Modelling the Git core system with Alloy

Cláudio Lourenço, Renato Neves
Universidade do Minho

June 27, 2012

1 Introduction

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 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 different 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.
Alloy description goes here
Nowadays git is having great success, it is efficient, fast and very easy to use. The problem here is that, most of the users do not dive into the git internals, they are just happy that it works. The ones that actually try to go deep into git to see how it works and why git behaves in a specific way, are fronted with the lack of formal or even informal specifications. So, the purpose of this project is to formally model the git core using alloy, analyse and model some git operations and then check which properties the model does (not) guarantee. This report presents to the reader a formal description of git internals, the description of some operations and it ends with some properties that the model guarantee.

This reports starts by explaining the organization of git, how git is structured. Then we cover each component of git. We begin with the object model, following the index and working directory. 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 formal specification of some operations. We finish the report with the analysis of some properties.

2 Git Structure

The git is divided mainly in three components. The working directory, the index and the repository. The connection between this components can be seen in figure 1.

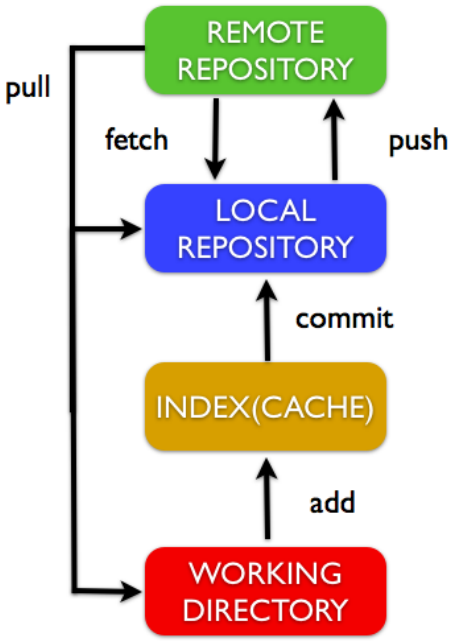


Figure 1: Git Structure

The working directory is a subset of a file system that contains the files of the project that you are currently working on. When retrieving an older snapshot of the project, the working directory is updated to reflect the project that state. The index is a component between the repository and the working directory. It contains the files that will be on the next commit (on the next snapshot). Finally the repository is like a database. It contains all the information about your system, all the snapshots taken till the moment. In figure 1 there are a local repository and a remote repository. In reality there are no difference between them, they have the same structure. Both

of them are local repository, and at the same time remote repository. Each one sees the other as remote repository.

For a matter of time and complexity, in this project we turned our attention mainly to the index and repository. We are not modeling what is going on in the working directory. We just model that there are files in the working directory, anything else.

3 The Repository

3.1 Git Object Model

As we have presented before git keeps snapshoots 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.

We will show the specification of the git object model, using as basis two manuals, [1] and [2].

Only the key parts of the model will be presented.

3.2 Identification of Objects

All objects are identified by a sha defined by their contents.

"All the information needed to represent the history of a project is stored in files referenced by a 40-digit **"object name"**..." (page 7)

We don't need to specify this, as Alloy already can uniquely identify atoms, so we can reutilize that functionality.

3.2.1 The four types of Objects

The objects are defined as in [1] (page 7) wich says: "...and there are four different types of objects: blob, tree, commit, and tag." Also: "A blob is used to store file data - it is generally a file" [1] (page 8).

For trees: "A tree is basically like a directory - **it references a bunch of other trees and/or blobs**..." (page 8)

```
abstract sig Object {
        objects : set State
}

sig Blob extends Object {}

sig Tree extends Object {
```

```
        contains : Name -> lone (Tree+Blob)
}
```

The tag will be discarded, as it doesn't affect the main operations of git.

Now for the commit: "As you can see,a commit is defined by : parent(s):**The SHA1 name of some number of commits which represent the immediately previous step(s) in the history of the project**..." "...merge commits may have more than one. A commit with no parents is called a root commit, and represents the initial revision of a project. **Each project must have at least one root. A project can also have multiple roots, though that isn't common (or necessarily a good idea)**". [1] (page12) A commit also points to a certain tree that represents the state of the repository at a given state.

```
sig Commit extends Object {
        points : Tree,
        parent : set Commit,
}
```

4 Branches in Git

One of the advantages of git compared to others VCS is the operation of creating a new branch. While in others VCS, a copy of the hole project is done each time we create new branch, in git, only a new pointer is created, as said in the next quotations: "A branch in Git is **simply a lightweight movable pointer to one of these commits**." [2] (pag 39)

"The special pointer called **Header points to the branch we are working on**". [2] (pag 40)

```
sig Branch {
        marks : Commit one -> State
        branches : set State,
        head : set State
}
```

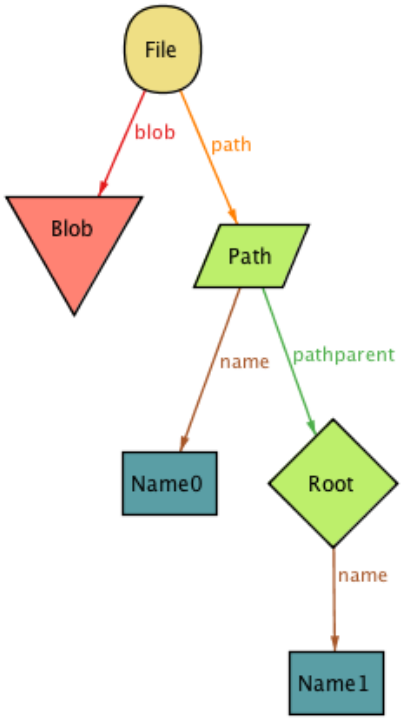
5 Specification of files

In order to model Git, we need the notion of a file. In our view a file should have a path associated and a content. The path (along with it's parents) will uniquely determine the full name of a file (e.g. /x/y/example.txt), the content will be just a Blob.

```
        sig File {
                path : Path,
                blob : Blob
        }
```

```
sig Path {
        pathparent : lone Path,
        name : Name
}
```

