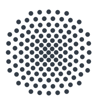
Software Lab
Institute of Software Engineering
University of Stuttgart
Universitätsstraße 38, 70569 Stuttgart

Master Thesis

Metamorphic Testing of Version Control Systems

Maximilian Reichel

Course of study:	Computer Science
Examiner:	Prof. Dr. Michael Pradel
Supervisor:	Prof. Dr. Michael Pradel
	Dr. Maria Christakis
Started:	November 15, 2021
Completed:	May 15, 2022


University of Stuttgart
Germany

Abstract

Currently, no approach exists to automatically and systematically test the implementation of version control systems (VCS). VCS are widely used in the software development industry, where a lot of people rely on these systems to perform as specified and not to lose any data. However, problems with the implementations of such systems are found frequently which causes risks and inconveniences to the users. The *oracle problem* renders traditional testing impractical.

In this thesis, we present an approach to automatically test the implementation of VCS using metamorphic testing. Metamorphic testing is an approach to mitigate the *oracle problem* by splitting it up into smaller parts. Instead of knowing the expected behavior of a program for a specific input, this approach uses the knowledge of how a certain change of the program input will be reflected in its output. We develop such transformations of inputs of VCS and the corresponding relations of the outputs. Since this requires initial inputs to work with, we also develop a random input test generator. We combine these two components into an automated testing tool. As inputs, we use bash-scripts containing commands to interact with the VCS and the file system. As outputs, we use the files in the working directory after executing the input scripts. Additionally, we develop a test minimizer to reduce the number of commands in the inputs when a relation gets violated. This eases the manual analysis later. Since *git* is complex and currently the most popular VCS, it makes a good first target to test our approach.

We find five real-world bugs in the implementation of *git* and archive a precision of 90.79%. Additionally, we are able to minimize the test inputs to $\frac{1}{5}$ of the original size on average.

Zusammenfassung

Derzeit gibt es keinen Ansatz, um Versionskontrollsysteme (VCS) systematisch und automatisiert zu testen. VCS sind weitverbreitet in der Softwareentwicklung, wo sich viele Leute darauf verlassen, dass diese Systeme wie beschrieben funktionieren und nicht zu Datenverlust führen. Dennoch werden immer wieder Fehler in den Implementierungen von VCS entdeckt, was Risiken und Unannehmlichkeiten für die Nutzer bedeutet. Aufgrund des *Orakel-Problems* ist konventionelles Testen dieser Systeme nicht praktikabel.

In dieser Thesis stellen wir einen Ansatz zum Testen der Implementierungen der VCS mithilfe von metamorphischem Testen vor. Metamorphisches Testen vereinfacht das *Orakel-Problem*, indem das Problem in kleinere Teilprobleme aufgeteilt wird. Anstatt das erwartete Verhalten des Programms auf eine bestimmte Eingabe kennen zu müssen, wird bei diesem Ansatz das Wissen darüber verwendet, wie sich eine bestimmte Änderung der Eingabe auf die Ausgabe des Programms auswirkt. Wir entwickeln solche Transformationen für die Eingaben von VCS, sowie die erwarteten Beziehungen der Ausgaben. Da wir initiale Eingaben benötigen, um diese transformieren zu können, entwickeln wir auch einen Zufallseingaben-Generator. Ebenso entwickeln wir auch ein Verfahren, welches die Test-Eingaben minimiert, wenn eine Relation verletzt wird, um die manuelle Auswertung hinterher zu vereinfachen. Wir verwenden *git*, um unseren Ansatz zu testen, da es komplex und das derzeit meist genutzte VCS ist.

Wir finden fünf Bugs in der Implementierung von *git* und erreichen eine Präzision von 90,70%. Zusätzlich haben wir die Test-Eingaben im Durchschnitt auf $\frac{1}{5}$ der ursprünglichen Größe reduziert.

Contents

1 Introduction **1**
1.1 Goals . . . 2
1.2 Example . . . 2
1.3 Thesis Structure . . . 2

2 Background **3**
2.1 Metamorphic Testing . . . 3
2.2 Version Control Systems . . . 4
2.3 *Git* . . . 4

3 Approach **9**
3.1 Overview . . . 9
3.2 Script Generation . . . 9
3.2.1 Command Model . . . 10
3.2.2 Script Generator . . . 14
3.3 Relations . . . 17
3.3.1 Exit Relation . . . 19
3.3.2 Source Independent Checkout Relation . . . 19
3.4 Test Execution . . . 20
3.5 Transformations . . . 21
3.5.1 Command Transformations . . . 21
3.5.2 Command Insertion . . . 25
3.5.3 Transformation Selection . . . 28
3.6 Test Minimization . . . 28

4 Implementation **31**
4.1 Architecture . . . 31
4.2 Script and Command Model . . . 32
4.3 Transformation . . . 34

5 Evaluation **35**
5.1 Approach Effectiveness . . . 35
5.1.1 File Move Bug . . . 35

5.1.2 Commit ID Collision Bug 37
5.1.3 Checkout No-Op Bug 38
5.1.4 Stash Push Fail Bug 40
5.1.5 Pull Fails After Commit `--dry-run` Bug 41
5.1.6 Result 42
5.2 Approach Efficiency 42
5.3 Approach Precision 44
5.4 Test Minimization Effectiveness 44
5.5 Transformation Effectiveness 45
5.5.1 Result 46

6 Related Work **49**

7 Future Work **51**
7.1 Script Parsing 51

8 Conclusion **55**

Bibliography **57**

1 Introduction

Developing software without using some kind of version control system (VCS) is risky, therefore practically every serious software development effort uses a VCS today. Within such projects, the developer's code is the most important asset, so it is critical to keep track of it; this can be accomplished by a VCS. It allows a developer to turn back the time to fix mistakes and review past changes. Additionally, it helps the team with the challenges of concurrent and distributed software development.

In the early years of the Linux kernel development, changes were passed around as patches or archive files [15, p. 12]. At this time, no software existed that could meet all the requirements of the developers. In 2005, this provoked them to start the development of today's most popular version control system: *git*. Since bugs in such systems could cause serious problems for a lot of people, it is important that these systems work properly. *Git* supports more than 140 commands and its source code consists of more than two-hundred-thousand lines of code. Such complexity increases the risk of bugs. The fact, that this risk is real is shown by a recent bug in *git*. In a specific scenario, the bug could cause files to disappear and changes to get lost[1]. Another recent bug with the `git mv` command for moving files might cause an unintended state and some subsequent *git* commands to fail[2]. This shows the need for testing these systems, which is not trivial in general. Currently, there is no way to systematically and automatically test the implementation of VCS. This encouraged us to tackle this issue.

When testing software, we need to know the expected behavior of each test case, which is known as the *oracle problem* [1]. For each test case, we need an oracle that tells us whether the observed behavior is correct or not. *Metamorphic testing* [4][11] is a technique to address this problem by splitting it up into smaller parts that are easier to solve. In metamorphic testing, we do not need to know the correct output for a specific test input. Instead, we use the knowledge of how a change of the test input is reflected in the test output. We use such relations to derive a so-called follow-up test case from the original test case and check if the difference in the program output of both test cases meets the predictions according to the applied relation. We will use this technique in combination with an automated test case generator for the automatic testing of version control systems. Due to its popularity and complexity, *git* is the ideal target for the first implementation and the evaluation of our approach.

[1] `https://github.com/git/git/pull/1039`

[2] `https://github.com/gitgitgadget/git/releases/tag/pr-1187%2Fvdye%2Freset%2Fmerge-inconsistency-v2`

1.1 Goals

The goal of this project is to design, implement, and evaluate an automated metamorphic testing tool targeted at version control systems. Additionally, we want to find real-world bugs in version control systems to prove the effectiveness of the approach. To do so,

- we design metamorphic relations and transformations aimed at version control systems,
- we design an automated input test case generator aimed at version control systems,
- we implement the relations, the transformations, and the generator within an automatic testing tool to test the implementation of the VCS *git*,
- we test this tool on the latest version of *git*, and
- we report the true positive violations to the developers and evaluate the approach.

1.2 Example

In the introduction, we have already mentioned the recent bug related to the `git mv` command. We will describe a scenario in which our approach can detect this bug. At first, the random input generator generates a sequence of commands. These commands interact with *git* and the file system, for example by creating some files and moving them by the `git mv` command as a real user might do. Second, the approach applies transformations to this command sequence and observes the behavior during the execution of the original and the transformed sequence. Afterward, it checks if the observed behavior matches our expectations according to the used relation. In this example, a `sleep 1` command is inserted at a particular point within the sequence. This command pauses the execution of the sequence for one second. We expect that the working directory contains the same files after the execution of both sequences. Since this bug is timing-related, it can be affected by the inserted `sleep` command. After the executions, the approach observes a difference in a file in the working directory. This observation contradicts our expected relation between these files, so we have found a potential bug. The approach minimizes the sequences to only contain these commands necessary to reproduce the difference of this file.

1.3 Thesis Structure

This thesis is structured as following: Chapter 2 gives an overview of the main topics required to understand the concepts described in this thesis. Our approach for an automatic metamorphic testing tool aimed at version control systems is described in Chapter 3. In Chapter 4, we explain the used technologies as well as interesting details of our implementation of the supposed approach. Chapter 5 is dealing with the evaluation of our approach and the explanation of bugs found in *git*. Afterward, we discuss some previous work that is related to our work in Chapter 6. In Chapter 7, we discuss current limitations and possible improvements of the approach. Finally, we give a short conclusion in Chapter 8.

2 Background

This chapter gives an overview of the main topics required to understand the concepts described in this thesis.

2.1 Metamorphic Testing

The concept of metamorphic testing was introduced with a demo paper by Chen et al. in 1998 [4]. An intuitive approach to test software would be to execute a program and compare the actual behavior against the desired behavior. But how do we know what the *desired* behavior should look like? The *test oracle problem* deals with the question of how we can decide whether the observed behavior is correct. It is still a current research topic [18][8][1]. Metamorphic testing is one technique to address this problem. In metamorphic testing, we do not need to know what the expected behavior is. For this technique, we will use knowledge about the relation between the test input and the behavior of the program. For example, we can start with some test input and observe its behavior. Then we can use our knowledge to transform this input in such a way, so that we can predict the difference between the behavior of this transformed test input and the original test input. If we execute the test with the new input and the observed behavior does not match our prediction, there must be either a fault in the program or our relation is wrong.

Let us assume we want to use this approach to test the calculation of the sinus function $sin(x)$ for some value for x. We can use the fact that we can add integer multiples of 2π to x without altering the result of the calculation. So, a possible transformation of some input x would be to add an integer multiple of 2π to it. And the relation results in the fact, that this transformation should lead to the same output as with x as an input. So, $sin(x) = sin(x + 2 \cdot i \cdot \pi)$ must hold true for all natural numbers $i \in \mathbb{N}$. If we observe a case where this relation does not hold true, the calculation is not correct. This would enable us to find a bug within the sinus calculation even without knowing a single concrete result of the sinus function for reference.

Formally, metamorphic testing can be defined as follows: Let P a program under test. Let t be a transformation for the inputs of P. Let A and $B := t(A)$ be two inputs for P where B is the transformed input A. Let r be a specific relation between the two outputs of P when using A and B as input. If we have found such an input pair for which the corresponding outputs of P are related as described by r, we have found a potential bug in P.

2.2 Version Control Systems

Version control systems (VCS) are used to track all changes that were applied to one or more documents. It enables the user to revert the content of the document to any state that was checked in at some point in time. This is beneficial for working on large and complex documents. When a modification introduces errors that cannot be fixed easily, the document can be reverted to some state before the error was introduced, even if the error was detected a long time after it was introduced. Distributed VCS can also be used to enable independent collaborative work on documents in some instances. They are considered as a important component of the software life cycle in modern software development [6]. VCS enable a lot of people to work concurrently on different parts of the same project. Additionally, they enable the identification of the author for each change.

2.3 *Git*

Git is a popular decentralized VCS. Into 2005, its development started as an open-source project by Linus Torvalds. Since then, it rapidly becomes one of the most popular VCS [9]. *Git* enables distributed work and non-linear workflows. *Git* itself is a command line tool, but graphical user interfaces also exist which can be used to interact with *git*.

Git stores all its information in so-called repositories. Since *git* is a distributed version control system, the information in different repositories can be synchronized. The main difference between distributed version control systems compared to centralized ones is that each user has a local copy of the whole repository and can therefore work completely independent. In *git*, a history of changes is called a branch and one repository can contain multiple branches. Each branch has a name. The name of the default branch is typically "master" or "main". The history is made up of commits. A commit is composed of the changes between itself and its parent commit. These changes are applied on top of each other to recreate the desired state of the files at the current commit. The history forms a directed acyclic graph (DAG) where the commits are the nodes and the parent's relation between the commits are the edges. Each outgoing edge of a commit is referring to one or more parents. Each commit is referenced by a unique ID, which is calculated from a hash of the current timestamp, the commit message, the ID of the parent commit(s), the contents of the change, and other meta information. A new branch can be created at any point in time. Per default, it is based on the current commit. A branch is similar to a label that points to the commit that is the latest commit of this branch. Changes to the new branch will not affect the former branch. For example in software development, branches are typically used to implement new features. This allows the developer to track new changes without affecting the master branch with unstable or untested changes. Additionally, changes in the master branch will not affect the developer. When the feature is ready to be included in the master branch, these changes can be pulled into the master branch by a merge operation. During the merge operation, the changes must be combined in a way that the outcome is the same as when the changes were applied linearly. To do so, *git* will analyze the changes on each branch since the last common commit of them. For example,

if each file is only altered in one of the two branches, the two sets of changes will not interact with each other and thus can be combined by a union set. If files are altered in both branches, *git* offers different merge strategies to combine the changes. Third-party applications can merge changes while using contextual knowledge of the merged content, for example for source code files of specific programming languages. If the two branches contain conflicting changes that cannot be resolved by *git*, it will ask the user to resolve these conflicts manually. This is called a *merge conflict*. The merge itself will create a merge commit in the history of the branch where the changes were pulled in. In some instances, it is possible to use a fast-forward merge instead of this explicit merge. Simplified, the fast-forward merge can only be used if the affected branch is not yet pushed to a remote server, as it will change the history and is therefore causing problems for other users who may have already added commits to this history. A fast-forward merge can only be used in the first example where each file is altered within at the most one branch. This enables *git* to reapply the changes of the merged-in branch to the target branch as if they were committed to this branch in the first place. Since the commit IDs depend on the parent commit, this causes the IDs of all following commits to change. Therefore, this should only be used for local histories.

Git Commands

Git will typically be invoked by the `git` base command on the command line. For example the `git --version` command shows the version number of the installed binary. All the different functions of *git* are split up into different subcommands. For example `git merge` to merge different branches or `git log` to show the commit history of the current branch. All these subcommands have their own options and arguments. Some subcommands have subcommands themselves. For example `git notes` is split into 9 subcommands, such as `git note add` to add a note to a commit. At this point in time, the latest *git* version 2.36.0 supports 142 subcommands[1]. Due to the high number of features and high complexity, this software may suffer from bugs.

Git Trees

When making changes to a history in *git*, the change has to go through three different stages. *Git* calls these stages *trees*. Think about these trees more as a collection of files than the data structure. These trees are called *HEAD*, *index* and *working directory*. The HEAD is a pointer to the latest commit of the current branch, which will be used as the parent of the next commit. The index contains the proposed next commit. It is also called *gits staging area*. When a commit is checked-out, the index is filled with the file contents of this snapshot. Finally, the working directory, also called the *working tree*, is the directory on the file system in which *git* creates the files of the snapshot of the checked-out commit. This is the directory in which we can access the files and make changes to them.

Let us assume we want to check out a branch and edit an existing file. In the typical *git* workflow, we first use the `git checkout` command to check out the master branch. This sets the HEAD to point to the latest commit on the master branch, populates the index with the file

[1] `https://www.git-scm.com/docs/git/2.36.0#_git_commands`

contents of this snapshot, and *git* creates the files in the working directory to match the index. We can then edit the file within the working directory and use the `git add` command to add the altered content of this file to the index. This is also called *staging* the changes. After the staging, we can use the `git commit` command to create a new commit containing the files from the index. After this, the HEAD will be updated to point to this new commit.

