Modelling the Git core system with Alloy

Cláudio Lourenço, Renato Neves
Universidade do Minho

April 30, 2012

1 Abstract

2 Introduction

This project aims to build a precise model of how the git works.

The purpose of this model is to check some properties that the model (not) guarantees.

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

After the specification process, 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.

Finnaly some conclusions will be presented to the readers.

2.1 A brief explanation of git

3 Git Static Object Model

In this section we will show the specification in Alloy of the git static object model. It will be mostly based on the textual specification present in [1] and [2].

3.1 The 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)

However Alloy has a mechanism to uniquely identify atoms, so we can use it instead of the sha.

The objects are as defined in [1] (page 7) "...and there are four different types of objects: "blob", "tree", "commit", and "tag". "

```
abstract sig Object {
}
sig Blob extends Object{}
sig Tree extends Object {}
sig Commit extends Object{}
sig Tag extends Object{}
```

“A ”blob“ is used to store file data - it is generally a file [1] (page 8). We can abstract from it.

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

```
sig Tree extends Object {
        contains : Name -> one (Tree+Blob)
}
```

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
}

sig RootCommit extends Commit{}
```

```
fact {
  no ^parent & iden
  no RootCommit.parent
  all c : Commit - RootCommit | some c.parent
}
```

The branches are one of the things that differs git from others version control systems. While they copy a snapshot of the project when making a new branch. In git the same it’s not true. "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 {
        mark : Commit,
```

```
        head : lone Branch
}{
        some Commit <=> one head
}
```

3.2 Implicit specifications

As the book [1] says, a "tree" acts like a directory, so we know that it or it's descendants cannot point to themselves. Also, only sees files and their paths. So it can't deal with empty folders, and because of this, there cannot be trees without sons.

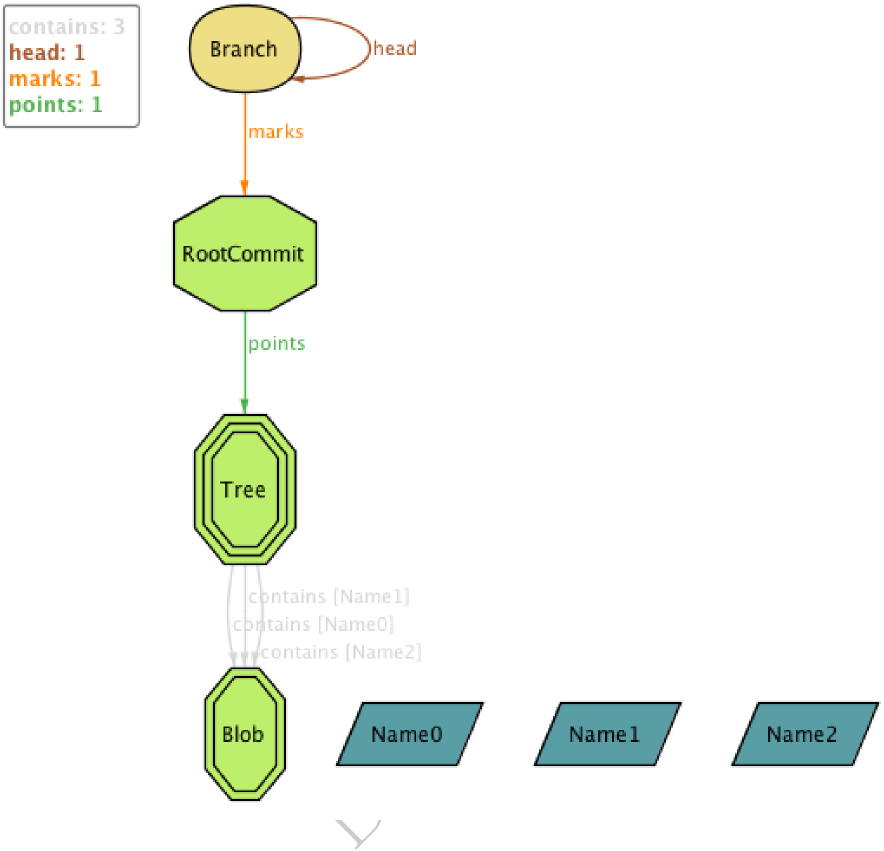
```
let r = {x :Tree, y : Tree+Blob |
                some n : Name | x->n->y in contains} {
        no ^r & iden
        all t : Tree | some t.r
}
```

A tree needs to have at most one parent. Because is part of a file's path.

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

Only with this we already have see some interesting examples, like the following.

Figure 1: A simple commit



4 Index

4.1 The notion of path

A path here is defined as part of a filesystem path. Each file on git has a blob and a path associated. The later and it's parents define a filesystem path.

