

Universidade do Minho  
Departamento de Informática  
Mestrado em Engenharia Informática

Metódos Formais em Engenharia de Software  
Group F  
Cohesive Project: CSAIL/MIT Proposal

## **ValidAlloy**

**A tool for validating a git alloy specification using test-case generation**

Authors:

José Pinheiro	Tiago Guimarães
<code>pg23208@di.uminho.pt</code>	<code>pg22832@di.uminho.pt</code>

Supervisors:

Alcino Cunha	Enusuk Kang
<code>alcino@di.uminho.pt</code>	<code>eskang@csail.mit.edu</code>

August 11, 2013

## **Abstract**

In the world of software, formal models are continually getting more important, but a lot of software is being built without a formal model to back it up. In this document we will talk about our approach to validate models of existing software, model validation through test case generation, and our tool that implements it. We used git as our case study, mainly because of its popularity and its bad documentation, and worked to get an accurate model of it, and tried to find git bugs while doing it.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Motivation . . . . .	3
1.1.1	Git . . . . .	3
1.2	Solution . . . . .	4
<b>2</b>	<b>ValidAlloy</b>	<b>5</b>
2.1	Case Study: Git . . . . .	5
2.2	ValidAlloy Features . . . . .	5
2.3	Implementation Details . . . . .	6
2.4	How it works . . . . .	7
2.4.1	Alloy API . . . . .	8
2.4.2	Git plumbing commands . . . . .	8
2.4.3	Unix Diff . . . . .	9
2.4.4	Configuration File . . . . .	9
2.4.5	Logs . . . . .	9
2.5	Problems encountered . . . . .	10
<b>3</b>	<b>Results</b>	<b>11</b>
3.1	Git add . . . . .	11
3.2	Git branch . . . . .	12
3.3	Git rm . . . . .	12
<b>4</b>	<b>Conclusions</b>	<b>15</b>
<b>5</b>	<b>Possible future work</b>	<b>16</b>
<b>6</b>	<b>Bibliography</b>	<b>17</b>
	<b>Appendix</b>	<b>18</b>
<b>A</b>	<b>Config File Grammar</b>	<b>18</b>

# List of Figures

1	Model validation through test case generation . . . . .	4
2	Pre and Pos state . . . . .	6
3	A run of validalloy . . . . .	6
4	How it works . . . . .	7
5	A run of validalloy with bad modelled add, showing folders with differences . . .	8
6	Example diff file . . . . .	9

# Chapter 1

## Introduction

### 1.1 Motivation

#### 1.1.1 Git

Git[2] is a famous distributed revision control and source code management system with an emphasis on speed.

One of it's main features is its fast branching and merging. That is because branches in git are very lightweight, a branch in git is only a reference to a single commit.

It has complete history and full revision tracking capabilities;

Its commands are divided into two types:

- Porcelain commands: The type of commands that the normal user should be using most of the time, very high-level;
- Plumbing commands: The low-level commands, commonly used for some tweaks.

But git is not a perfect world, as we can see by some random internet quotes:

“What a pity that it's so hard to learn, has such an unpleasant command line interface, and treats its users with such utter contempt.”

*10 things I hate about Git*

“The man pages are one almighty “f\*ck you”. They describe the commands from the perspective of a computer scientist, not a user.”

*10 things I hate about Git*

“(…) there are lots of ”cheat sheets” floating around about how to use git tools - I think this is evidence that [...] many people have to write ”GIT for mortals” pages...”

*Perl Mailing List*

Git is a complex tool, it is not simple to use and its manual is obscure. Its commands are multi-purpose and sometimes they won't work as you expect them to, or even do something that isn't even mentioned on the manual.

For those reasons, we purpose the creation of a formal git model to alleviate some of these problems. With an actual git model it would be easier to predict the system behaviour, as well as verify some of the system's properties. A model would also help with the finding of bugs, because you could say that an operations didn't run as expected by the formal model. The main problem would be how accurate the model would be. It needs to be validated!