Git also provides the option to configure multiple working directories for one repository. Each working directory has its own index and HEAD.

Remote Synchronization

Git is a decentralized version control system. Therefore, repositories can interact with each other to share changes and synchronize with each other. For example, there is some repository on the Internet with source code that we want to use. The typical workflow would be to use `git clone` command to make a local copy of the remote repository. This will also check out the default branch. Let us assume we have made some changes and have created new commits at the master branch. So, in order to synchronize our local master branch with the master branch of the remote repository, we can use the `git push` command.

Suppose, we have the permission to write to the remote repository, the push operation will try to append our new commits to the top of the remote master branch. If nobody else has added changes to this remote branch in the meantime, this operation will succeed. If somebody has made changes in the meantime and the remote branch does not match our local version anymore, it will fail. In this case, we first have to synchronize our local copy with the remote one. This can be done by using the `git pull` command which will first fetch the remote changes and in the following merge our changes on top or fast-forward if possible. This may also result in a merge conflict, which must be resolved manually. After finishing the merge operation, we will be finally able to push our changes to the remote branch.

Git also allows the user to interact with more than one remote repository. Each remote repository is called *upstream* in our local repository. Local repositories can also be configured to use multiple upstreams. This enables us to pull and push changes at will.

Git Configuration

Git offers a lot of configuration options. Looking at all *git* commands there are about 2000 optional arguments in total. Some of these options can be permanently set within the *git* configuration file. This configuration also contains other *git* properties, such as upstream targets. When a local *git* repository is created, either via `git init` as an empty repository or via `git clone` as a clone from a remote repository, *git* will create a directory called *.git*. *Git* will use this directory to store all its internal states, such as the configuration file, the index, and the history. Besides this configuration file within the repository, *git* also uses a global configuration file. The configuration file is text-based and uses a hierarchical key-value system. For example, when an empty repository is created via `git init`, *git* will also create and check out a new branch called “master” by default. The default name of this first branch can be changed via the configuration *init.defaultBranch* key. The

command `git config --global init.defaultBranch main` can be used to set the value of this key to “main” in the global configuration file.

3 Approach

This chapter describes our approach for an automatic metamorphic testing tool aimed at version control systems.

3.1 Overview

In metamorphic testing we start with some test input as our source test case. A follow-up test case is derived by applying transformations to this source test case. These transformations are chosen in such a way that the specific relations between the outputs of the two test cases hold true. Section 3.2 describes our approach to tackle the first problem of getting test inputs by generating test cases from scratch. In Section 3.3, we describe the relations used for this approach and how we check them. Section 3.5 explains the transformations used with this approach. If a relation does not hold true for a source and follow-up test case, we have to evaluate the cause of the violation and figure out if this is a false positive or a true positive warning. Since the test cases can be large and difficult to understand, we use an automated test minimizer that minimize the test cases which is described in Section 3.6. These steps are visualized in Figure 3.1.

3.2 Script Generation

To be able to run metamorphic tests, we need to have some test input to work with. In our context, a test input is a shell script with commands that interact with *git* and the file system. One approach to obtain test inputs is to generate them at random. The challenging part is about how much to guide the generator and how much should be random. A very simple and naive approach would be to use a pure random text generator that knows nothing about the application context. Of course, this generator is probably not producing a valid shell command most of the time. Therefore, we need a generator with some knowledge of the application context to make sure, it generates at least syntactical valid shell scripts and also increases the probability of generating valid *git* commands. For this, we will use a logical model for concrete commands, so that they can be generated from

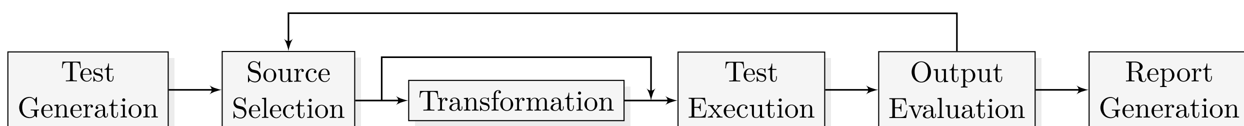


Figure 3.1: Approach Overview.

command templates. To generate whole scripts, the generator will first insert commands to prepare the current working directory and then repeatedly insert a new generated concrete command and check if the script still executes without errors. If an error occurs, the generator will go back to the last insertion and try again. This section will explain the terminology, the model, and the details of the generator.

3.2.1 Command Model

To be able to describe our command model we have to distinguish between command templates and concrete commands. A **command template** describes general properties of a command, such as its name, possible arguments, and possible subcommands. These properties are used to derive instances of **concrete commands** from the command template. A concrete command contains valid values for the properties defined by the template. It can be transformed into a string representation to be used in a shell script. You can assume the command templates as being a formal grammar that describes how a valid command of this type looks like on the command line.

The generator will use the command templates with some information from the current generation context to generate concrete commands. A command template $t \in T$ is defined as a four-tuple (N_t, OD_t, AD_t, S_t) with the following properties:

Name ($N_t \in \Sigma^*$) The name of the binary to invoke the command on the command line or to invoke the subcommand.

Optional argument definitions ($OD_t \subset \hat{OD}$) A set containing the definitions of all individual optional arguments or flags which can be used as a concrete instance of a command generated by this template. These definitions contain at least all the possible names of the corresponding options and their type. These definitions are used to generate concrete values for each option when instantiating a command from the template and the definitions can also contain other attributes which will be specified later.

Positional argument definitions (AD_t) This is a function that provides a list with the type of each positional argument used to instantiate a concrete command from the template. The output of this function can be non-deterministic. Positional arguments will be appended to the command after the optional arguments.

Subcommand templates ($S_t \subset T$) If the command supports subcommands, a template for each subcommand is defined in this set. Otherwise, this set is empty.

Each concrete command c generated from a command template $t \in T$ is defined by a five-tuple $(N_c, O_c, A_c, R_c, P_c)$ with the following properties:

Name ($N_c \in \Sigma^*$) The name of the binary to invoke the command on the command line or the subcommand as specified by N_t.

Optional arguments (O_c) A set of optional arguments where each option is instantiated according to some option definition OD_t.

Figure 3.2: Example of a concrete command on the command line with its annotated parts.

Positional arguments (A_c) A list of positional arguments which will be passed to the command during runtime. In contrast to the option's set, the order of these arguments is important. The list will be filled with values of the types provided by AD_t.

Redirections (R_c) A list of redirections to redirect the output of the command to a file.

Parent (P_c) If this concrete command is a subcommand of another concrete command, P_c will contain the other command. Otherwise, P_c will be ω.

Figure 3.2 shows an example of a concrete `git commit` command as it can be invoked on the command line. The first token is the name of the `git` command followed by an optional argument `-C` which itself takes the file path *repo1* as arguments. This argument specifies the path that *git* should use as the current working directory for this single invocation pretending to be in the directory when invoking the command. The next token `commit` is the name of the subcommand of *git* that should be invoked. This subcommand itself has two optional arguments and two positional arguments. The `-m` option itself takes also a string as an argument, which is the commit message for this commit. The `--allow-empty` option allows us to make a commit that does not contain any changes. This is also called a flag as it does not take any arguments themselves. It can be either specified or not. The last part `>out.txt` is a redirection of the standard output stream of the command into the file *out.txt*. This part is interpreted by the shell and is not passed to the program.

To generate commands with arguments, the generator needs to know their types and meaning. For example, the `git commit -m` takes some argument but the `git commit --allow-empty` option does not. The `git -C` option expects the name of a directory. We could try to use some random string as a value for this option, but there would be a very low success rate actually to hit the name of an existing directory. To overcome this, we assign a type to each argument that will be used to choose an appropriate value while taking the current generation context into account. This approach uses the following types (OT):

File The path of a file that was created by the generator at some point in time during the generation of the current script

Directory The path of a directory that was created by the generator at some point in time during the generation of the current script

Commit ID The ID of a commit that was done during the generation of the current script. Since the commit hash ID changes each execution, we use some logical identifier that is resolved to the actual commit hash during runtime. This will be explained in detail in Section 3.5.1.3.

Random word Some random alphanumeric string with a length of 10 characters.

Selected word This type will be defined with a set of strings from which one instance is chosen at random. This can be used for options such as `git commit --untracked=<mode>` where mode can be *all*, *normal* or *no*.

Branch name A randomly chosen name of a branch from the finite set of possible branch names used by the generator.

Current working directory The current working directory has the task to enable multi-user scenarios. We use multiple working directories with different *git* repositories which can interact with each other.

Bool This type is used for optional arguments that do not take any arguments themselves.

During the generation, we retrieve a value for each type from the generation context. Each option definition $od \in \hat{OD}$ of the command templates is defined as a six-tuple (cli_{od}, sep_{od}, ot_{od}, $cond_{od}$, env_{od}, $conf_{od}$) with the following attributes:

Cli names ($cli_{od} \subset \Sigma^*$, $0 < |cli_{od}| \leq 3$) A set of the names used to specify this specific option. Typically, an option can be used with its short form e.g. `git commit -v` or with its long form e.g `git commit --verbose` which is semantically equivalent. In addition, some options provide even a third name.

Cli name seperators ($sep_{od} : cli_{od} \to \Sigma$) A function that provides the separator character for each option name $x \in cli_{od}$. The separator character is inserted between the name of an option and its argument if it has an argument. This is necessary since *git* uses different notations per option and even per argument name, for example for `git commit --untracked=<mode>` we need to use the equal sign between the name of the option and the mode value, but the short form of this option needs to be used as `git commit -u<mode>`. There does not have to be any character between the name and the mode value. As *git* still uses the equal sign as separator most of the time, let $sep_{od} : x \mapsto$ '=' $\forall x \in cli_{od}$ if not stated otherwise.

Type ($ot_{od} \in OT$) The value type as explained above.

Preconditions ($cond_{od}$) A set with preconditions that are checked before generating this option. If one precondition fails, the generation of this option will be skipped. They are used to specify restrictions about the other options that are allowed to be used simultaneously. For example, while one has the possibility to use `git commit -m "some message"` to specify the commit message directly, it is also possible to use `git commit -F messageFile.txt` and provide the path to a text file to read the message from. Is it not allowed to use both of these options at the same time. Respecting such restrictions helps to guide the generator to generate valid commands.

Env option name ($env_{od} \in \Sigma$) Particular options of some *git* commands can be specified by setting the corresponding environment variable. For example, the path to the working tree can

be passed to *git* as `git --work-tree=<path>`, but this option can also be specified via an environment variable with `GIT_WORK_TREE=<path> git`. If this option can also be specified via an environment variable, env_{od} will contain the name of the corresponding environment variable or ϵ otherwise.

Config key name ($conf_{od} \in \Sigma$) Similar to the environment variables, particular options can be defined within the *git* configuration file instead of specifying them directly on each use. For example the `git --work-tree=<path>` option can also be written permanently to the configuration file by using the command `git config core.worktree <path>` where *core.worktree* is the name of the configuration key for this option. If this option can also be specified via a configuration file entry, $conf_{od}$ contains the name of the corresponding configuration key or ϵ otherwise. The env_{od} and $conf_{od}$ values will be used to transform commands which is explained in Section 3.5.

Let an option value $o \in O$ of a concrete command be a tuple (d_o, c_o, v_o) with

$$d_o \in \hat{OD}: \text{ the definition of this option} \tag{3.2.1}$$

$$c_o \in cli_{d_o}: \text{ the used cli option name} \tag{3.2.2}$$

$$v_o \in \Sigma^*: \text{ the assigned value or } \epsilon \text{ if empty} \tag{3.2.3}$$

With all these definitions, we can finally describe the first templates. As an example we define part of the `git` and `git commit` command template. Let t_{commit} be $(\text{commit}, OD_{\text{commit}}, ad_{\text{commit}}, \omega)$ with

$$\begin{aligned}
OD_{\text{commit}} &:= \{od_{\text{message}}, od_{\text{allow-empty}}, od_{\text{reuse-message}}, od_{\text{file}}, ...\} \\
od_{\text{message}} &:= (\{\texttt{-m}, \texttt{--message}\}, sep_{\text{message}}, \text{random word}, \\
&\qquad \{\text{not with } od_{\text{file}} \text{ option}\}, \epsilon, \epsilon) \\
sep_{\text{message}}(\texttt{-m}) &:= \text{'}\llcorner\!\lrcorner\text{'} \\
sep_{\text{message}}(\texttt{--message}) &:= \text{'='} \\
od_{\text{file}} &:= (c\{\texttt{-F}, \texttt{--file}\}, sep_{\text{file}}, \text{file}, \{\text{not with } od_{\text{message}} \text{ option}\}, \epsilon, \epsilon) \\
sep_{\text{file}}(\texttt{-F}) &:= \text{'}\llcorner\!\lrcorner\text{'} \\
sep_{\text{file}}(\texttt{--file}) &:= \text{'='} \\
od_{\text{allow-empty}} &:= (\{\texttt{--allow-empty}\}, sep_{\text{allow-empty}}, \text{bool}, \emptyset, \epsilon, \epsilon) \\
sep_{\text{allow-empty}}(\texttt{--allow-empty}) &:= \epsilon \\
od_{\text{reuse-message}} &:= (\{\texttt{-C}, \texttt{--reuse-message}\}, sep_{\text{reuse-message}}, \text{commit ID}, \emptyset, \epsilon, \epsilon) \\
sep_{\text{reuse-message}}(\texttt{-C}) &:= \text{'}\llcorner\!\lrcorner\text{'} \\
sep_{\text{reuse-message}}(\texttt{--reuse-message}) &:= \text{'='} \\
ad_{\text{commit}} &:= \text{randomFrom}(\{\emptyset, [\text{file}], [\text{file}, \text{file}]\})
\end{aligned}$$

And let t_{git} be $(\text{git}, \{od_C\}, \emptyset, \{t_{\text{commit}}, ...\})$ with

$$od_C := (\{\texttt{-C}\}, sep_C, \text{current working directory}, \emptyset, \epsilon, \epsilon)$$
$$sep_C(\texttt{-C}) := \text{'␣'}$$

According to this template, the message option of the commit command can be either specified with the short form `-m` followed by a space and the message or with its long-form `--message` followed by an equal sign and the message. Additionally, we specified the condition for the command generator that the option cannot be added to a commit command if the `-F` option is already present since they are incompatible with each other. The command can be specified with zero to two file paths as arguments. These definitions can be used to model a concrete instance of the command shown in Figure 3.2. Let $c_{\text{commit3.2}}$ be $(N_{t_{commit}}, \{o_{\text{allow-empty}}, o_{\text{message}}\}, a_{\text{commit}}, [\texttt{>out.txt}], c_{\text{git3.2}})$ with

$$o_{\text{allow-empty}} := (od_{\text{allow-empty}}, \texttt{--allow-empty}, \epsilon)$$
$$o_{\text{message}} := (od_{\text{message}}, \texttt{-m}, \texttt{"fix"})$$
$$a_{\text{commit}} := [\text{foo.txt}, \text{bar.txt}]$$
$$c_{\text{git3.2}} := (N_{t_{git}}, \{o_{\text{C}}\}, [], [], \omega) \text{ with}$$
$$o_{\text{C}} := (od_{\text{C}}, \texttt{-C}, \texttt{repo1})$$

3.2.2 Script Generator

The script generator component is used to create concrete commands from the command templates and assemble multiple concrete commands to form a script. As mentioned before, the generator uses the so-called generation context to store a part of its internal state during the generation of scripts. The generation context contains the following information:

Created files The relative paths of the files that are generated by a file creation operation within this context at some point in time.

Current working dir The name of the current working directory. The algorithm switches between different working directories and updates this property accordingly.

Number of instantiations per template The generator tracks how many times each command was generated to prioritize less often used commands.

Number of created commits The generator assigns a logical ID to each commit command. This ID will be used for the relations later.

