Modelling the Git core system with Alloy

Cláudio Lourenço, Renato Neves
Universidade do Minho

April 7, 2012

1 Abstract

2 Modelling Git 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
}
fact {
  no ^parent & iden
}
```

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 "filesystem".

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

3 Conclusions

References

[1] *Git Community Book*.