But validating a formal model of an existing system is not a easy task. Deriving the model from the code is a option, just as well as trial and error, our approach is test case generation.

## 1.2 Solution

Model validation through test case generation is an approach where we use a model to generate test runs with the input of a modelled operation and its expected result, which is then compared with a run of the actual system operation for the same input, if they do not match then the model is not accurate or the system is bugged.

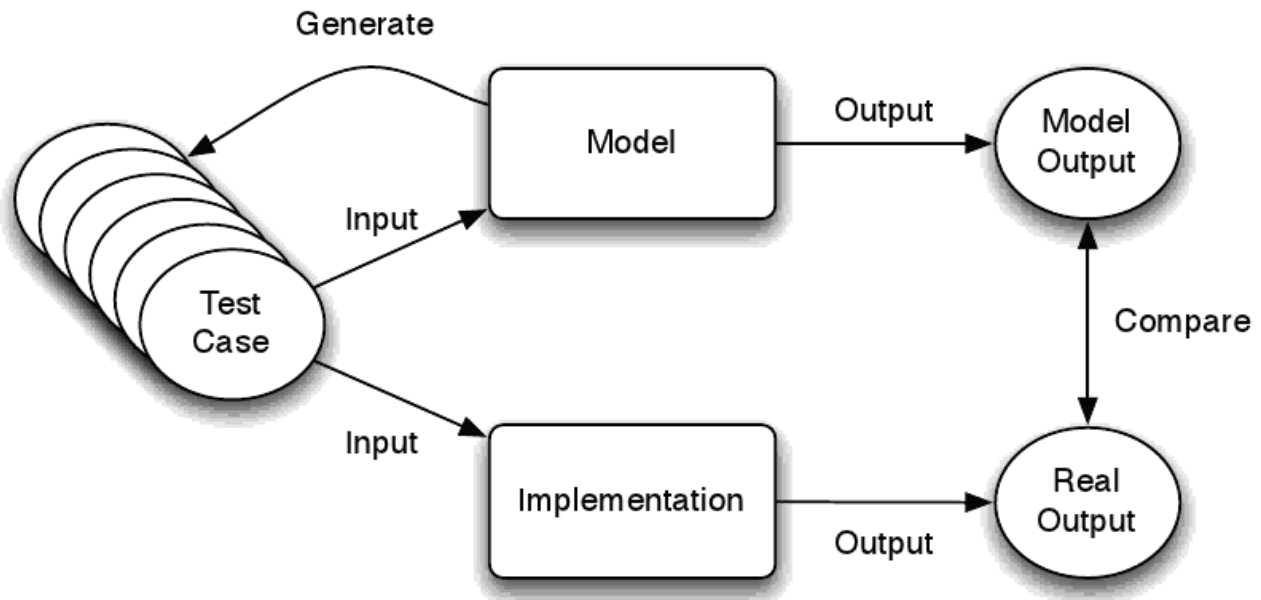


Figure 1: Model validation through test case generation

We are using a Alloy[1] model to represent git, because Alloy provides us the features needed for the use of model validation through test case generation, we can use predicates model the operations and make use of its solving to generate the test cases. It also has a Java API which we used to automate the process.

This approach makes the validation highly automatic, but we must have caution about our pre-conditions strength. Pre-conditions too strong won't fully test the operation and it is hard to recognize when a pre-condition is too strong, so we might miss some bugs.

# Chapter 2

## ValidAlloy

### 2.1 Case Study: Git

We started with a legacy Git model, and aimed to validate and improve it. Our goal was to create a testbench that implements model validation through test case generation applied to this git model so that we could test git itself, and improve the model. Our testbench uses the Git Alloy model and the Alloy API to implement model validation through test case generation

### 2.2 ValidAlloy Features

Our Testbench is capable of:

- Creating a git repo from the modelled instance;
- Associating any git modelled command with actual command;
- Comparing the modelled version with the git command;
- Generic configuration file;
- Handles git errors.