The generation of a script begins with creating an empty command list and initializing the generation context. The initialization operation inserts handcrafted commands into the command list to prepare the initial working directory with a *git* repository that contains an initial commit. Algorithm 1 shows this algorithm in pseudo code. The algorithm gets a seed for the random number generator and the desired number of commands as input. It uses a stack to save checkpoints of different states of the command list, generation context, and error counter during the script generation. After the initialization step, the algorithm creates a checkpoint of the initial state on the stack and enter the generation loop on line number 6. This loop is executed until the desired number of

Algorithm 1 Generate random script.

```
 1: INPUT random seed r, number of commands n
 2: stack ← [ ]                  ▷ Stack with tuple of command list and generation context
 3: ctx ← new generation context
 4: cmds ← ctx.initialize(r)
 5: stack.push((ctx, cmds, 0))
 6: while countCmds(cmds) < n do                                        ▷ Generation loop
 7:     ctx, cmds, errCount ← stack.peek()
 8:     ctx, newCmds ← generateCommands(ctx)
 9:     cmds ← cmds ‖ newCmds
10:     if canExecuteWithoutErrors(cmds) then
11:         ctx, cmds ← ctx.maySwitchWorkingDir(ctx, cmds)
12:         stack.push((ctx, cmds, 0))                                ▷ Create checkpoint
13:     else
14:         stack.last.errCount ← errCount +1
15:         if stack.last.errCount > threshold then
16:             stack.pop()                                                 ▷ Backtrack
17:         end if
18:     end if
19: end while
20: cmds ← collectRuntimeStates(cmds)
21: return scriptOf(cmds)
22: OUTPUT script
```

N_{cmds} commands is reached. We use a value of $N_{\text{cmds}} = 40$ per default. At first, the current checkpoint is loaded from the stack, then new commands are generated and added to the command list. The generation of the commands will be described later. On line number 10, a script is generated based on the command list, and the generator will try to execute this script. If the execution of all commands within this script is successful, we consider the whole execution as being successful. In this case, some probability is used to optionally switch the current working directory, then create a new checkpoint with the current state of the command list and the generation context on line number 12, and start another round of the generation loop. If the execution was not successful, the algorithm will record this error, and check if we reached the maximum number of errors for the current checkpoint. In this case, the algorithm removes the current checkpoint from the stack on line number 16 to prevent the algorithm from getting stuck. Afterward, it continues with the generation loop. When the desired number of commands is present, the generation loop is exited. In that case, the generator will execute an extended version of the script to collect runtime information before the execution of each command on line number 20 and return the script subsequently. The runtime information is used within the precondition checks within the transformation steps later.

Within the *maySwitchWorkingDir* function call on line number 11, the algorithm checks if we have already created the maximum number of working directories and create a new one if this is not the case. During this action, it chooses one of the other working directories at random to be configured as the default origin for push and pull operations of this working directory, and pulls the

history from them. Independent of the creation, the algorithm commits all staged changes present in the current working directory, saves the name of the currently checked-out branch, and checks out a new branch with a random name. It has to check out another branch because *git* refuses to push into a branch currently checked out at the remote repository. This would cause the HEAD of the remote working directory to become detached. On the switched-to working directory, the algorithm checks out the branch that was checked out before switching to the random branch.

Within the *generateCommands* function the generator chooses and instantiates one or more command templates. To increase the possibility to generate command sequences that would occur in normal use, we use special handling for the `git add` and the `git mv` commands. Additionally, the generator can insert `echo` commands to create, append or override files and can create symbolic links. After a file or symbolic link is touched, there is some chance that the generator also inserts a `git add` command to add the update of this file to the index. The `git mv` command is used to move or rename files and therefore takes the path of the file and the destination. If the destination path already exists and the command is used without the `--force` flag, it fails. Therefore, the generator uses a fifty percent chance to choose a likely unused path as the destination. For path handling, the generator uses a fixed list with possible paths and taints them as used after the first use. Since we use multiple working directories and the file could also be deleted at some point, it is not guaranteed that a *used file* actually exists in the current working directory at this point in time. The generation context contains information about how many times each command or case was chosen. When choosing a case, the generator uses the inverse of this occurrence count as weight.

Algorithm 2 function genCmdFromTemplate: Generate command from template.

1: **INPUT** generation context ctx, command template t
2: $O \leftarrow$generateOptionValues(OD_t, ctx)
3: $A \leftarrow [\,]$
4: **for all** $ad \in AD_t()$ **do**
5: $\quad A \leftarrow A \parallel ctx$.getValueForType($ad$)
6: **end for**
7: $P \leftarrow \omega$
8: **if** $\exists t' \in T : t \in S_{t'}$ **then** ▷ Check for parent
9: $\quad t' \leftarrow$ randomFrom($\{t' \in T | t \in S_{t'}\}$)
10: $\quad P \leftarrow$ genCmdFromTemplate(ctx, t')
11: **end if**
12: $R \leftarrow [\,]$
13: $N \leftarrow N_t$
14: **return** (N, O, A, R, P)
15: **OUTPUT** command

The generator uses Algorithm 2 to derive concrete commands from a command template. The algorithm takes the current generation context and a command template as input. At first, it instantiates a set of valid option assignments on line number 2. We will have a look into the generation of these option values later. Then the algorithm gets a list of types from the argument types function of the command template on line number 4 and retrieves a value for each type

from the generation context to be used as positional arguments of the concrete command on line number 5. Afterward, the algorithm checks if the provided template is used as a subcommand within another template and instantiates the parent command recursively.

To generate a set of options for a command, the script generator uses the algorithm shown in Algorithm 3. It generates an option assignment for up to $max(\lfloor\frac{|OD|}{2}\rfloor+1, |OD|)$ optional arguments. The exact number is chosen at random to ensure variation. Also, the order in which the options are assigned will vary since the elements in the set do not have a fixed order. Since the preconditions are checked before an option is assigned, it is essential that the order is not fixed. If the order would be fixed and the preconditions of two options would not allow them to be generated with each other, such as the od_{message} and od_{file} options of the t_{commit} command template, each time they are chosen both, only the one further in the list would be assigned. This would cause some bias on the probabilities.

Algorithm 3 Generate random optional arguments from option definitions and generation context.

1: **INPUT** option definitions $OD \subset \hat{OD}$, generation context ctx
2: $options \leftarrow \emptyset$
3: $n \leftarrow$ choose random from $0, \ldots, max(\lfloor\frac{|OD|}{2}\rfloor + 1, |OD|)$
4: $OD' \leftarrow$ choose n elements from OD
5: **for all** $od \in OD'$ **do**
6:     **if** all $cond_{od}$ satisfied **then** ▷ Check preconditions
7:         $c \leftarrow$ choose random from cli_{od} ▷ Choose cli name
8:         $val \leftarrow ctx$.getValueForType(ot_{od}) ▷ Retrieve value for type
9:         $option \leftarrow (od, c, val)$
10:         $options \leftarrow options \cup \{option\}$
11:     **end if**
12: **end for**
13: **return** $options$
14: **OUTPUT** options with $\forall o \in$ options : $d_o \in OD$

Table 3.1 shows a list of the *git* command templates used in the approach along with the number of optional arguments that can be generated per command and information about them.

3.3 Relations

The relations are one of the core parts of metamorphic testing. When we execute the source and follow-up test case, we check how both outputs relate to each other. Also, we have to define what an *output* is in regard to a specific relation. Our approach uses two types of relations which both assume that their respective outputs will be the same for both test cases. We call them *exit relation* and *source independent checkout relation*. This implies that each test case itself must be deterministic with respect to the outputs relevant to us.

Table 3.1: List of all *git* commands implemented in the generator.

Command name	Options	Generatable options	Cli names	Config keys	Environment variables
git	16	8	18	1	9
git add	14	11	22	1	0
git branch	0	0	0	0	0
git checkout	19	17	24	1	0
git clean	5	4	8	0	0
git clone	0	0	0	0	0
git commit	19	13	36	4	0
git config	0	0	0	0	0
git diff	0	0	0	0	0
git fetch	18	17	26	1	0
git init	0	0	0	0	0
git log	0	0	0	0	0
git merge	0	0	0	0	0
git mv	4	2	7	0	0
git notes	0	0	0	0	0
git notes add	6	5	11	0	0
git notes prune	2	1	4	1	0
git notes show	0	0	0	0	0
git pull	28	25	36	3	0
git push	18	15	24	2	0
git remote	0	0	0	0	0
git reset	9	2	11	1	0
git restore	16	16	21	2	0
git rev-parse	1	1	1	0	0
git rm	7	6	10	0	0
git stash	0	0	0	0	0
git stash apply	2	2	3	0	0
git stash pop	2	2	3	0	0
git stash push	7	6	14	0	0
git status	0	0	0	0	0
git switch	0	0	0	0	0
git tag	8	5	16	1	0
git update-index	1	1	1	0	0
Total	202	159	296	18	9

(a) Checkout of A after adding B. (b) Checkout of A after adding C.

Figure 3.3: The checkout of A must lead to the same output in both cases.

3.3.1 Exit Relation

This relation looks at the files left behind in the working directory after the execution of the test case. When collecting the output, it recursively traverses the working directory and collects information about the found files and directories, but ignores *.git* directories. *Git* uses the *.git* directory in the working directory to store the internal state of the repository. Since this state also includes dynamic information, such as timestamps, the contents of these files vary on each execution. Therefore, from our point of view, we consider the contents of these files as being non-deterministic and ignore them during the collection of our output. During the collection, the approach differentiates between regular files, directories, symbolic links, and other files. Other files can be for example named pipes or sockets. For all types, the approach collects their path relative to our working directory and their type. For regular files, it also collects their contents. For symbolic links, it collects their corresponding targets. Symbolic links are similar to aliases for files or directories within the file system. If the target path is an absolute path within our working directory, the approach converts it into a path relative to our working directory. Since it ignores *.git* directories, a directory that only contains such a directory is treated as an empty directory. The contents of directories are collected implicitly through their contained files.

When comparing the outputs of two test executions, the approach considers the *exit relation* fulfilled, if their outputs contain the same entries.

3.3.2 Source Independent Checkout Relation

The *source independent checkout relation* is used to check an invariant within a single test case execution. It uses the same approach to collect the contents of a working directory as in the output of the *exit relation*, but instead of collecting a single output after the execution of a test case, it collects multiple sets of outputs during the execution of the test case. This relation checks the invariant, that the commits within a history do not change their contents, even when new commits are added on top of the history. Figure 3.3 shows an example of this relation. The checkout of commit A must lead to the same output of the working directory regardless of whether commit C was added. The approach collects these outputs during the script execution each time when a new commit is added to the history. When this happens, it checks out all former commits of the history into a temporary directory and collects the contents of this directory as in the *exit relation*. After the execution of the test case, it checks if the output for a specific commit was equal at each point in time. If this is not the case for at least one output, the approach considers this relation as violated.

Formally we describe this relation as follows: Let $s \in S$ be a commit and let $prev : S \to \mathcal{P}(S)$ be the set of commits that happened at a point in time before the commit itself. Let

$op : S \times S, (s, k) \mapsto o$ be the state of s checked-out right after recording commit k. Considering the situation of Figure 3.3, we would have $S := \{A, B, C\}$ with $prev(A) := \emptyset$, $prev(B) := \{A\}$ and $prev(C) := \{A, B\}$. The relation holds true if $\forall s \in S : \forall a \in S \setminus prev(s) : b \in S \setminus prev(s) : op(s, a) = op(s, b)$.

3.4 Test Execution

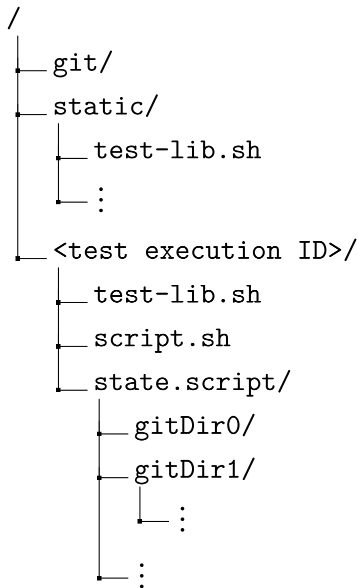


Figure 3.4: Structure of the directories for test execution.

After retrieving a test script as described, the script is used as a source test case for the test execution. A transformation is applied to the source test script to retrieve a follow-up test script. These two scripts are executed, and the outputs are analyzed afterward. If the two outputs relate to each other as expected, the execution is considered successful. Otherwise, a violation report is generated. For the successful executions, we use the approach of iterative metamorphic testing [19] proposed by Peng Wu where the follow-up test script of the current execution is reused as the source test script for the subsequent test execution, in a chain-style fashion. After N_{iter} iterative repetitions, the last N_{iter} applied transformations are reverted to start with the original source test script again. When N_{total} transformations have been applied to a test script in total, the approach discards the script and generates a new one from scratch to get more diversity. Our default values are $N_{\text{iter}} = 15$ and $N_{\text{total}} = 250$.

It is important that the execution of each test does not affect the execution of other tests. For this, each test is executed within its own test directory and the *git* testing harness is used to configure *git* to not touch any configuration files outside of this working directory. Figure 3.4 shows the structure of the file system for the test execution. The *git* directory contains the built version that is used for the tests. This allows us to test arbitrary *git* versions without modifying the locally installed *git* version. The *static* directory contains the file and directories of the testing harness. We create a test directory with a unique name per execution which is used as test directory. The test script is exported to this directory and symbolic links to all files and directories of the *static* directory are created. Also, the *script.state* working directory for the test is created within this directory. In normal use, the test harness deletes the working directory after executing a test. Since we have to analyze the working directory before it can be removed, we use a slightly modified version of the test harness that leads to an intact working directory afterward.

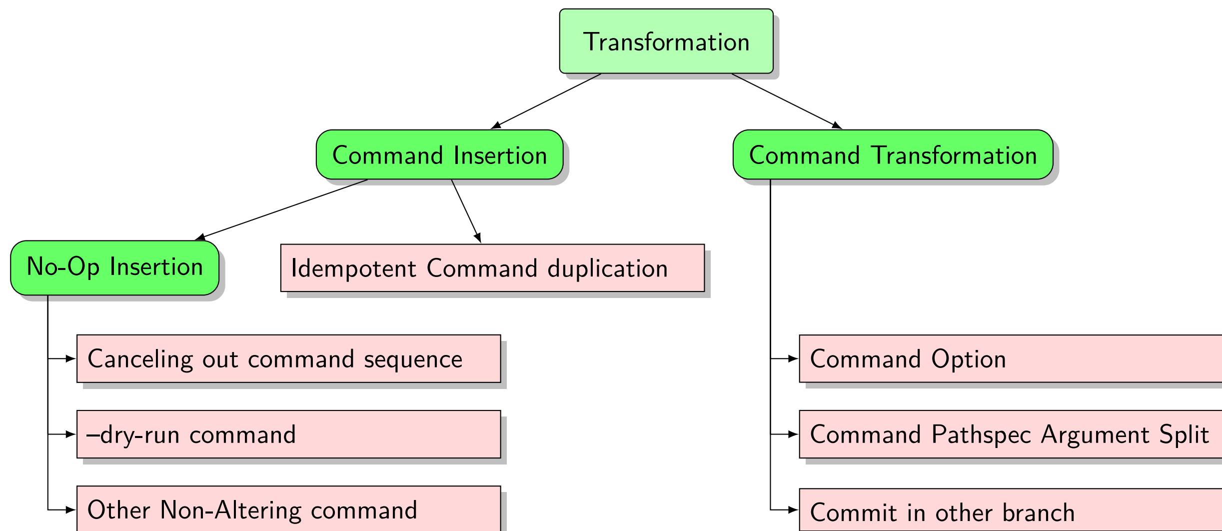


Figure 3.5: Transformation hierarchy.

3.5 Transformations

The transformations are applied to a source test case to derive a follow-up test case. The transformations of this approach are grouped into different variants. Figure 3.5 shows the hierarchy of our transformations. At first, there are two general types of transformations: The *Command Insertion* and the *Command Transformation*. The difference is that *Command Transformations* alter existing commands or their effect, while *Command Insertion* transformations do not alter existing commands at all. The *Command Insertion* transformations are further grouped into the *No-Op Insertion* transformations and the duplication of idempotent commands. This section will describe all these transformations.

