Modelling the Git core system with Alloy

Cláudio Lourenço, Renato Neves
Universidade do Minho

April 15, 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 : 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 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 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
}
```

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.~~ Maybe in the future we can define Blob in Tree.references + Index.stage

```
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". In this case the navigational style is more compact than the relation style.

```
//references.(iden & (Tree->Tree)).~references  in iden
all t:Tree | lone references.t
```


3 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.

In order to understand them, the definition of the git Working Directory and git Index is in debt, but first we give a small the notion of branch.

3.1 Introducing 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
}
```

A branch in git, is a pointer for a commit. We are always working on a certain branch, and all the commits made on that branch are independent from the others branch until we make a merge. A new branch can be created using *git branch branch_name*. The special pointer called Header points to the branch we are working on. [2] (pag 40)

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

We can navigate by different branches using *git checkout branch_ name.*

Trying to understand branches using git:

- if we modify a tracked file, we cannot change branch if we do not add that file to the index and commit it;
- if we create a new file on a branch and then we change branch (without adding the file to the index) that file will be present;
- if we create a new file, add it to the index, change branch, commit, go back to the other branch, that file will not be visible;
- master branch can be deleted;

We can conclude that the index is not connected to a branch. It is something is common to all branches.

3.2 Working Directory

The definition of Working Directory comes as follows: "The Git 'working directory' is the directory that holds the current checkout of the files you are working on.".

The working directory is basically what we see in our file system. When we change to a certain branch, the files from the working directory are replaced by the files from the commit corresponding to that branch. So, a working directory is composed by: last commit + added files - removed files.

3.3 Index

"The Git index is used as a staging area between your working directory and your repository." [1] (pag 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)

```
sig PathName{}
one sig Index{
   stage: PathName -> one Blob
}
```

3.4 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).

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

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

```
abstract sig Type {}
one sig Mod,Add,Del extends Type {}

sig File{
        diff : some Type
}
```

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.

That said, in the Working Directory can only appear modified ou deleted files.

```
fact {

 not Add in (WorkingD.contents).diff
}
```

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 simply refers the files that will be commited.

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

What is the pratical difference between the Index and the Working Directory? The simple answer is that there can be cases when we don't want to commit something, previously changed.

3.4.1 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**."

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.

```
pred add [s,s' : State, f : File] {

    f.diff in (Add+Mod)
    s'.workD = s.workD
    s'.index.toCommit = s.index.toCommit + f
    (s'.index.toCommit).Mod in (s'.workD.contents).Mod
}
```

```
sig State {
  workD : WorkingD,
  index : Index
}
```

3.4.2 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.