It is configured through a configuration file that lets you make those associations and other important configurations. The tool also leaves the Alloy instance xml in the git repo that was generated from it, and saves extensive logs, so that the user is capable of fully analyse the results of the testbench. It distinguishes between an error occurring in an operation and an operation returning different results than those expected. This is important to know what change in the model (errors means the current pre-conditions are too weak, and different results means wrong post-conditions), unless it is an actual git bug.

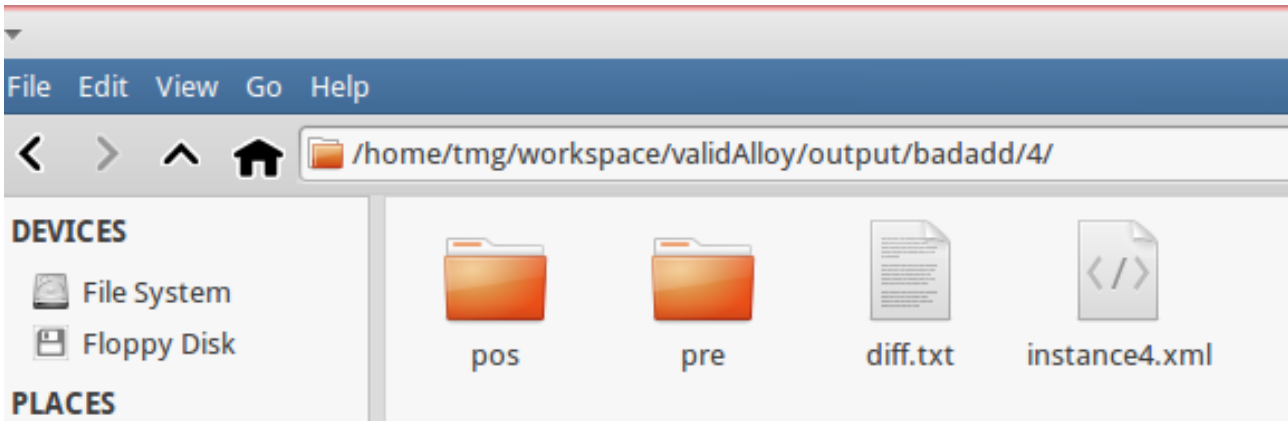


Figure 2: Pre and Pos state

## 2.3 Implementation Details

We decided to implement ValidAlloy with Java7 mainly because we are using an Alloy model, and Alloy has a Java API that we find very helpful to make the process automatic. Our tool depends on the Git Alloy model, this implies that if you change some parts of the model, you might have to change the testbench. (The structural part of the model (signs and facts) must remain the same for the testbench to work but predicates can be changed without need to change the testbench) It only supports predicates written on that model. It requires as input a configuration file.

```

Terminal - tmg@ubuntu: ~/workspace/validAlloy
File Edit View Terminal Tabs Help
tmg@ubuntu:~/workspace/validAlloy$ ./validAlloy.sh src/example.cfg
=====
                          ValidAlloy                          =====
=====
                          Parsing + Typechecking               =====
Config File                : src/example.cfg
Number of runs              : 500
Number of commands         : 1
===== Getting solutions from git_dynamic.als =====
Variables                  : {#p=path}
Predicate                  : add
Arguments                  : [p]
Scope                     : for 4 but exactly 2 State
Commands                   : rm
Options                    : [#p]
Expected Errors            : null
=====
                          Running Instances                    =====
Instance                   : 499
=====
                          Command terminated                   =====
                          ValidAlloy terminated                 =====
tmg@ubuntu:~/workspace/validAlloy$ █

```

Figure 3: A run of validalloy



## 2.4 How it works

A configuration file is passed as input to the testbench, in this file should be the associations between the model predicates and the actual git commands, it should also be present the number of test runs. Through the Alloy API, the testbench generates the model instances, for each instance it is created two different folders, one with the a git repository to be used to run the command, the other contains the git repository that is expected as result. We use Git plumbing commands to generate the git objects. After everything need is created, the git command runs, if the commands reports an error, it is saved in an error file, otherwise the testbench will compare both folders and create a file with all differences listed if they exist. By default our testbench deletes successfully runs(runs with no errors, and no differences between the expected result and the actual result).