3.5.1 Command Transformations

The command transformations replace an existing command by one or more other commands. The various types are described in this section.

3.5.1.1 Command Option Transformation

As explained in Section 3.2.1, there are *git* commands which allow us to specify one logical option over multiple different channels. For example, the *git* `--work-tree` command-line option can be also specified via the `GIT_WORK_TREE` environment variable as well as by setting the `core.worktree` key in the *git* configuration file. Additionally, a configuration value can also be specified directly at the command invocation via the *git* `-c` option. Since all of these variants are semantically equivalent, we are able to switch between the different representations without changing the behavior of the commands and therefore are not violating any relation by doing so. In Section 3.2 we have already included the $conf$ and *env* properties in our option definition. If a concrete command contains an option that has any of these properties set within its definition, we are able to switch the way this option is used with this command. Therefore, let $\hat{M} := \{m_{\text{cli}_0}, m_{\text{cli}_1}, m_{\text{cli}_2}, m_{\text{env}}, m_{\text{conf}_{\text{file}}}, m_{\text{conf}_{\text{override}}}\}$

the possible output modes in general and let $M : \hat{OD} \rightarrow \hat{M}$ be the set of possible output modes for an option definition *od* of a command *c* with

$$\begin{aligned} m_{\text{cli}_i} &\in M(od),\ 0 \leq i < |od_{cli}| \\ m_{\text{env}} &\in M(od) \leftrightarrow od_{env} \neq \epsilon \\ m_{\text{conf}_{\text{file}}} &\in M(od) \leftrightarrow od_{conf} \neq \epsilon \\ m_{\text{conf}_{\text{override}}} &\in M(od) \leftrightarrow (od_{conf} \neq \epsilon \land c_{\text{git}} \in parentsOf(c)) \end{aligned}$$

were each $m_{\text{cli}_i},\ 0 \leq i < |od_{cli}|$ corresponds to exactly one element in od_{cli}. Also let $opm : O \rightarrow \hat{M}$ be the output mode that should currently being used for an option of a concrete command, with $opm(o) := m_{\text{cli}_0}$ for each option $o \in O$ per default.

Cli output mode For the current active output modes m_{cli_0}, m_{cli_1} or m_{cli_2}, the option is used with the corresponding cli option name as cli option. For example `git commit -F foo.txt` for m_{cli_0} and `git commit --file=foo.txt` for m_{cli_1}, or `git --work-tree=somePath status`.

Env output mode If $opm(o) = m_{\text{env}}$ is active, the final command string is prepended with the definition of the corresponding environment variable. For example `GIT_WORK_TREE=somePath git status`.

Config file output mode If $opm(o) = m_{\text{conf}_{\text{file}}}$ is active, the command is enclosed between helper commands before and after the command. Commands to back up the current local *git* configuration file and add the corresponding key-value-pair to the configuration file are inserted before the command, and commands to restore the former configuration file are inserted after the command. This could result in the following commands:

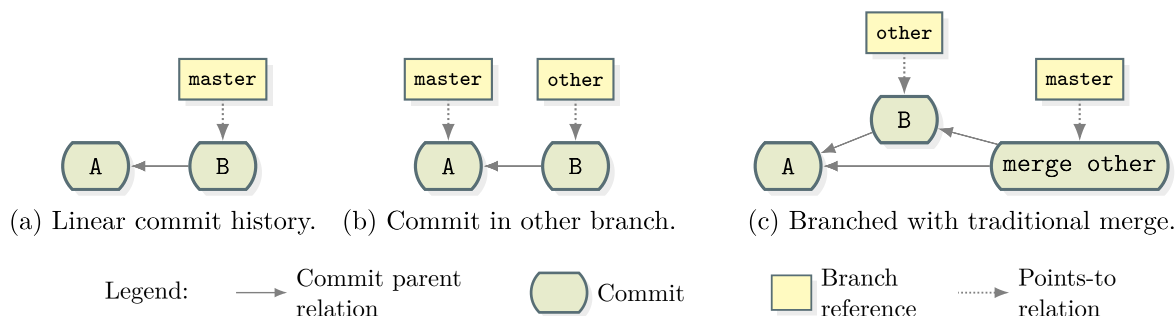
```
backup_git_config
git config --local core.worktree somePath
git status
restore_git_config
```

Config override output mode If $opm(o) = m_{\text{conf}_{\text{override}}}$ is active, the configuration option is added via the *git* `-c` option to the `git` command. This option is intended to temporarily override some configuration values of the configuration file. For example `git -c core.worktree=somePath status`.

Therefore, $pt : O \rightarrow \mathcal{P}(\hat{M}), o \mapsto M(o_{od}) \setminus opm(o)$ is the set of the output modes to which we can switch at some point in time for the option of a concrete command.

3.5.1.2 Command Pathspec Argument Split

Some *git* commands take a file path as a positional argument to do some actions regarding this file. To a lot of these commands, we can also pass multiple file paths to do this action to all of these files. For example the `git add` command can be used to add files to the index, the `git rm` command can be used to remove files from the index and working tree in some instances, and the



(a) Linear commit history. (b) Commit in other branch. (c) Branched with traditional merge.

Figure 3.6: Commit histories.

`git restore` command can be used to restore files in the working tree with their contents at some point in the history. If a script contains a concrete command with such a "for each" semantic and multiple file path arguments are used, the approach divides the list of file path arguments into multiple parts and replaces the original command by multiple commands with one of these parts, each. For example the command `git add file1 file2` can be replaced by the two commands `git add file1` and `git add file2`.

One special case is the handling of so-called *magic signatures*. They are path specifications with a special syntax interpreted by *git*. For example the command `git add *.txt ':!.old.txt'` is used to add all files ending with `.txt` to the index except those ending with `.old.txt`. In this case, we are not allowed to split them into two commands, because these two path specifications influence each other. To overcome this problem, the approach takes only non-magic file paths into account when dividing the file path list and keeps the magic file paths on all of the inserted commands. For example the command `git add a/*.txt b/*.txt ':!.old.txt'` can be split into `git add a/*.txt ':!.old.txt'` and `git add b/*.txt ':!.old.txt'`.

3.5.1.3 Commit in Other Branch Transformation

Committing some changes inside a branch and merging them back to the parent branch should lead to the same state as if the changes would be committed directly within the parent branch. Figure 3.6a shows a simple history with two commits on the master branch. If we create and checkout another branch before committing the changes of B, we would end up with a history as shown in Figure 3.6b. If the branch is merged back to the master branch as shown in Figure 3.6c, the merge commit should contain the same changes as if B would be committed within the master directly. The merging can be done with a traditional merge which produces a merge commit or it can be done using a fast-forward merge which would apply the changes of the other branch one after another to the master without creating a merge commit as if they were committed to the master in the first place.

When specific conditions are met, the approach can switch to another branch before a commit command in our test script and merge back to the former branch afterward, either traditional or fast-forward. To be able to apply this transformation there must not be a merge in progress since the checkout of another branch would mess up this merge operation. Also, the HEAD must point to a branch, otherwise there is no branch to be switched back later. Another important aspect is

to pay attention to the untracked files and tracked files with pending changes after the commit. Since we need to check out the former branch first to merge in the new branch afterward, *git* may refuse the checkout due to conflicts caused by pending changes of files that are tracked by the latest commit of the former branch. To restore the state of the files of this commit, *git* needs to override these files. But since the files contain changes which are not committed, the current state of the files cannot be restored by *git* later. Therefore, *git* refuses the checkout in this case by default to avoid the loss of data. An example of such a scenario is shown in Table 3.2. The table shows the execution of the original test script and its execution after applying the transformation. Lines number 8, 10, and 11 are not part of the original test script. It also shows the content and status of the two files *a* and *b* as well as the current branch after the execution of each line. On lines number 1 to 5 we initialize the repository, create the file *a* with the content "foo", add it to the index, and commit the change. Then on line number 6 the file *a* is renamed to *b*. Afterward, a new file with the name *a* and the content "bar" is created. Since the former file *a* was renamed and the new file has not been added yet, *git* does not track this file at the current state. Then the staged change is committed, the new file *a* is added to the index and this change is committed as well. All the commits took place on the master branch. In the transformed script we switch to another branch after creating the new file *a*. The fact that the old file *a* was renamed is committed to this branch. In the aftermath, we want to check out the master branch again, to merge the commit, in the other branch, back. During this checkout, *git* restores the contents of the files in the working directory as they were at the commit on line number 5. Since this state contains a file *a* with the content "foo" the checkout overrides the content of the new file *a*. As this file is untracked and the content is therefore not part of the *git* repository, there is no chance for *git* to revert to the current state later. The changes would be lost. Therefore, the checkout command refuses to proceed and aborts the checkout operation. The rest of the script now continues within the wrong branch which easily results in the wrong output state and therefore violates a relation.

The condition in which the transformation can by applied is formally described as follows: Let F_A and F_B be the paths of the tracked files of commits A and B. Let $D_A(f)$ be the content of a file $f \in F_A$ at commit A with $D_F(f) := \Diamond$ for $f \notin F_A$ and $D_B(f)$ analogue for commit B. Let F_W be the paths of the files present in the working directory after commit B and $D_W(f)$ their contents analogue to D_A. A conflict will occur if

$$\exists f \in (F_A \cup F_B) : D_B(f) \neq D_W(f) \tag{3.5.1}$$

After the generation of each input test case, the approach executes the script once again to collect information about the state before the execution of each command. This runtime state does contain the following information:

Staged and unstaged changes of tracked files The approach uses the `git status --porcelain -uall` command to collect information on whether there are currently staged and unstaged changes of tracked files. The command outputs this information in a parsable format.

Merge active The approach uses the `git merge` command to check if there is currently an ongoing merge operation. It fails if there is already an ongoing merge.

Table 3.2: Example scenario in which the *checkout in other branch transformation* cannot be applied.

	Command	Original script: Contents of file *a*	Original script: Contents of file *b*	Original script: Current branch	Transformed script: Contents of file *a*	Transformed script: Contents of file *b*	Transformed script: Current branch
1	`git init`			master			master
2	`git commit -m init --allow-empty`			master			master
3	`echo foo > a`	foo		master	foo		master
4	`git add a`	foo		master	foo		master
5	`git commit -m "added A"`	foo		master	foo		master
6	`git mv a b`		foo	master		foo	master
7	`echo bar > a`	bar	foo	master	bar	foo	master
8	`git checkout -b other`				bar	foo	other
9	`git commit -m "renamed a to b"`	bar	foo	master	bar	foo	other
10	`git checkout master`				bar	foo	other ⚡
11	`git merge --no-ff other`				bar	foo	other
12	`git add a`	bar	foo	master	bar	foo	other
13	`git commit -m "new a"`	bar	foo	master	bar	foo	other

Legend: untracked modified unchanged

First commit present The approach uses the `git log` command to check whether it exists at least one commit in the current branch. The command fails if there is no commit present.

HEAD on branch The approach uses the `git symbolic-ref HEAD` command to check if the current HEAD points to a branch or if the HEAD is detached. The command fails if the HEAD is detached.

The use of runtime information is inspired by the approach of *interactive metamorphic testing* by Tolksdorf et al. [17]. In their approach, they used runtime information to determine the next transformation and the expected output relation on the fly while executing the tested program to overcome the dynamic nature of this program. Our approach differs in executing each original source test case once to gather the required runtime information to be able to statically determine the next transformations and relations of the output during the whole life of this input test case. This is because we need to interact with *git* to collect this runtime information which would force us to assume, that the commands used for collecting the state will not influence the behavior of the execution due to their nature or bugs. Also, since the collection of the runtime state adds a computational overhead, it could reduce the effectiveness of finding timing-related bugs.

3.5.2 Command Insertion

Git provides some commands that can be executed at some point during the script, which are not to influence the output. As a transformation, we can add such commands or command sequences to the script without violating the relations. For this, the approach either duplicates idempotent

commands, so that the second invocation of the command does not influence the output, or it generates and inserts such commands or command sequences from scratch.

Idempotent Command Duplication If the script contains commands as `git add <some file>` and `git checkout <some branch>`, the approach can duplicate them without changing the resulting state. If a file is already added to the index with no unstaged changes, the second `git add` command leads to the same result as before. This also holds true for the `git checkout` command because when checking out the same branch twice, the second command does not alter the result. To avoid duplicating the same command over and over again, we limit this transformation to once per concrete command. If a command is duplicated, the command itself and the duplicate will not be duplicated again within the current script.

3.5.2.1 No-Op Insertion

It is the goal of *no-op insertions* to generate and insert a command or command sequence at some point in the script, so that the insertion does not alter the output of the script.

To accomplish this, the approach either inserts commands which intentionally do not alter the state in general: for example, commands to just gather some information about the repository. Alternatively, it uses sequences of commands where each command itself may change the state, but within the sequence these changes will cancel out each other, so that the same state is left behind as before when specific conditions are met. For example, the `git reset --soft <some commit>` command is used to move the current HEAD to some other commit in the tree, followed by a `git reset --soft <former HEAD ID>` command with the ID of the HEAD before to revert to the state of the beginning.

`--dry-run` Commands Some *git* commands take a `--dry-run` option which is typically used to get the console output and result of a command, without actually changing anything. For example, `git add --dry-run foo.txt` is used to check if *foo.txt* can be added to the index without actually adding it. The approach generates such commands with the `--dry-run` flag and adds it at any point to the script. If the `--dry-run` flag of some command does not work as expected we would likely observe some violation.

Other Non-Altering Commands *Git* also provides many commands to show information about the repository, such as `git status` to show the status of the files in the working directory and `git log` to show the commit history. The approach adds `sleep 1` commands to the script which will wait for one second before continuing the execution, to probably find timing-related bugs. This does obviously not change the state since *git* is designed to be used by people whose timing is inconsistent in invoking the commands. Also, the invocation of `git init` within an existing non-corrupted repository should not alter the state.

Canceling out Command Sequences As mentioned, pairs of the `git reset` commands can be used to move the HEAD to another commit and back to the former commit afterward. The

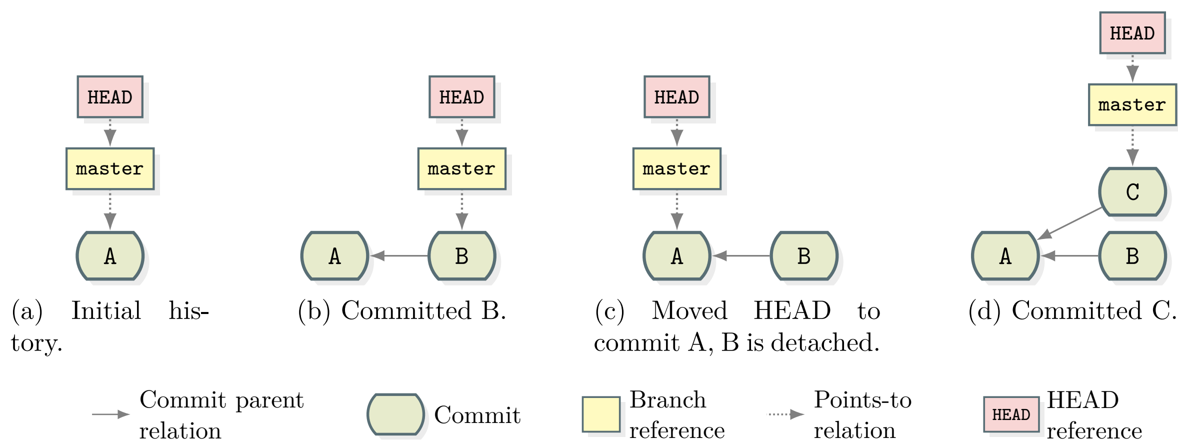


Figure 3.7: *Git* reset example scenario.

HEAD is a pointer to the latest commit in the history of the current branch on which the next commit is based. Figure 3.7 shows an example of this. Starting with a history only containing the single commit A as shown in Figure 3.7a where the HEAD currently points to this commit. When new changes are committed at this state, these changes are based on commit A and the HEAD points to B afterward as shown in Figure 3.7b If we use the `git reset --soft` command to move the HEAD to A as shown in Figure 3.7c, commit B gets detached. This means the commit is not part of the history of any branch. If we commit new changes at this state, these changes are based on commit A and the HEAD points to commit C afterward. Commit B is not part of any branch.

