Modelling the Git core system with Alloy

Cláudio Lourenço, Renato Neves
Universidade do Minho

April 26, 2012

1 Abstract

2 Introduction

In this report, we will try to give a complete specification of the git core system.

We will focus on the following parts :

1. Object Model
2. Index

After the specification of these parts, we will make the model dynamic, in order to specify git most important operations.

Following this, some assertions on the model will be introduced, in order to verify the git robustness and safety.

3 Git Static Object Model

In this section we will show the specification in Alloy of the git object model. This will be mostly based on the textual specification present in [1].

Thus, we will divide this section in conditions that are specified in [1] and conditions that we didn't found explicity in the book but that are needed for the proper functioning of git.

3.1 Explicit specifications

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

```
sig Sha{}
```

"...and there are four different types of objects: "blob", "tree", "commit", and "tag". " (page 7)

```
abstract sig Object {
        namedBy : Sha
}
sig Blob extends Object{}
sig Tree extends Object {}
sig Commit extends Object{}
sig Tag extends Object{}
```

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

```
sig Tree extends Object {
        references : set (Tree+Blob)
}
```

"A "commit" **points to a single tree**...." (page 8)

```
sig Commit extends Object{
        points : Tree
}
```

"A "tag" is a way to **mark a specific commit**..." (page 8)

```
sig Tag extends Object{
        marks : Commit
}
```

As the book [1] says, a "tree" acts like a directory, so we know that it or it's descendants cannot point to themselves.

```
no ^references & iden
```

"...two "trees" have the **same SHA1 name if and only if their contents (including, recursively, the contents of all subdirectories) are identical.** " (page 11)

```
all t,t' : Tree |
   t.namedBy = t'.namedBy <=> t.references = t'.references
```

"What that means to us is that is virtually **impossible to find to different objects with the same name**". (page 7)

However we know that there is an exception. There are cases (defined above) that allow different trees to have same name.

```
namedBy.~namedBy - (Tree->Tree) in iden
```

3.1.1 Commit structure

Again from [1].

"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**. The example above has one parent; 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)**". (page 12)

```
sig Commit extends Object{
  points : Tree,
  parent : set Commit
}
some sig RootCommit extends Commit{}
fact {
  no ^parent & iden
  no RootCommit.parent
  Commit in *parent.RootCommit
}
```

3.2 Implicit specifications

A commit must point only root trees, unless we want losses of information.

```
no Commit.points & Tree.references
```

In the object model we don't have orphan Blobs.

```
Blob in Tree.references
```

The same for Trees.

```
Tree in Tree.references + Commit.points
```

Sha's that don't point to anywhere are useless.

```
Sha in Object.namedBy
```

We can't have the same tree shared by commits. Git doesn't work that way ("it stores a snapshot of what all the files in your project looks like..." [1] (page 7)).

```
points.~points in iden
```

A tree has at most one parent. If this is not true, than there is inconsistency in the git "filesystem".

```
all t:Tree | lone references.t
```

4 Index

4.1 Explicit specifications

Now for 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).

"The index is a binary file (generally kept in .git/index) **containing a sorted list of path names**, each with permissions and the SHA1 of a blob object" [1] (pag 120).

```
one sig Index{
  stage: Sha one -> File
}{
  Index.stage.File in Blob.namedBy
}
```

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

This description tells us that the index saves a snapshot of the system, not the differences !

What is the practical difference between the Index and the Working Directory? The Working Directory is just the root folder of a repository, that we work on. When we want to save something in a commit, we have to explicitly tell git to add that to the Index. Once added, it will be saved in the git database at the next commit. If we don’t want to save something in a commit, we simply don’t add it to the Index.

4.2 Branches

Definition of branch: "A branch in Git is **simply a lightweight movable pointer to one of these commits**." [2] (pag 39)

```
 sig Branch{
   on: Commit
 }
```

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

```
  one sig Head{
        current: Branch
  }
```

5 Modelling Git Dynamic Object Model

The static object model, only by itself, is not very useful. And so the model would be much more interesting if we specify the most important git operations. That said in this section we will focus on modelling the most important and critical of them.

For this purpose, we will associate a State to each relation with potential to change it's content. Thus, this way a relation can change it's content over time.

5.1 add

This command is one of the most used in git. [1] (page 26) tells us that "git add is **used both for new and newly modified files**, and in both cases it takes a snapshot of the given files and **stages that content in the index, ready for inclusion in the next commit**."

As already said above a newly modified file, it's tracked by the Working Directory, so all references to modified files in the Index must also appear in the Working Directory. For newly added files that is not the case.

5.2 rm

This one is generally less used. It's usefullness can be seen in the following [2] (page 21) says "To remove a file from Git, you have to remove it from your tracked files (more accurately, **remove it from your staging area**) and them commit. The **git rm command does that and also removes the file from your working directory**...".

Note that the working directory refered in the citation is not the git Working Directory, it's the directory of the filesystem itself. In other words a 'rm <file>' is used inside the 'git rm <file>'.

When using git rm that modification will only appear in the Index. Unless, before the git rm we delete the file from the filesystem (e.g. rm).

Another restriction that git imposes is that, when a git rm is used, the content of the file to remove must be equal to the current commit.

5.3 commit

6 Conclusions

References

[1] *Git Community Book.*

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