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

With this we introduce some new invariants. We can't have cicles, and two paths with the same name as in any normal filesystem. Only the leafs of a path can have a file (blob) associated.

```
fact {
         no ^pathparent & iden
         all disj p,p' : Path | p.pathparent = p'.pathparent
         implies p.name != p'.name
```

```
        no Path.pathparent & blob.Blob
}
```

4.2 The notion of Index

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).

```
sig Path {
        ...
        index: lone Path
}{
        no Path.parent & index.Path
}
```

The index relation will define which files are in the index. And we know that only files can be in the index.

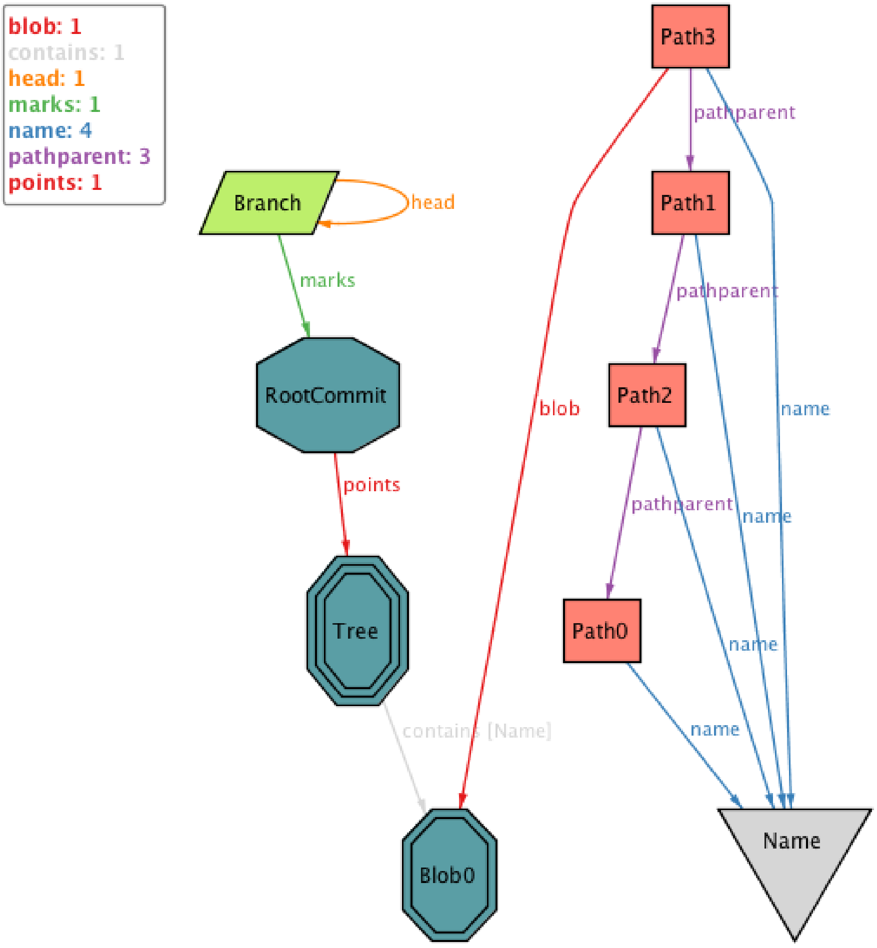
Also : “The index **contains all the information necessary to generate a single (uniquely determined) tree object**” [1] (pag 121).

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.

Now we can see a more complete model of git.

Figure 2: A simple commit and a file



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 some git operations. In order to do that, we need to specify in the model, that it can change over time.

5.1 The notion of state

From now on, all relations with potential to change must have a state associated. So, a relation that has some information before an operation, can have other contents after it.

For example :

```
sig Commit extends Object {
```

```
        ...
        parent : Commit set -> State
}
```

We can see that a commit can have a determined set of parents, but, after some operation it can have another set of parents. Thus, the state will define the relation contents in a certain "instant".

The model, now, needs some adaptations to support the notion of state.

5.2 Git operations

5.2.1 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**."

Thus, we can see the add operation, only as the adition of a file to the index, and nothing else can change between the two states.

```
        //path needs to be a file
        no pathparent.p

        // a path is added to the index
        objects.s' = objects.s + p.blob
        index.s' = index.s + p

        //all other things are kept
        parent.s' = parent.s
        marks.s' = marks.s
        branches.s' = branches.s
        head.s' = head.s
```

5.2.2 rm

The usefullness of rm can be seen in the following [2] (page 21) "To remove a file from Git, you have to remove it from your tracked files (more accurately, **remove it from your staging area with git rm**) and then commit. An important restriction that git imposes when using git rm is used, it that the content of the file to remove must be equal to the last commit.

So, we can see the rm operation as a removal operation of a file that is in the index, and it cannot have changed since the last commit.

5.2.3 commit

The commit operation is simply explained. "A commit is usually created by git commit, which creates a **commit whose parent is normally the**

current HEAD, and whose **tree is taken from the content currently stored in the index**" [1] (page 13). But hard to model. The culprit is the transformation of the index into a tree that is pointed by the new commit.

5.2.4 push

5.2.5 pull

6 Conclusions

References

[1] *Git Community Book*.

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