The `--soft` mode causes the command to leave the index and the working tree files untouched and only moves the HEAD. Therefore, a no-op with the `--soft` mode can be used by the approach at each point if there is no merge in progress. During a merge, the command refuses to work. We can also use the `--mixed` mode for a no-op if we take care. This mode is also the default mode if the mode is not specified explicitly. At this mode *git* keeps the state of the files in the working directory but may change the index. If changes to some files are staged these changes can possibly get lost during the reset, and we are not able to revert them at the second reset command. Therefore, this no-op can only be inserted by the approach when there are currently no staged or unstaged changes in the index and no merge in progress.

The `git stash` command enables the user to commit changes to a temporary place to check them out later. Technically the stash is just a branch within *git* that can be accessed by convenience methods. In case a user had made some experimental changes that are not ready to commit yet and the user has just noticed an error in the last commit, the user has to put these experimental changes aside in order fix the former commit. In this scenario, the user can use the `git stash push` command to push the current changes to the stash to get a clean working tree, fix the error, commit the changes and use the `git stash pop` command afterward to restore the stashed changes to the working tree to continue the work. Since `git stash push` and `git stash pop` cancel out each other under specific circumstances, the approach uses them as a no-op command sequence. To apply this transformation, there must be some staged or unstaged changes, and there must be no merge in progress. If staged changes are present, the `git stash pop` command in the

transformation is used with the `--index` option to restore the state of the index and cause these changes to be staged again.

3.5.3 Transformation Selection

To choose one specific transformation, the approach first evaluates how many transformations for each type can be applied to the current test script. Then it multiplies the number of available transformations per type with a type-specific constant factor and uses this number as weight per type. Afterward, a type is chosen and one of the available transformations of this type will be randomly selected and applied by the approach.

3.6 Test Minimization

The minimization of test scripts is helpful to reduce the manual effort to analyze violation reports. For each report, we need to isolate the cause of the violation to be able to decide if the violation is caused by a real bug in *git* or if it is a false positive. A programmatic test minimization component saves a lot of time when narrowing down such a cause. For example, if we assume that no file is altered by the transformation but this does not hold true for a single file, we limit our attention to this file only. During the minimization, we do not need to keep other commands in the test scripts if their presence does not affect this file. So, the general idea of the minimizer is to remove as many commands as possible while not affecting the contents of the files which caused the violation, the files we are interested in. In the ideal case, the minimizer produces a slice of the original program, that is a minimal example to trigger the bug which caused the violation. With respect to the *exit relation*, we assume that the transformed script has the same semantics as the source script. Following this assumption, each command which was not altered by the transformation has to have the same semantic in both scripts. Therefore, during the minimization, the approach only removes commands which are common on both scripts and it removes them either in both scripts or none. With respect to our *exit relation* this is formalized as follows: Let F_i be the file paths in output i and let $D_i(x)$ be the content of file x in output i. Given a source test case t_s with the output o_s and a follow-up test case t_f with output o_f where $o_s \neq o_f$ violates the *exit relation*, for every pair of minimized source and follow-up test cases $t_{s'}$ and $t_{f'}$, the following conditions must hold true: Files removed in o_f compared to o_s must still be removed in $o_{f'}$ and added and modified files in o_f compared to o_s must be equal in o_f and $o_{f'}$. Analogue this must hold true for o_s compared to o_f. This leads to the following equations:

$$\forall a \in F_{o_s} \setminus F_{o_f} : a \notin F_{t_{f'}} \tag{3.6.1}$$

$$\forall a \in F_{o_f}, D_{o_s}(a) \neq D_{o_f}(a) : a \in F_{o_{f'}} \wedge D_{o_f}(a) = D_{o_{f'}}(a) \tag{3.6.2}$$

$$\forall a \in F_{o_f} \setminus F_{o_s} : a \notin F_{t_{s'}} \tag{3.6.3}$$

$$\forall a \in F_{o_s}, D_{o_f}(a) \neq D_{o_s}(a) : a \in F_{o_{s'}} \wedge D_{o_s}(a) = D_{o_{s'}}(a) \tag{3.6.4}$$

For the minimization itself, the approach first uses the Myer algorithm [13] to find the unaltered parts of the source and follow-up test case and then uses a divide and conquer approach to remove

these commands. Our approach is based on the general idea of *delta debugging* introduced by Zeller and Hildebrandt [20]. Thereby our approach recursively first tries to remove a whole block of commands from the scripts and split these blocks into halves until either the block can be removed without affecting the differing files or if one block is empty after the split. This technique does not work if the outcome of a script is not deterministic. Therefore, the approach runs the untouched source and follow-up test case $N_{\text{min_exec}}$ times each, and checks that the behavior is the same for each run. We use a value of $N_{\text{min_exec}} = 100$, determined by our testing. We have to balance between the performance impact and the remaining probability of not detecting a non-deterministic test case. The remaining probability would never be zero anyway. Additionally, the approach uses a second minimization stage to minimize the optional arguments of the remaining commands, analog to the command minimization. This minimization of the optional arguments massively reduces the time to manually analyze the test cases and find similarities between different violations.

If we assume that a fixed number of executions reveals all the different outcomes of a test case and the number of different outcomes is finite, we can derive a new version of this algorithm that can be used to minimize non-deterministic test cases. In this non-deterministic version, we only have one test script that is able to produce two outputs. In contrast, the deterministic version of this algorithm executes each pair of test scripts only once to check if the outputs are as expected. In the non-deterministic version, we execute the script multiple times until we observe all the possible outputs or until we reach the limit of N_{exec} executions, assuming the unobserved outputs will never be produced by this script. After the minimizing, we are left with a minimized script that produces the same outputs in regard to the differing files as the original script before.

4 Implementation

In this chapter, we describe the used technologies, libraries, and the architecture of the application. The approach is implemented in Java for the most part, since Java is available for various platforms and offers various libraries which are beneficial for rapid prototyping. We also use bash for implementing helper functions for the test execution. We use a docker[1] container to compile *git* from its source code, to minimize the dependence on pre-installed software and therefore increase the portability. The application is also designed to run inside a docker container to protect the host computer from the effects of misbehaving test scripts. We use Prometheus[2] to collect statistical data during the execution and a Grafana[3] to visualize this data.

4.1 Architecture

The application is designed in two parts: The worker and the controller component. The worker connects to the controller via Java RMI and it generates, transforms, and executes the test cases. Multiple instances of the worker application can be connected to one instance of the controller at the same time, which enables scalability. When a worker obverses the violation of a relation, it minimizes the corresponding source and follow-up test scripts and reports this information to the controller. To help us later to understand what happened during the test execution, it also provides other information, such as the console output, the contents of the working directory, and the difference between the outputs for both test cases. The controller exports all available information into a directory on the file system to be reviewed by us later. The controller is also able to retrieve test cases from the worker and delegate them to other workers when a worker exits. Each worker executes a user-configurable number of concurrent test cases. The concrete number must be used according to the available resources on the computer. If the number is set too high, it is likely that timeouts occur during test executions. The timeout time can also be configured by the user. When the current number of concurrent test cases already makes use of all available resources, the further increase of the concurrent test cases will result in lower performance. This is because the number of test case executions per time cannot increase, but since each worker thread generates its own input test case, the ratio between the used resources for generating input test cases and the actual execution of the test cases will decrease.

[1] `https://www.docker.com`
[2] `https://prometheus.io`
[3] `https://grafana.com`

The architecture is logically split into two parts: a generic framework for modeling and the concrete models. The framework enables the modeling of commands, transformations, and relations. Additionally, it is able to generate and execute these tests, derive follow-up test cases, and report violations. The other part consists of the concrete models of the *git* commands, the transformations, and relations. This separation increases the likelihood that the application can be adapted to future versions of *git* and to support additional commands.

4.2 Script and Command Model

The modeling of commands and their possible arguments is a crucial part of the application. We need these models to generate concrete commands during the script generation as well as at transforming the commands. Due to the high number of optional arguments of the commands in total, it is desirable to minimize the effort of specifying each option. Therefore, we implement custom Java runtime annotations that allow us to define the optional arguments in a descriptive manner.

All implemented commands, *git* commands as well as other bash or helper commands, implement a common interface called *Command*. This interface provides methods to be used to export the command as string into a bash script, as well as methods to enable the command transformations. The transformation methods are implemented by the *AbstractCliCommand* class, that is the base class for all implemented *git* commands and it also takes care of assembling the command into its string representation to be used in the bash scripts. To do so, it handles the *command option transformation* in a generic and transparent manner and it provides hooks which can be implemented by the inheriting classes to implement command-specific transformations, such as the *commit in other branch* transformation of the `git commit` command or the *command pathspec argument split* transformations. Since the *command pathspec argument split* transformation is used by multiple commands, the splitting functionality itself is implemented by an abstract class which inherits the *AbstractCliCommand* class and is inherited for example by the implementing class for the `git add`, `git mv`, `git rm` and the `git restore` command.

To specify the options as in our option template in Section 3.2.1, we use various annotations which are used to configure values within the *Option* class. Listing 4.1 shows an example of how the definitions of the options from the example in Section 3.2.1 looks similar to the *GitCommitCommand* class of our implementation. Each option definition consists at least of a field of the type *Option* with the type parameter of either *Boolean*, *Boolean[]*, *String* or *String[]* and the *OptionMarker* annotation above this field. This annotation contains at least a formatter that is used to create a string representation of the option value(s) and the definition of the cli names as defined by cli_{OD} of the option definition. The separating character is '=' by default, but it is changed when a special character is used as the last character in the name definition. The environment variable name and configuration key name can be specified as well. The *StringArgumentType* annotation specifies the type of the argument, which tells the generator what kind of value can be used for this argument. The *InterlockedOptionsGroups* annotation specifies which options are not allowed to be generated at the same time. At the most one option per group is set by the generator. Additionally,

Listing 4.1: Part of the option definitions of the `git` base and `git commit` command used to define the same options as in the definition of t_{git} and t_{commit} in the example in Section 3.2.1.

```
1  // Class: GitCommitCommand
2  @InterlockedOptionsGroups(groups = {1})
3  @StringArgumentType(argumentValueType = ArgumentValueType.FILE)
4  @OptionMarker(cliNames = {"-F ", "--file"}, fmt = StringFormatter.class)
5  private Option<String> file;
6
7  @InterlockedOptionsGroups(groups = {1})
8  @StringArgumentType(argumentValueType = ArgumentValueType.SOME_STRING)
9  @OptionMarker(cliNames = {"-m ", "--message"}, fmt = StringArrayFormatter.class)
10 private Option<String[]> message;
11
12 @OptionMarker(cliNames = {"--allow-empty"}, fmt = PositiveBooleanFormatter.class)
13 private Option<Boolean> allowEmpty;
14
15 @StringArgumentType(argumentValueType = ArgumentValueType.COMMIT_ID)
16 @OptionMarker(cliNames = {"-C ", "--reuse-message"}, fmt = StringFormatter.class)
17 private Option<String> reuseMessage;
```

```
1  // Class: GitBaseCliCommand
2  @OptionMarker(cliNames = "--work-tree", envName = "GIT_WORK_TREE", configKey =
   ↪  "core.worktree", fmt = StringOptionFormatter.class)
3  private Option<String> workTree;
```

the generator supports an annotation called *OptionsGenerationSubsetGroups* which takes a set of groups, as the *InterlockedOptionsGroups* annotation does, but functions the opposite way round. The generator will first randomly choose one group and then set arguments of this group only.

The *TestScript* class contains a list of commands and a set of helper functions. These helper functions can be optionally provided by the commands, and they consist of a string with bash code. This string is inserted at the start of the script, before the first command string. They are typically used to define helper functions in bash and to prepare the test directory. For example, the `git commit` command defines such helper functions to collect the commit ID after each commit, which is used to map the real commit IDs to our own logical commit IDs. To collect the ID, the command appends itself with a line similar to `recordCommit "1" "-C gitDir0"; checkoutRecordedCommits "1"`. This stores the ID of the last commit in the repository in *gitDir0* at the logical ID 1. Afterward, the real commit ID can be read again by calling `printCommitId "1"` in the script. For example the approach can checkout this commit with `git checkout $(printCommitId "1")`. The second part of the line after the semicolon is used to gather the information for the *source independent checkout relation*.

When the approach generates a command, it provides an instance of the *GenerationContext* interface to the constructor. This context provides arguments during the option generation and other information, such as the current working directory. During the generation, each command passes a lambda function to the constructor of the *AbstractCliCommand* class, that contains the

constructor of its parent command. Per default, the approach uses the constructor of the *GitBaseCliCommand* as the lambda function, since most of the commands of our implementation are subcommands of `git`.

Additionally, the implementation is able to parse commands from strings into the command objects, while also setting the *Option* fields according to the provided optional arguments. This is implemented for another variant of our approach, described in Section 7.1. We use the picocli[4] library to parse the commands and configure this library with the information of the *OptionMarker* annotations.

4.3 Transformation

As explained before, the command transformations are implemented by the commands themselves. Therefore, the *Command* interface features a method to retrieve the number of possible transformations, which can be applied to this concrete command, and a method to trigger one of these transformations by random and return a list containing the new commands. In the follow-up test script, the current command is replaced by these new commands. The transformation does not alter the internal state of the former command. The immutability of commands allows us to share the command objects of an input test case across all derived follow-up test cases. This has the advantage, that the approach does not need to duplicate all commands of the test case when the follow-up case is derived, which saves time and main memory.

To derive a follow-up test case, the approach evaluates which types of transformation can be applied and how many per type. The transformations are implemented in a tree-like fashion somehow similar to the tree in Figure 3.5. At each node in the tree, the approach counts the number of transformations that can be applied by each child recursively. The transformation count of a node is calculated by the sum of the transformation counts of its children. Since the approach needs to choose a single transformation of all available transformations at the end, it uses these counts as weighting at each branching point, to apply each transformation with the same possibility. This design enables us to easily implement further transformations.

[4] `https://picocli.info`

5 Evaluation

To evaluate the approach we will answer the following research questions within this chapter:
RQ1 How effective is the approach at finding bugs in *git*?
RQ2 How efficient is the approach at finding bugs in *git*?
RQ3 How precise is the approach?
RQ4 How effective is the test minimization?
RQ5 How effective are the individual transformations?

Before answering the questions we will describe the setup used for the evaluation in detail. All tests are executed on a server with two Intel Xeon E5-2650v4 CPUs @2.2GHz with 12 cores each, two threads per core, and 256GB main memory installed. The server runs *Ubuntu 18.04.6 LTS* with docker version 20.10.14 and Java 1.8.0_312

5.1 Approach Effectiveness

During the development, the program has been running on the server for approximately three months. The worker application is started within a docker container and the controller application runs natively on the server. The worker is configured to use 50 worker threads for the concurrent execution of the test cases and the test case generator is configured to use 40 successful command insertions per test case. The number of 50 worker threads is chosen because the server provides 48 CPU threads in total. Since the test cases will generate a lot of disk read and write operations, all test cases are executed on a RAM disk to avoid bottlenecks in the file system access.

To measure the effectiveness for **RQ1** we count the number of found bugs over the course of this project. Within this time frame, we were able to find five bugs in *git*. Typically, we used the minimized test cases from the minimizer to analyze them until we understand the relationships of the commands and manually write a simpler script, to trigger the bug or the violation. These bugs will be explained in the following.