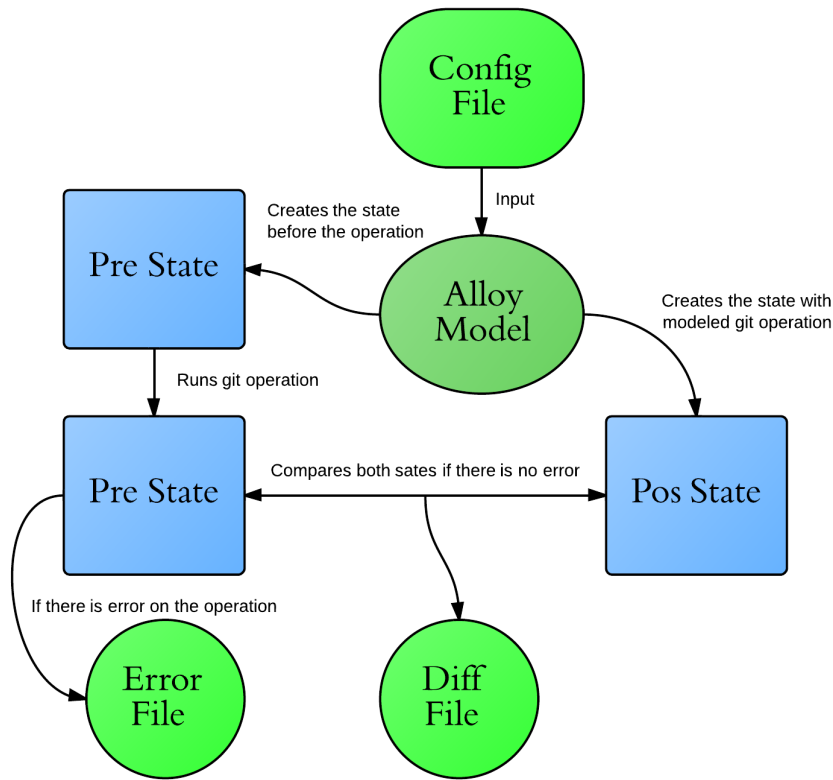


Figure 4: How it works

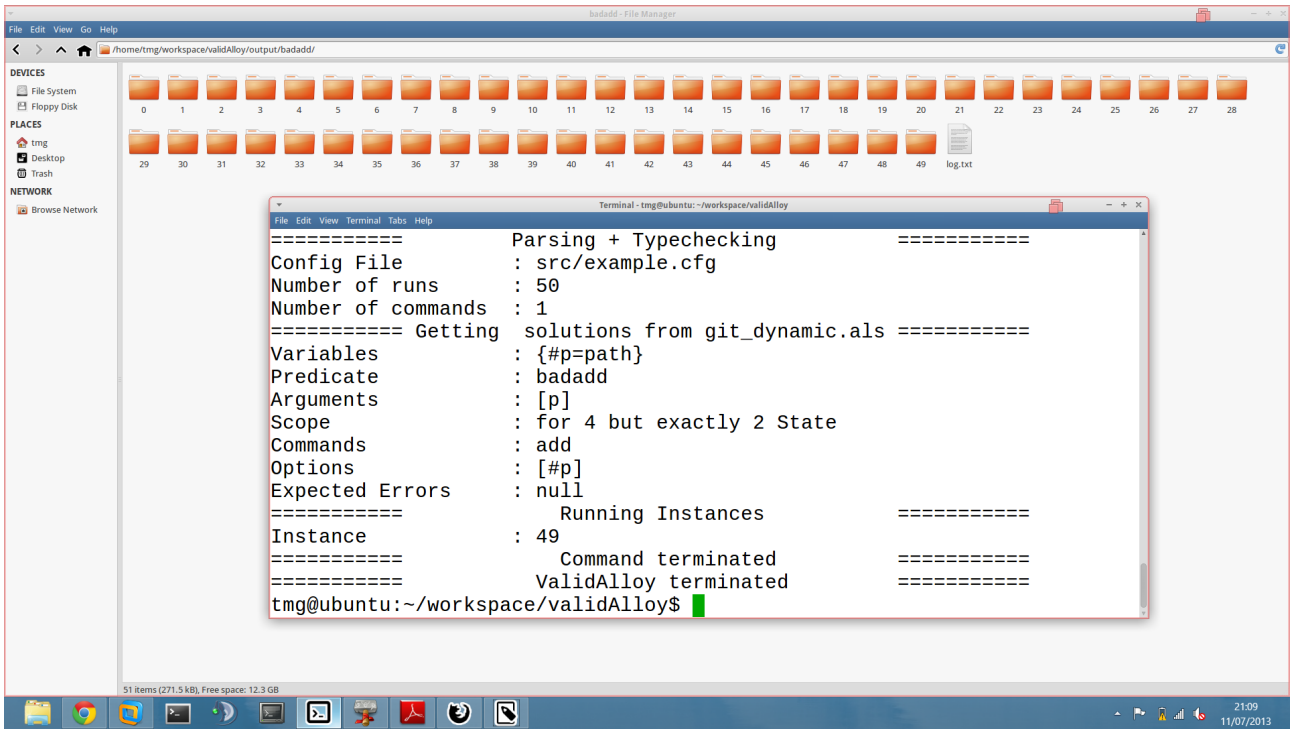


Figure 5: A run of validalloy with bad modelled add, showing folders with differences

### 2.4.1 Alloy API

Our testbench relies a lot on the Alloy API, every alloy run is executed through it, and it is used to iterate the result models so that we could create the git repositories. We use the API to parse the model from the file, then we use the the method `CompUtil.parseOneExpression_fromString` to create the expressions that we needed to use, but we still had to build some expressions manually, this was