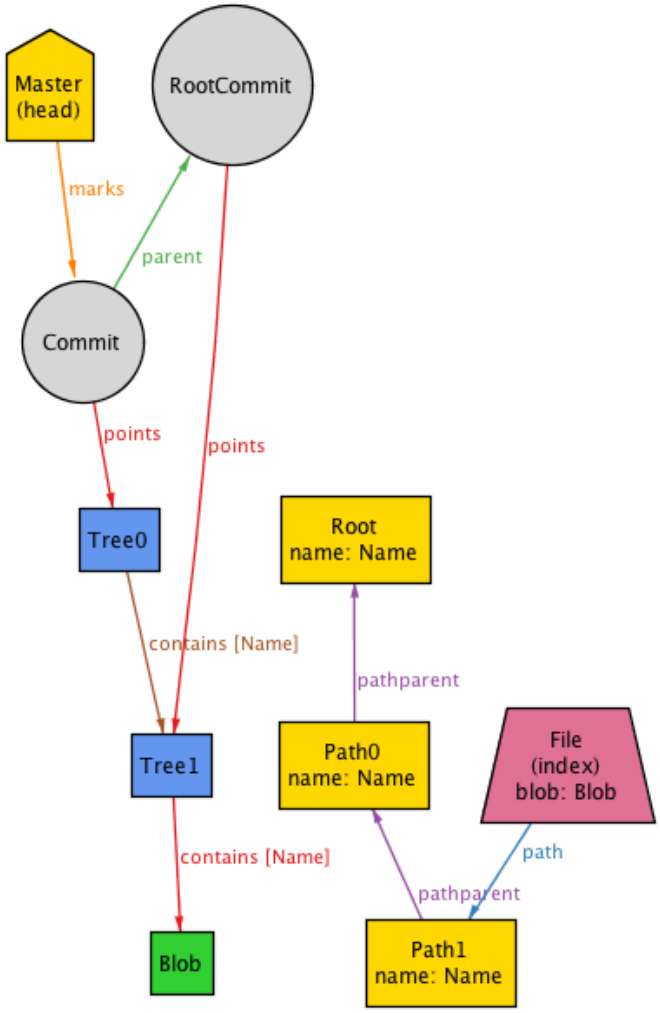
4.2.2 Pre requisites

The only restriction that Git has about this operation it's that your new commit must have something different from the previous commit, or, if it's the first commit, then there must exist some content in current index.

4.2.3 Examples

The figure 5 is a typical example of a RootCommit. There is a file with the path Name/Name, and a content Blob. It is currently on the index. The

Figure 6: A Commit



current branch is Master, and it is pointing to a commit, which represents the index at the current state.

The figure 6 represents the case of a normal commit, where the newest commit comes after the RootCommit. The line of thought is the same as in the last image. We can see that the difference between these two commits in the figure, is that the only file was moved to a folder named "Name".

We can also see the efficiency of Git, about the sharing property of objects in the commits, thus minimizing the space occupied in the repository.

4.3 Branch

4.3.1 What it does?

4.3.2 Pre requisites

4.3.3 Examples

Git branch operation has several forms, that to different things on git. The ones which we care are the forms that create and delete branches.

When a new branch is created it will point to the current HEAD by default.

The man git-branch says that when deleting a branch, it must be fully merged in its upstream branch, or in HEAD if no upstream was set.

4.4 Checkout

4.4.1 What it does?

4.4.2 Pre requisites

4.4.3 Examples

From man git-checkout : "Switches branches by updating the index, working tree, and HEAD to reflect the specified branch or commit".

4.5 Merge

4.5.1 What it does?

When you do a 'git merge <branch>', a merge will happen between the current branch and the branch selected on the command. The result will be a new merge commit, or a partially merged index.

Git knows three types of merge :

- A fast-forward merge, that will happen when the current commit from the other branch, is more recent than the current commit from the current branch.

  This type is not a true merge, in the sense that the only thing that changes it's the pointer of the current branch is moved to the newest commit.

- A 2-way merge, that happens when the conditions for a fast-forward don't exist, and there isn't a common ancestor for both commits.

- A 3-way merge, happens when the conditions for a fast-forward don't exist, but, there is a common ancestor for both commits.

  The difference between these two last types, is that the last is much more automatic having much less conflicts than the previous type.

Knowing these 3 types of merge, it's important to know that Git tries to make the merge process as automatic as possible. However conflicts do exist, and when they happen it leaves to the user the process of resolving those conflicts.

When the Git/user, finishes the merging of files in the index, he can commit the resulting merge, in a merge commit.

Note that while there are unmerged files in the index, a commit cannot be made.

4.5.2 Pre requisites

- A merge can't be made, if the current commit is more recent then the other commit.
- A merge can't be made, while there are unmerged files.
- In the case of a fast-forward merge, the index can't have uncommitted files that can be changed by the merge operation.
- In the case of a 2-way or 3-way merge operation there can not be uncommitted files.

4.5.3 Examples

4.5.4 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 and to place your content on 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 branch will be merged.

If a 'git push' is done, the remote branch will be uploaded to the remote repository and will be merged there.

However, a 'git push' can only be done when the resulting merge will be a fast-forward, simply because there is no one on the other side to resolve the merge conflicts. For the case of 'git pull' it's pre requisites are the same of the merge operation.

References

Figure 7: A typical case of a fast-forward commit

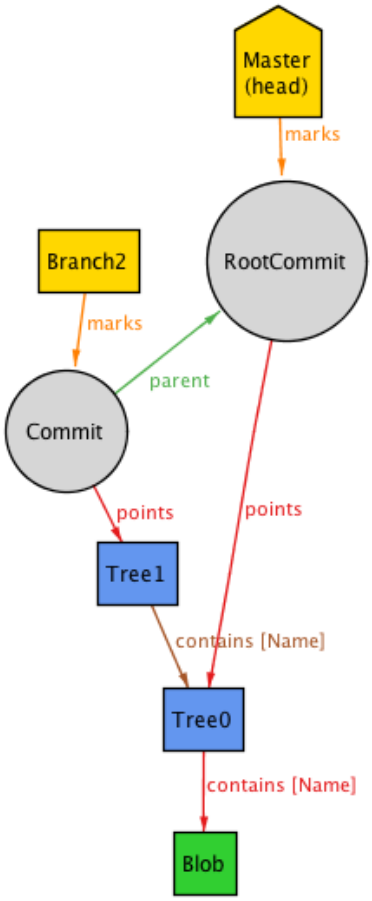


Figure 8: Normal case of a 3-way merge