5.1.1 File Move Bug

The `git mv <from> <to>` command is used to rename or move tracked files and directories in the working directory and updates them in the index as well. To do so, the implementation updates the information of the existing files to correspond to the new location and moves the files or directories on the file system. On the file system, the timestamp of the last modification of each file or directory is stored within an attribute, called *ctime*. When a new file or any modifications to a file are added

Listing 5.1: Script to showcase the *git* move bug.

```
#!/bin/bash
git init                    # create git repository
touch bar                   # create empty file bar
git add bar                 # add the file to the index
git commit -am "add bar"    # commit
sleep 1                     # wait 1 second to ensure other ctime
git mv bar foo              # rename bar to foo
git update-index --refresh  # reset fails without this command
git reset --merge HEAD      # reset to last commit
```

Table 5.1: Values of the *ctime* attribute in the file system and the index during the execution of the script in Listing 5.1.

		ctime			
		Without line 8		With line 8	
After line	File	File system	Index	File system	Index
3	bar	1	-	1	-
4 - 6	bar	1	1	1	1
7	foo	2	1	2	1 ⚡
8	foo			2	2

to the index, the value of the *ctime* attribute of this file is also stored in the index. If a file is being moved on the file system, its *ctime* attribute is updated to the current time. As a result of the bug, the *ctime* attributes in the index are not updated during the moving of the files by the `git mv` command. If a file is added to the index and moved with `git mv` within the same tick of the *ctime* value, than the index contains the correct *ctime* value regardless of this bug.

To avoid merge conflicts, the `git reset --merge` command first checks if there are unstaged changes present for files that are different between the working directory and the HEAD. During this check `git reset` compares the *ctime* of the files in the working directory against the *ctime* stored in the index. If they do not match, the command assumes unstaged changes for this file and therefore refuses to proceed. Therefore, a `git reset --merge` fails, if it is executed after a `git mv` command that caused an incorrect index.

Listing 5.1 shows a simple script to reproduce the bug. Table 5.1 shows the *ctimes* values of the index and file system during the execution of this script accordingly. The `sleep 1` command after the commit command on line number 5 ensures that the *ctime* value of the moved file on the file system will be different than before. The time precision of this attribute is platform-dependent. The `git update-index --refresh` command refreshes various attributes of changed files in the index. Since the file was not modified after the renaming, the command updates the *ctime* value in the index to match the *ctime* value on the file system. This causes the `git reset --merge HEAD` command to succeed. If `git update-index --refresh` is removed from this script, the `git reset --merge HEAD` command fails.

For all our transformations the *exit relation* must hold true. Therefore, this must also hold true if we apply no transformation at all. Each script generated by the generator is known to be

Listing 5.2: Script to showcase the effect of the commit ID collision. Line 8 needs to be removed to trigger the bug.

```
1  #!/bin/bash
2  git init                           # create git repository
3  git commit --allow-empty -m init   # initial commit
4  git checkout -b otherBranch        # switch to new branch
5  git commit --allow-empty -m first  # commit in new branch
6  git notes add -m foo               # add note "foo" to last commit
7  git checkout master                # switch back to master
8  sleep 1                            # sleep one second to ensure new timestamp for
                                      ↪  next commit, line not present in source test
9  git commit --allow-empty -m first  # commit in master
10 git notes add -f -m bar            # add note "bar" to last commit (may override)
11 git checkout otherBranch           # switch to other branch again
12 git notes show > myNote.txt        # write note message of last commit to a file
```

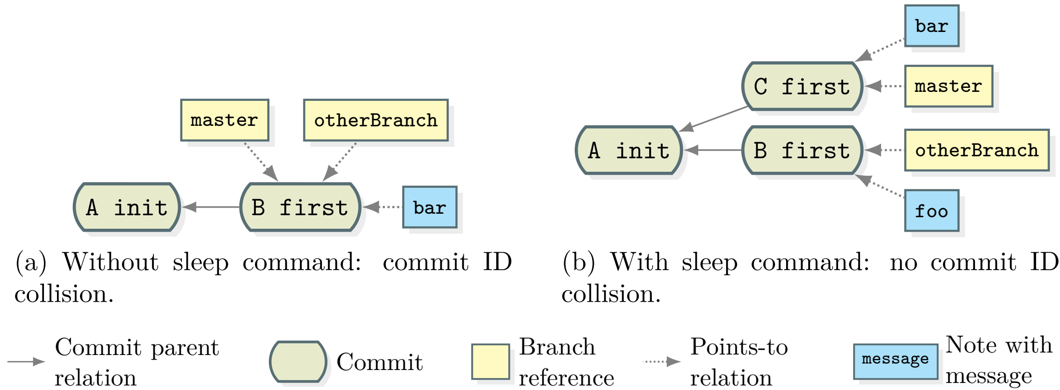
deterministic per design and the output is reproducible. If the violation of any relation is detected, the program will execute the source and follow-up test script $N_{\text{min_exec}}$ times in a row and check if the output is stable, as explained in Section 3.6. Since this bug could be triggered by the timing of the commands, our approach has reported some scripts as being non-deterministic. Thus, we were able to reduce the number of commands of the script while keeping the non-deterministic behavior and filed a bug report[1]. After this, the bug has been confirmed and patched[2] in *git* version 2.36.0.

5.1.2 Commit ID Collision Bug

The `git notes add` command is used to add a textual note to some commit in the history. To add the note, the commit is referenced by its unique ID. The ID is a hash based on the contents of the commit, the ID of the parent, the message, the current timestamp, and other meta information. So, if two commits share the same parent, contents, message, and the other meta information and additionally they are made close to each other regarding clock time, it is possible that they will have the same ID. In fact, in this case they are the same commit since *git* itself uses the commit ID as a unique key to access them. So, it depends on the timing whether the later commit actually adds a second commit to the repository or if they are the same. Since we can attach at the most one note to each commit, we are only able to attach a note to both commits each if they have different IDs. If the second commit gets the same ID, we are unable to add the second note to it. Listing 5.2 shows a script in which the problem occurs. In the first four lines, we create a new repository within the current directory, make an empty initial commit, create a new branch and switch to this new branch. Within lines number 5 to 7, we create an empty commit within the new branch, add a note with the message “foo” to this commit and then switch back to the master branch. On line number 8, we pause the execution for one second. This line is only part of the follow-up test case. On line number 9, we make a new commit within the master branch with the same commit

[1] https://lore.kernel.org/git/84FF8F9A-3A9A-4F2A-8D8E-5D50F2F06203@icloud.com/T/#u

[2] https://github.com/gitgitgadget/git/releases/tag/pr-1187%2Fvdye%2Freset%2Fmerge-inconsistency-v2



(a) Without sleep command: commit ID collision.

(b) With sleep command: no commit ID collision.

Figure 5.1: Histories after the execution of the script of Listing 5.2 with and without the sleep command on line 8 present.

message as on line number 4. In general, we expect that these commits have their own unique ID, which is the case when the sleep command on line number 8 is present. Otherwise, this commit gets the same ID as the commit on line number 4. On line number 10 and 11, we want to add a note with the message “bar” to this new commit and switch back to *otherBranch*. On line number 12, we read the message of the note of the last commit of this branch and write it to a file. When the commit IDs happened to be equal, the second note overrides the first one and the content of the file is “bar”. With the sleep command present, the commit IDs will be different, therefore the message of the first note will still be intact after adding the second note and the content of the file would be “foo”. The resulting histories after executing the test case with and without the `sleep 1` command present is visualized in Figure 5.1.

We have reported this bug[3] but still have not received a response 50 days later when publishing this master thesis, so the error is not officially confirmed.

5.1.3 Checkout No-Op Bug

The *git* documentation states that the `git checkout` command without any arguments is “[...] a glorified no-op with rather expensive side-effects to show only the tracking information, if exists, for the current branch”[4]. This inspired us to use the insertion of plain `git checkout` commands within the *no-op insertion* transformation. Using our approach we discovered that the promise of the documentation does not hold true. We have found scenarios in which the insertion actually caused the violation of the *exit relation*.

Imagine we are currently on the master branch on a clean state, and we are able to merge another branch into the master with the command `git merge --no-ff otherBranch` without getting a conflict. The `--no-ff` option forces *git* to use traditional merging with a merge commit even if a fast-forward merge is possible. In this case, we can add the `--no-commit` option to the command to tell *git*, to prepare but not actually commit the merge commit. This option is intended for the user to be able to verify the result of the merge before committing it. When the user decides that

[3] `https://lore.kernel.org/git/1BD801F4-B2BA-4D6C-A450-5EEB14E8A58A@icloud.com/T/#u`

[4] `https://www.git-scm.com/docs/git-checkout/2.36.0#Documentation/git-checkout.txt-emgitcheckoutemltbranchgt`

Listing 5.3: Minimized test case to trigger the checkout no-op bug.

```
1  #!/bin/bash
2  git init repo1                      # initialize 1st repository
3  cd repo1                            # switch into 1st repository
4  git commit --allow-empty -m first   # first commit in 1st repository
5  git clone . ../repo2                # clone 1st repository
6  touch a                             # create file a in 1st repsitory
7  git add a                           # add 'a' to index of 1st repository
8  git commit --allow-empty -m second  # second commit in 1st repository
9  cd ../repo2                         # switch into 2nd repository
10 git pull --no-ff --no-commit        # pull in changes from 1st repository into 2nd
                                       ↪  repository
11 git checkout                        # should be a no-op
12 git commit --no-edit                # commit merge in 2nd repository
13 git reset --hard                    # remove untracked files of 2nd repository
```

the merge has worked as expected, the user can use the command `git commit`, which is just a shorthand for `git merge --continue`, to actually commit the merge. If we invoke `git checkout` in between `git merge --no-commit --no-ff otherBranch` and `git commit`, the `git checkout` command causes the merge to be aborted and the following `git commit` to fail, since there are no changes to be committed. As already explained, the `git pull` command is internally a fetch and a merge operation, therefore we observe the same problem when invoking `git pull` with the `--no-commit --no-ff` options in a similar scenario. Our approach has inserted a `git checkout` command in the context of the *no-op insertion transformation* right after a `git pull` command with the `--no-commit --no-ff` options which has caused the violation of the *exit relation*. The failed merge operation causes missing changes in the history and when the branch is checked out at a later point, these changes are missing in the working directory which causes the violation of the *exit relation* in the end.

Listing 5.3 shows a minimal example to trigger the bug. In the lines number 2 to 4, we create an empty repository, enter it, and make a first commit. Then we create a second repository which is a clone from the first one, in the next line. On lines number 6 to 8, we create and add a new file to the index of the first repository and commit this change afterward. Then we switch to the second repository and pull this change from the first repository into this repository while using the `--no-commit --no-ff` options for the pull operation. Line number 11 contains the checkout command as documented. This line is not present in the source test case. On line number 12 we commit the merged changes to the history. This operation fails when the `git checkout` command of line number 11 is present. The last line cleans the working directory by removing the untracked files. During the pull operation, the new changes were checked out into the working directory. The `git reset` command cleans the working directory and index to match the HEAD. When the merge operation fails, this clean operation will remove file *a* from the working directory of the second repository.

Listing 5.4: Test case to trigger the stash push fail bug.

```
1  #!/bin/bash
2  git init                          # initialize repository
3  git commit -m init --allow-empty  # initial commit
4  touch a                           # create empty file 'a'
5  git add a                         # add 'a' to the index
6  git stash push                    # stash the creation of 'a'
7  touch b c                         # create empty files 'b' and 'c'
8  git add b                         # add 'b' to the index
9  git add --intent-to-add c         # add the presence of 'c' to the index
10 git stash push                    # push changes to stash
11 git stash pop --index             # pop changes back and restore index
12 git commit --all -m second        # commit changes to all tracked files
13 git clean -f                      # remove untracked files
```

We have reported this bug[5] but still have not received a response 41 days later when publishing this master thesis, so the error is not officially confirmed.

5.1.4 Stash Push Fail Bug

As described in Section 3.5.2.1, the *stash* in *git* is intended to be used to remove changes from the index and working directory, in order to get a clean index and working directory but also with the ability to reapply these changes to the working directory and index at a later point in time. The `git add` command supports the `--intent-to-add` (`-N` in the short version) flag which can be used to make *git* aware of a file that has not yet been tracked so fare, without adding the file to the staging area. After adding a file using this option, `git status` reports an unstaged change of the type *new file* for this file. The `git diff` command shows the difference between the staged files and the working directory. When a new file is created which is not yet added to the staging area, it is untracked and therefore not included in this output. If the user desires the file to be included in that output, the `git add --intent-to-add` command is intended to accomplish this.

The problem was discovered by our approach after inserting a sequence of a `git stash push` and a `git stash pop` command within the context of the *no-op insertion* transformation. The `git stash push` command failed because a file was added via the `git add --intent-to-add` earlier in the test case, which has caused the following `git stash pop` command to try to pop other changes from the stash than intended. The difference of the files in the working directory has caused the *exit relation* to get violated.

Listing 5.4 shows a test case which triggers this bug. On lines number 2 to 5, we create the repository, make an initial commit, create a new empty file *a* and add this file to the index. Afterward, we stash the changes of the staging area. This operation removes file *a* from the index and working directory, and creates a stash entry. On lines number 7 to 9, we create two empty files *b* and *c*, add file *b* to the index and use the `git add --intent-to-add` command to add the creation of the file *c* to the unstaged changes. The commands on lines number 10 and 11 were inserted by

[5] `https://lore.kernel.org/git/D0A0CC41-C41E-4856-B969-2A6DD3C14079@icloud.com/T/#u`

Listing 5.5: Minimal example to showcase the pull fails after commit `--dry-run` bug.

```
1  #!/bin/bash
2  git init repo1                     # initialize 1st repository
3  cd repo1                           # switch to 1st repository
4  touch a                            # create empty file a in 1st repository
5  git add a                          # add file a to the index in 1st repository
6  git commit -m one --allow-empty    # first commit in 1st repository
7  git init ../repo2                  # initialize 2nd repository
8  cd ../repo2                        # switch to 2nd repository
9  git commit -m two --dry-run        # dry-run commit in 2nd repository
10 git pull --rebase "../repo1"       # try to pull from 1st repository
```

the *no-op insertion* transformation and were not present in the source test case. These commands are supposed to push the changes to the stash and pop them back afterward. Due to the `git add` command on line number 9, the `git stash push` command will fail. If the command would succeed, the working directory and index are in a clean state afterward. The remaining changes in the index, caused by the failed stash push, causes the `git stash pop --index` command on line number 11 not to work as expected. The stash pop operation requires the index to not contain any unstaged changes of tracked files. This failed pop operation causes the staged changes of the files to get lost, which causes the files *b* and *c* to not get tracked anymore. The `git commit` command on line number 12 commits the changes of all tracked files and the `git clean` command on line number 13 removes all untracked files. Since the files *b* and *c* are not tracked anymore, they are removed by this command in the follow-up test case, which leads to an empty working directory. In the source test case without lines number 10 and 11, these files are still tracked and therefore not removed by the clean operation.

We have reported this bug[6] but still have not received a response 41 days later when publishing this master thesis, so the error is not officially confirmed.

5.1.5 Pull Fails After Commit `--dry-run` Bug

We have found this bug by using another technique to generate the input test cases, which is described in Section 7.1. A `git commit` command with the `--dry-run` option is supposed to show the information about the files, that are supposed to be included in the next commit, without creating a commit. The approach has detected a violation of the *exit relation* after inserting a `git commit --dry-run` command in the context of the *no-op insertion* transformation.

Listing 5.5 shows a minimal test case to trigger the bug and cause the violation of the *exit relation*. On lines number 2 to 6, we create the first repository, create an empty file and add it to the index, and create a commit. On lines number 7 and 8, we create the second repository and switch to it. Line number 9 with the `git commit --dry-run` command is only present in the follow-up test case. On line number 10, we pull the changes from the first repository into the second one. In the source test case, this operation succeeds and file *a* is created within the second

[6] https://lore.kernel.org/git/111D7753-AE53-4906-A7AF-F39EA7455CA3@icloud.com/T/#u

repository. In the follow-up test case, after the insertion of the `git commit --dry-run` command, this operation fails. Therefore, file *a* is not created and the *exit relation* violated.

This bug is reported[7] and a patch is proposed by a *git* contributor.

5.1.6 Result

The fact that the approach was able to discover five real-world bugs in the latest version of *git* proves that it is effective in finding bugs in *git*. Since the *commit ID collision bug* only occurs in special edge cases, it may not have a lot of impact on real-world use-case scenarios of *git*. The *pull fails after commit* `--dry-run` *bug* also has a relatively small impact on real-world use-case scenarios. However, the other three bugs are likely to be triggered in real-world use-case scenarios. Especially the *file move bug* has a high impact since it is likely to be triggered at any usage of the `git mv` command without showing any symptoms.