because we couldn't get the instance atoms with this method, so after we evaluate we had to look up on an previously created map, to get the corresponding expression to an atom.

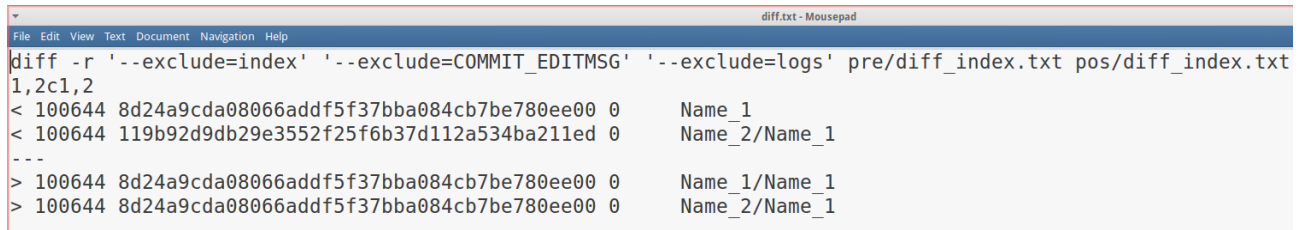
### 2.4.2 Git plumbing commands

To create the git objects, we used several git low level[3][4] commands those where:

- `git hash-object` : creates the blobs;
- `git mktree` : creates the git filesystem tree;
- `git commit-tree` : creates the git commit tree;
- `git symbolic-ref HEAD` : sets the HEAD to a ref;
- `git update-ref` : creates the refs;
- `git update-index` : creates the index file.