Figure 2: A typical example of a file, where is name is /Name1/Name0



6 Index in Git

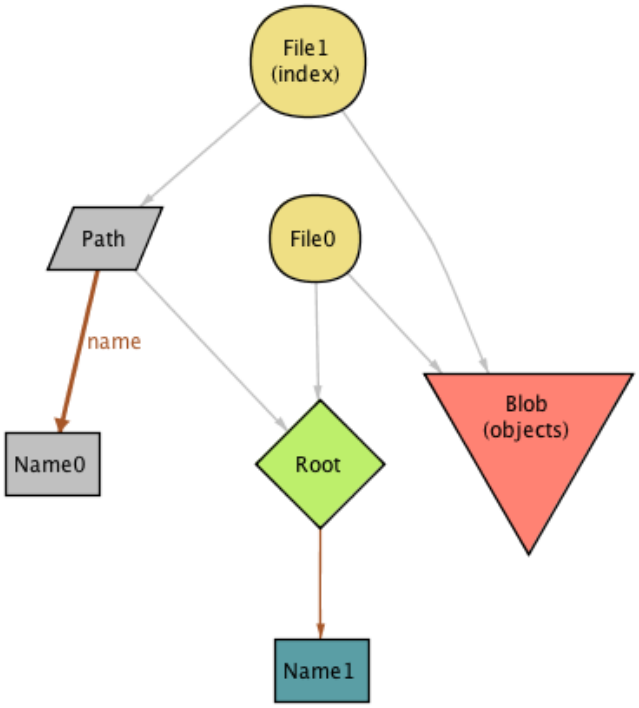
The definition of Index: "...staging area between your working directory and your repository. You can use the index to **build up a set of changes that you want to commit together**. When you create a commit, **what is committed is what is currently in the index, not what is in your working directory.**" [1] (page 17).

Also : "The index **contains all the information necessary to generate a single (uniquely determined) tree object**" [1] (pag 121).

Thus, what is in the index for a given state, is just a set of files.

```
sig File {
        path : Path,
        blob : Blob,
        index : set State
}
```

Figure 3: A typical example of a file (file1) that is in the index



7 Git operations

7.1 Add and Remove

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.

Figure 4: Before the add

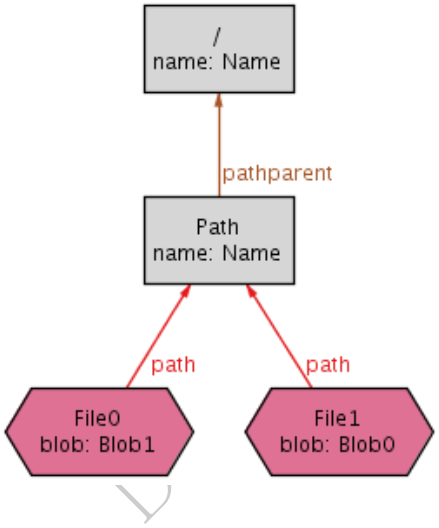
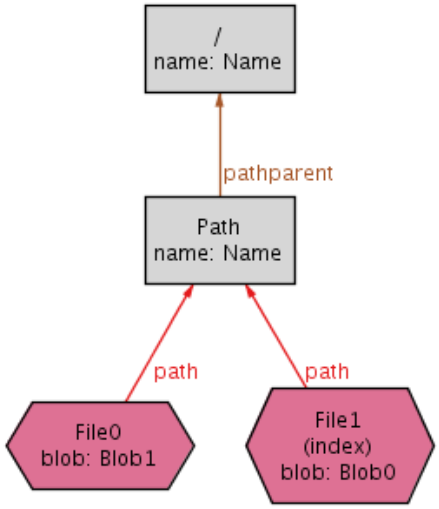


Figure 5: After the add



'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).

However, there are a few cases that Git prevents you from removing a file :

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

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.

7.2 Add and Remove operation properties

- Adding a file after adding the same file, has no effect
- Adding a file, committing, removing the same file and committing brings to a commit equal to before of adding the file
- Removing a file and then adding the same file brings to an index equal to the one before removing the file

7.3 Branch

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.

```
pred branch[s,s':State,b:Branch]{

        b not in branches.s

        head.s' = head.s
        branches.s' = branches.s + b
        objects.s' = objects.s
        index.s' = index.s

        marks.s' = marks.s + b -> (head.s).(marks.s)
}
```

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.

```
pred branchDel[s,s':State, b:Branch]{

        b in branches.s
        b not in (head.s)
        b.marks.s in (head.s).(marks.s).*parent

        head.s' = head.s
        branches.s' = branches.s - b
        objects.s' = objects.s
        index.s' = index.s

        marks.s' = marks.s - b -> Commit
}
```

7.4 Branch operations properties

7.5 Commit

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

The only restriction that Git has about this operations 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.

7.6 Checkout

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

7.7 Merge

When you do a 'git merge <branch>', a merge will happen between the current branch and the branch selected on the command. That merge can be automatic, or must have intervention from the user. There are also, three types of merge. A fast-forward merge a 2-way or a 3-way merge.

References

[1] *Git Community Book.*

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