5.2 Approach Efficiency

To evaluate **RQ2**, we measure various data during a 48-hour execution period of the implementation of our approach on the server. The setup consists of four docker containers. One for the controller application, one for the worker application, one for Prometheus to collect the statistics, and one for Grafana to observe and visualize the measurements. As before, the worker is configured to use 50 worker threads for the concurrent execution of the test cases, the test case generator is configured to use 40 successful command insertions per test case, and a RAM disk is used. We stop the program after 48 hours and evaluate the data. The metrics are collected all 15 seconds by Prometheus from the worker. The worker implements a Prometheus endpoint to provide these metrics.

We measure the following metrics:

Transformation time We measure the time it takes to derive each follow-up test case from the source test case by applying a transformation. This also includes the time it takes to evaluate which transformations can be applied to this source test case. This is implemented as a sum of all measured time intervals and a counter that holds the number of summed intervals.

Test case execution time and number of executed test cases We measure the time it takes to prepare the working directory, export the test case script, execute the test case script, and collect the output according to our relations. This is implemented as a sum of all measured time intervals and a counter that holds the number of summed intervals.

End-to-end test run execution time After the execution of each test case script, we need to check the relations and either derive a new follow-up test case or even generate a new test case from scratch when the former one has reached the maximum number of executions. This metric measures the time of such a full test run cycle. This is implemented as a sum of all measured time intervals and a counter that holds the number of summed intervals.

Input test case generation time and number of generated test cases We measure the time it takes to generate a new test case from scratch as well as the number of generated test cases.

[7] `https://lore.kernel.org/git/B0458F2D-C6B9-41AE-8F2F-39C1D2AEE6BD@icloud.com/`

Table 5.2: Measured metric values.

Metric	Sum [s]	Count	Average ($\frac{\text{sum}}{\text{count}}$) [$s$]
Transformations	1,047.355	2,077,819	0.000504
Test case executions	7,324,738	2,094,586	3.497
End-to-end test run execution time	8,617,006	2,086,176	4.131
Input test case generations	1,233,757	8,450	146.007
Script executions during test generations	1,229,618	894,575	1.375
Successful script executions during test generations	-	218,923	-

This is implemented as a sum of all measured time intervals and a counter that holds the number of summed intervals.

Number of script executions during test case generation We count the number of script executions during the generation of new test cases. The generator appends new commands to the test case and tries to execute it. We also measure the time for these executions. This is implemented as a sum of all measured time intervals and a counter that holds the number of summed intervals.

Number of successful script executions during test case generation We count the number of scripts that possibly can be executed successfully, after the addition of newly generated commands to a test case during its generation. This number also represents the number of successful command insertions.

Table 5.2 shows the measurement results of the experiment. Within the observed time, the approach executed 2,094,586 test cases. This leads to an average of $\frac{2{,}094{,}586}{172{,}800s} \approx 12.121$ executions per second and an average execution time of 3.497 seconds. One end-to-end test run execution takes 4.131 seconds on average by $\frac{2{,}086{,}176}{172{,}800s} \approx 12.073$ executions per second. The application of a transformation takes 0.504 milliseconds on average. A number of 8,450 input test cases have been generated, which take about 146 seconds per test case on average. This leads to an average of $\frac{1{,}233{,}757s}{2{,}086{,}176} \approx 0.591$ seconds per generated script per end-to-end test run execution. During the input test case generation 218,923 of 894,575 script executions were successful. This is a success rate of 24.5% and a duration of $\frac{1{,}229{,}618s}{218{,}923} \approx 5.617$ seconds on average per command insertion.

When a new test case is generated and being used as source test case for the first time, we execute both the source and follow-up test case within this first end-to-end test run. In the subsequent end-to-end test runs, we use the former follow-up test case as source case. Since this test case had been executed before, we just need to execute the new follow-up test case. This is reflected in the difference between the number of end-to-end test run executions compared to the number of test case executions of $2{,}094{,}586 - 2{,}086{,}176 = 8{,}410$ which almost matches the number of generated test inputs. The iterative application of n transformations onto a source test case results in a chain of $n + 1$ test cases in total. This is reflected in the difference between the number of end-to-end test run executions and the number of applied transformations of $2{,}086{,}176 - 2{,}077{,}819 = 8{,}357$ which is also about the same as the number of generated test inputs.

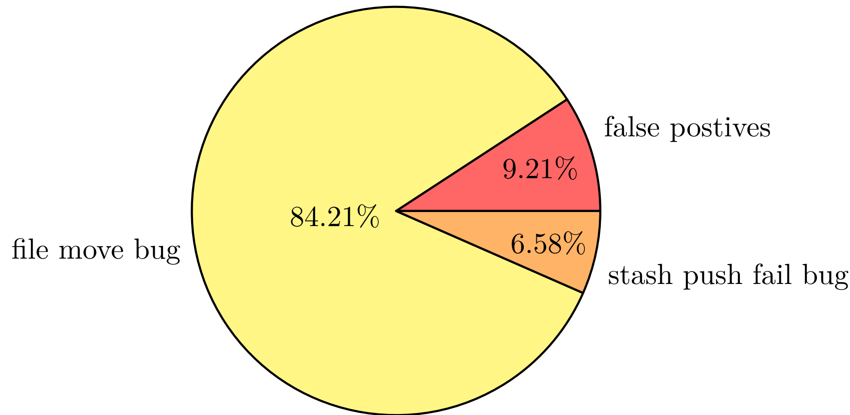


Figure 5.2: Distribution of the various causes of the warnings generated by the approach during the 48-hour period.

5.3 Approach Precision

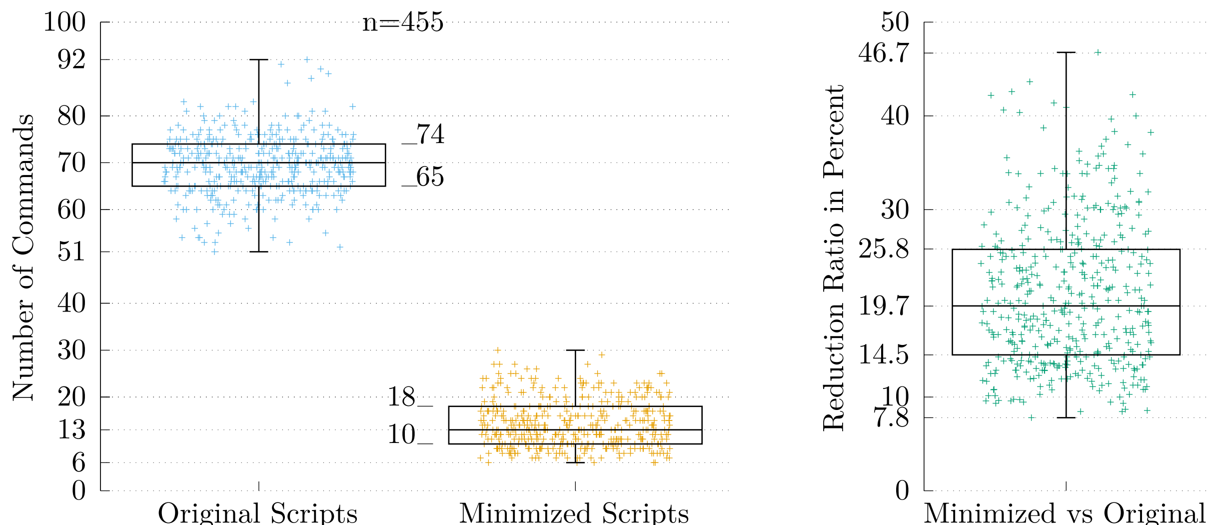
To address **RQ3**, we analyze all warnings produced by the approach within a 48-hour execution period on the server. The setup consists of two docker containers: one for the controller application and one for the worker application. As before, the worker is configured to use 50 worker threads for the concurrent execution of the test cases, the test case generator is configured to use 40 successful command insertions per test case, and a RAM disk is used. After the 48-hour period, we have manually traced the cause of each generated warning.

We observe 76 warnings. The cause of 64 warnings is the *file move bug*, which is 84.21%. The *stash push fail bug* is the cause of 5 warnings, which is 6.58%, and the remaining 7 warnings are false positives, which is 9.21%. Figure 5.2 shows the distribution as a pie diagram.

As seen, the approach results in a precision of 90.79%. A high precision is important for the real-world use of the approach since the manual analysis of warnings is a time-consuming task. So, we want to waste as little time as possible by analyzing false positive warnings.

5.4 Test Minimization Effectiveness

To answer **RQ4**, we compare the number of commands in the original follow-up test script of 455 violation reports against the number of commands in the corresponding minimized follow-up test script. These violations have been collected during the development of the approach within approximately 2 months, using the setup as described in section 5.1. We measure the number of logical commands described by our model, excluding the special commands inserted as convince to provide us orientation during manual evaluation of the reports. These commands just output numbers to the console between each other command, which enables us to relate lines of the console outputs to lines in the test script. The minimization of the optional arguments of the commands themselves is not taken into account for this evaluation.



(a) Number of commands in the original and minimized test scripts.

(b) Reduction ration of the number of commands in the original test scripts compared to the minimized ones.

Figure 5.3: Evaluation of the test minimizer. Comparison between original and minimized test scripts with box plots. Whiskers indicate the minimum and maximum. The two boxes in between indicate the first and third quantile.

Figure 5.3a shows the distribution of the counted commands. Each colored +-sign represents a single data point. The vertically centered line in each box represents the median value. The top line of each box represents the median of the upper half of the values and the bottom line of each box represents the median of the lower half of the values. The whiskers indicate the minimum and maximum value.

The original test scripts consist of 52 to 92 commands with a mean value of 70 commands. In comparison, the minimized test scripts consist of 6 to 30 commands with a mean value of 13 commands. Even in a worst-case scenario, a minimized script contains less than 50% of the commands of the original script. In a best-case scenario, we archive a reduction of up to 7.8% in the number of the commands of the original test script. On average, we archive a reduction of $\frac{1}{5}$. The archived reduction of each script pair is plotted in Figure 5.3b. The reduction is calculated by dividing the number of commands in a minimized test script by the number of commands in the corresponding original test script.

5.5 Transformation Effectiveness

To address **RQ5**, we analyze one violation report for each bug described in Section 5.1. We analyze which transformations can be found in the minimized test case and are therefore relevant to trigger the bug. To do so, we manually compare the commands of the minimized test case against the corresponding input test case. Additionally, we look at the number of transformations applied to

the input test case until the bug was triggered. These numbers are collected by the program and are part of the violation reports.

File Move Bug For the *file move bug* we cannot answer this question because of the non-deterministic nature of this bug. It is possible that the bug is already present in a generated input test case and but its symptoms are triggered after multiple executions only now for the first time. So, the last applied transformation before the symptoms are observed may not be involved in triggering the bug and therefore cannot be made responsible. Additionally, it is possible that the symptoms are observed even before the generation of the input test case is finished. In this case, there is even no transformation to blame.

Commit ID Collision Bug This bug was triggered after applying the *checkout in other branch* command transformation. A `git commit` command was transformed to be made in another branch, which is merged back via a traditional merge into the former branch. This was the 13th transformation in the iterative transformation chain and the 133rd transformation for this input test case in total. So, it is the 8th chain that starts with this input test case. The minimized test case contains this single transformation only, so the other 12 transformations are not relevant for triggering the bug.

Checkout No-Op Bug The *checkout no-op bug* has been triggered after the insertion of a `git checkout` command in the context of the *no-op command insertion* transformation. This was the 8th transformation in the iterative transformation chain and the 218th transformation of this input test case in total. This inserted checkout command is the only transformation present in the minimized test case, so the 7 other transformations are not relevant for triggering this bug.

Stash Push Fail Bug This bug was triggered after inserting a `git stash push -a` command followed by a `git stash pop --index` command in the context of the *no-op command insertion* transformation. These commands are inserted in the 4th transformation in the iterative command chain and the 169th transformation of the input test case in total. As before, this transformation is the only transformation present in the minimized test case.

5.5.1 Result

As seen, one of these bugs is triggered by the *checkout* insertion of the *no-op insertion* transformation, one is triggered by the *stash-push stash-pop* insertion of the *no-op insertion* transformation, and one is triggered by the *checkout in other branch* command transformation. The first is a simple no-op insertion without any preconditions and the latter two are more complex, as they can only be applied in particular circumstances, and they consist of multiple operations. So, there is a tendency that more complex transformations are more effective than less complex ones. There is no evidence for the other transformations to be effective in the current configuration. We are surprised that the various output modes of the *command option* transformation have not caused any violation,

since *gits* change log regularly mentions the fix of problems of ignored configuration keys in special instances.

6 Related Work

Peng Wu proposed the approach of *iterative metamorphic testing* and compared it against traditional metamorphic testing and special case testing. *Iterative metamorphic testing* applies multiple transformations to a single test case. To be precise a source test is used to generate a follow-up test and after testing, this follow-up test will be reused as the source test again for the next iteration, for n times in total. This approach leads to a high test case generation efficiency with a higher fault detection capability than traditional metamorphic testing. [19] We have used this technique in our approach with a n-value of 15.

Chen et al. introduced a technique to increase the fault detection rate with fewer test cases called *Equivalence-Class Coverage for Every Metamorphic Relation* (ECCEM). For their method, they first divide the input domain of the tested program into different equivalence classes based on their properties. Then they analyze the relation between the class of the source and follow-up test case and the applied metamorphic relation, to decide if the relation can be abandoned for some source test case classes. This will reduce the number of generated test cases while increasing the fault detection rate. [3] We were not able to use this approach because of the huge size of our input domain. It is not feasible to manually partition our input domain into equivalence classes.

Zeller and Hildebrandt proposed two *delta debugging* algorithms to automatically minimize failing test cases and isolate the failure cause. The approach works on chunks of text. A given failing test case is reduced until the removal of any further character causes the fault to disappear. The result is an isolated failure. In case a pair of failing and non-failing tests are given, the algorithm can isolate the failure-inducing difference between these two. [20] The algorithm is greedy and can therefore output a local minimum rather than the global minimum. Our test case minimization algorithm is based on this approach.

Regehr et al. proposed an approach to use the concept of delta debugging to minimize C-programs that trigger bugs in the compiler. The original delta debugging approach can lead to wrong results on C-programs, caused for example by the creation of uninitialized variables which lead to unpredictable results. To solve this, they extend the original approach by adding a second criterion to determine if the generated slice is a valid C-program, according to their definition. This must hold true for each result candidate. [14] In our test case minimization algorithm, we remove parts within the logical model and then regenerate the script from this representation. This enforces the validity of each test case.

Segura et al. have released a survey on metamorphic testing in which they collect and analyze the techniques of 119 related papers and give a forecast about future trends. They have shown

the development and expansion of metamorphic testing techniques since the introduction by Chen et al. in 1998. [16]

Murphy et al. have designed and have implemented a framework for the automated metamorphic testing of machine learning programs, called *amsterdam*. The framework supports six types of transformations and various relations, which can be configured by the user. [12] The high-level idea is similar to our approach. Unfortunately, this framework cannot be used for VCS, since its transformations are applied to machine learning models and its relations use the corresponding output data.

Botella and Gotlieb have introduced the approach of *automated metamorphic testing* to test programs written in a subset of the C-language. First, they transform the program into an equivalent *constraint logic program* (clp). Second, they convert a fault-based model of the relation into a goal to be solved by the rules of the clp. To find violations, they use a solver to solve this constraint system. [2] This approach cannot be applied to our problem of testing VCS, since it is limited to programs written in a subset of C. Our approach aims at VCS in general. Even if *git* had been written in this subset of C, it is likely that the resulting constraint system would contain too many variables to be solved in a feasible amount of time.

Dong et al. proposed an algorithm that uses a criterion based on the path-coverage to decide on the next follow-up test case. They have shown an increased effectiveness by using this approach compared to classical n-iterative metamorphic testing. [7] Since they use symbolic execution for the evaluation of the paths, it is not trivial to apply this approach onto a large and complex program as *git*, which uses hash functions, network communication, and file system access.