### 2.4.3 Unix Diff

The Unix diff is the key to our repository comparison, not only because it precisely reports differences in files, but also because if you use the `-r` flag it will compare everything that exist on the given folders. Because we relay on diff, our testbench will only work on Unix systems.

A screenshot of a text editor window titled 'diff.txt - Mousepad'. The window shows the output of a 'diff -r' command comparing two directories. The output includes file names, line numbers, and SHA-1 hashes. The files shown are 'Name\_1' and 'Name\_2/Name\_1'. The diff shows that 'Name\_1' is identical in both directories, while 'Name\_2/Name\_1' is also identical. The output is as follows:

```
diff -r '--exclude=index' '--exclude=COMMIT_EDITMSG' '--exclude=logs' pre/diff_index.txt pos/diff_index.txt
1,2c1,2
< 100644 8d24a9cda08066addf5f37bba084cb7be780ee00 0      Name_1
< 100644 119b92d9db29e3552f25f6b37d112a534ba211ed 0      Name_2/Name_1
---
> 100644 8d24a9cda08066addf5f37bba084cb7be780ee00 0      Name_1/Name_1
> 100644 8d24a9cda08066addf5f37bba084cb7be780ee00 0      Name_2/Name_1
```

Figure 6: Example diff file

### 2.4.4 Configuration File

To generate the parser to parse our config file, we used Another Tool For Language Recognition [6].

Our language is very simple, you define a list of commands, where each command has a a list of variables, the predicate that must be run, its scope, the associated git command and its parameters and a list of expected errors, finally you have the number of iterations that validalloy must run. For a formal description of the grammar go to appendix A. Bellow follows a config file example:

```
#p : path
pred rmNOP[#p]
scope for 4 but exactly 2 State
cmd git rm #p
errors ("has changes staged in the index")

runs 100
```

2.1: Config file example

### 2.4.5 Logs

Early during the development of our testbench, we realized that we needed to record our operations and the output of the the git commands, we had to implement logs.

We were concerned that due to the big amount of io operations that our testbench does, adding a powerful but complex log system(like log4j) would stress the testbench even more, so we choose java tinylog, mainly because it is fast, has a small memory footprint and supports writing threads.

Our logs support different levels:

- TRACE: Result of a successfully operation;
- ERROR: When a operation returns an error;
- INFO: Normal behaviour of the testbench.

When a predicate is run, all the logs of the testbench are written to that predicate output folder in a file named logs.txt.

## 2.5 Problems encountered

During the development of ValidAlloy we had some problems related to the approach we were using that we had to resolve:

- Timestamps in commit objects: We resolved this problem, by fixing the same time-stamp in all commit objects;
- Index is binary file and difficult to compare: Instead of comparing the index file, we ignore it, and compare the output from `git ls-files -staged` instead, this command already gives us everything we modelled about the index.
- Errors that are expected while running the git operations: We added to the config file a list of expected errors;
- Git doesn't report some "errors" as errors, but to standard input: We could not find a solution to this problem, but you can manually check the logs, and see if the output from the operation was actually an error.

# Chapter 3

## Results

The model we used is a legacy one, and it describes the file system and the git objects(blob, tree, commit, tag), index and heads(including the HEAD). We modelled the git add,branch and remove operations and tested them with our testbench, the results follows.

