Modelling the Git core system with Alloy

Cláudio Lourenço, Renato Neves
Universidade do Minho

April 12, 2012

1 Abstract

2 Modelling 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.

2.1 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 : one 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 : one Tree
}
```

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

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

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

```
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)

We know that there is an exception. That is, there are cases (defined above) that allow different trees to have same name.

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

2.1.1 Commit structure

Again from [1].

"As you can see,a commit is defined by : ...

parent(s) : **The SHA1 name of some number of commits wich 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 ideia)**". (page 12)


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

}
```

2.2 What we assume for the proper functioning of git

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

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

We can't have orphan Blobs, or else, there is the possibility for garbage data.

```
Blob in Tree.references
```

The same for Trees.

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

This is only appears because we don't need names that don't belong to someone, in the git system.

```
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 there is inconsistency in the git "filesystem".

```
references.(iden & (Tree->Tree)).~references  in iden
```

3 Modelling Git Dynamic Object Model

The static object model, only by itself, is not very usefull. 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.

In order to understand them, the definition of the git Working Directory and git Index is in debt.

3.1 Working Directory and Index

The definition of Working Directory comes as follows: "... simply a **temporary checkout place** where you can modify the files **until your next commit**." [1] (pag 17).

To complement the definition above, git will only track files that existed in the last commit, thus new ones will not appear in this directory.

So, the Working Directory, does nothing more than telling us wich files have been modified since the last commit.

```
sig File{}
sig WorkingD {
        contents : set File
}
```

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). So, the Index, in a certain manner, points to a subset of files that are present in the Working Directory, and is this subset of files that wil be commited.

```
sig Index {
        toCommit : set File
}
```

However there will be a few exceptions!(present in the last parapgrah of the sections git add command and git rm command)

We can see that a commit depends on the contents of git Index, as the last depends on the Working Directory. So it will be important to add those definitions to our model when defining the commit operation .But, a question arises. What is the pratical difference between the Index and the Working Directory? Shouldn't all that we modify in the repository be commited? We will see that next.

3.1.1 The git add command

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**."

When doing a git add of a newly added file, that addition will not appear in the Working Directory, only in the Index.

3.1.2 The git rm command

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 present 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).

4 Conclusions

References

[1] *Git Community Book.*

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