Zolkipli et al. have analyzed different kinds of VCS in general and have summarized the ideas which have been established in these systems. [21] Additional, Koc and Tansel have published a survey describing the inner working of typical operations within such systems with a formal representation. [10] These papers are related to our testing of VCS, since these papers thoroughly describe VCS, which is required for testing.

7 Future Work

In this section, we will address various limitations of the approach and discuss possibilities for improvement.

One limitation of the approach is caused by the limited amount of available runtime information during the follow-up test case derivation. Since commit IDs differ in each execution of a test case, we are not able to collect them with our runtime information collection approach. To overcome this during the initial generation of the test cases, we use our own logical commit IDs, which are substituted by the actual commit IDs during the execution. Since we are unaware of which commits are present in the current repository, it is possible that we use the reference of a commit ID at a point in time even when this commit is not reachable. This is feasible during the input test case generation since we check if the generated command can be executed successfully. But we cannot use this strategy to decide if a commit reference is valid when we derive a follow-up test case since we are unable to differentiate if the command fails as a result of an invalid commit reference or as a result of a bug.

A possible way to increase the efficiency of the approach is the use of metrics to measure the quality of the applied transformations and the generated input test cases. An obvious candidate is the measurement of code coverage. During the transformation and generation, we can generate multiple candidates and choose one to maximize the coverage of poor covered parts of *git*.

Another improvement is the implementation of additional relations. Possible relations are the comparison of the exit code of each command in the source and follow-up test case, the comparison of the number of commits in the histories in both test cases, or the comparison of the console outputs of specific commands in both test cases.

In addition, the approach can be improved by implementing more transformations. A transformation could export a part of the history as a patch, reset the history to the start of the patch and import the patch again. It is also possible to create a second working directory via *gits working directory* feature, to make changes within this working directory, revert them and remove the working directory again.

In this master thesis, we tested our approach only with *git*, however, the approach focuses on the testing of VCS in general and should therefore be adapted to other VCS, for example, *mercury*.

7.1 Script Parsing

The *git* project itself uses automated tests within its CI/CD pipeline to ensure that new changes do not introduce bugs. When a bug is reported and fixed, the fix will often contain new test cases

Listing 7.1: Example of a simple test script.

```
#!/bin/sh
test_description='demo test'
. ./test-lib.sh

test_expect_success 'setup' '
    git init && :>a &&
    git add a &&
    git commit -m a'

test_expect_success 'rm test' '
    # test one code
    git rm a &&
    ! test_path_is_file a'

test_expect_success 'echo test' "
    # test two code
    echo \$ > b &&
    test_path_is_file b"

test_done
```

Setup code (lines 2–8)

Code of the first test case (lines 10–13)

Code of the second test case (lines 15–18)

to ensure that this bug cannot be silently reintroduced by a future change. The tests are written as shell scripts and use a self-written testing harness. At this point in time, the test scripts consist of about twenty-five thousand test cases split across about one-thousand files[1] with about two hundred thousand lines of code. As the tests contain scenarios targeting edge cases as well as the basic functionality, it will be logical to use the test cases as input test cases for our program. This section describes our approach to making use of these test cases and the encountered problems.

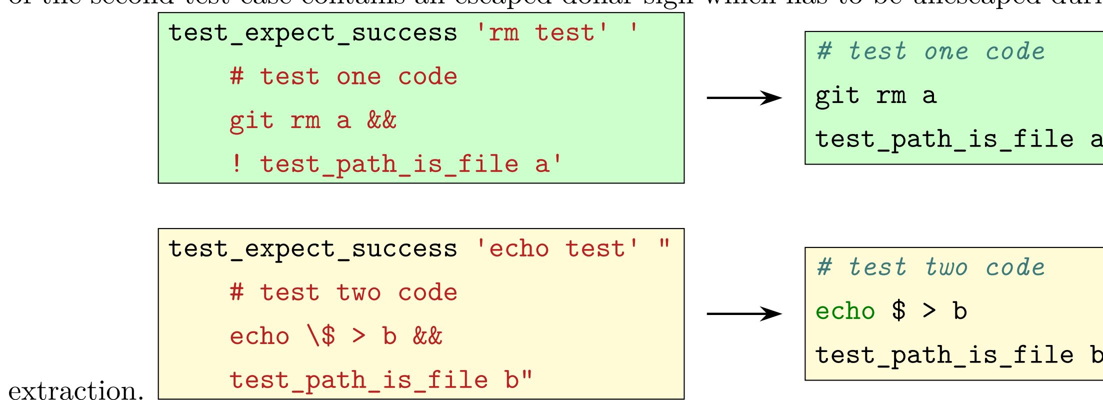
To extract the test cases, we need to parse these shell scripts. Due to the complexity of the shell syntax, it is not trivial to parse and interpret shell scripts statically. The testing harness will create a working directory for each test initialized with a *git* repository, providing many helper functions, and will for example set the path of the global *git* configuration file to a file within the test directory, so that the interactions with *git* do not alter any files outside of the testing directory. The two functions `test_expect_success` and `test_expect_failure` are the most relevant helper functions for us since they are used to execute test cases. They both take two string arguments, one with the name of the test case and the other with the commands for this test case. They also take a third optional argument with preconditions to run this test case, for example, on specific platforms only. The commands in this string are typically chained by the `&&`-operator, so that the whole chain will fail when the first command fails. Since we need to represent the single commands with our command model to be able to transform them later, we need to split up these chains and parse every single command on its own. Besides the testing harness, the *git* project also contains

[1] `https://github.com/git/git/tree/v2.35.3/t`

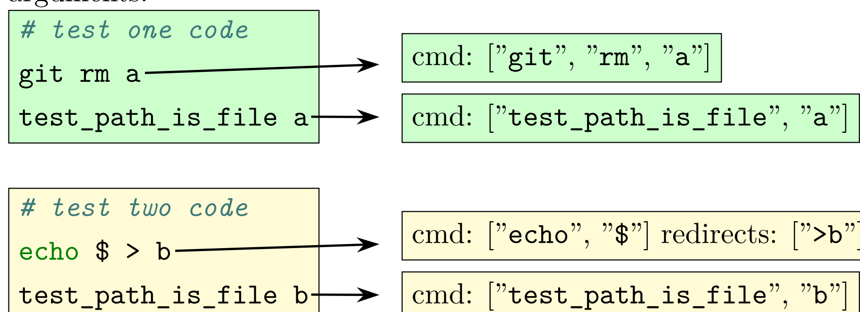
other smaller helper scripts that provide helper functions for specific operations and are loaded by some of the tests. Additionally, some of the tests will define their own helper functions or will contain code to prepare the test cases. If we want to extract some test cases on their own out of a script, we need to keep these imports or the setup code with the extracted test case. Summarized, to represent the scripts with our logical model, we need to apply the following processing steps to the scripts. We show this with an example.

1. Parse each test script and determine the test cases and their setup code as shown in Listing 7.1. All the test cases with special words in their names, such as *setup*, as well as the code above the test case, that is not part of another test case, will be tainted as setup-code.

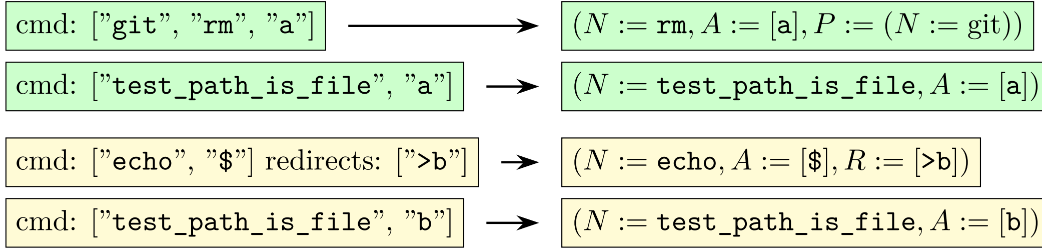
2. Extract the string contents, parse the content as a script, and split up the chains into single command strings. In our example, the last command of the first script contains an exclamation mark which will negate the result of the command. Since we split up the chains, the result of a command does not matter anymore, so we can discard the exclamation mark. During the extraction, the contents must be unescaped according to the used quotes. The `echo` command of the second test case contains an escaped dollar sign which has to be unescaped during the extraction.



3. Parse each command string as script to split up the command into the parts, as the shell is doing. These parts are for example redirects, environment variables and list of command arguments.



4. Parse the list of command arguments to find the corresponding command template and fill the values of the template according to the command arguments.



Unfortunately, we have encountered various limitations using this approach. At first, we have had problems extracting the test cases. Static bash parsing itself is not a trivial problem, as seen by the approach of Davis et al. to convert bash scripts to python [5]. We have found a single library for Java capable of parsing bash. Also, it does not support all the syntax used in the test scripts. We have solved this by applying manual and automatic pre-processing steps to these scripts. The next problem is the detection of the test cases and the related code for each test case. These scripts are not indented for automatic extraction of the test cases. Each script is a little different from the other ones. In some test scripts, the test cases consist of one setup block and each test case will revert the state to this initial state at the beginning. Other test cases depend on the state produced by the test case before. Another problem is the use of self-defined functions within the scripts. Some scripts define functions to invoke the test cases. These test cases are not found by our approach, since we search for the invocation of the `test_expect_success` or `test_expect_failure` function. Additionally, these self-defined functions of the test scripts are also often used within the test cases themselves. When we parse them, we do not know the semantic of the function and we cannot transform the command within the function as well. A possible solution can be to rewrite scripts by substituting all function calls with the code of the function itself. Due to the complexity of bash, this is not trivial to accomplish and beyond the scope of this thesis.

The next problem is the semantic of the test cases themselves. Since *git* is so complex, it is difficult to implement the relations and transformations in such a way, that they function as desired in every possible case. When using scripts generated by our random input test generator, we have control of how the commands interact with *git* and which assumptions can be made about the state of the repositories during the test executions. It enables us to start the development with a small subset of commands and implement more commands while taking care of our assumptions. Additionally, many test cases are not deterministic regarding the output that we are using for the relations. As soon as a command pipes its output to a file in the working directory and the output contains either a commit ID or the current timestamp, the content of this file is likely to change for each execution.

While using this approach to gather input test cases, we have found a single bug in *git* as described in Section 5.1.5. Nevertheless, this approach features poor effectiveness and it requires future work to address the open issues.

8 Conclusion

In this thesis, we design and evaluate an approach to use metamorphic testing within an automated testing tool targeted at version control systems. We implement our approach to test the popular version control system *git*. The approach consists of two parts: a random test input generator and a set of metamorphic relations and transformations.

We define a logical model to describe command templates and use them to instantiate concrete commands for the test input generator. We also use this model to transform these commands.

Using our approach, we found five bugs in the current version of *git*. This shows that it is possible to use our approach, and metamorphic testing in general, to automatically test VCS.

The main challenge of applying the approach to a real-world application, such as *git*, is to develop stable transformations and relations. This is a difficult and time-consuming task since the documentation of *git* is in general not precise about the specific conditions in which an operation is expected to succeed and which behavior is valid, especially for rare edge cases. More precise documentation or specification enables more efficient development of metamorphic relations and transformations and will therefore allow exploiting more complex scenarios.

Bibliography

[1] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(05):507–525, may 2015. ISSN 1939-3520. doi: 10.1109/TSE.2014.2372785. URL `https://doi.org/10.1109/TSE.2014.2372785`.

[2] B. Botella and A. Gotlieb. Automated metamorphic testing. In *2013 IEEE 37th Annual Computer Software and Applications Conference*, page 34, Los Alamitos, CA, USA, nov 2003. IEEE Computer Society. doi: 10.1109/CMPSAC.2003.1245319. URL `https://doi.ieeecomputersociety.org/10.1109/CMPSAC.2003.1245319`.

[3] L. Chen, L. Cai, J. Liu, Z. Liu, S. Wei, and P. Liu. An optimized method for generating cases of metamorphic testing. In *2012 6th International Conference on New Trends in Information Science, Service Science and Data Mining (ISSDM2012)*, pages 439–443, 2012.

[4] T. Y. Chen, S. C. Cheung, and S. Yiu. Metamorphic testing: A new approach for generating next test cases. *CoRR*, abs/2002.12543, 1998. doi: 10.48550/arXiv.2002.12543. URL `https://doi.org/10.48550/arXiv.2002.12543`.

[5] I. J. Davis, M. Wexler, C. Zhang, R. C. Holt, and T. Weber. Bash2py: A bash to python translator. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 508–511, 2015. doi: 10.1109/SANER.2015.7081866. URL `https://doi.org/10.1109/SANER.2015.7081866`.

[6] N. Deepa, B. Prabadevi, L. Krithika, and B. Deepa. An analysis on version control systems. In *2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE)*, pages 1–9, 02 2020. doi: 10.1109/ic-ETITE47903.2020.39. URL `https://doi.org/10.1109/ic-ETITE47903.2020.39`.

[7] G. Dong, C. Nie, B. Xu, and L. Wang. An effective iterative metamorphic testing algorithm based on program path analysis. In *Seventh International Conference on Quality Software (QSIC 2007)*, pages 292 – 297, 11 2007. ISBN 978-0-7695-3035-2. doi: 10.1109/QSIC.2007.4385510. URL `https://doi.org/10.1109/QSIC.2007.4385510`.

[8] G. Jahangirova. Oracle problem in software testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 444–447, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350761. doi: 10.1145/3092703.3098235. URL `https://doi.org/10.1145/3092703.3098235`.

[9] S. Just, K. Herzig, J. Czerwonka, and B. Murphy. Switching to git: The good, the bad, and the ugly. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 400–411, 2016. doi: 10.1109/ISSRE.2016.38. URL `https://doi.org/10.1109/ISSRE.2016.38`.

[10] A. Koc and A. U. Tansel. A survey of version control systems. *ICEME 2011*, 2011.

[11] H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen. How effectively does metamorphic testing alleviate the oracle problem? *IEEE Transactions on Software Engineering*, 40(1):4–22, 2014. doi: 10.1109/TSE.2013.46. URL `https://doi.org/10.1109/TSE.2013.46`.

[12] C. Murphy, K. Shen, and G. Kaiser. Automatic system testing of programs without test oracles. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, page 189–200, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583389. doi: 10.1145/1572272.1572295. URL `https://doi.org/10.1145/1572272.1572295`.

[13] E. W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(1):251–266, 1986.

[14] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for c compiler bugs. *ACM SIGPLAN Notices*, 47, 06 2012. doi: 10.1145/2254064.2254104. URL `https://doi.org/10.1145/2254064.2254104`.

[15] B. S. Scott Chacon. *Pro Git*. Apress, 2014. ISBN 1484200772. URL `https://github.com/progit/progit2/releases/download/2.1.339/progit.pdf`.

[16] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9):805–824, 2016. doi: 10.1109/TSE.2016.2532875. URL `https://doi.org/10.1109/TSE.2016.2532875`.

[17] S. Tolksdorf, D. Lehmann, and M. Pradel. *Interactive Metamorphic Testing of Debuggers*, page 273–283. Association for Computing Machinery, New York, NY, USA, 2019. ISBN 9781450362245. URL `https://doi.org/10.1145/3293882.3330567`.

[18] E. J. Weyuker. On Testing Non-Testable Programs. *The Computer Journal*, 25(4):465–470, 11 1982. ISSN 0010-4620. doi: 10.1093/comjnl/25.4.465. URL `https://doi.org/10.1093/comjnl/25.4.465`.

[19] P. Wu. Iterative metamorphic testing. In *29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, volume 1, pages 19–24, 2005. doi: 10.1109/COMPSAC.2005.93. URL `https://doi.org/10.1109/COMPSAC.2005.93`.

[20] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, feb 2002. ISSN 0098-5589. doi: 10.1109/32.988498. URL `https://doi.org/10.1109/32.988498`.

[21] N. N. Zolkipli, A. Ngah, and A. Deraman. Version control system: A review. *Procedia Computer Science*, 135:408–415, 01 2018. doi: 10.1016/j.procs.2018.08.191. URL `https://doi.org/10.1016/j.procs.2018.08.191`.

Selbsständigkeitserklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Datum	**Unterschrift**