### 3.1 Git add

Git add is used to track files and fix merge conflicts. Or Alloy model is too abstract, we don't have on our modelled index, the information need for merge and its conflicts, so we are only testing the tracking part of git add. With this in mind, git add becomes a rather simple operation that just puts the files on the index if they are not already there. With our current Testbench we couldn't find any difference.

```
pred add [s,s' : State ,p : Path] {  
  s != s'  
  p in (node.s).path  
  path.p in File  
  object.s' = object.s + (path.p).blob  
  index.s' = index.s ++ p->(path.p).blob  
  ref.s' = ref.s  
  head.s' = head.s  
  HEAD.s' = HEAD.s  
  node.s' = node.s  
}
```

3.1: Predicate add

## 3.2 Git branch

Git branch's speed is one of git's selling points. Its speed comes from its simplicity, branching in git is just creating another pointer to the actual commit. So its post-conditions are not that complicated, and its pre-conditions are also simple: you cannot branch before your first commit, and you can't give the new branch the same name as an existing one.

Our approach could not find any problems with git branch. And we also believe that git branch is fine, specially because of its simplicity.

```
pred branchPC[s:State, b:Ref]
{
  some (HEAD.s).head.s
  b not in (head.s).univ
}

pred gitBranch[s,s':State, n:Ref]{
  branchPC[s,n]

  head.s' = head.s + (n->((HEAD.s).head.s))

  HEAD.s' = HEAD.s
  object.s' = object.s
  node.s' = node.s
  index.s' = index.s
}
```

3.2: Predicate branch

## 3.3 Git rm

Git remote, according to the man pages, should remove the entry of a file from the index and, if it exists on the file system, deletes it. But with our testbench we found out that if a file is the last one in a directory, when you remove it with git remove, it will also delete the containing directory and every other ancestor directory that also becomes empty, this is unmentioned on the man pages and on the Git Pro Manual.

Git manual states that the files being removed have to be identical to the tip of the branch and no updates to their contents can be staged in the index. But we noted that sometimes git remove would run even if one of these conditions didn't hold, so we decided to test when some of the pre-conditions don't hold as well. This was the motivation for the error expecting part of the testbench, so that we wouldn't get our results spammed with the expected result of a negated pre-condition.

With this our approach to this operation became:

- Run git rm with both pre-conditions holding true;
- Negate one pre-condition alternatively and test, while the other held true;
- Test with both pre-conditions being false.

```

pred rmConditionA [s:State ,p:Path]{
  (((HEAD.s).head.s).path.s).p & Blob = (path.p & node.s).blob
}

pred rmConditionB [s:State ,p:Path]{
  p.index.s in (((HEAD.s).head.s).path.s).p
}

pred rmAB[s,s':State ,p:Path]{
  rmConditionA [s,p]
  rmConditionB [s,p]
  rmBehaviour [s,s',p]
}

pred rmA[s,s':State ,p:Path]{
  rmConditionA [s,p]
  not rmConditionB [s,p]
  rmBehaviour [s,s',p]
}

pred rmB[s,s':State ,p:Path]{
  not rmConditionA [s,p]
  rmConditionB [s,p]
  rmBehaviour [s,s',p]
}

pred rmNOP[s,s':State ,p:Path]{
  not rmConditionA [s,p]
  not rmConditionB [s,p]
  rmBehaviour [s,s',p]
}

pred rmBehaviour [s,s':State ,p:Path]{
  p in (index.s).univ
  some path.p & node.s ==> some path.(p.siblings) & node.s
  path.p in File & node.s
  index.s' = index.s - p->Blob
  node.s' = node.s - path.p
  object.s' = object.s
  head.s' = head.s
  HEAD.s' = HEAD.s
}

```

### 3.3: Predicate rm

With this approach we where able to identify a weird error that could be replicated with this trace:

```

$ git init
$ mkdir D
$ echo "Hi" > D/F
$ git add D/F
$ rm -r D
$ echo "Hey" > D

```

```
$ git rm D/F
warning : 'D/F': Not a directory
rm 'D/F'
fatal: git rm: 'D/F': Not a directory
```

If we instead did:

```
$ git init
$ mkdir D
$ echo "Hi" > D/F
$ git add D/F
$ rm -r D
$ echo "Hey" > F
$ git rm D/F
```

This works as expected.

We asked at [git@vger.kernel.org](mailto:git@vger.kernel.org)<sup>[5]</sup>, if this was the correct behaviour from `git rm`, in the first reply, they suggested a patch for it. It should also be noted that because of our post they also identified a similar problem but with symbolic links instead of files, that could cause some trouble.



# Chapter 4

## Conclusions

Git is not nearly perfect, its documentation is incomplete, and the functionality of some operations makes it hard to understand what they actually do. We are satisfied with model validation through test case generation, on the simpler commands it showed not apparent errors, and on the complex one it provided us some really weird behaviour, as it should be expected. We believed that we used this well to validate our model, and to create and run all those tests, and we found an actual bug in git. But there are still some problems: model depth limits how accurate the tests are. And of course, you must be good with modelling to use it well. You can check out our work at <https://github.com/eskang/validAlloy>.

# Chapter 5

## Possible future work

After this, we believe that there is some work can still to be done. The model needs to be refined, and there are some operations that are yet to be done( merge and commit for example). The testbench needs an update on the documentation(we are working on it). We also believe that it should be possible to create a framework to make testbenches, like this one that use alloy and the test case generation approach. We believe to a big part of the work could be saved.

# Chapter 6

## Bibliography

- [1] Official alloy website, <http://alloy.mit.edu/>.
- [2] Official git website, <http://git-scm.com/>.
- [3] Scott Chacon, *Pro Git*. Apress, Berkely, CA, USA 2009.
- [4] Git man pages, <https://www.kernel.org/pub/software/scm/git/docs/>.
- [5] Discussion of the git rm bug, <http://git.661346.n2.nabble.com/Behavior-of-git-rm-td7581485.html>.
- [6] Official antlr website, [www.antlr.org](http://www.antlr.org).

# Appendix A

## Config File Grammar

Here follows a formal description of the grammar that recognizes the config file language. It is in extendBNF notation.

```
cfg
: command ( ';' b=command)* runs
;

command
: vars? 'pred' pred 'scope' sp 'cmd' cmdgit errors?
;

errors
: 'errors' '(' list_errors ')'
;

list_errors
: error (',' error)*
;

error
: STRING
;

runs
: 'runs' INT
;

vars
: var (var)*
;

pred
: name arguments
;

arguments
```

```

    : '[' args? ']'
    ;

args
  : arg (',' arg)*
  ;

sp
  : 'for' INT sigs
  | 'for' INT 'but exactly' sigs
  ;

sigs
  : sig (',' sig)*
  ;

sig
  : INT ID
  ;

var
  : arg ':' type
  ;

type
  : ID
  ;

cmdgit
  : 'git' name opts
  ;

opts
  : opt (opt)*
  ;

opt
  : ID
  | arg
  | STRING
  ;

name
  : ID
  ;

arg
  : '#' ID

```

```

;

ID : ('a'..'z'|'A'..'Z'|'_'|'-' ) ('a'..'z'|'A'..'Z'|'0'..'9'|'_'|'-' ) *
;

INT : '0'..'9'+
;

WS : ( ' '
      | '\t'
      | '\r'
      | '\n'
      ) {$channel=HIDDEN;}
;

STRING
: '"' ( ESC_SEQ | ~('\\"'|'"') ) * '"'
;

CHAR: '\'' ( ESC_SEQ | ~('\''|'\\') ) '\''
;

fragment
HEX_DIGIT : ('0'..'9'|'a'..'f'|'A'..'F') ;

fragment
ESC_SEQ
: '\\' ('b'|'t'|'n'|'f'|'r'|'"'|'\''|'\\')
| UNICODE_ESC
| OCTAL_ESC
;

fragment
OCTAL_ESC
: '\\' ('0'..'3') ('0'..'7') ('0'..'7')
| '\\' ('0'..'7') ('0'..'7')
| '\\' ('0'..'7')
;

fragment
UNICODE_ESC
: '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
;

```